Algorithmes pour les graphes

Christine Solnon

INSA de Lyon - 3IF

2015

Positionnement du module / U.E. d'IF

Unités d'enseignement du département IF :

- Système d'Information
- Réseaux
- Architectures matérielles
- Logiciel Système
- Formation générale
- **Développement logiciel (DL)**
- **Méthodes et Outils Mathématiques (MOM)**

Modules de l'U.E. DL en 3IF :

- Introduction à l'algorithmique
- Modélisation UML
- Programmation OO / C++
- Génie logiciel

Modules de l'U.E. MOM en 3IF :

- Algèbre linéaire
- Bases de l'I.A.
- Images
- Probabilités
- Théorie de l'info. et crypto.
- Traitement du signal
- **Algo. pour les graphes**

Référentiel des compétences

Approfondissement de compétences abordées au semestre 1 :

- Choisir les structures de données adaptées à la situation
- Déterminer la complexité d'un algorithme
- Prouver la correction d'un algorithme

$\rightsquigarrow$ Implémenter de bons logiciels

Nouvelles compétences :

- Modéliser et résoudre des problèmes à l'aide de graphes
  - Reformuler un nouveau problème à résoudre en un problème connu de la théorie des graphes
  - Choisir le bon algorithme pour résoudre le problème
  - Savoir adapter un algorithme connu de la théorie des graphes à un contexte particulier
- Identifier la classe de complexité d'un problème

Organisation

4 séances de cours

- du 2 février au 12 mars

4 séances de travaux dirigés (TD)

- du 17 février au 16 mars

1 devoir surveillé (DS)

- le 26 mars

Pour en savoir plus...

- **Sur l'algorithmique en général :**
  - *Algorithmique*
    T. Cormen, C. Leiserson, R. Rivest, C. Stein
    Editions Dunod - 2010
- **Sur les graphes :**
  - *La théorie des graphes*
    Aimé Sache
    Collection "Le sel et le fer", n°22
    Editions Cassini - 2003

Modélisation basée sur les graphes

Euler : Il était une fois, en 1735, dans la ville de Koenigsberg. . .

[Image empruntée à Wikipedia]

Définition

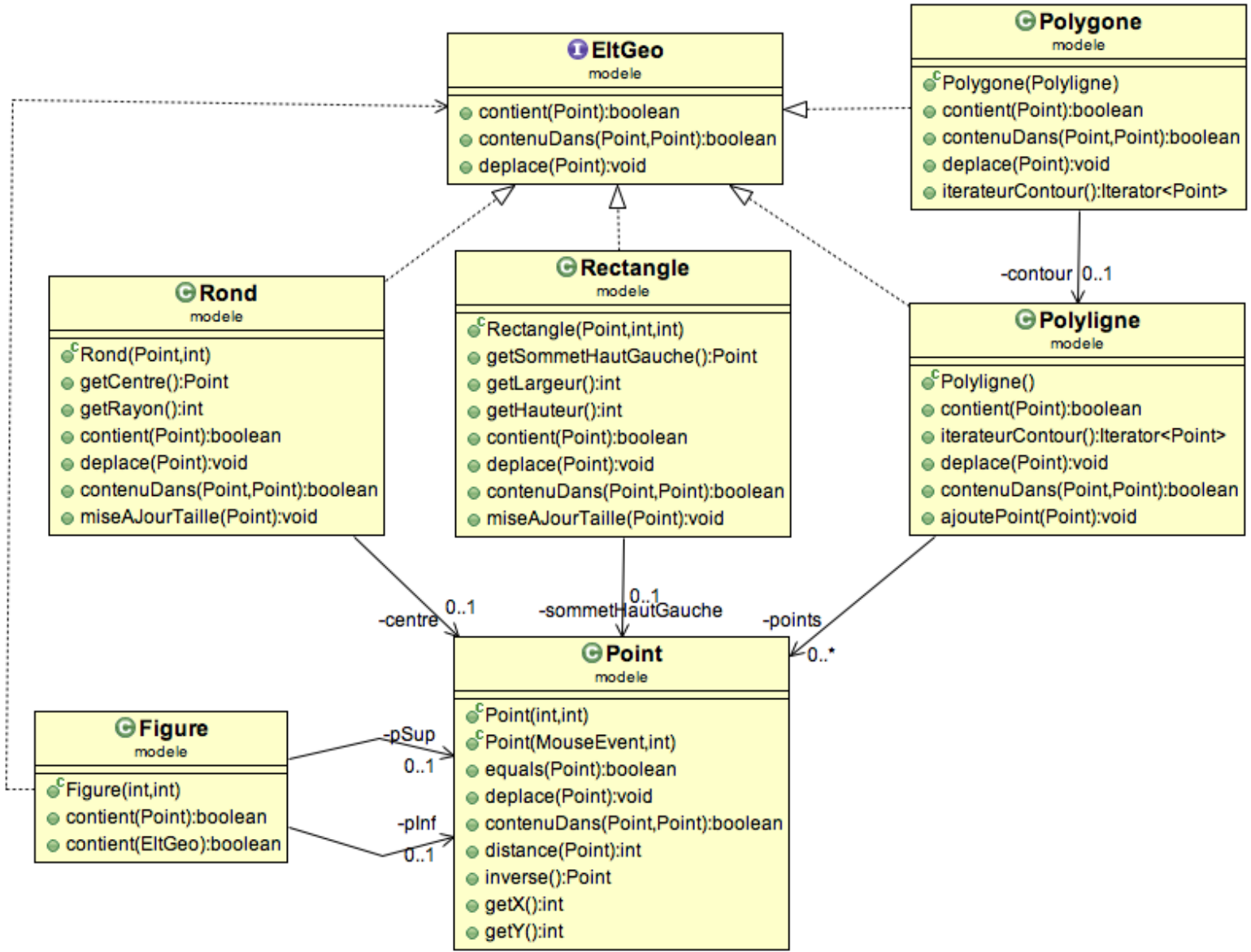
Un graphe est défini par un couple (S, A) tel que

- S est un ensemble de sommets (ou nœuds)
  $\rightsquigarrow$ Composants du modèle
- $A \subseteq S \times S$ est un ensemble d'arêtes (ou arcs)
  $\rightsquigarrow$ Relation binaire entre les composants du modèle

Sommets et arêtes peuvent être étiquetés par des propriétés

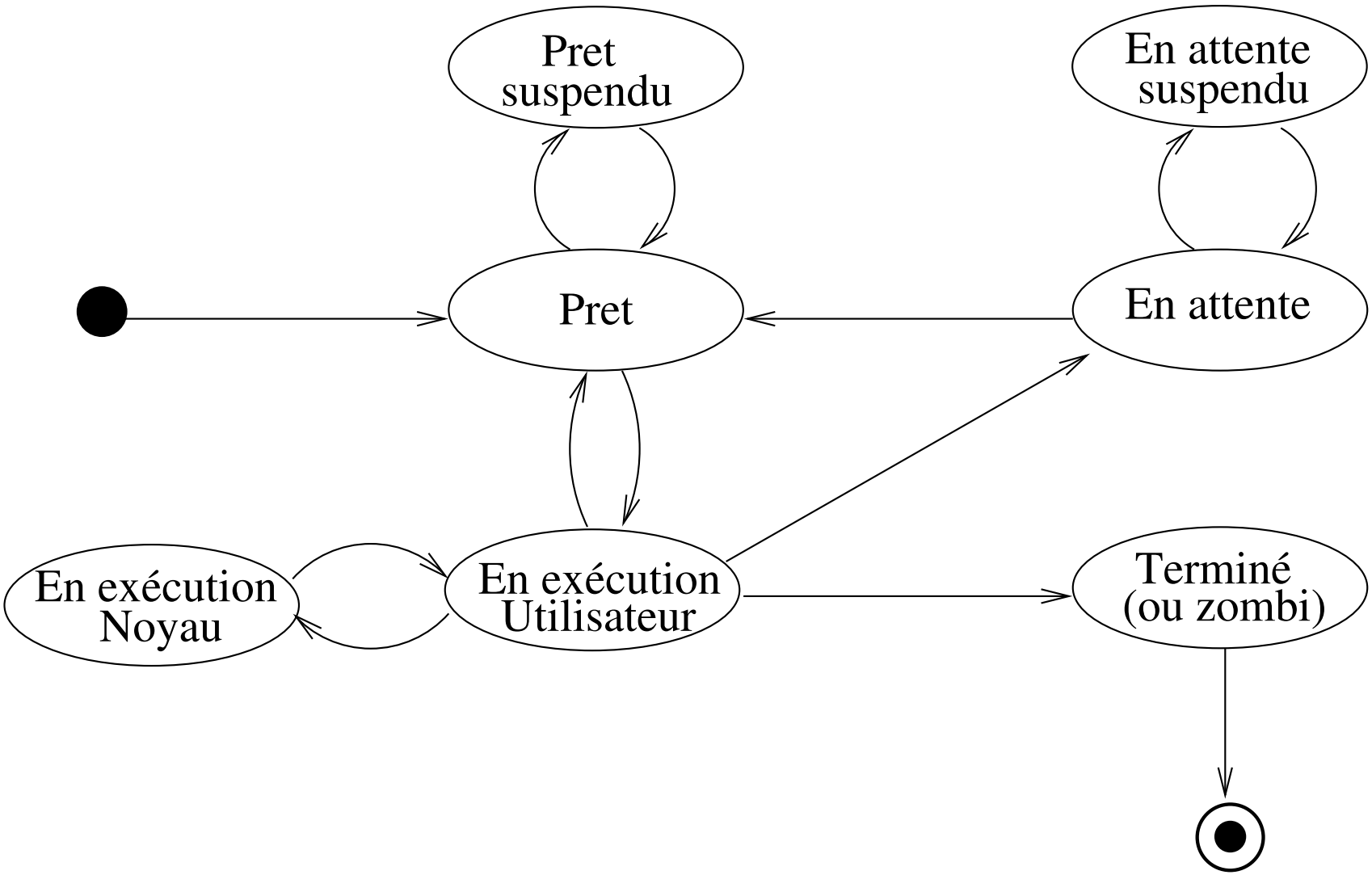
Exemples de modélisation par des graphes

Diagrammes de classes UML



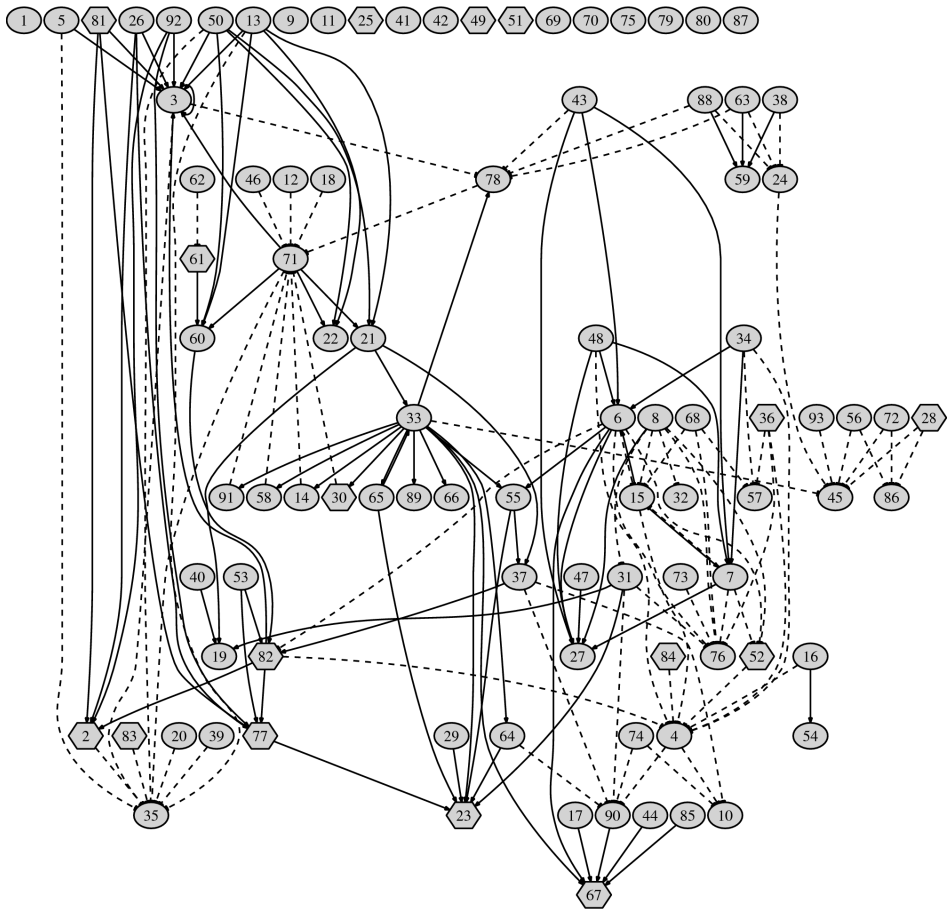
Exemples de modélisation par des graphes

Diagrammes d'états-transitions UML modélisant les états d'un processus



Exemples de modélisation par des graphes

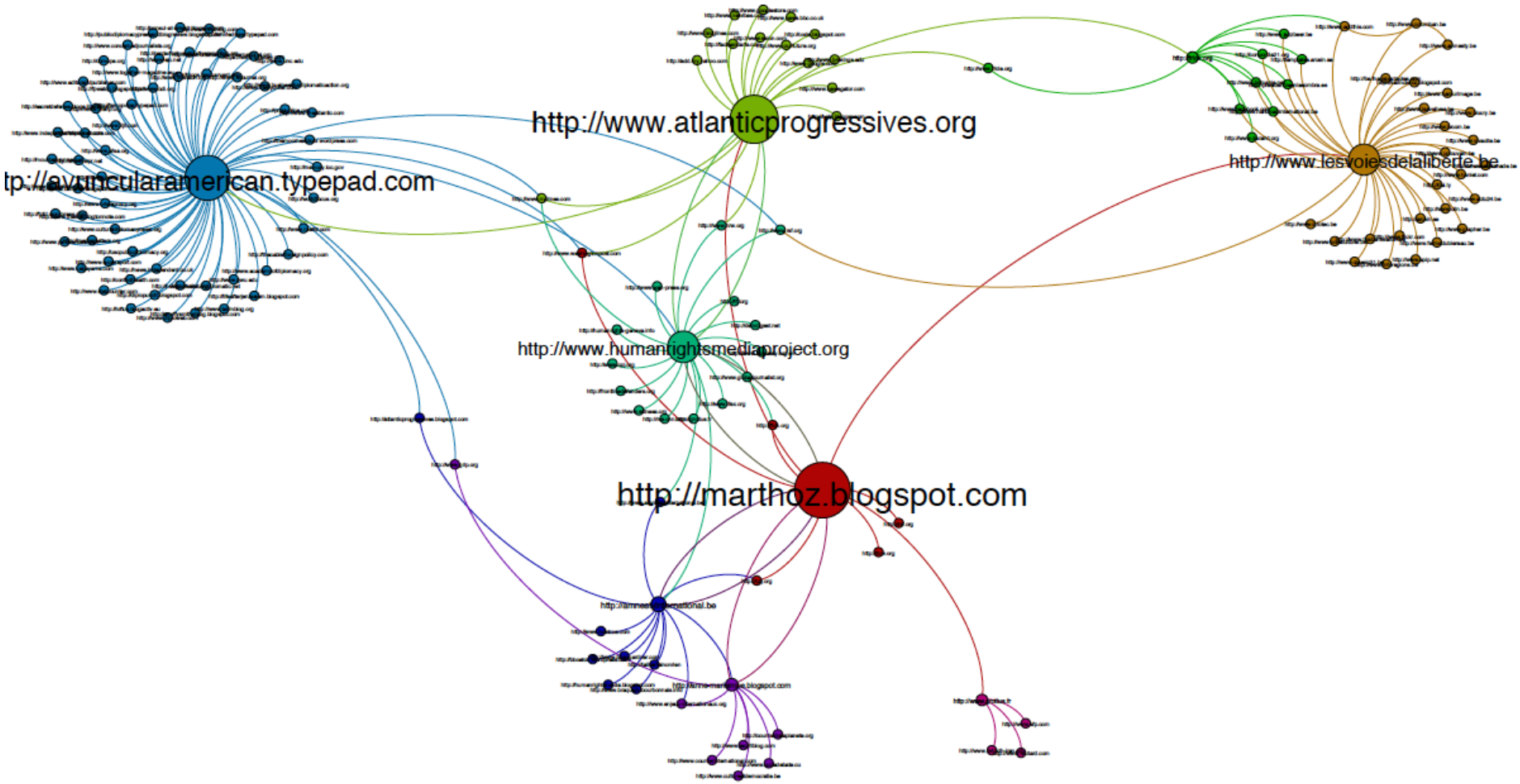
Réseaux de régulation génétique



- Sommets = gènes
- Arcs = influence entre gènes

Exemples de modélisation par des graphes

Réseaux sociaux



- Sommets = URL de blogs
- Arcs = Hyper-liens

[Image empruntée à 7bis.wordpress.com/tag/reseaux-sociaux/]

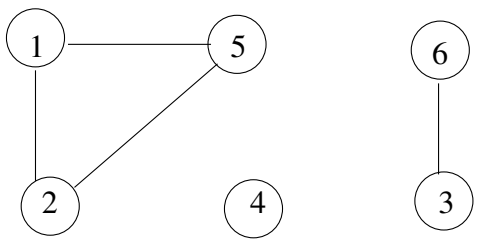
Graphes non orientés

Définition

$G = (S, A)$ est non orienté si $\forall (s_i, s_j) \in S \times S, (s_i, s_j) \in A \Leftrightarrow (s_j, s_i) \in A$

$\rightsquigarrow$ La relation binaire définie par A est symétrique

Exemple :



$$\begin{aligned} S &= \{1,2,3,4,5,6\} \\ A &= \{(1,2),(2,1),(1,5),(5,1), \\ &\quad (5,2),(2,5),(3,6),(6,3)\} \end{aligned}$$

Terminologie :

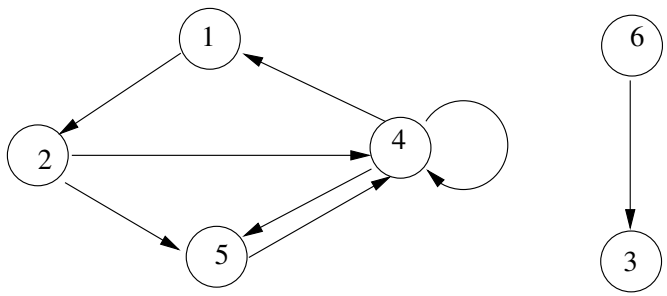
- Les éléments de A sont appelés arêtes
- s_i adjacent à s_j si $(s_i, s_j) \in A$: $adj(s_i) = \{s_j | (s_i, s_j) \in A\}$
- degré d'un sommet = nombre de sommets adjacents : $d^\circ(s_i) = |adj(s_i)|$
- Graphe complet si $A = \{(s_i, s_j) \in S \times S | s_i \neq s_j\}$

Graphes orientés

Définition

$G = (S, A)$ est orienté si $\exists (s_i, s_j) \in S \times S, (s_i, s_j) \in A$ et $(s_j, s_i) \notin A$
$\rightsquigarrow$ La relation binaire définie par A n'est pas symétrique

Exemple :



$$\begin{aligned} S &= \{1,2,3,4,5,6\} \\ A &= \{(1,2),(2,4),(2,5),(4,1),\\ &\quad (4,4),(4,5),(5,4),(6,3)\} \end{aligned}$$

Terminologie :

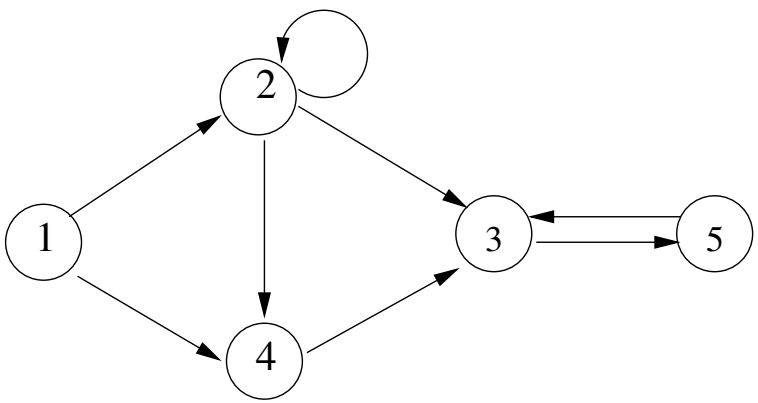
- Les éléments de A sont appelés arcs
- s_j successeur de s_i si $(s_i, s_j) \in A$: $succ(s_i) = \{s_j | (s_i, s_j) \in A\}$
- s_j prédécesseur de s_i si $(s_j, s_i) \in A$: $pred(s_i) = \{s_j | (s_j, s_i) \in A\}$
- demi-degré extérieur = nombre de successeurs : $d^{\circ +}(s_i) = |succ(s_i)|$
- demi-degré intérieur = nombre de prédécesseurs : $d^{\circ -}(s_i) = |pred(s_i)|$

Graphes partiels

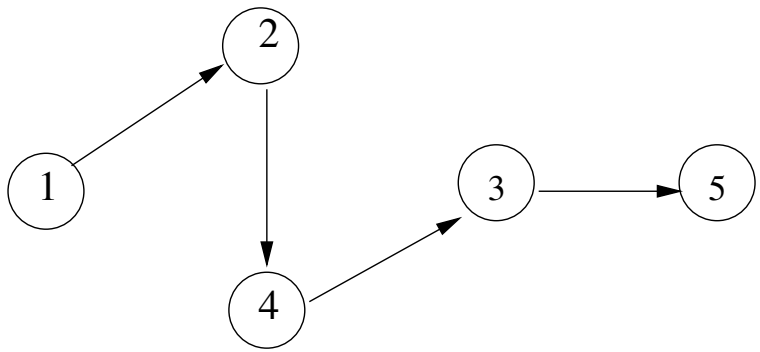
Définition

$G' = (S, A')$ est un graphe partiel de $G = (S, A)$ si $A' \subseteq A$

Exemple :



Graphe $G = (S, A)$



Graphe partiel de G

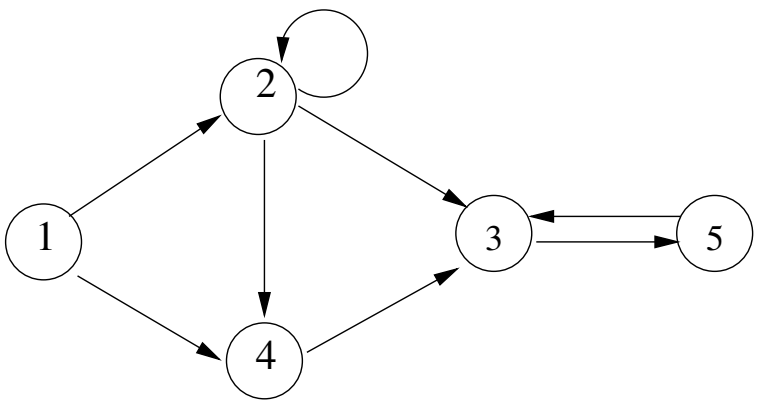
Sous-graphes

Définition

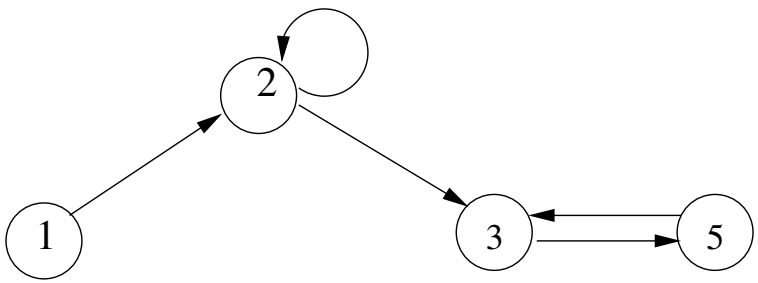
$G' = (S', A')$ est un sous-graphe de $G = (S, A)$ si $S' \subseteq S$ et $A' = A \cap S' \times S'$

$\rightsquigarrow$ G' est le sous-graphe de G induit par S'

Exemple :



Graphe $G = (S, A)$

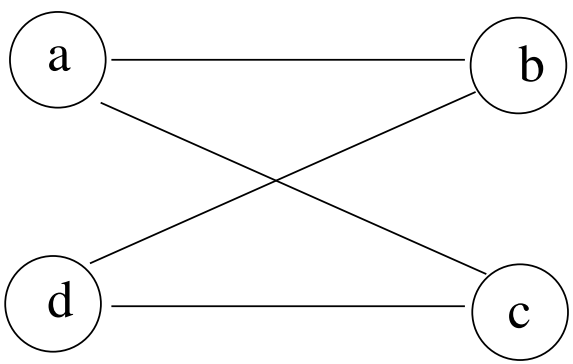


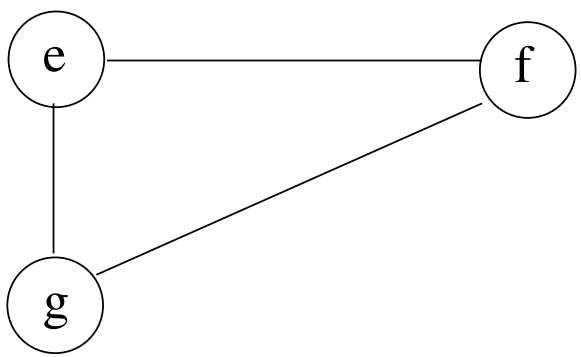
Sous-graphe de G induit par $\{1, 2, 3, 5\}$

Cheminements et connexités

Définitions dans le cas d'un graphe non orienté $G = (S, A)$

- Chaîne = Séquence de sommets $< s_0, s_1, s_2, ..., s_k >$ (notée $s_0 \sim s_k$) telle que $\forall i \in [1, k], (s_{i-1}, s_i) \in A$
- Longueur d'une chaîne = Nombre d'arêtes dans la chaîne
- Chaîne élémentaire = Chaîne dont tous les sommets sont distincts
- Cycle = Chaîne commençant et terminant par un même sommet
- Boucle = Cycle de longueur 1
- $G = (S, A)$ est connexe si $\forall (s_i, s_j) \in S^2, s_i \sim s_j$
- Composante connexe de G = sous-graphe de G connexe et maximal

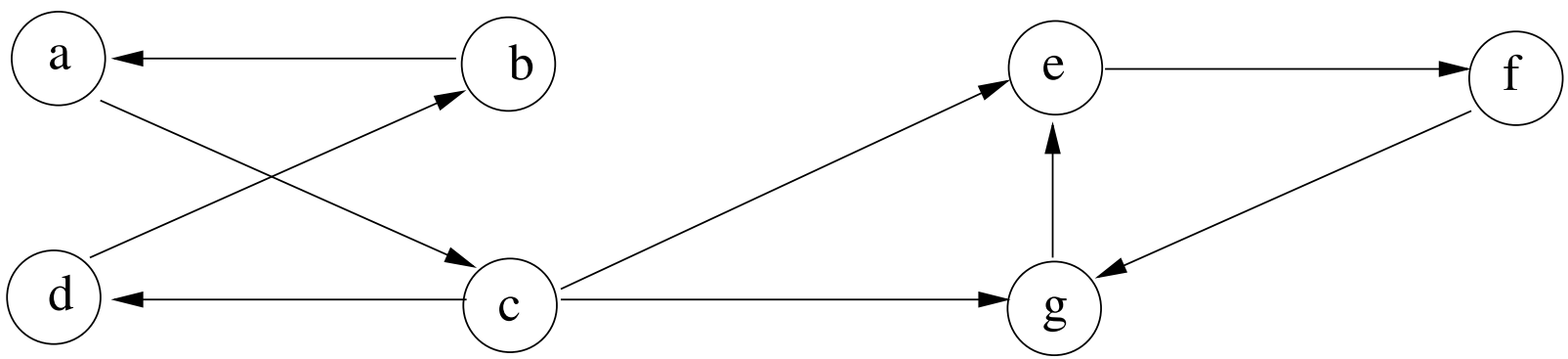




Cheminements et connexités

Définitions dans le cas d'un graphe orienté $G = (S, A)$

- Chemin = Séquence de sommets $< s_0, s_1, s_2, ..., s_k >$ (notée $s_0 \rightsquigarrow s_k$) telle que $\forall i \in [1, k], (s_{i-1}, s_i) \in A$
- Longueur d'un chemin = Nombre d'arcs dans le chemin
- Chemin élémentaire = Chemin dont tous les sommets sont distincts
- Circuit = Chemin commençant et terminant par un même sommet
- Boucle = Circuit de longueur 1
- $G = (S, A)$ est fortement connexe si $\forall (s_i, s_j) \in S^2, s_i \rightsquigarrow s_j$
- Composante fortement connexe = ss-graphe fortmnt connexe maximal



Arbres et Arborescences

Définition d'un arbre :

Graphe non orienté $G = (S, A)$ vérifiant une des propriétés suivantes :

1. G est connexe et sans cycle
2. G est sans cycle et possède $|S| - 1$ arêtes
3. G est connexe et admet $|S| - 1$ arêtes
4. G est sans cycle, et en ajoutant une arête, on crée un cycle élémentaire
5. G est connexe, et en supprimant une arête, il n'est plus connexe
6. $\forall (s_i, s_j) \in S \times S$, il existe exactement une chaine entre s_i et s_j

Si 1 des propriétés est vérifiée, alors les 5 autres le sont aussi

Définition d'une forêt :

Graphe non orienté dont chaque composante connexe est un arbre.

Définition d'une arborescence :

Graphe orienté sans circuit admettant une racine $s_0 \in S$ tel que $\forall s_i \in S$, il existe un chemin unique allant de s_0 vers s_i

Exemple d'algorithme :

```
1 Procédure afficherSucc(g, s_i)
      Entrée          : Un graphe g et un sommet s_i de g
      Postcondition   : Affiche les successeurs de s_i
2     pour tout sommet s_j ∈ succ(s_i) faire
3         afficher(s_j)
```

Complexité de cet algorithme ?

Exemple d'algorithme :

```
1 Procédure afficherSucc(g, s_i)
      Entrée          : Un graphe g et un sommet s_i de g
      Postcondition   : Affiche les successeurs de s_i
2     pour tout sommet s_j ∈ succ(s_i) faire
3         afficher(s_j)
```

Complexité de cet algorithme ?

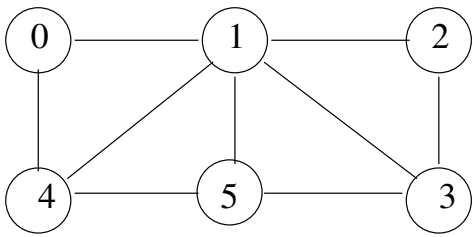
Dépend des structures de données utilisées pour représenter le graphe !

Matrice d'adjacence

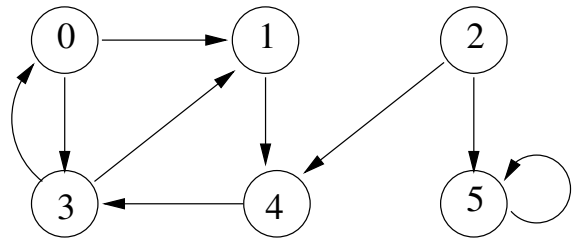
Définition : matrice d'adjacence d'un graphe $G = (S, A)$

Matrice M telle que $M[s_i][s_j] = 1$ si $(s_i, s_j) \in A$, et $M[s_i][s_j] = 0$ sinon

Exemples :



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



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

Complexité

Complexité en mémoire :

$\rightsquigarrow \mathcal{O}(n^2)$ avec $n =$ nombre de sommets de g

Complexité en temps pour déterminer si (s_i, s_j) est un arc :

$\rightsquigarrow \mathcal{O}(1)$

Complexité en temps de *afficherSucc*(g, s_i) :

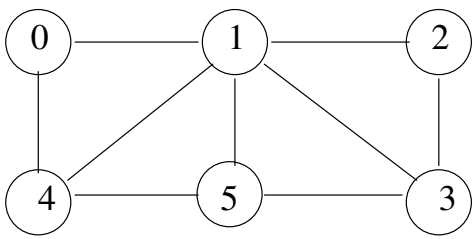
$\rightsquigarrow \mathcal{O}(n)$ avec $n =$ nombre de sommets de g

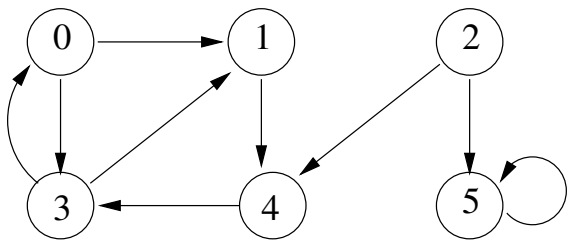
Listes d'adjacence

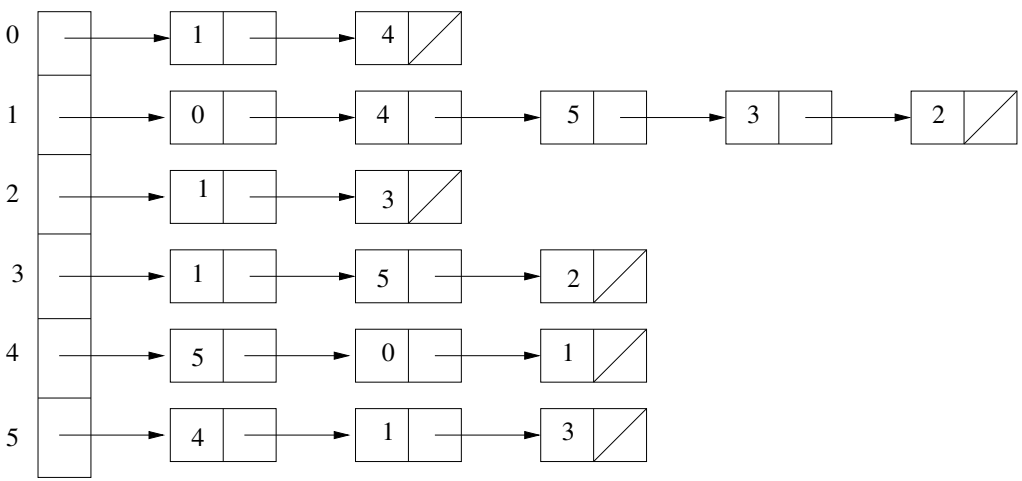
Définition : listes d'adjacence d'un graphe $G = (S, A)$

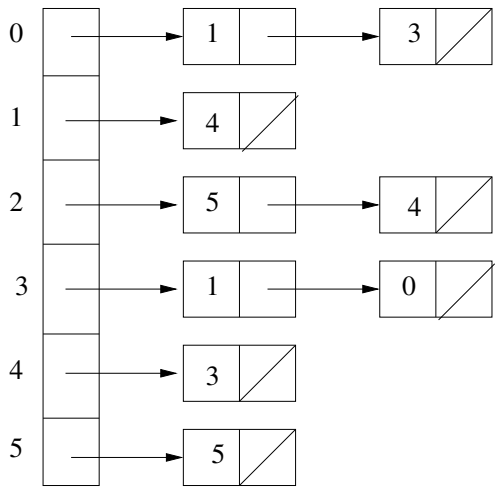
Tableau *succ* tel que $succ[s_i]$ = liste des successeurs de s_i

Exemples :









Complexité

Complexité en mémoire :

$\rightsquigarrow \mathcal{O}(n+p)$ avec $n =$ nombre de sommets de g et $p =$ nombre d'arcs

Complexité en temps pour déterminer si (s_i, s_j) est un arc :

$\rightsquigarrow \mathcal{O}(d^\circ(s_i))$

Complexité en temps de *afficherSucc*(g, s_i) :

$\rightsquigarrow \mathcal{O}(d^\circ(s_i))$

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

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

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

$\rightsquigarrow$ Jusqu'à ce que tous les sommets soient noirs ou blancs

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

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

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

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

$\rightsquigarrow$ Jusqu'à ce que tous les sommets soient noirs ou blancs

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

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

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

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

$\rightsquigarrow$ Jusqu'à ce que tous les sommets soient noirs ou blancs

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

Spécification d'un algorithme de parcours

1 **Fonction** *Parcours*(g, s_0)
- **Entrée** : Un graphe g et un sommet s_0 de g
- **Postcondition** : Retourne l'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
- Mémorisation dans un tableau π tel que $\pi[s_i] = \textit{null}$ si s_i est racine, et $\pi[s_i] =$ père de s_i sinon

Exemple :

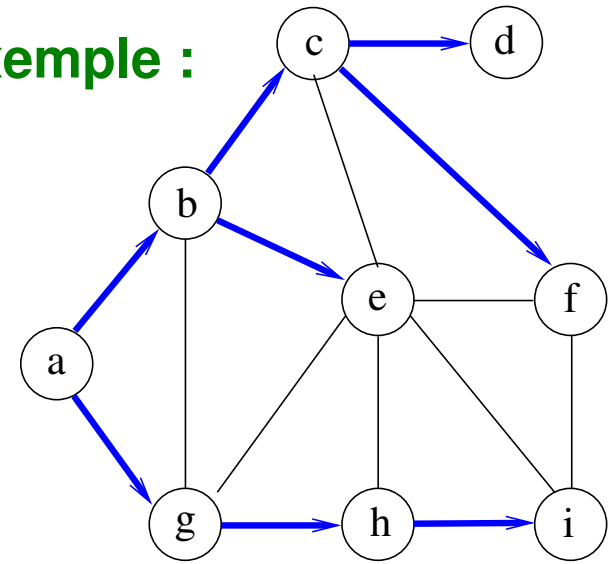
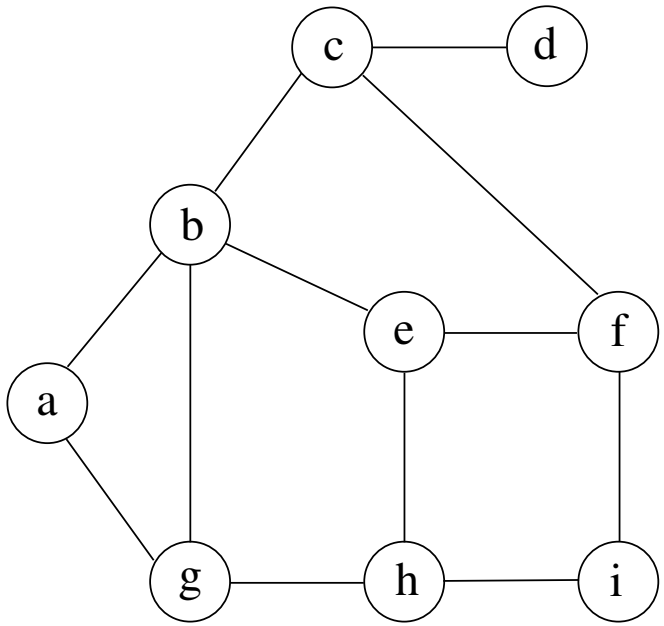


Tableau π correspondant :

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```

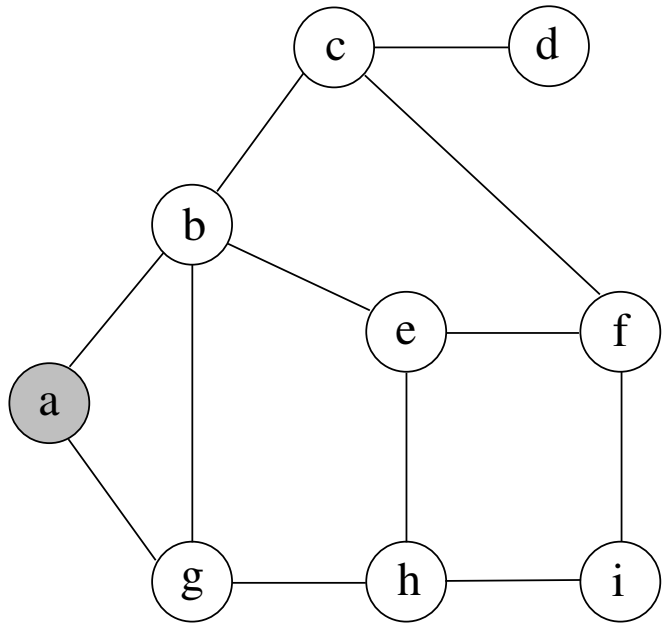


$f = <>$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```

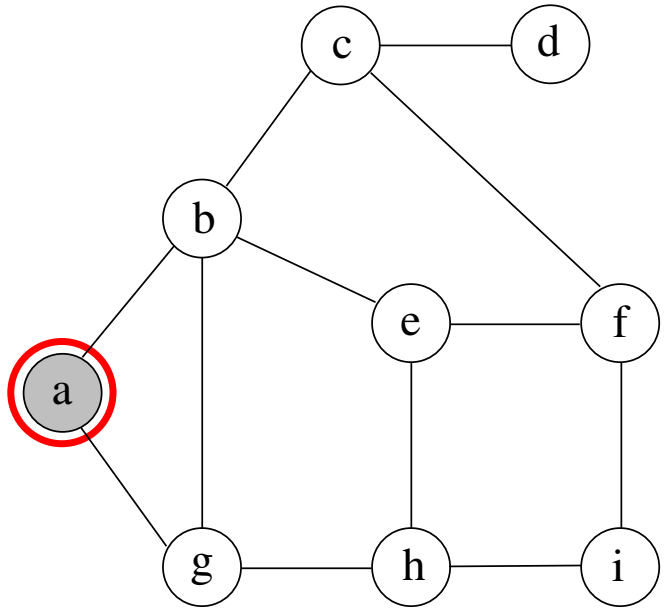


$f =< a >$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



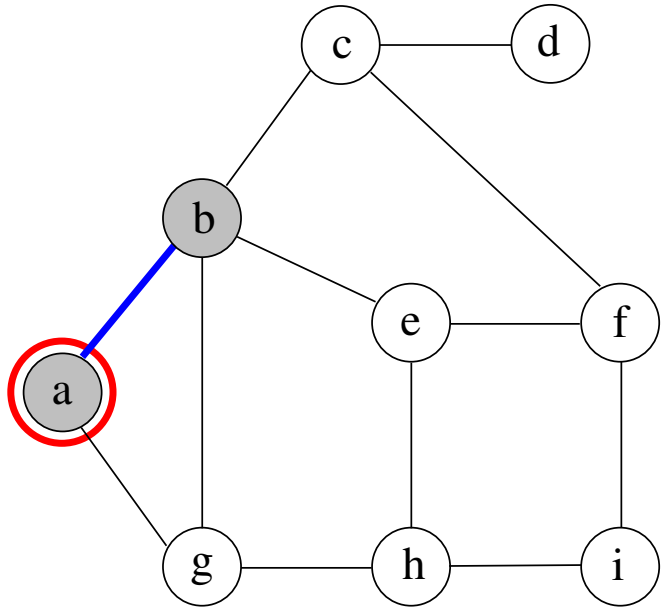
$f =< a >$

$s_k = a$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



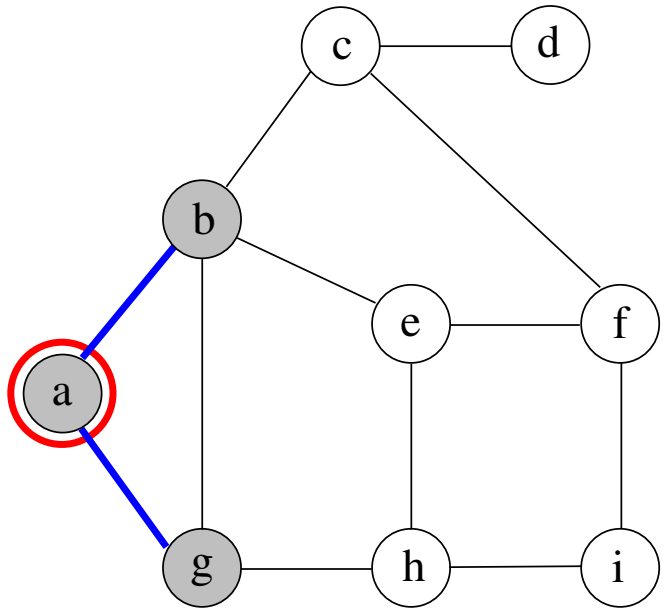
$f =< b, a >$

$s_k = a, s_i = b$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



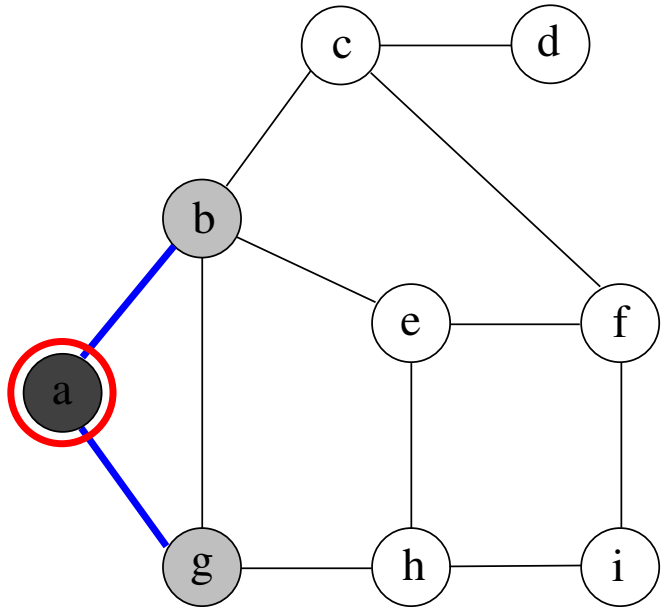
$f =< g, b, a >$

$s_k = a, s_i = g$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



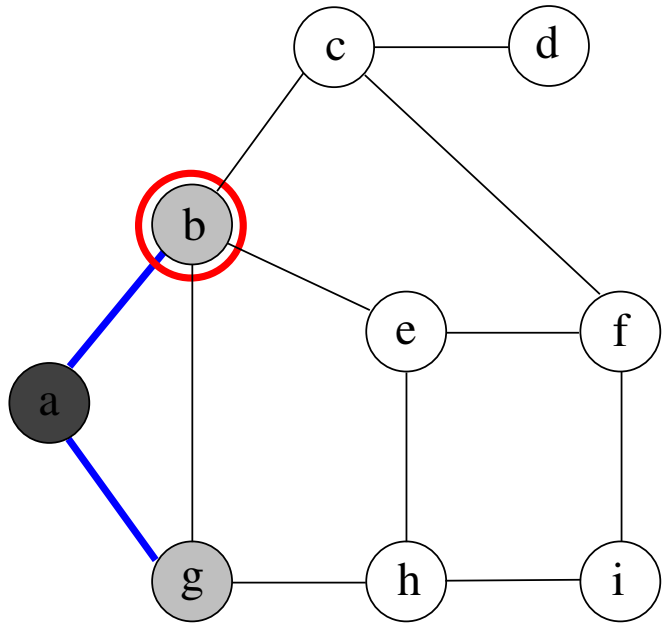
$f =< g, b >$

$s_k = a$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



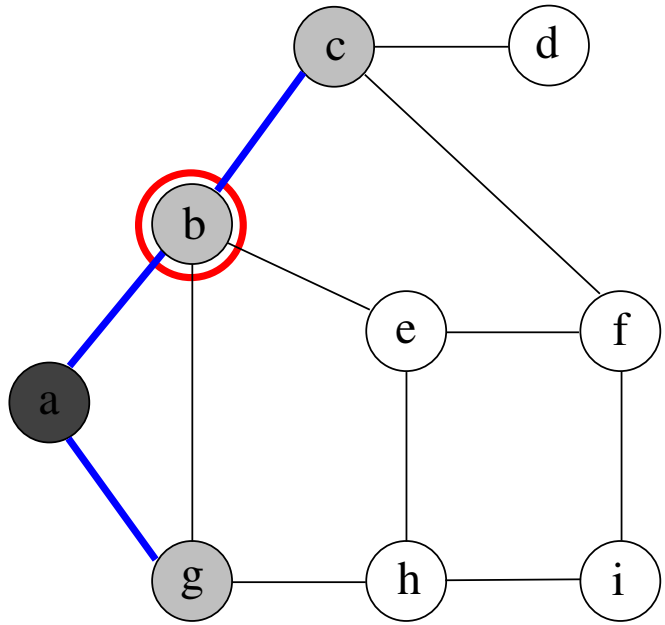
$f =< g, b >$

$s_k = b$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



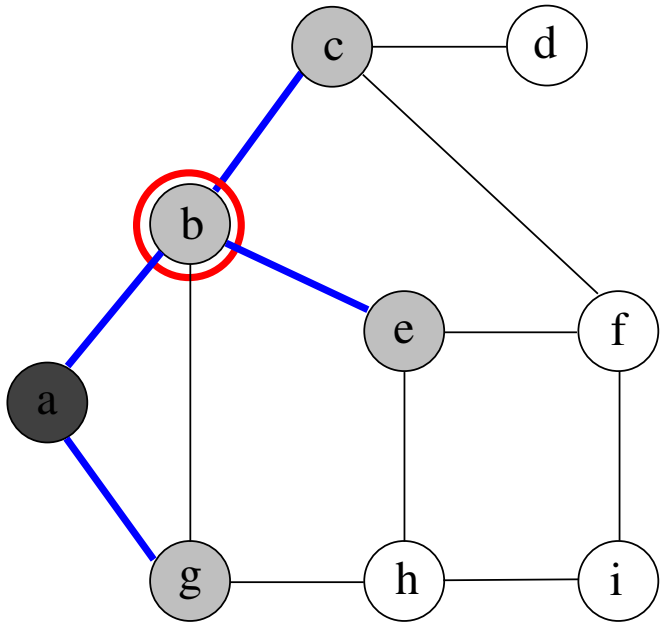
$f =< c, g, b >$

$s_k = b$, $s_i = c$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



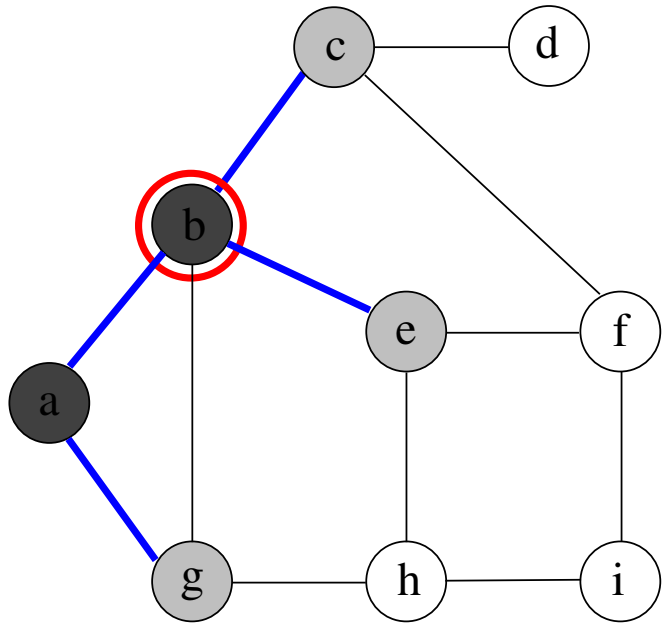
$f =< e, c, g, b >$

$s_k = b$, $s_i = e$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



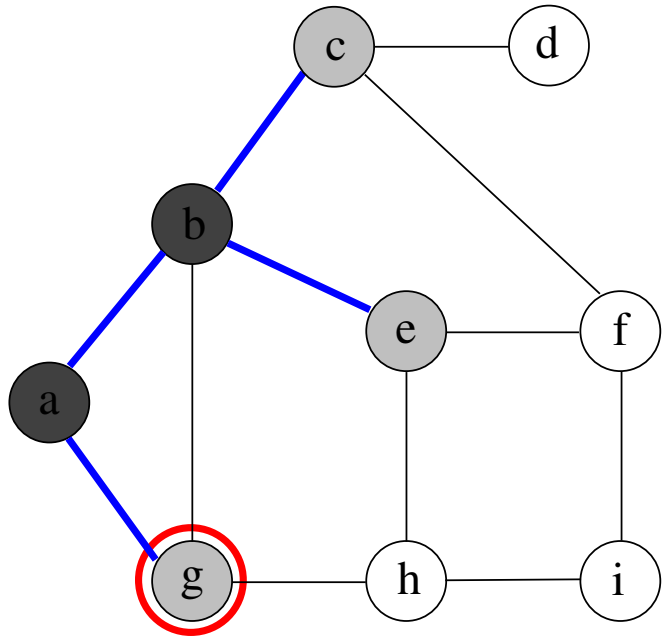
$f =< e, c, g >$

$s_k = b$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



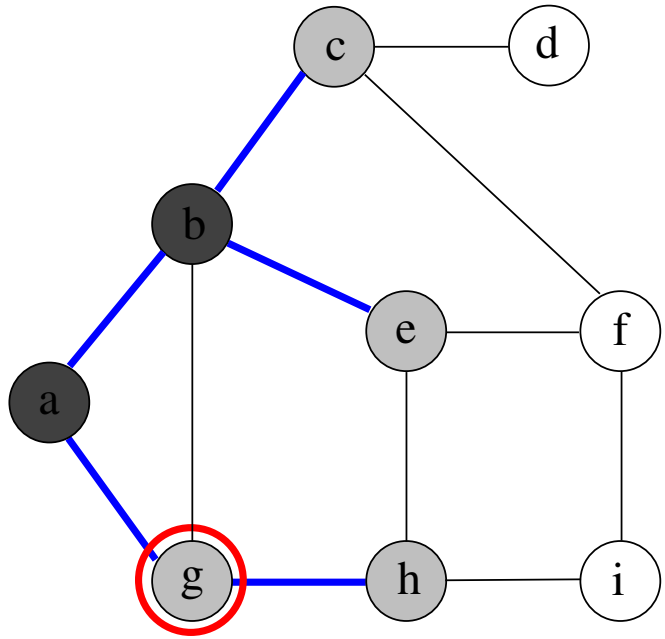
$f =< e, c, g >$

$s_k = g$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



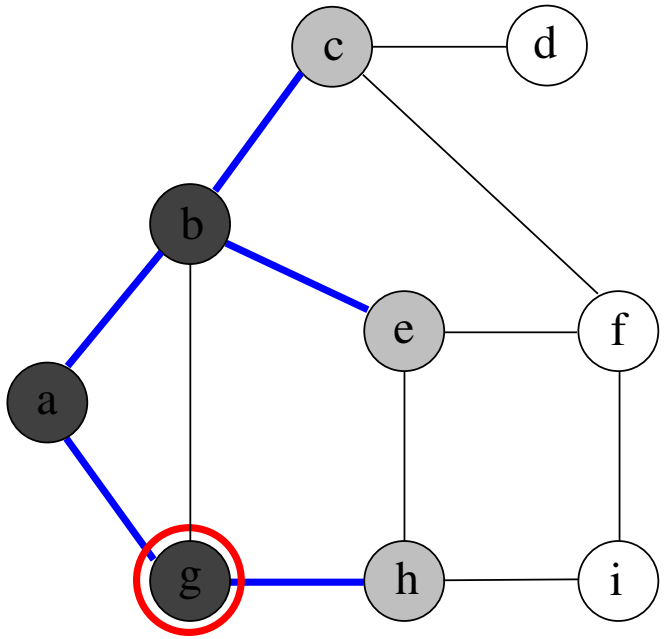
$f =< h, e, c, g >$

$s_k = g, s_i = h$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



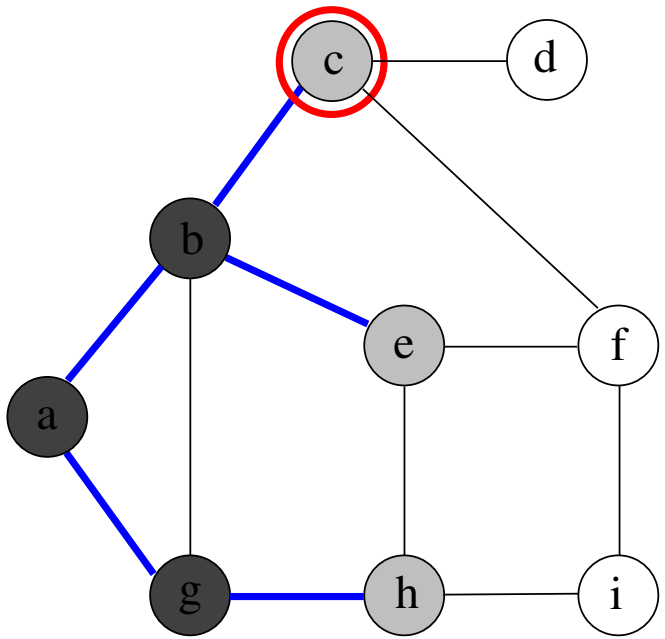
$f =< h, e, c >$

$s_k = g$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



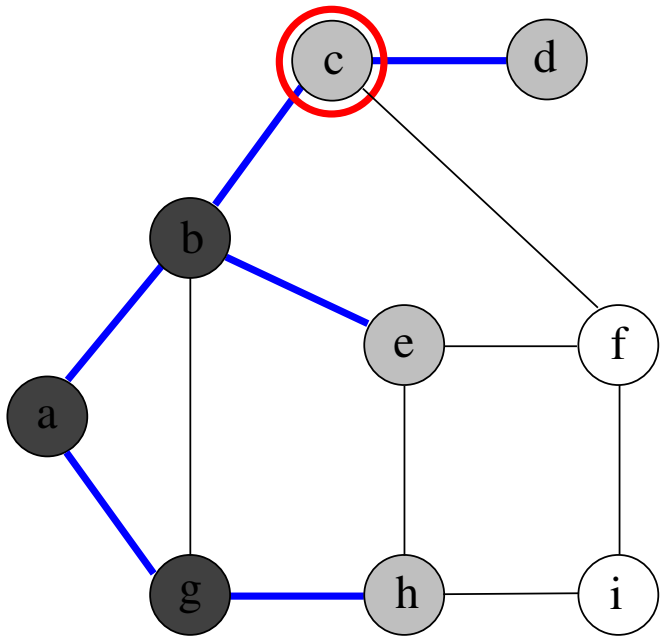
$f =< h, e, c >$

$s_k = c$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



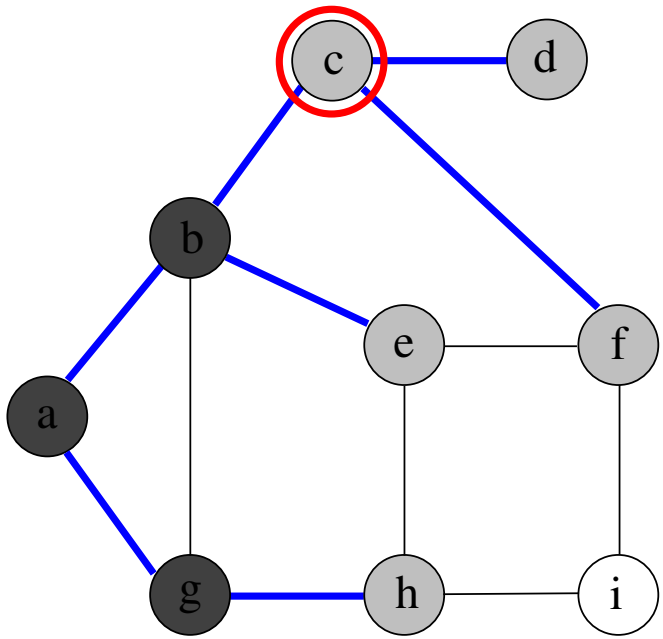
$f =< d, h, e, c >$

$s_k = c, s_i = d$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



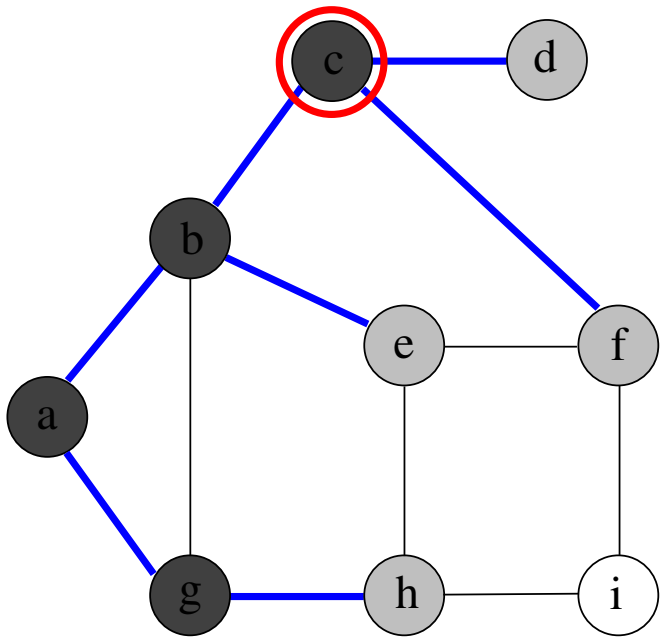
$f =< f, d, h, e, c >$

$s_k = c$, $s_i = f$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



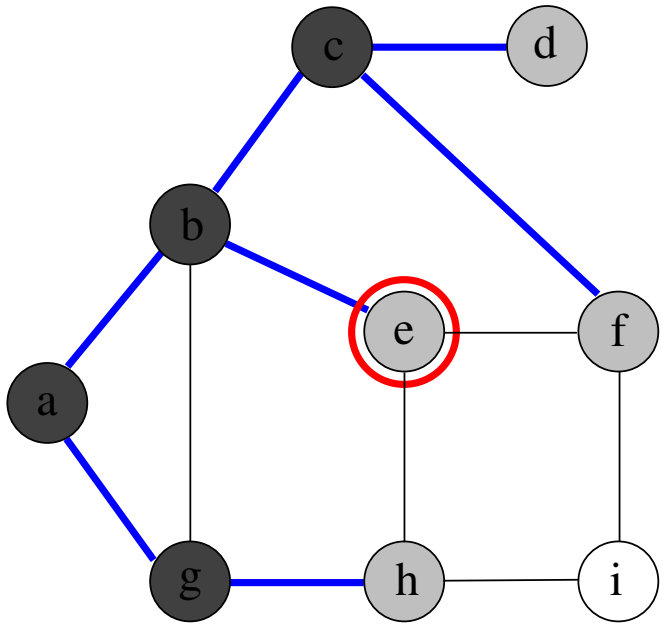
$f =< f, d, h, e >$

$s_k = c$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



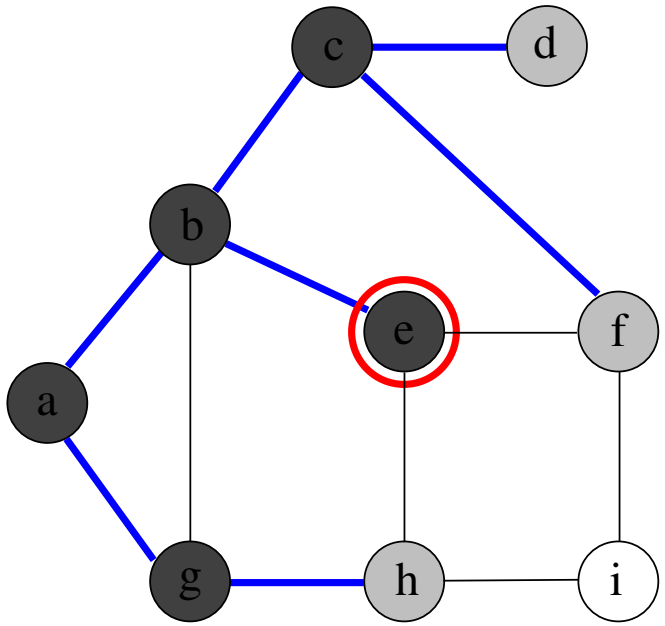
$f =< f, d, h, e >$

$s_k = e$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



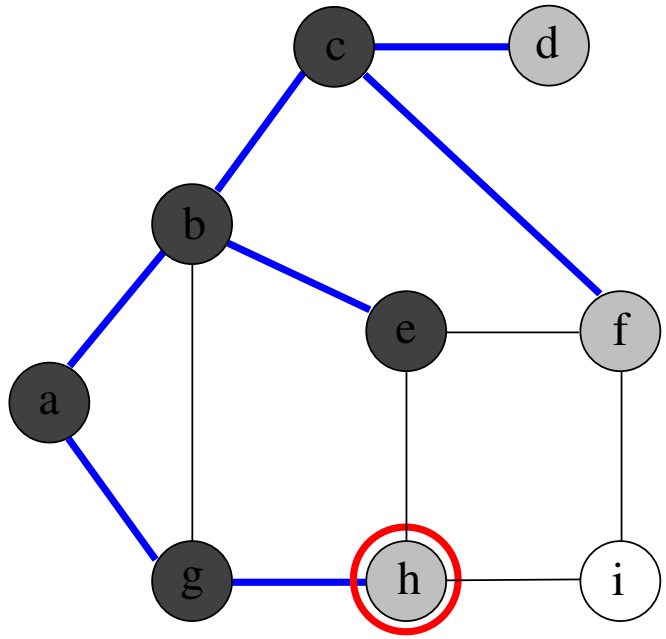
$f =< f, d, h >$

$s_k = e$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



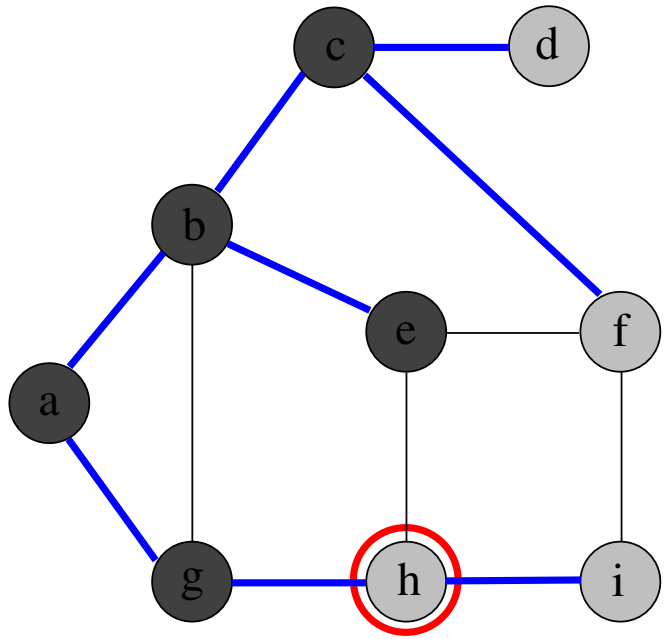
$f =< f, d, h >$

$s_k = h$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



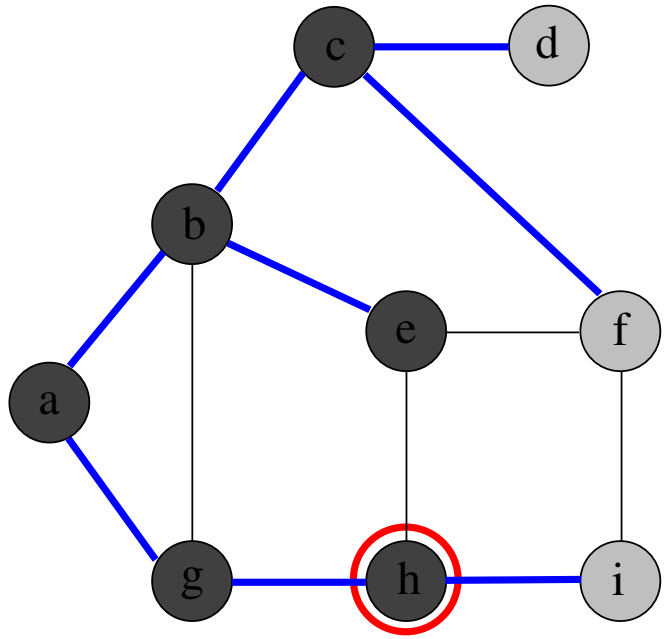
$f =< i, f, d, h >$

$s_k = h$, $s_i = i$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



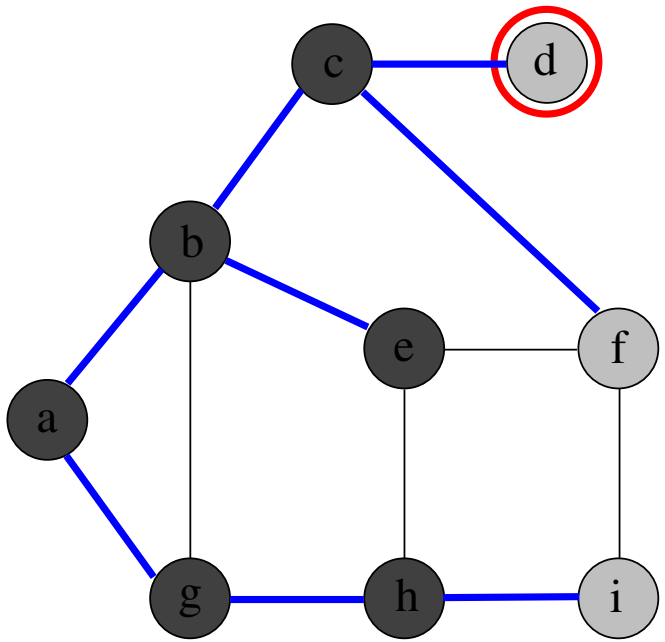
$f =< i, f, d >$

$s_k = h$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



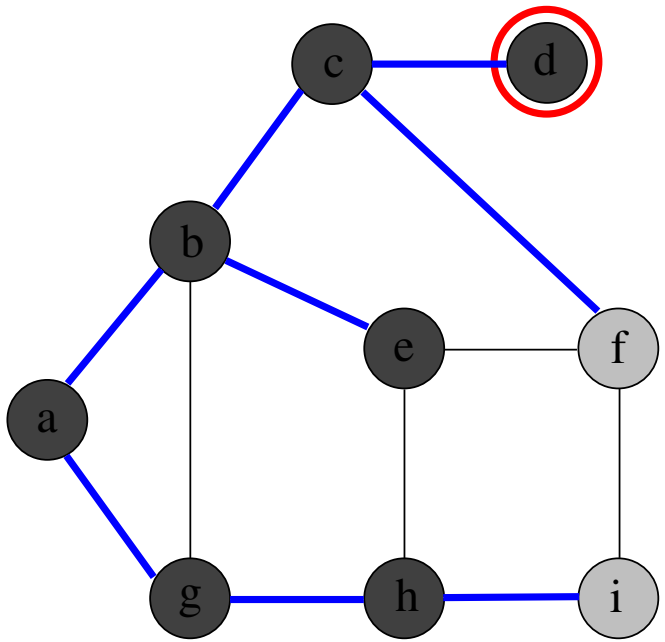
$f =< i, f, d >$

$s_k = d$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



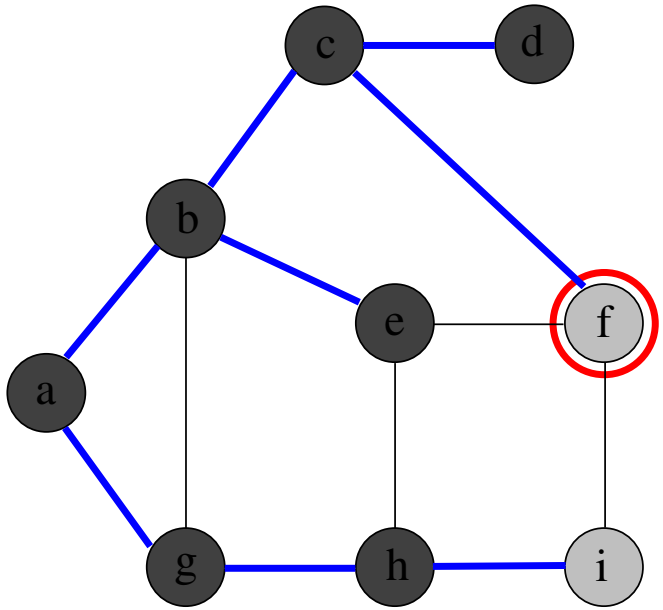
$f =< i, f >$

$s_k = d$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



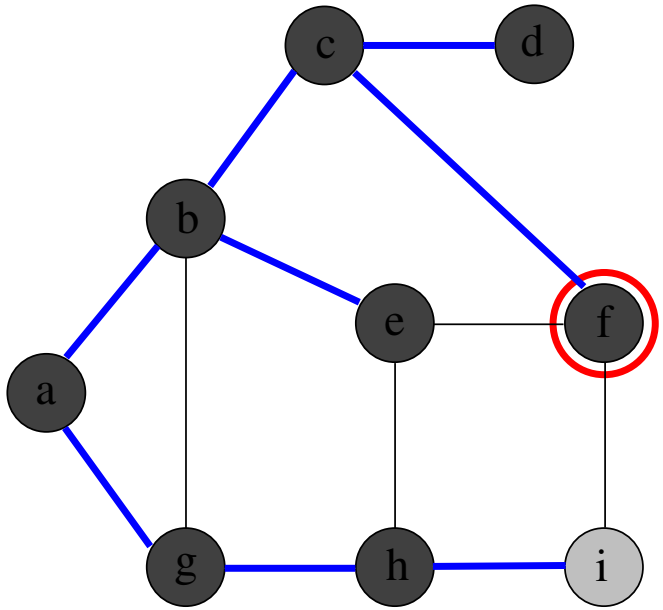
$f =< i, f >$

$s_k = f$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



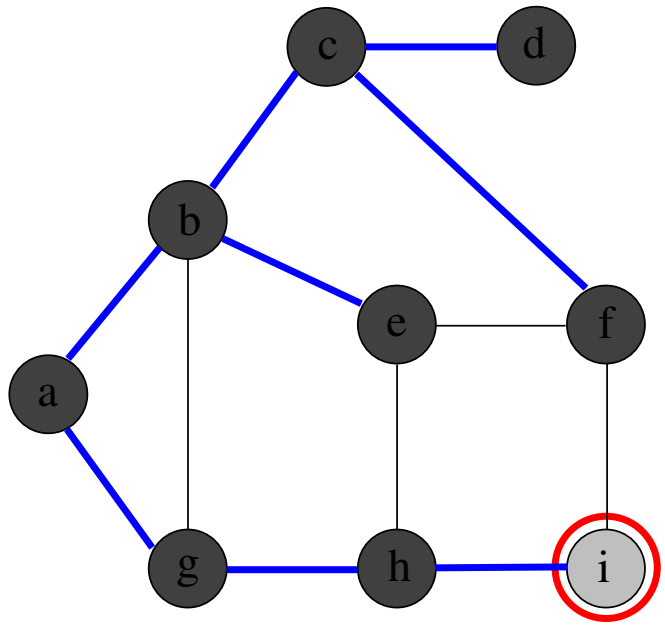
$f =< i >$

$s_k = f$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



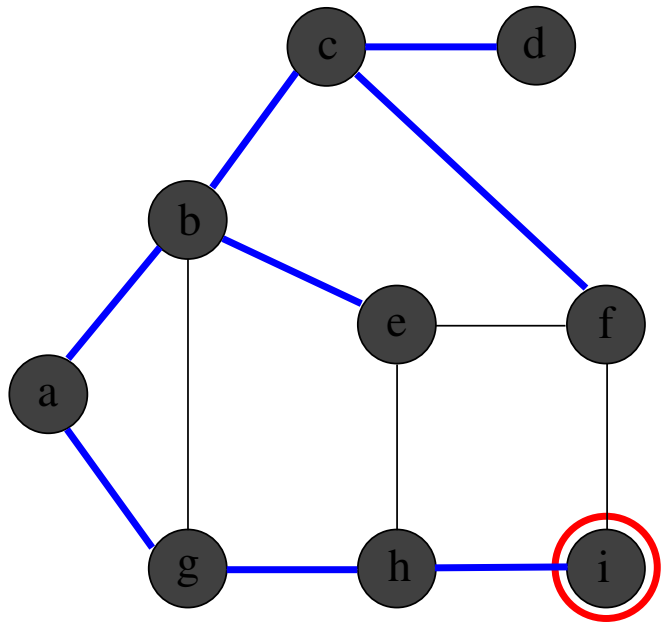
$f =< i >$

$s_k = i$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



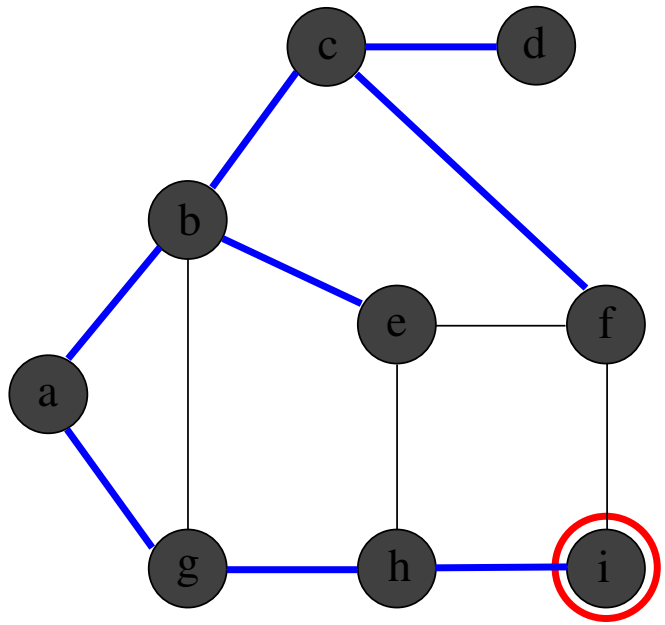
$f = <>$

$s_k = i$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



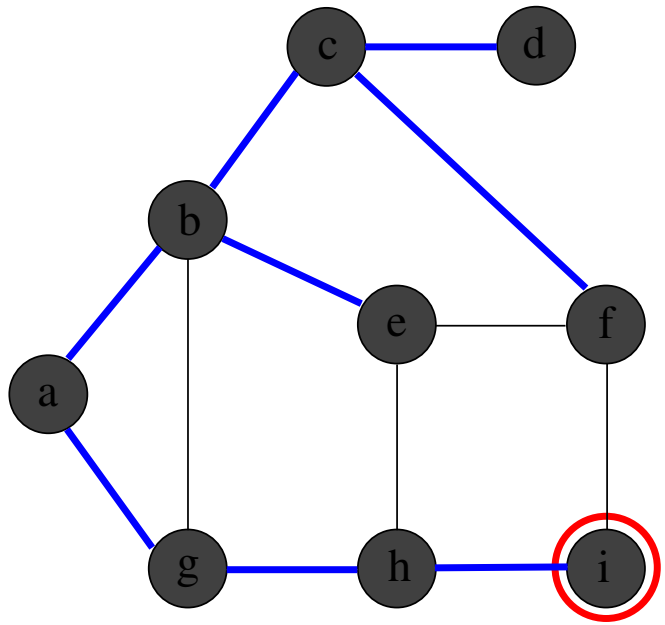
$f = <>$

$s_k = i$

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

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s_0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet s_i de g faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Ajouter s_0 dans f et colorier s_0 en gris
7      tant que f n'est pas vide faire
8          Soit s_k le sommet le plus ancien dans f
9          tant que ∃ s_i ∈ succ(s_k) tq s_i soit blanc faire
10             Ajouter s_i dans f et colorier s_i en gris
11             π[s_i] ← s_k
12         Enlever s_k de f et colorier s_k en noir
13     retourne π
```



$f = <>$

$s_k = i$

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

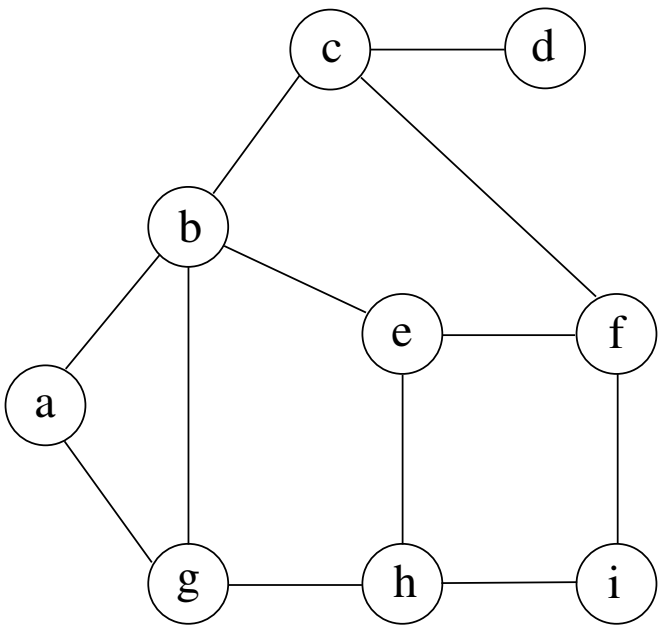
$\rightsquigarrow \mathcal{O}(n+p)$ (sous réserve d'une implémentation par listes d'adjacence)

Utilisation de BFS : recherche de plus courts chemins

Définition : soient s_0 et s_i deux sommets tels que $s_0 \rightsquigarrow s_i$

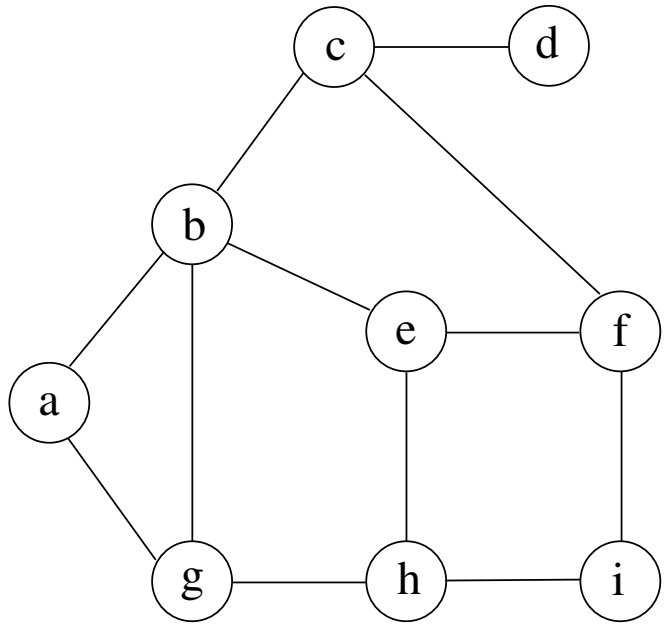
- Plus court chemin entre s_0 et s_i = chemin de longueur minimale
- Distance entre s_0 et s_i = $\delta(s_0, s_i)$ = longueur du plus court chemin

Exemple :



- $\delta(a, a) = 0$
- $\delta(a, b) = \delta(a, g) = 1$
- $\delta(a, c) = \delta(a, e) = \delta(a, h) = 2$
- $\delta(a, d) = \delta(a, f) = \delta(a, i) = 3$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

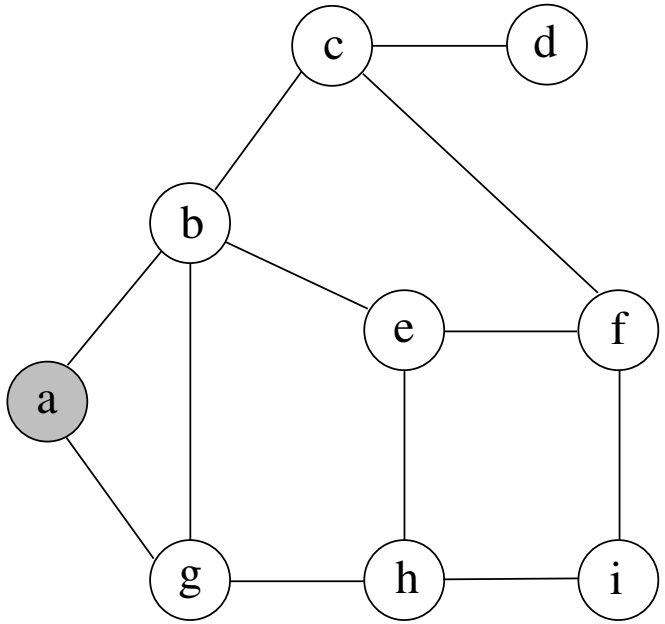


$f = <>$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



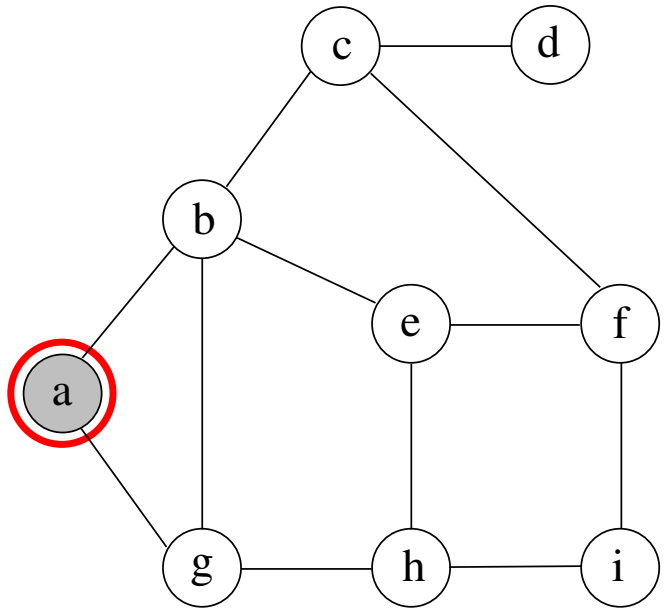
$f =< a >$

$d[a] = 0$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



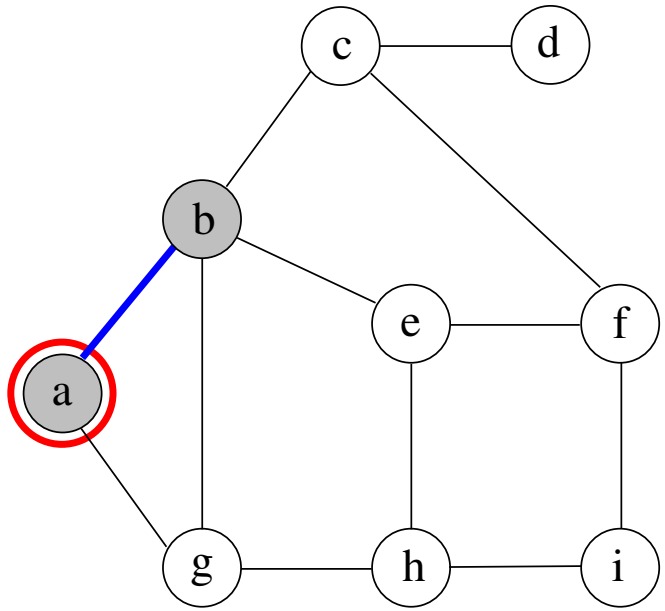
$f =< a >$

$d[a] = 0$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



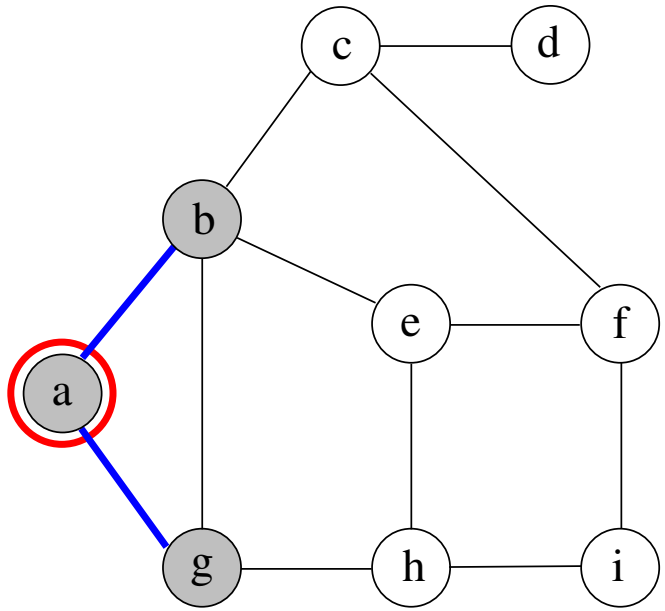
$f =< b, a >$

$d[a] = 0, d[b] = 1$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



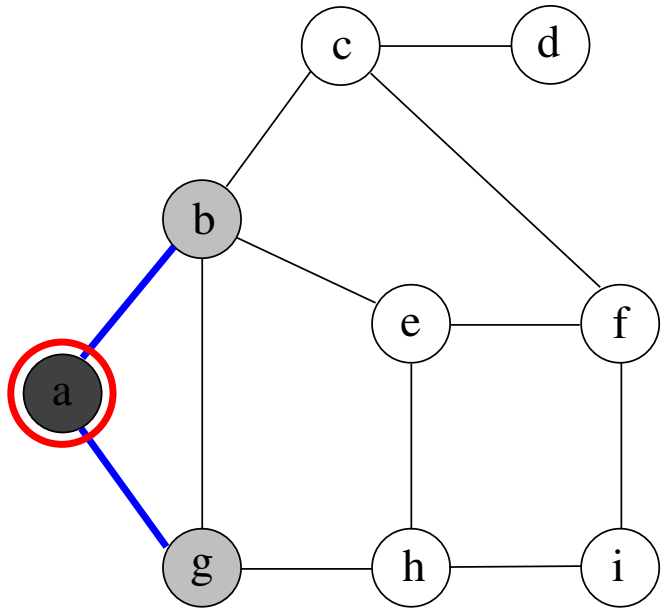
$f =< g, b, a >$

$d[a] = 0, d[b] = 1, d[g] = 1$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



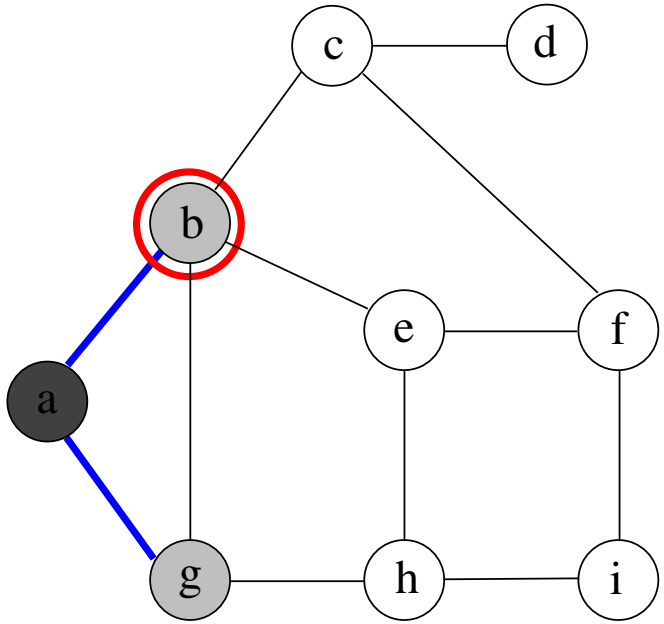
$f =< g, b >$

$d[a] = 0, d[b] = 1, d[g] = 1$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



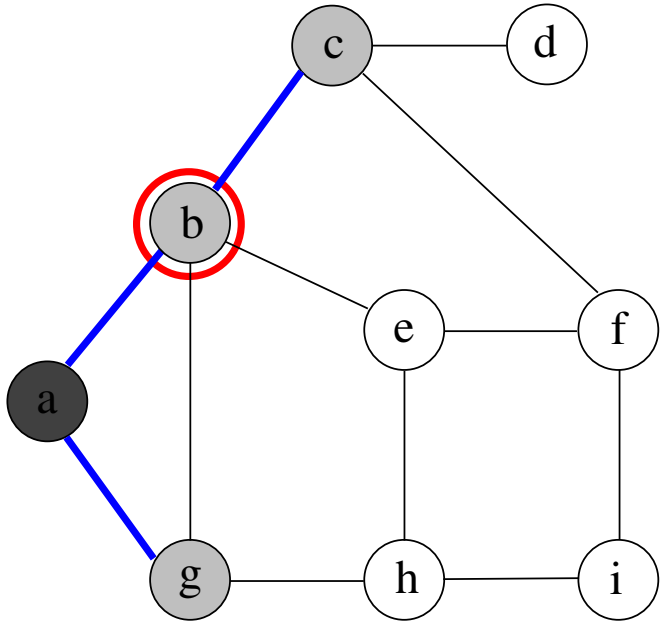
$f =< g, b >$

$d[a] = 0, d[b] = 1, d[g] = 1$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

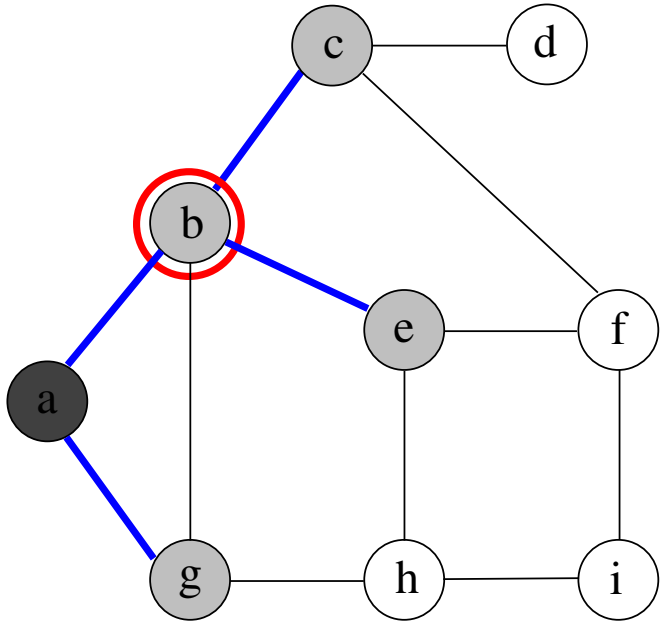


$f =< c, g, b >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

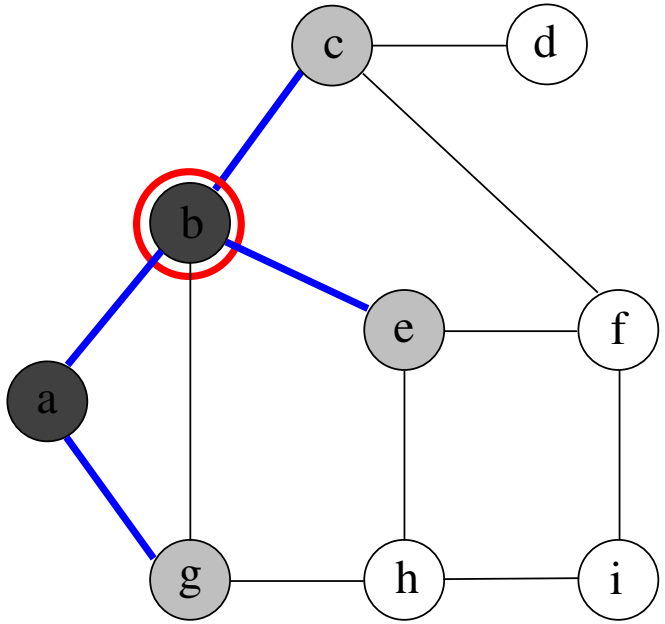


$f =< e, c, g, b >$
$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

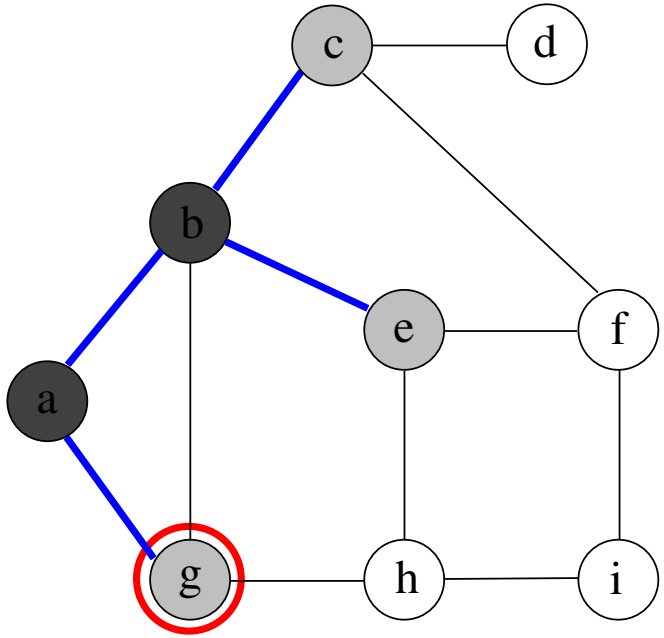


$f =< e, c, g >$
$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

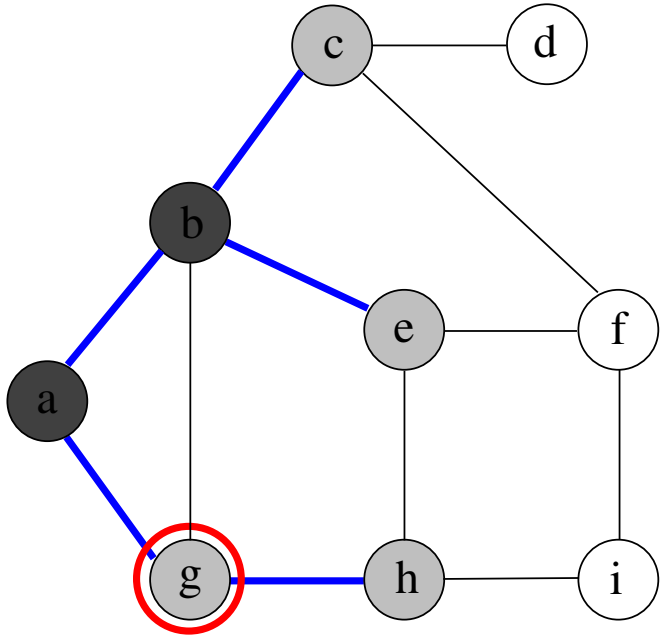


$f =< e, c, g >$
$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

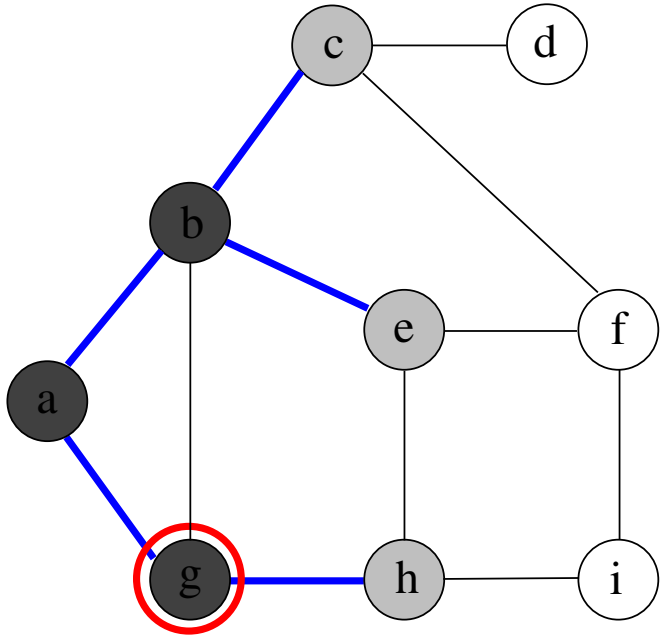


$f =< h, e, c, g >$
$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



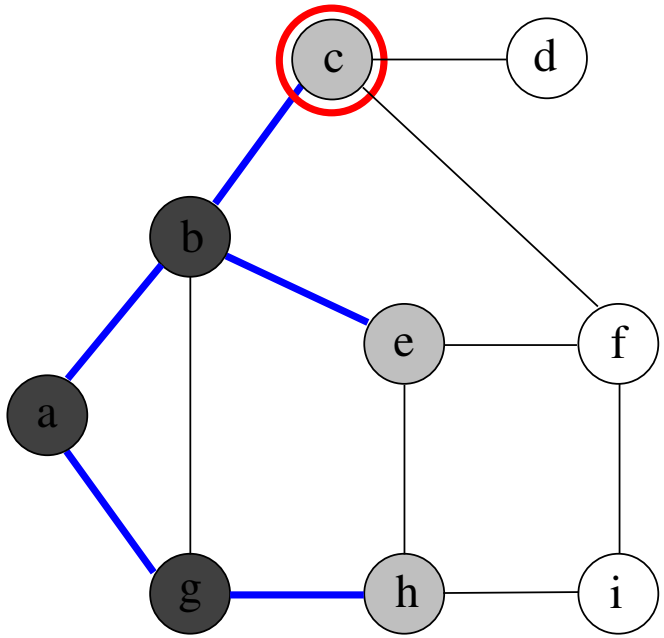
$f =< h, e, c >$

$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



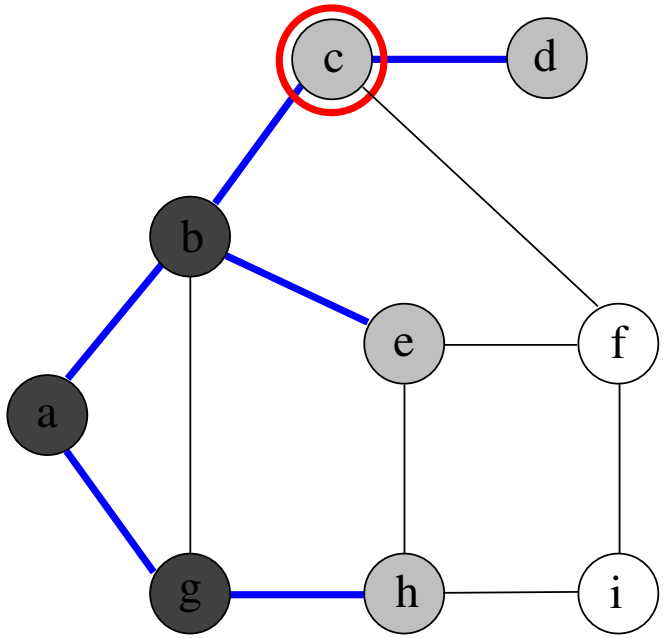
$f =< h, e, c >$

$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

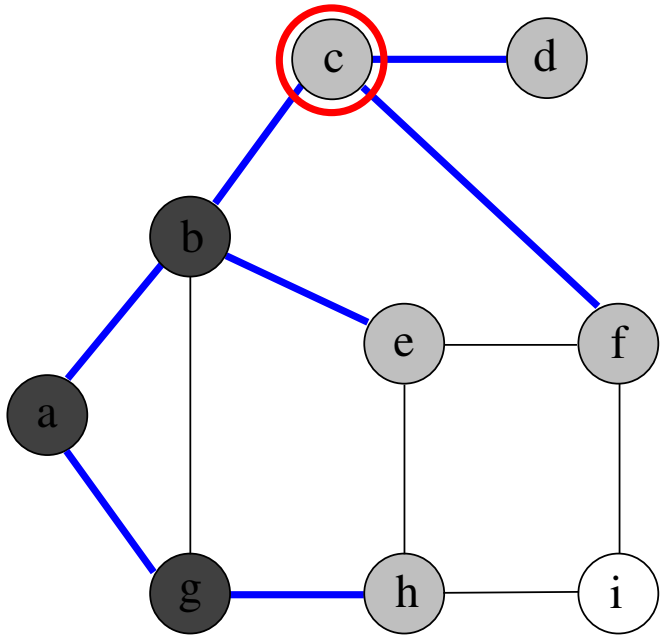


$f =< d, h, e, c >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

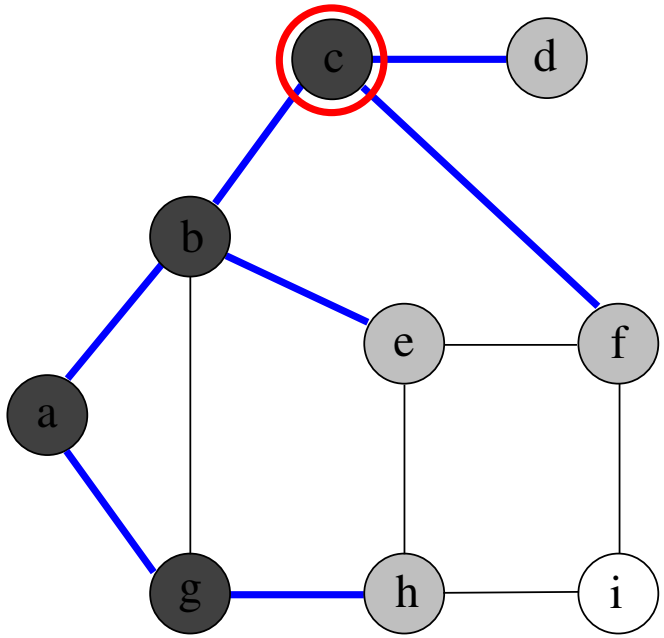


$f =< f, d, h, e, c >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$, $d[f] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

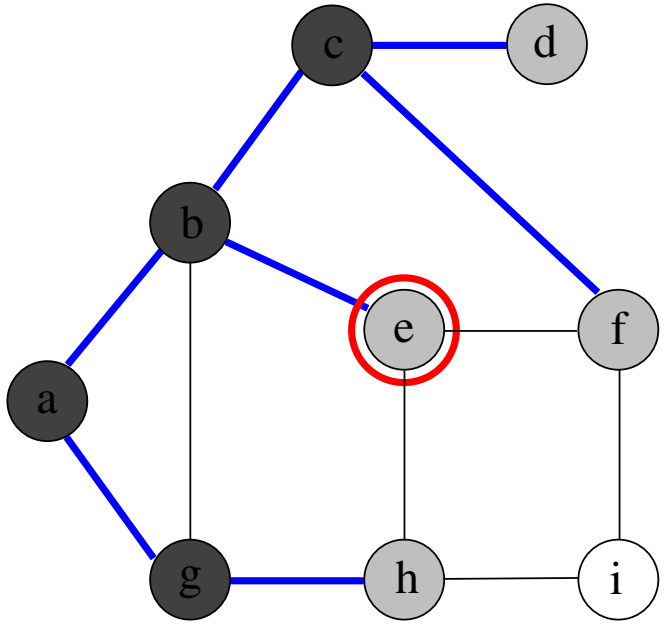


$f =< f, d, h, e >$
$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2,$
$d[d] = 3, d[f] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

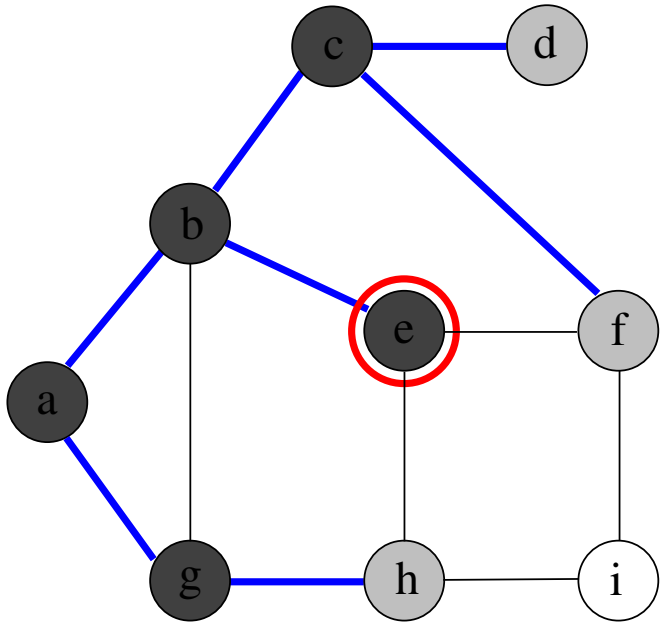


$f =< f, d, h, e >$
$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2,$
$d[d] = 3, d[f] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

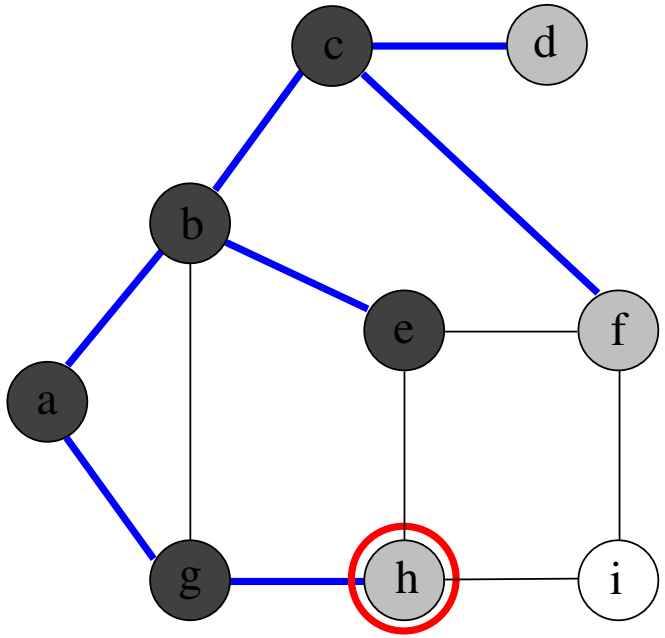


$f =< f, d, h >$
$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2,$
$d[d] = 3, d[f] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

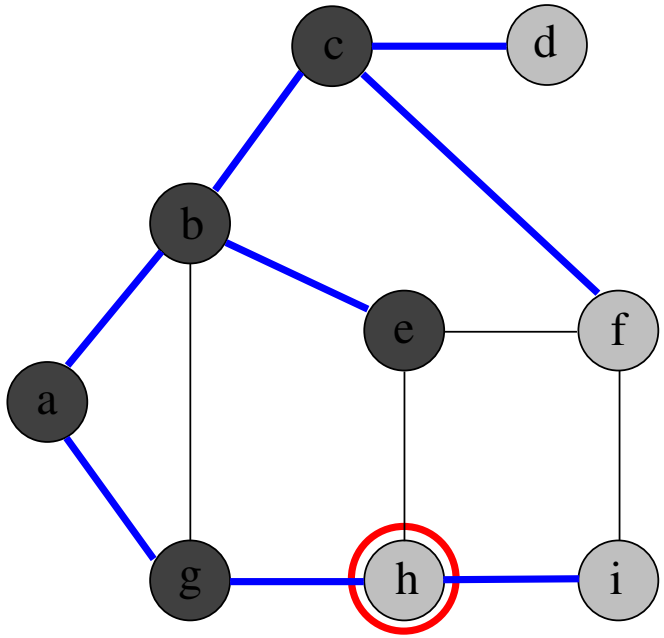


$f =< f, d, h >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$, $d[f] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

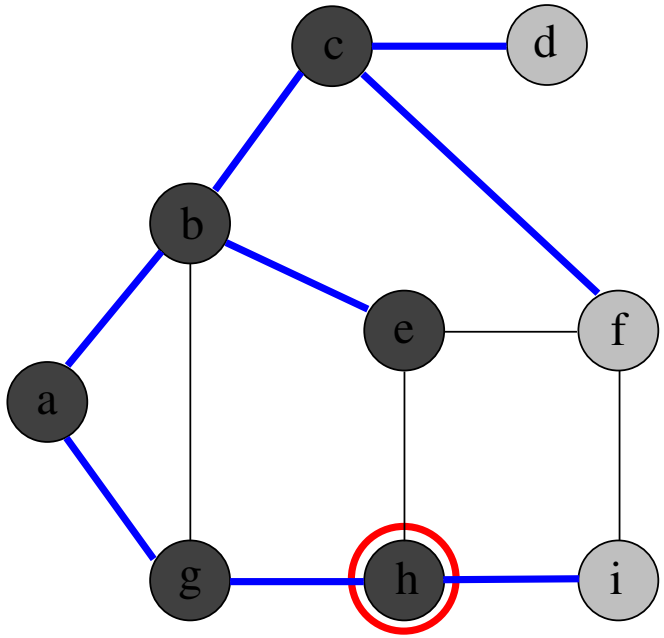


$f =< i, f, d, h >$
$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2,$
$d[d] = 3, d[f] = 3, d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```

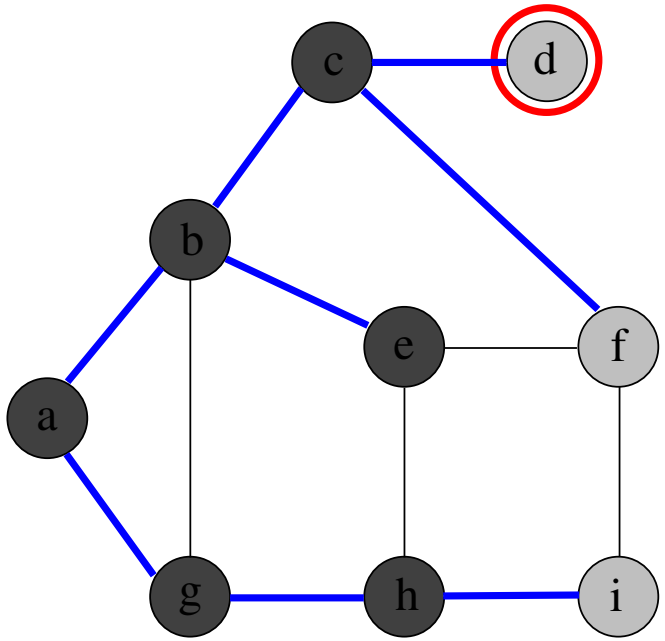


$f =< i, f, d >$
$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2,$
$d[d] = 3, d[f] = 3, d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



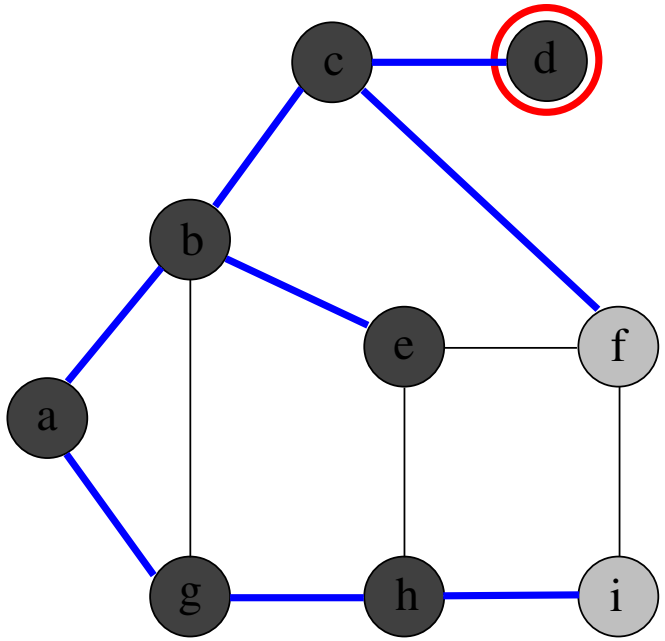
$f =< i, f, d >$

$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2,$
$d[d] = 3, d[f] = 3, d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



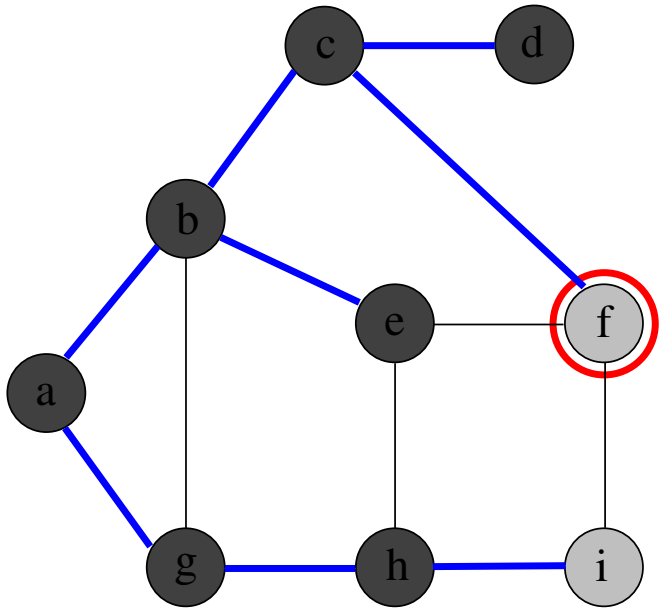
$f =< i, f >$

$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2,$
$d[d] = 3, d[f] = 3, d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



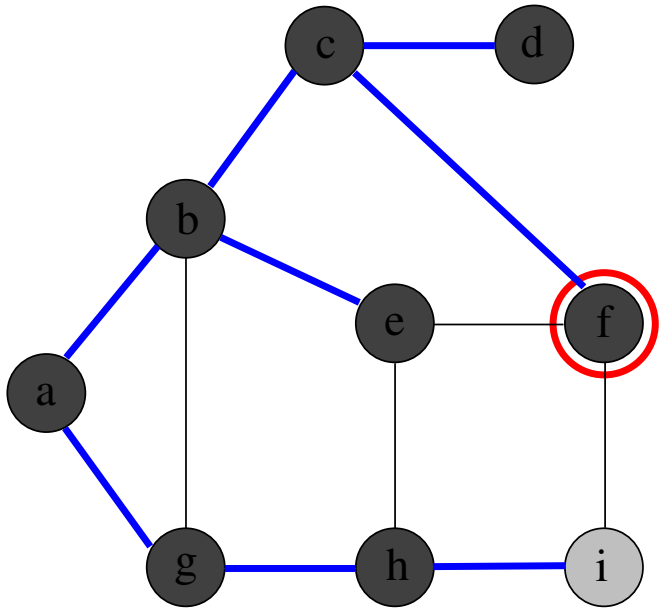
$f =< i, f >$

$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2,$
$d[d] = 3, d[f] = 3, d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



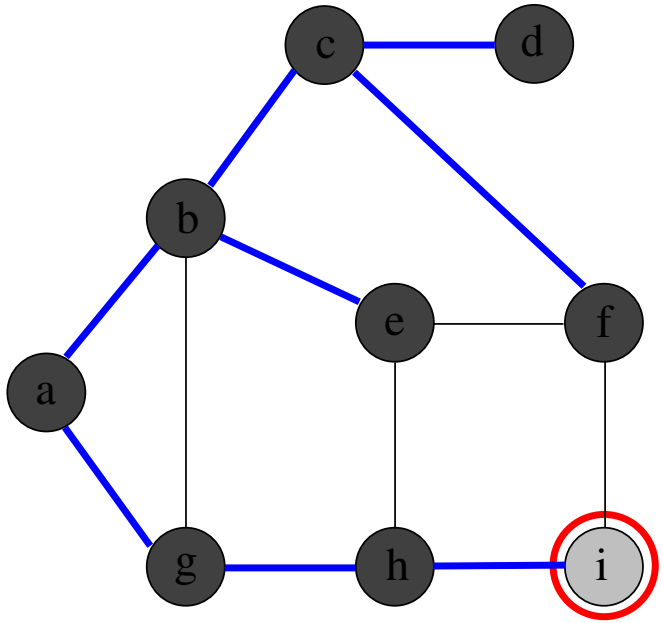
$f =< i >$

$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2,$
$d[d] = 3, d[f] = 3, d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



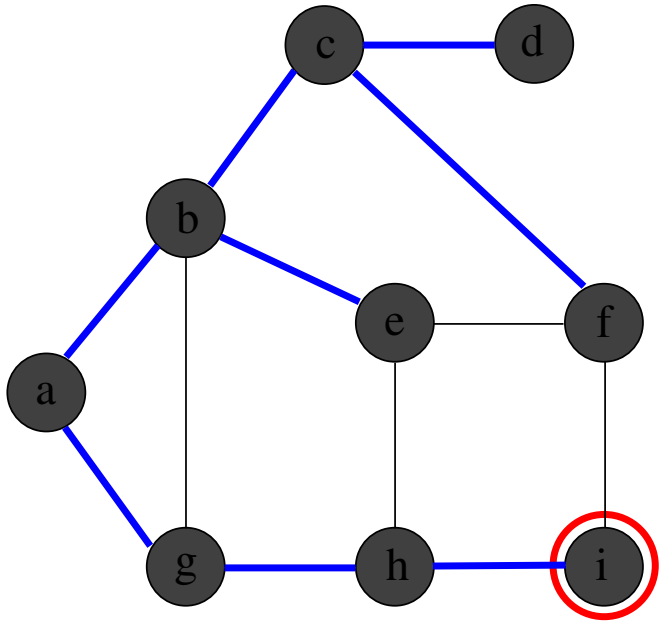
$f =< i >$

$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2,$
$d[d] = 3, d[f] = 3, d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

```
1  Fonction calculeDistances(g, s_0)
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5          d[s_i] ← ∞
6      Ajouter s_0 dans f et colorier s_0 en gris
7      d[s_0] ← 0
8      tant que f n'est pas vide faire
9          Soit s_k le sommet le plus ancien dans f
10         tant que ∃s_i ∈ succ(s_k) tq s_i soit blanc faire
11             Ajouter s_i dans f et colorier s_i en gris
12             d[s_i] ← d[s_k] + 1
13             π[s_i] ← s_k
14         Enlever s_k de f et colorier s_k en noir
15     retourne d
```



$f = <>$

$d[a] = 0, d[b] = 1, d[g] = 1,$
$d[c] = 2, d[e] = 2, d[h] = 2,$
$d[d] = 3, d[f] = 3, d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

1. Aucun successeur d'un sommet noir n'est blanc
2. Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
3. Soit $< s_1, s_2, \ldots, s_k >$ les sommets de f, du + récent au + vieux : $d[s_1] \geq d[s_2] \geq \ldots \geq d[s_k]$ et $d[s_1] \leq d[s_k] + 1$

Affichage du plus court chemin

```
1 Procédure plusCourtChemin(s_0, s_j, π)
      Entrée          : 2 sommets s_0 et s_j, et une arborescence π
      Précondition    : π = arborescence retournée par calculeDistance(g, s_0)
      Postcondition   : Affiche un plus court chemin pour aller de s_0 jusque s_j
2     si s_0 = s_j alors afficher(s_0);
3     sinon si π[s_j] = null alors afficher("Il n'y a pas de chemin de ",s_0," jusque ",s_j);
4     sinon
5         plusCourtChemin(s_0, π[s_j], π)
6         afficher(" suivi de ",s_j)
```

Exemple :

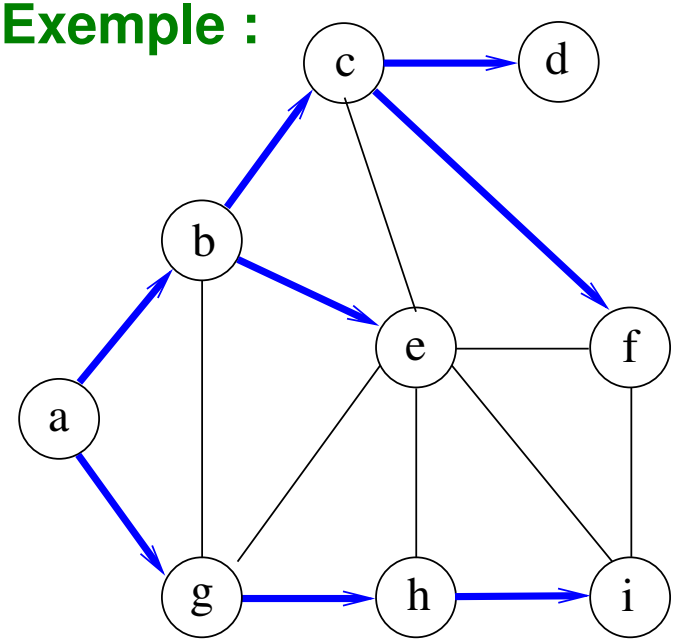
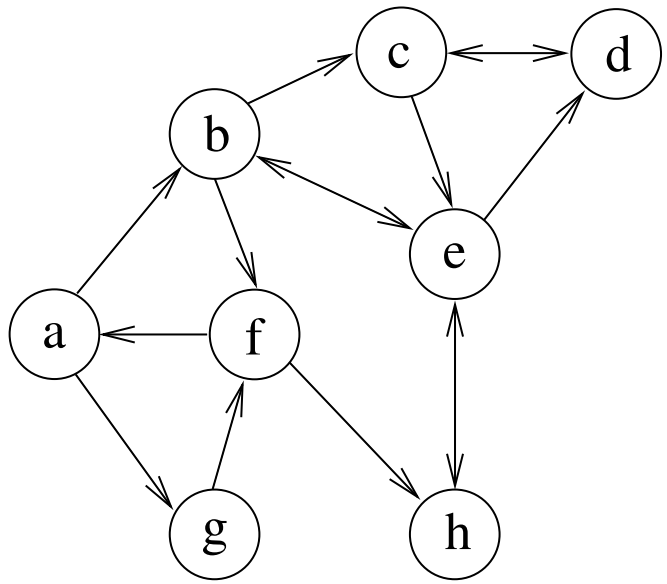


Tableau π correspondant :

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```

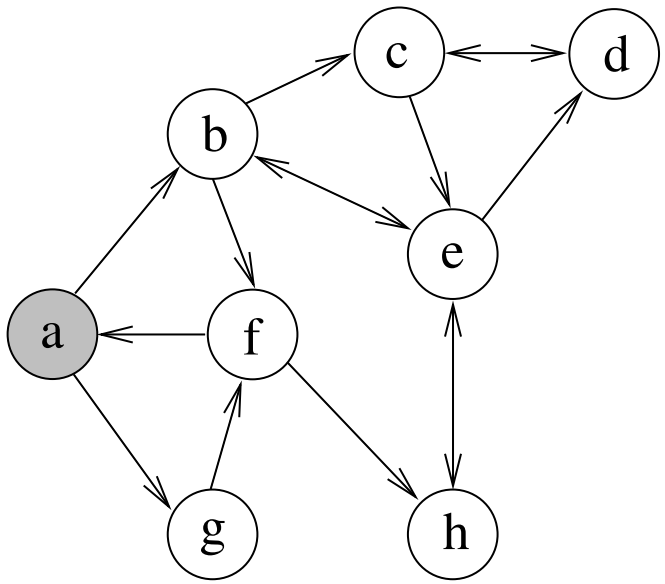


$p = <>$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```

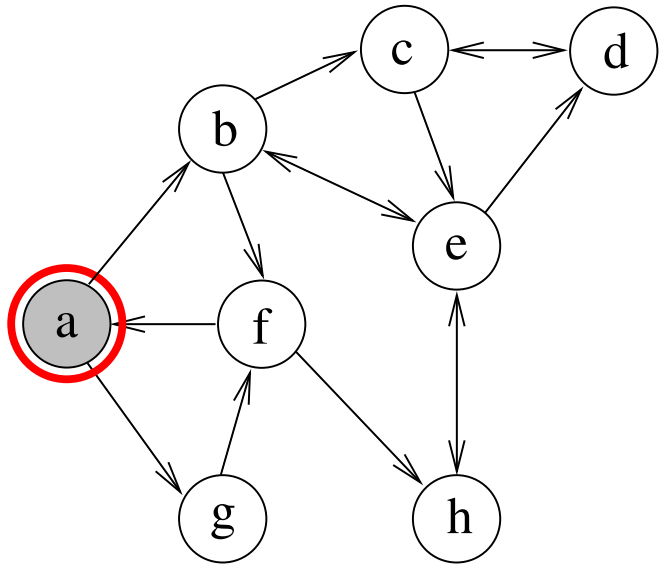


$p =< a >$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



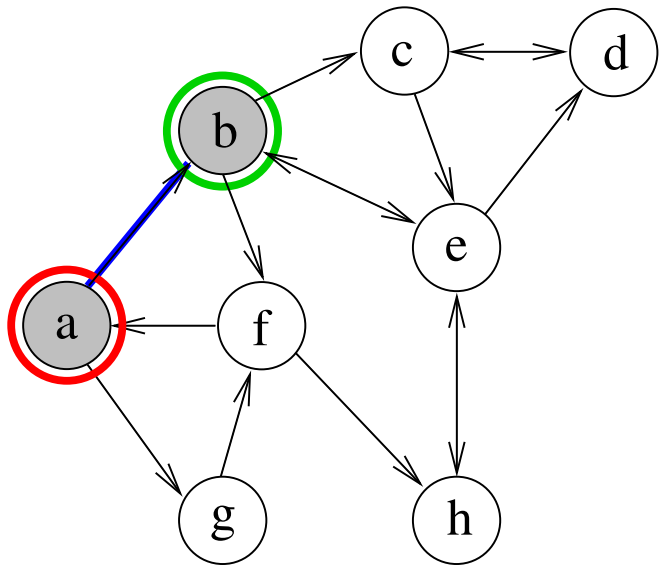
$p =< a >$

$s_i = a$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



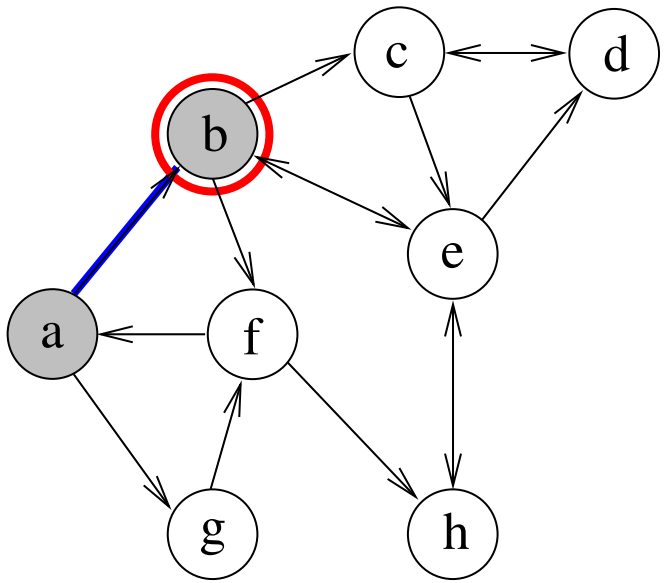
$p =< a, b >$

$s_i = a, s_j = b$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```

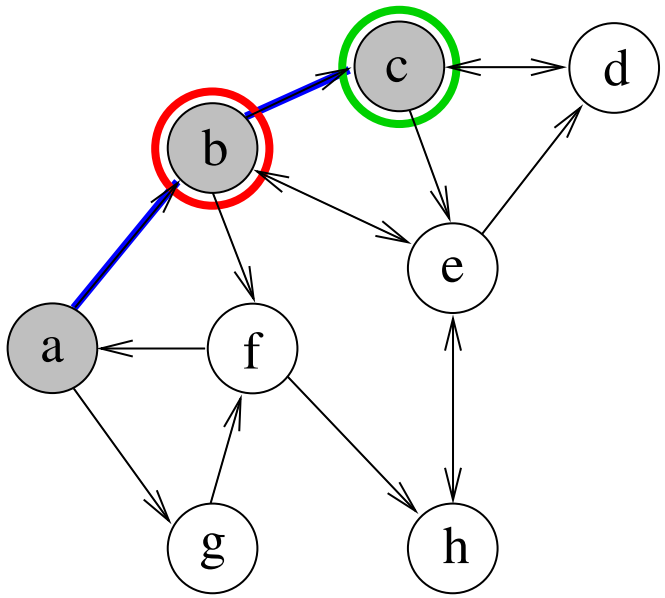


$p =< a, b >$
$s_i = b$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



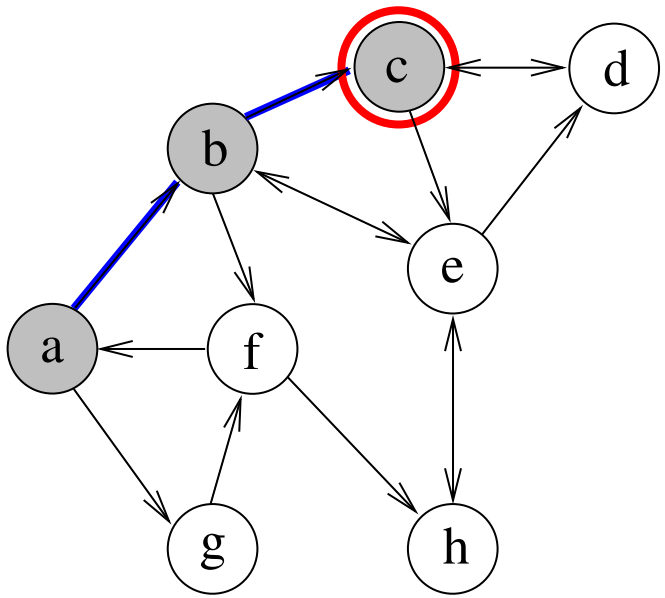
$p =< a, b, c >$

$s_i = b, s_j = c$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



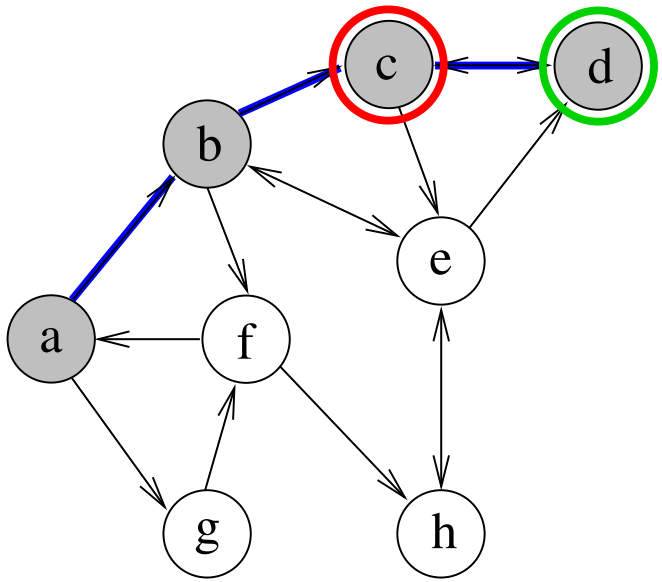
$p =< a, b, c >$

$s_i = c$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```

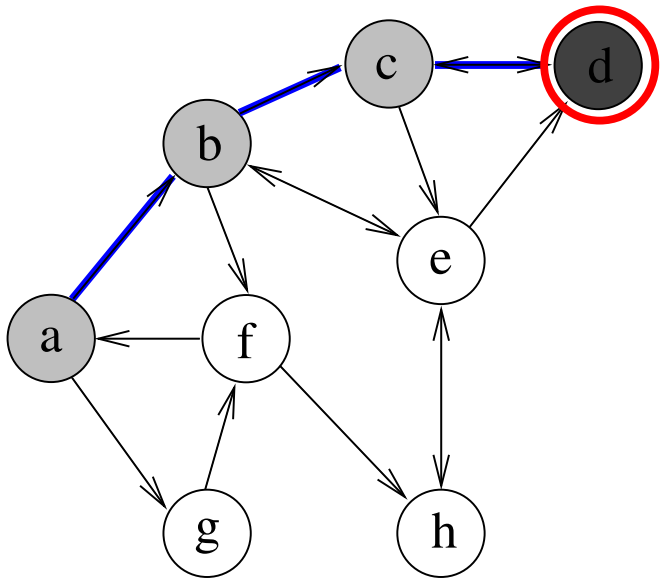


$p =< a, b, c, d >$
$s_i = c, s_j = d$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```

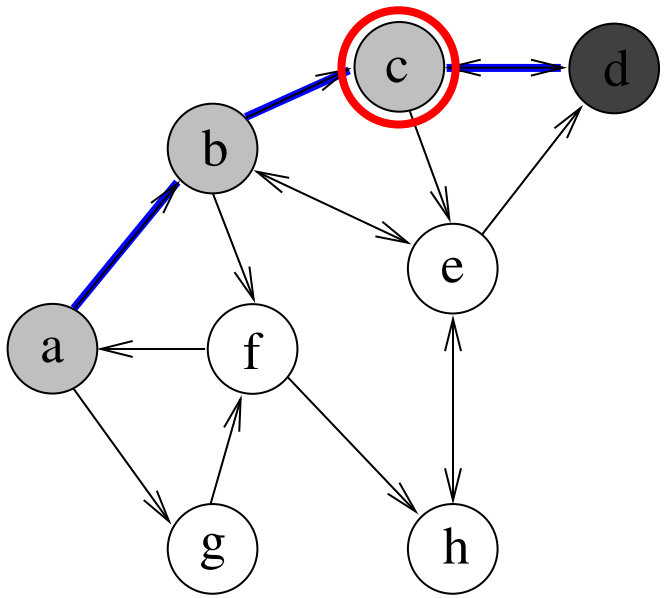


$p =< a, b, c >$
$s_i = d$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



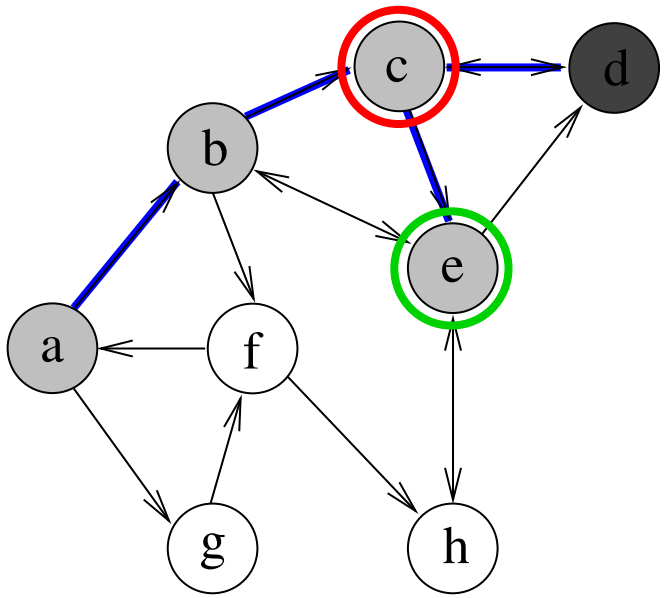
$p =< a, b, c >$

$s_i = c$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



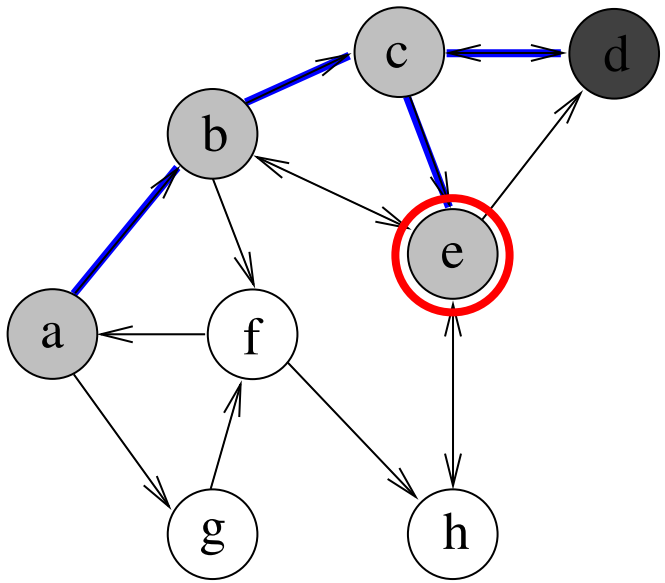
$p =< a, b, c, e >$

$s_i = c, s_j = e$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



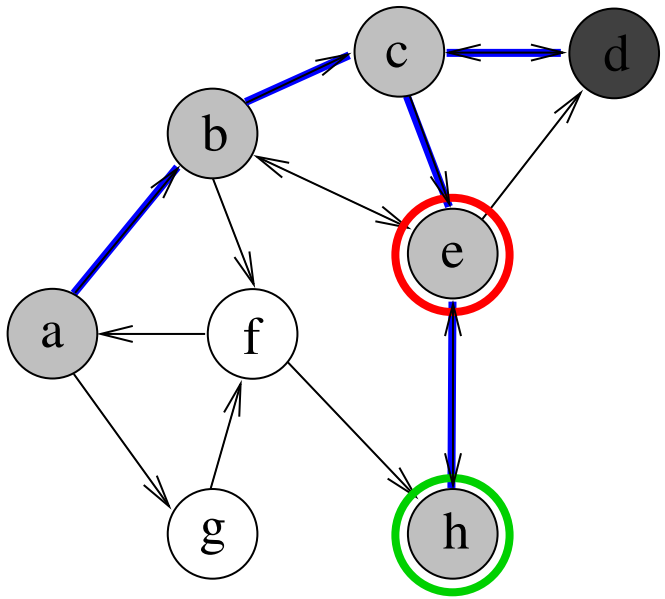
$p =< a, b, c, e >$

$s_i = e$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```

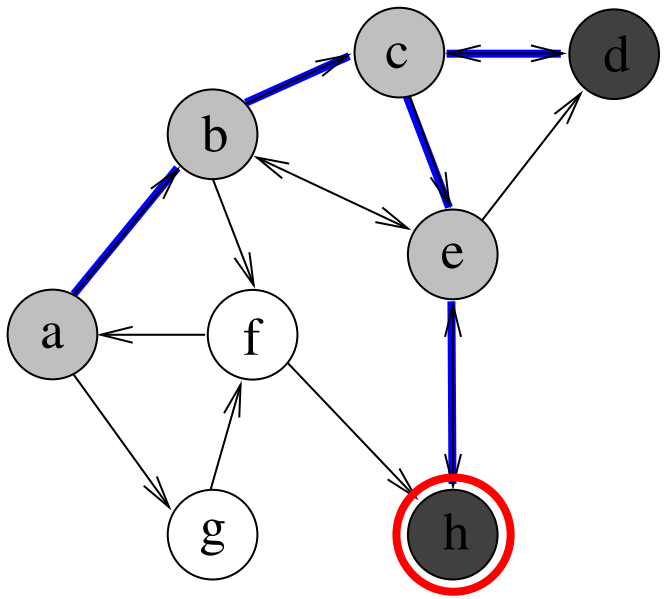


$p =< a, b, c, e, h >$
$s_i = e, s_j = h$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```

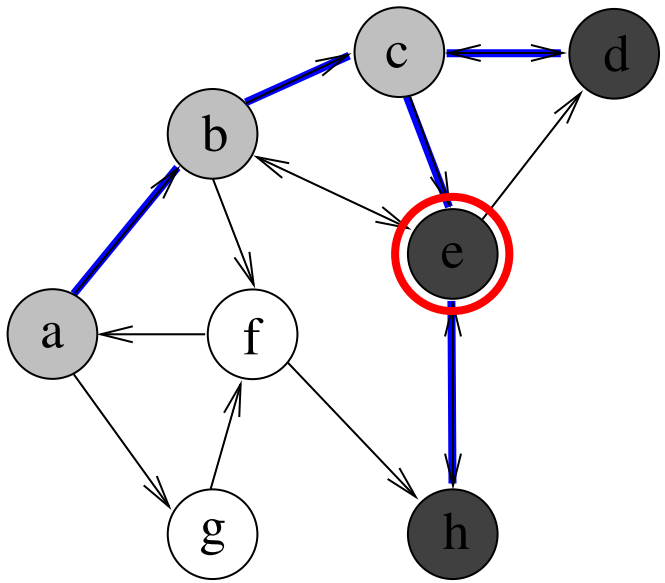


$p =< a, b, c, e >$
$s_i = h$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```

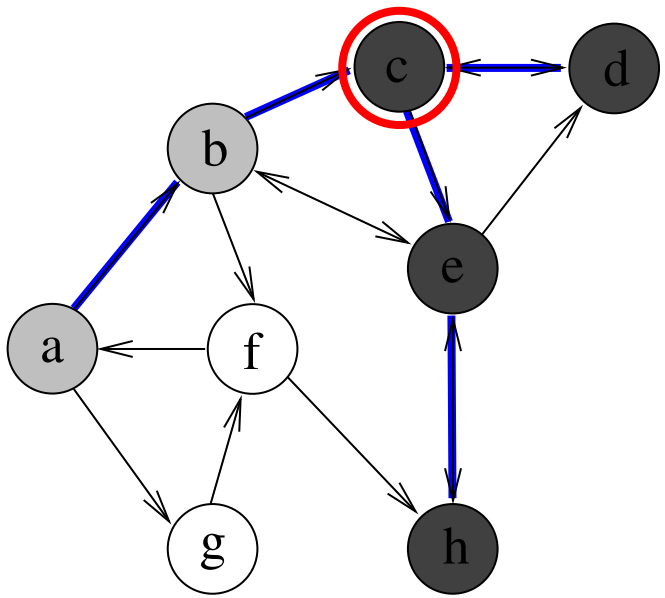


$p =< a, b, c >$
$s_i = e$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



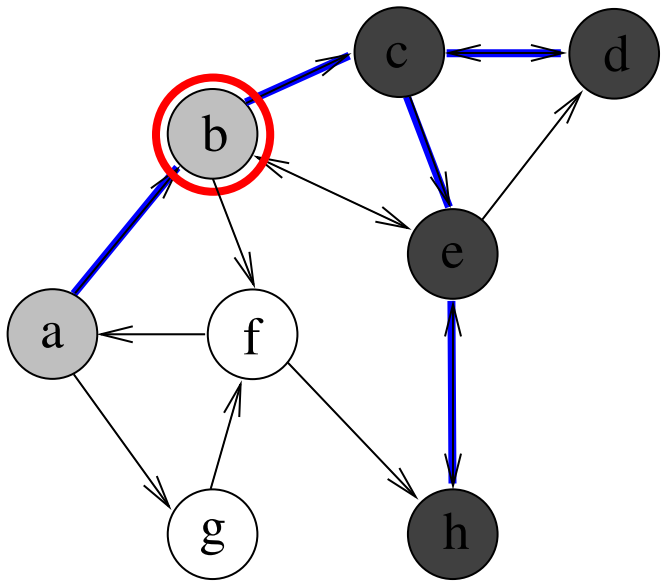
$p =< a, b >$

$s_i = c$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```

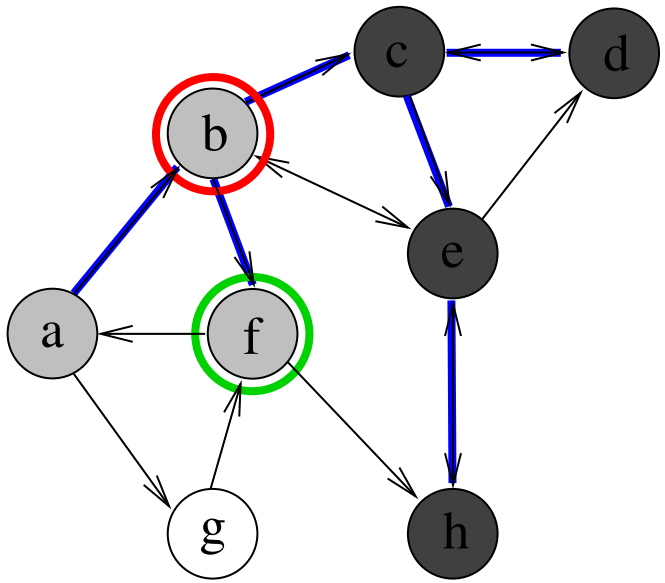


$p =< a, b >$
$s_i = b$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```

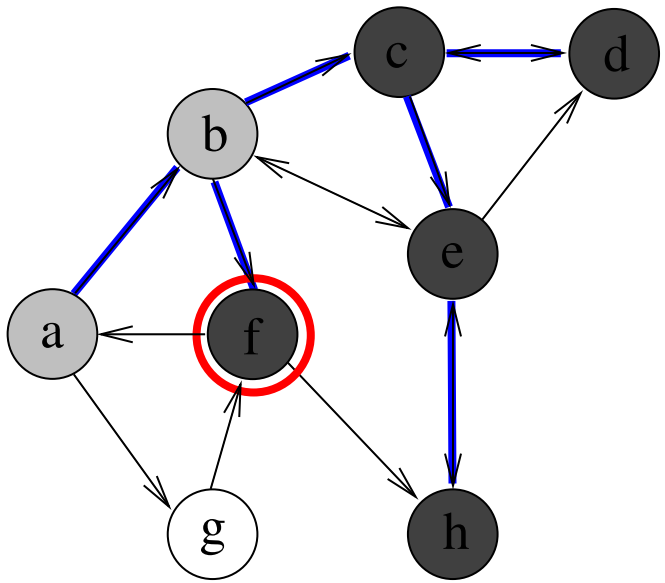


$p =< a, b, f >$
$s_i = b, s_j = f$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```

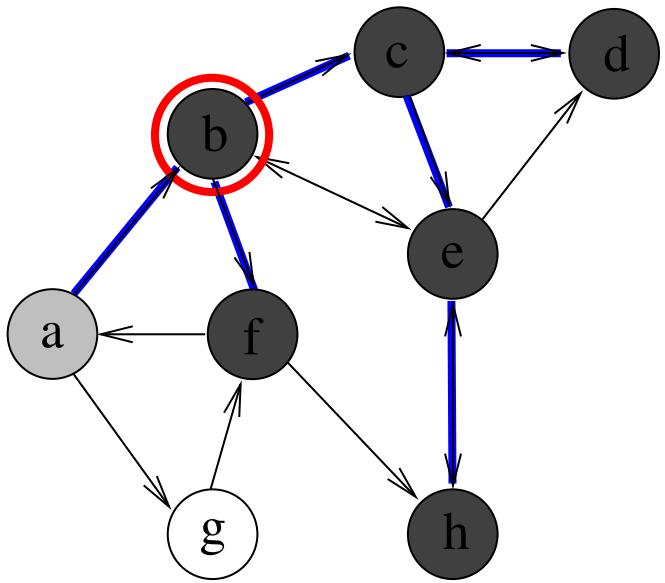


$p =< a, b >$
$s_i = f$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```

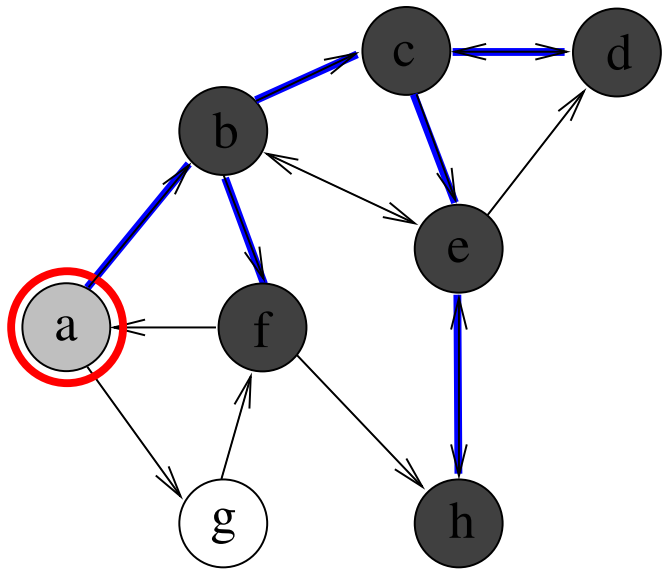


$p =< a >$
$s_i = b$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



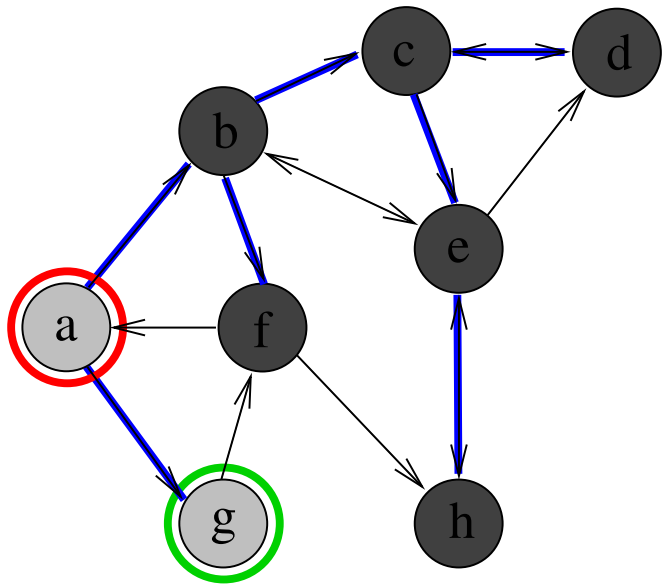
$p =< a >$

$s_i = a$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



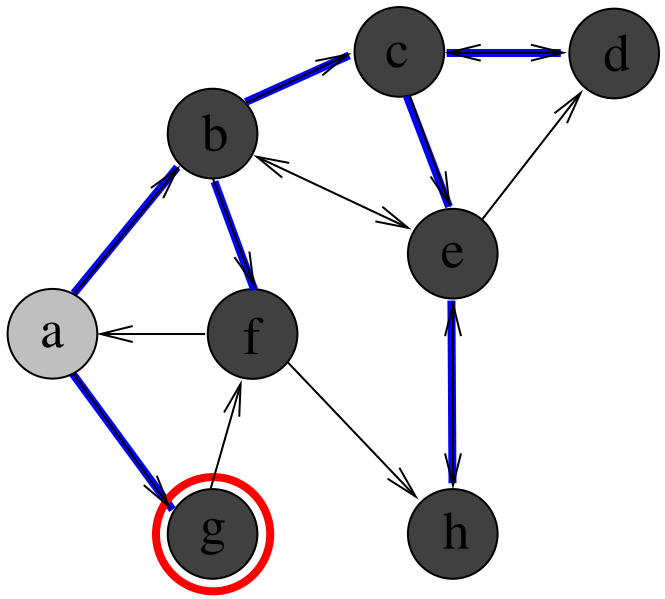
$p =< a, g >$

$s_i = a, s_j = g$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



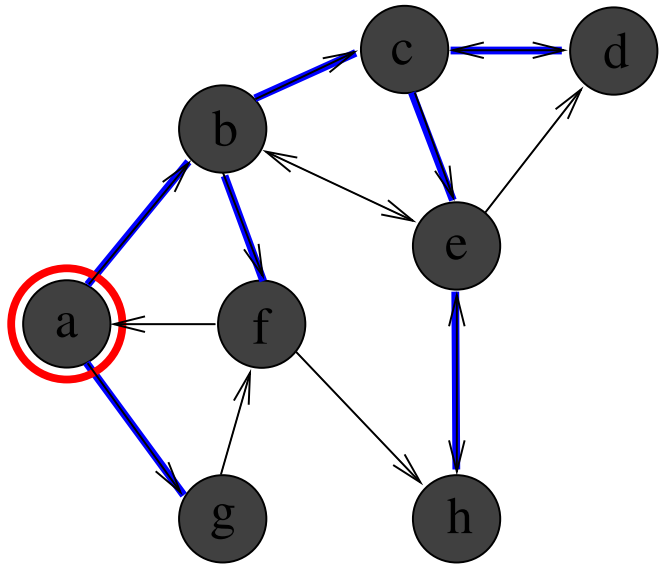
$p =< a >$

$s_i = g$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



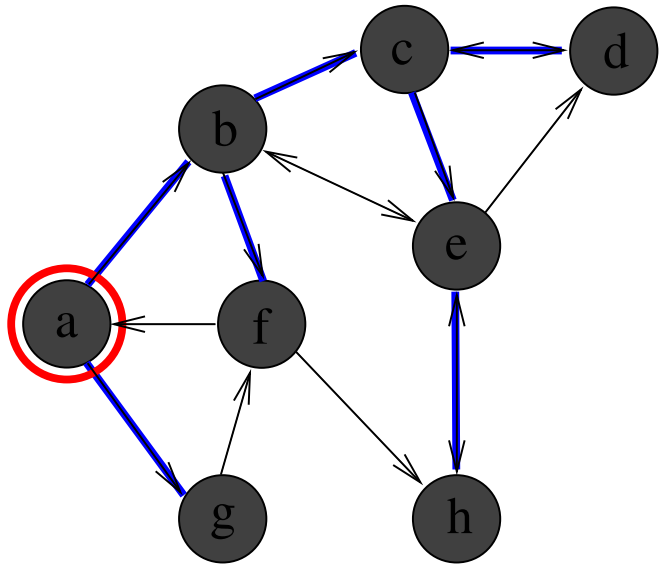
$p = <>$

$s_i = a$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



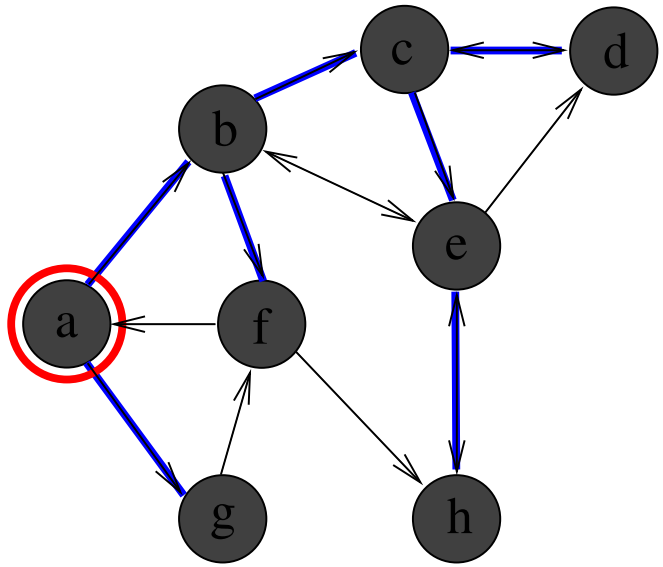
$p = <>$

$s_i = a$

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

Parcours en profondeur (Depth First search / DFS)

```
1  Fonction DFS(g, s_0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet s_i ∈ S faire
4          π[s_i] ← null
5          Colorier s_i en blanc
6      Empiler s_0 dans p et colorier s_0 en gris
7      tant que p n'est pas vide faire
8          Soit s_i le dernier sommet entré dans p
9          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
10             Empiler s_j dans p et colorier s_j en gris
11             π[s_j] ← s_i
12         sinon
13             Dépiler s_i de p et colorier s_i en noir
14     retourne π
```



$p = <>$

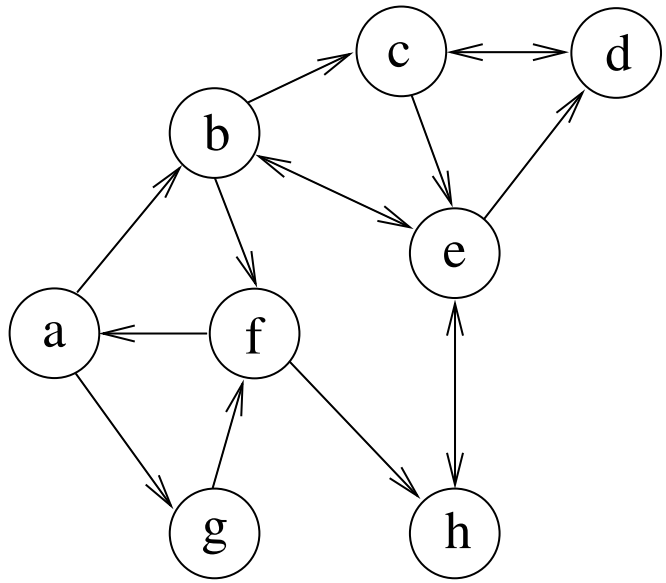
$s_i = a$

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

$\rightsquigarrow \mathcal{O}(n+p)$ (sous réserve d'une implémentation par listes d'adjacence)

Version récursive de DFS

```
1  Procédure DFSrec(g, s0)
       Entrée        : Un graphe g et un sommet s0
       Précondition  : s0 est blanc
2      début
3          Colorier s0 en gris
4          pour tout sj ∈ succ(s0) faire
5              si sj est blanc alors
6                  π[sj] ← s0
7                  DFSrec(g, sj)
8          Colorier s0 en noir
```



Variables globales :

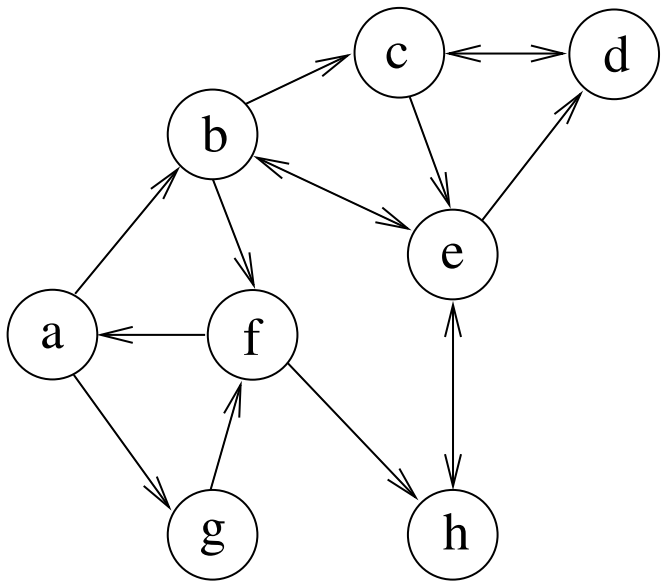
- Tableau π, initialisé à null avant le premier appel
- Couleur des sommets initialisée à blanc avant le premier appel

Remarque :

π correspond à l'arborescence des appels récursifs

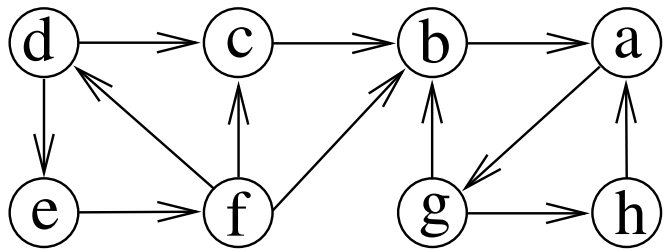
Recherche de circuits

```
1  Procédure DFSrec(g, s0)
       Entrée          : Un graphe g et un sommet s0
       Précondition    : s0 est blanc
2      début
3          Colorier s0 en gris
4          pour tout sj ∈ succ(s0) faire
5              si sj est gris alors afficher("circuit");
6              sinon si sj est blanc alors
7                  π[sj] ← s0
8                  DFSrec(g, sj)
9          Colorier s0 en noir
```



Numérotation des sommets

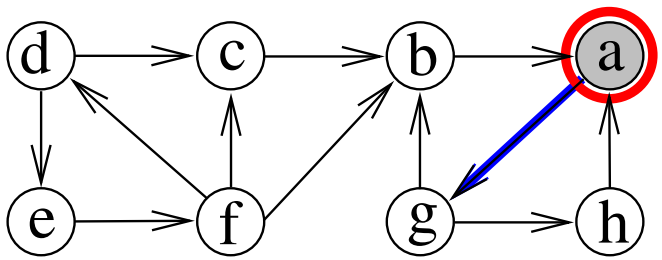
```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```



$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

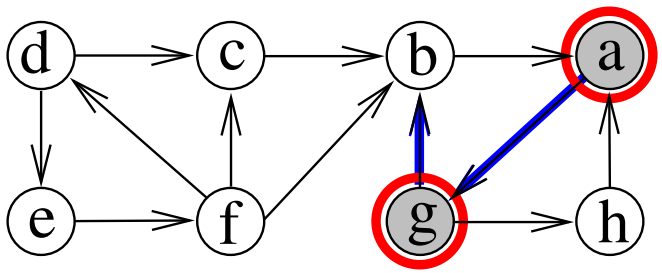


Pile = $< a >$
cpt = 1
$s_0 = a$; $s_j = g$

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

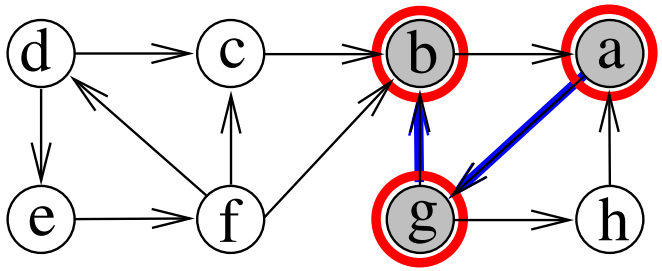


Pile = $< a, g >$
cpt = 1
$s_0 = g$; $s_j = b$

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

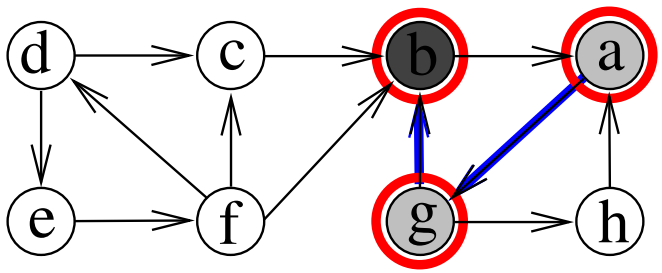


Pile = $< a, g, b >$
cpt = 1
$s_0 = b$

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

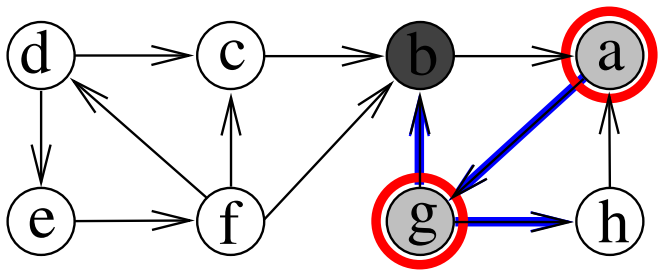


Pile = $< a, g, b >$
cpt = 2
$s_0 = b$
num[b]=1

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

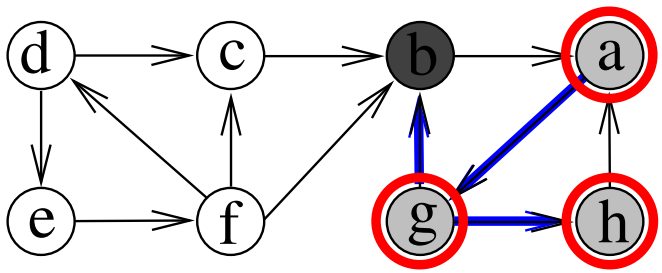


Pile = $< a, g >$
cpt = 2
$s_0 = g$; $s_j = h$
num[b]=1

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

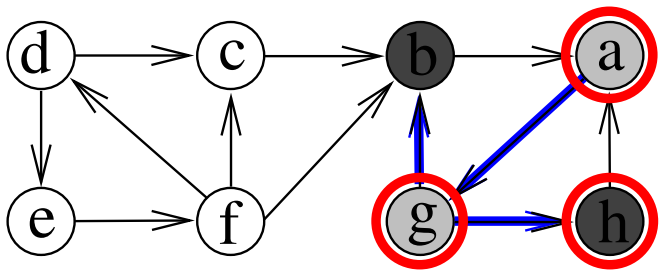


Pile = $< a, g, h >$
cpt = 2
$s_0 = h$
num[b]=1

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

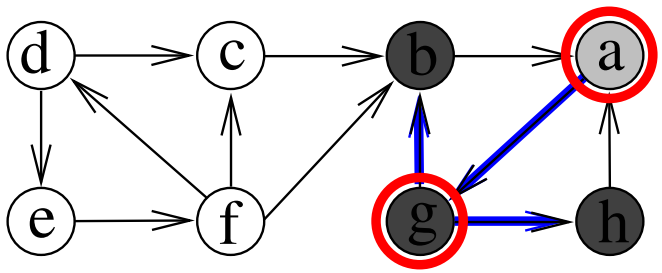


Pile = $< a, g, h >$
cpt = 3
$s_0 = h$
num[b]=1, num[h]=2

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

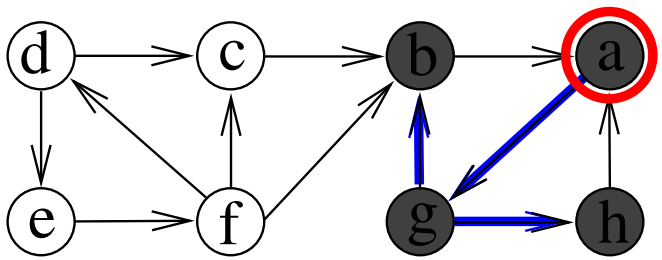


Pile = $< a, g >$
cpt = 4
$s_0 = g$
num[b]=1, num[h]=2, num[g]=3

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

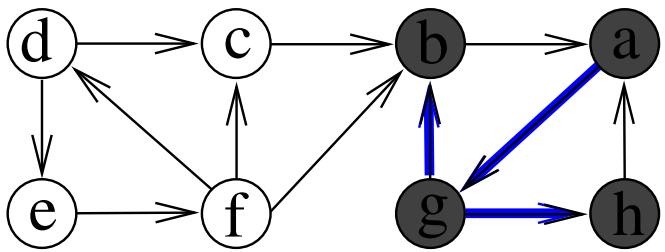


Pile = $< a >$
cpt = 5
$s_0 = a$
num[b]=1, num[h]=2, num[g]=3,
num[a]=4

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```



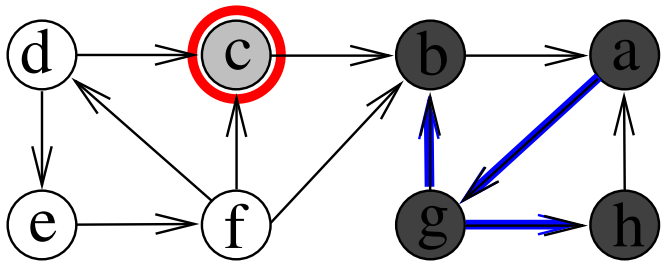
Pile = <>
cpt = 5

num[b]=1, num[h]=2, num[g]=3,
num[a]=4

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

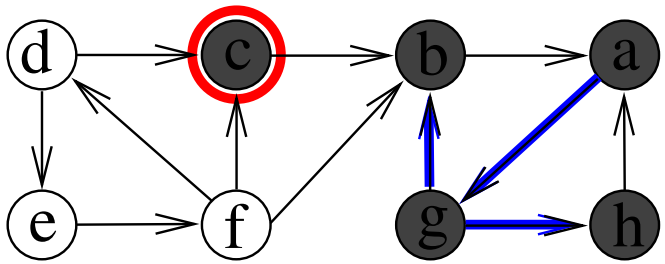


Pile = $< c >$
cpt = 5
$s_0 = c$
num[b]=1, num[h]=2, num[g]=3,
num[a]=4

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

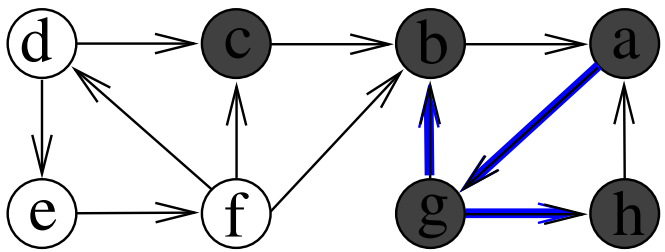


Pile = $< c >$
cpt = 6
$s_0 = c$
num[b]=1, num[h]=2, num[g]=3,
num[a]=4, num[c]=5

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```



Pile = <>
cpt = 6

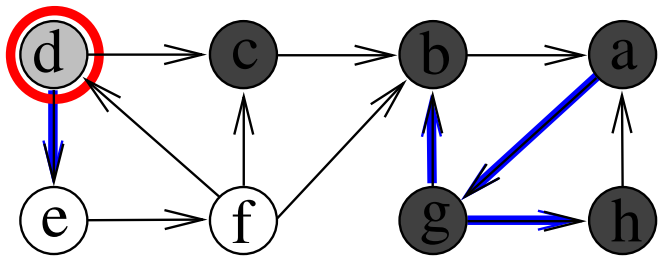
num[b]=1, num[h]=2, num[g]=3,
num[a]=4, num[c]=5

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

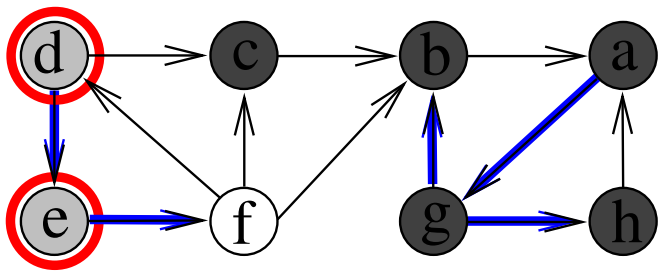
$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir



Pile = $< d >$
cpt = 6
$s_0 = d$; $s_j = e$
num[b]=1, num[h]=2, num[g]=3,
num[a]=4, num[c]=5

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

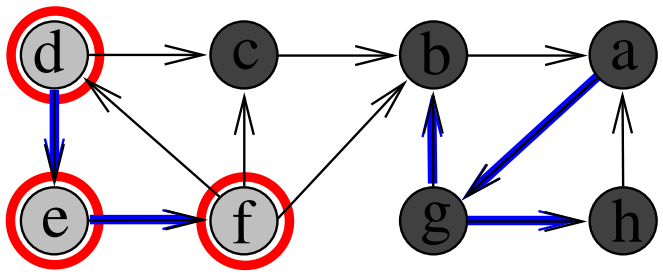


Pile = $< d, e >$
cpt = 6
$s_0 = e$; $s_j = f$
num[b]=1, num[h]=2, num[g]=3,
num[a]=4, num[c]=5

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

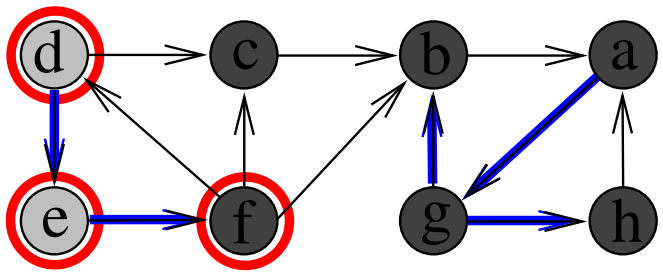


Pile = $< d, e, f >$
cpt = 6
$s_0 = f$
num[b]=1, num[h]=2, num[g]=3,
num[a]=4, num[c]=5

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

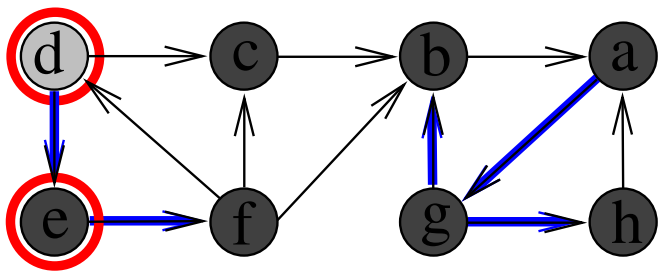


Pile = $< d, e, f >$
cpt = 7
$s_0 = f$
num[b]=1, num[h]=2, num[g]=3,
num[a]=4, num[c]=5, num[f]=6

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

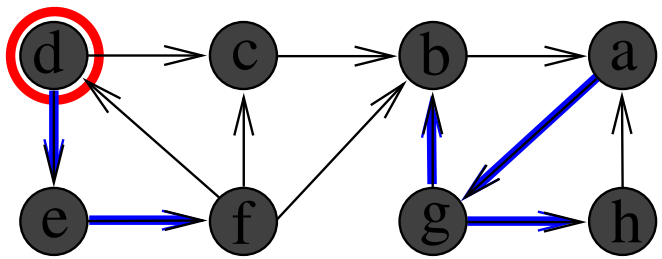


Pile = $< d, e >$
cpt = 8
$s_0 = e$
num[b]=1, num[h]=2, num[g]=3,
num[a]=4, num[c]=5, num[f]=6,
num[e]=7

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```

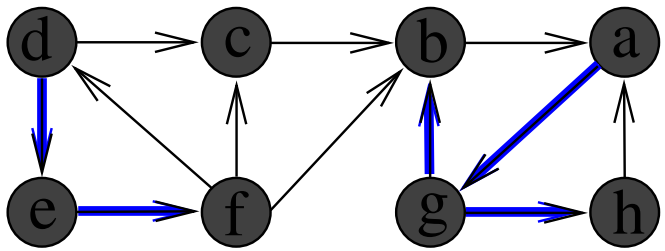


Pile = $< d >$
cpt = 9
$s_0 = d$
num[b]=1, num[h]=2, num[g]=3,
num[a]=4, num[c]=5, num[f]=6,
num[e]=7, num[d]=8

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

```
1  Procédure DFSnum(g)
2      cpt ← 1
3      Colorier tous les sommets en blanc
4      pour chaque sommet s_i de g faire
5          si s_i est blanc alors DFSrec(g, s_i);
6  Procédure DFSrec(g, s_0)
7      début
8          Colorier s_0 en gris
9          pour tout s_j ∈ succ(s_0) faire
10             si s_j est blanc alors
11                 π[s_j] ← s_0
12                 DFSrec(g, s_j)
13         Colorier s_0 en noir
14         num[s_0] ← cpt
15         cpt ← cpt + 1
```



Pile = <>
cpt = 9

num[b]=1, num[h]=2, num[g]=3,
num[a]=4, num[c]=5, num[f]=6,
num[e]=7, num[d]=8

$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Tri topologique d'un DAG

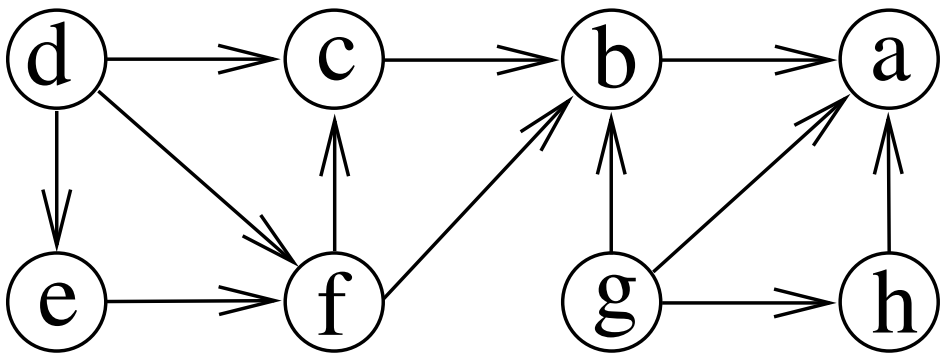
Définitions :

- DAG : graphe orienté sans circuit
- Tri topologique d'un DAG $G = (S, A)$: Ordre total sur S tel que $\forall (s_i, s_j) \in A, s_i < s_j$

Théorème :

Après l'exécution de DFSnum, pour tout arc $(s_i, s_j) \in A$: $num[s_j] < num[s_i]$

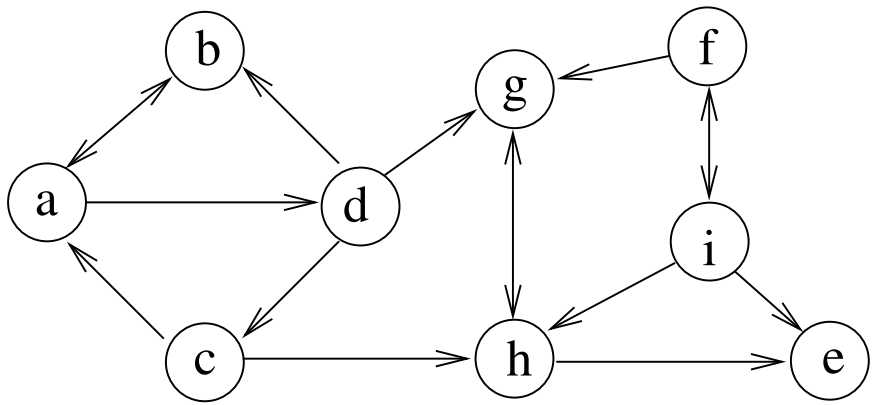
Exercice : Tri topologique du graphe suivant



Recherche des composantes fortements connexes

```
1  Fonction SCC(g)
2      SCC ← ∅
3      DFSnum(g)
4      Construire le graphe g^t = (S, A^t) tel que A^t = {(s_i, s_j) | (s_j, s_i) ∈ A}
5      Colorier tous les sommets de g^t en blanc
6      pour chaque sommet s_i pris par ordre de num décroissant faire
7          si s_i est blanc alors
8              Blanc ← {s_j ∈ S | s_j est blanc}
9              DFSrec(g^t, s_i)
10             Ajouter à SCC l'ensemble {s_j ∈ Blanc | s_j est noir}
11     retourne SCC
```

Exercice :



Recherche des composantes fortements connexes

```
1  Fonction SCC(g)
2      SCC ← ∅
3      DFSnum(g)
4      Construire le graphe g^t = (S, A^t) tel que A^t = {(s_i, s_j) | (s_j, s_i) ∈ A}
5      Colorier tous les sommets de g^t en blanc
6      pour chaque sommet s_i pris par ordre de num décroissant faire
7          si s_i est blanc alors
8              Blanc ← {s_j ∈ S | s_j est blanc}
9              DFSrec(g^t, s_i)
10             Ajouter à SCC l'ensemble {s_j ∈ Blanc | s_j est noir}
11     retourne SCC
```

Preuve de correction : Propriété vérifiée après la ligne 3

- Soit $g^{scc} = (S^{scc}, A^{scc})$ le DAG des composantes fortement connexes
- $\forall (scc_i, scc_j) \in A^{scc} : max\{num[s_i] | s_i \in scc_i\} > max\{num[s_j] | s_j \in scc_j\}$

$\rightsquigarrow \forall (scc_i, scc_j) \in A^{scc}, \exists s_i \in scc_i, \forall s_j \in scc_j$, s_i rencontré avt s_j (lignes 6-10)
$\rightsquigarrow$ DFSrec à partir de s_i permet de découvrir tous les sommets de scc_i

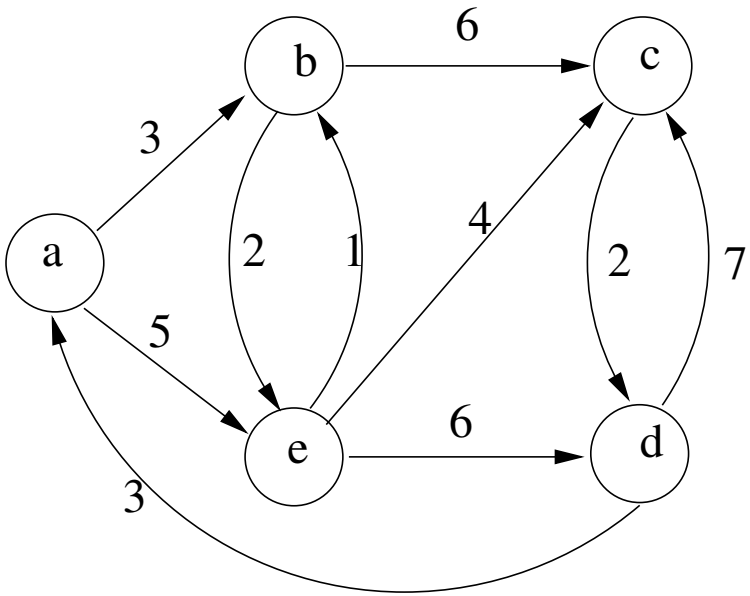
Définitions

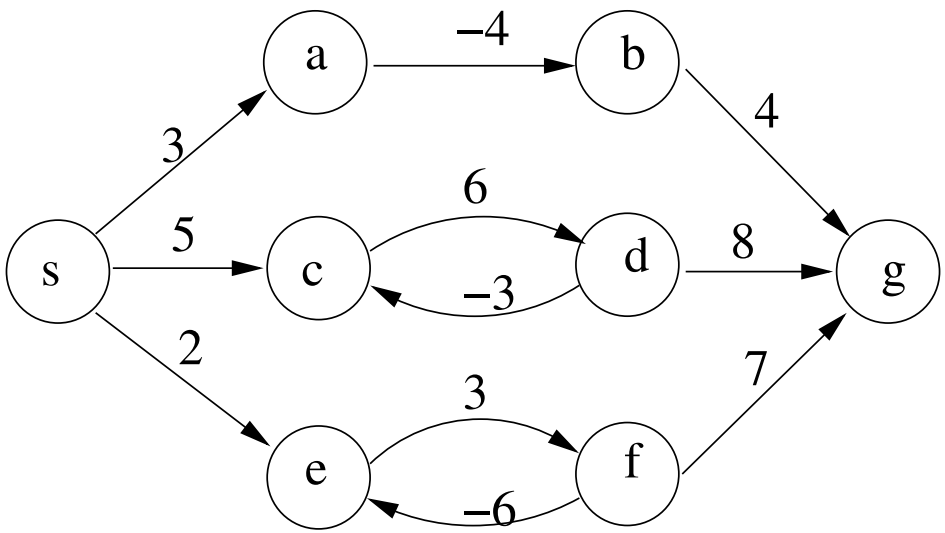
Soit $G = (S, A)$ un graphe (orienté ou non) muni d'une fonction $cout : A \to \mathbb{R}$

- Coût d'un chemin $p = \langle s_0, s_1, s_2, \ldots, s_k \rangle$: $cout(p) = \sum_{i=1}^{k} cout(s_{i-1}, s_i)$
- Coût d'un plus court chemin de s_i vers s_j = $\delta(s_i, s_j)$:

$$\begin{array}{lcll} \delta(s_i, s_j) & = & +\infty & \text{si } \nexists \text{ chemin de } s_i \text{ vers } s_j \\ \delta(s_i, s_j) & = & -\infty & \text{si } \exists \text{ circuit absorbant} \\ \delta(s_i, s_j) & = & \min\{cout(p) | p = \text{chemin de } s_i \text{ a } s_j\} & \text{sinon} \end{array}$$

Exemples :





Specification d'un algo de plus courts chemins à origine unique

1 **Fonction** *PlusCourtsChemins*$(g, cout, s_0)$

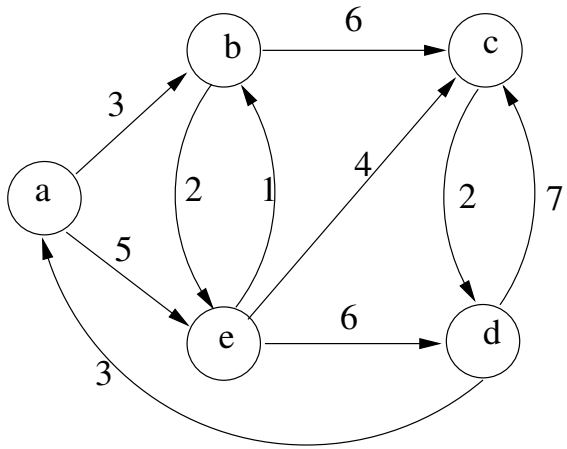
Entrée : Un graphe (orienté ou non) $g = (S, A)$, une fonction $cout : A \to \mathbb{R}$ et un sommet de départ $s_0 \in S$

Postcondition : Retourne une arbo. des plus courts chemins partant de s_0 et un tableau d tq $\forall s_i \in S, d[s_i] = \delta(s_0, s_i)$

Arborescence des plus courts chemins :

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

Exemple :



Principe des algos 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 $+\infty$
  $\rightsquigarrow$ $d[s_i]$ = borne supérieure de $\delta(s_0, s_i)$
- Raboter itérativement les bornes d par relâchements d'arcs

1 **Procédure** *relacher*$((s_i, s_j), \pi, d)$
  Entrée : Un arc (s_i, s_j)
  Entrée/Sortie : Les tableaux π et d
  Précondition : $d[s_i] \geq \delta(s_0, s_i)$ et $d[s_j] \geq \delta(s_0, s_j)$
  Postcondition : $\delta(s_0, s_j) \leq d[s_j] \leq d[s_i] + cout(s_i, s_j)$
2 **début**
3 **si** $d[s_j] > d[s_i] + cout(s_i, s_j)$ **alors**
4 $d[s_j] \leftarrow d[s_i] + cout(s_i, s_j)$
5 $\pi[s_j] \leftarrow s_i$

Algorithmes de recherche de plus courts chemins

Principe commun à tous les 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
  $\rightsquigarrow$ Chaque arc est relâché exactement une fois
  $\rightsquigarrow$ Ne marche que si tous les coûts sont positifs
- TopoDAG relâche un arc si tous ses prédécesseurs ont été relâchés
  $\rightsquigarrow$ Chaque arc est relâché exactement une fois
  $\rightsquigarrow$ Ne marche que si le graphe est acyclique
- Bellman-Ford relâche tous les arcs à chaque itér., jusqu'à convergence
  $\rightsquigarrow$ Chaque arc est relâché plusieurs fois
  $\rightsquigarrow$ Marche dans tous les cas

Principe de l'algorithme de Dijkstra

Généralisation d'un BFS à des graphes valués :

- Procède par coloriage des sommets :
  - s_i est blanc s'il n'a pas encore été découvert
    $\rightsquigarrow d[s_i] = +\infty$
  - s_i est gris s'il a été découvert et sa borne peut encore diminuer
    $\rightsquigarrow \delta(s_0, s_i) \leq d[s_i] < +\infty$
  - s_i est noir si sa borne ne peut plus diminuer
    $\rightsquigarrow d[s_i] = \delta(s_0, s_i)$
    $\rightsquigarrow$ Tous les arcs partant de s_i ont été relâchés
- A chaque itération, les arcs partant d'un sommet gris sont relâchés
  - Stratégie gloutonne pour choisir ce sommet gris
    $\rightsquigarrow$ Sommet gris minimisant d
  - Ne marche que si tous les coûts sont positifs
    $\rightsquigarrow$ Précondition à Dijkstra : Pour tout arc $(s_i, s_j) \in A$, $cout(s_i, s_j) \geq 0$

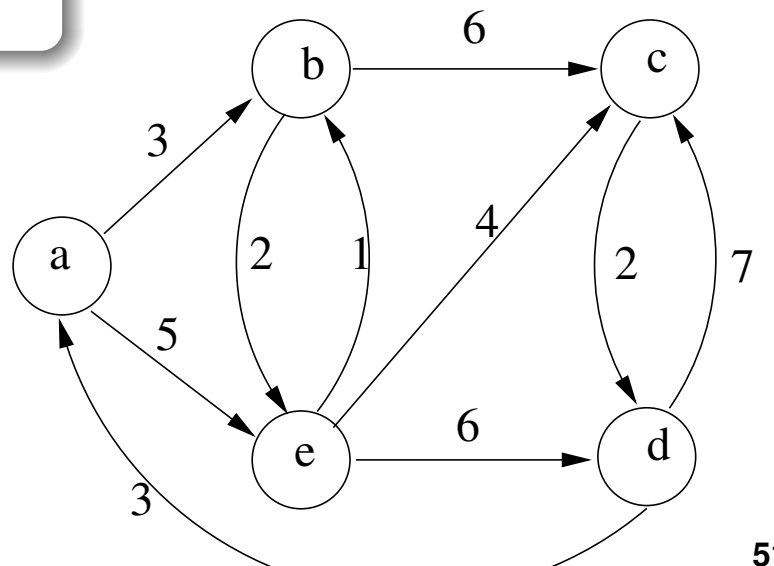
```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

Propriété invariante ligne 5 :

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

Exemple :



```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

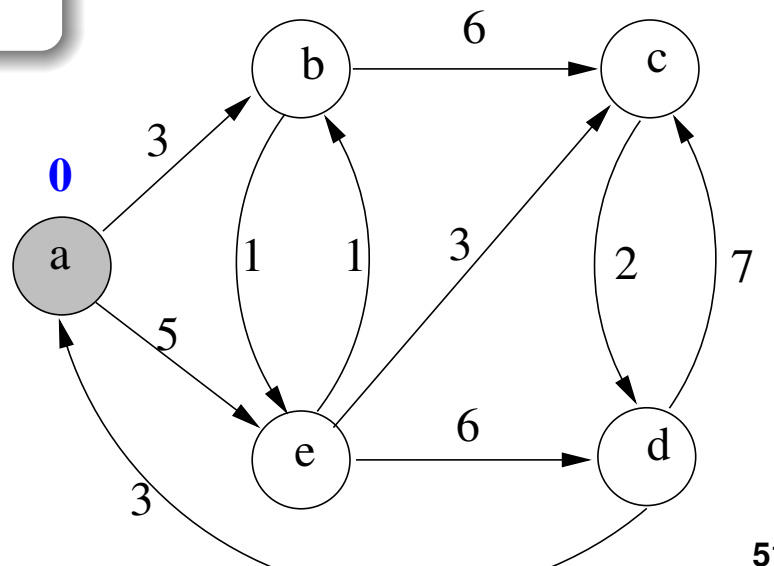
Propriété invariante ligne 5 :

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

Exemple :

$s_0 = a$



```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

Propriété invariante ligne 5 :

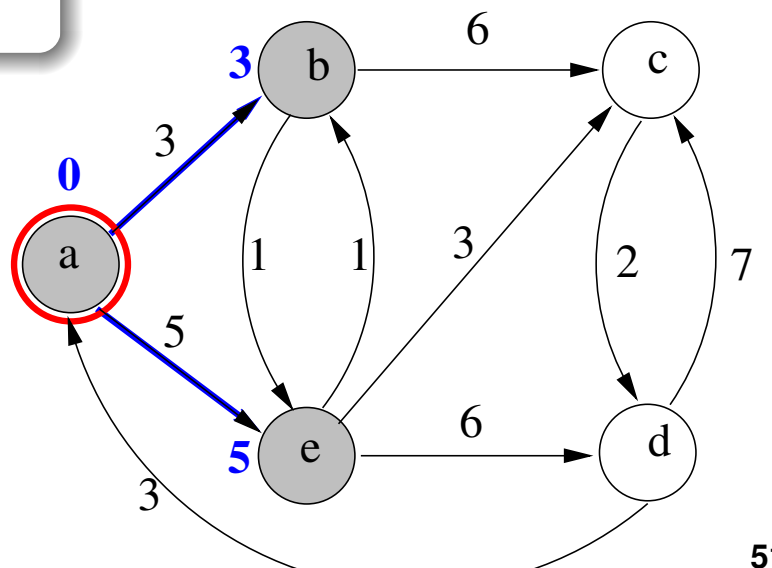
Pour tout sommet s_i,

- 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

Exemple :

$s_i = a$

Arcs relâchés : $(a, b), (a, e)$



```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

Propriété invariante ligne 5 :

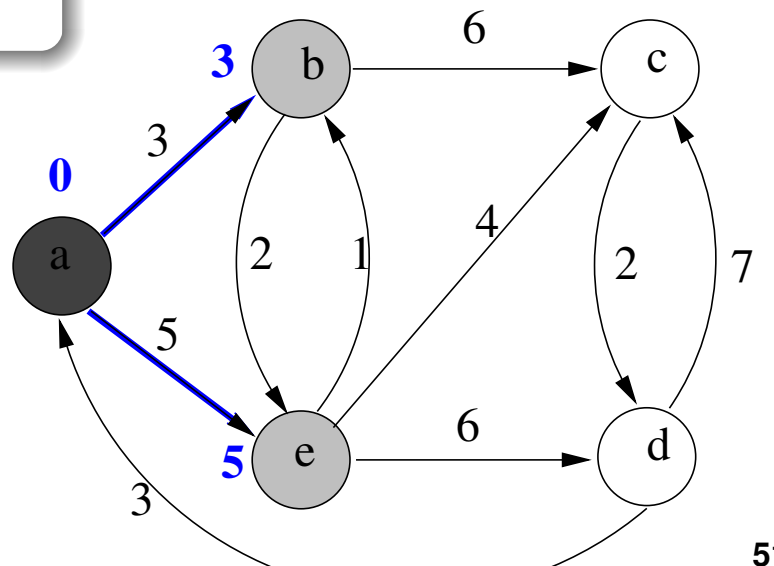
Pour tout sommet s_i,

- 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

Exemple :

$s_i = a$

Arcs relâchés : $(a, b), (a, e)$



```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

Propriété invariante ligne 5 :

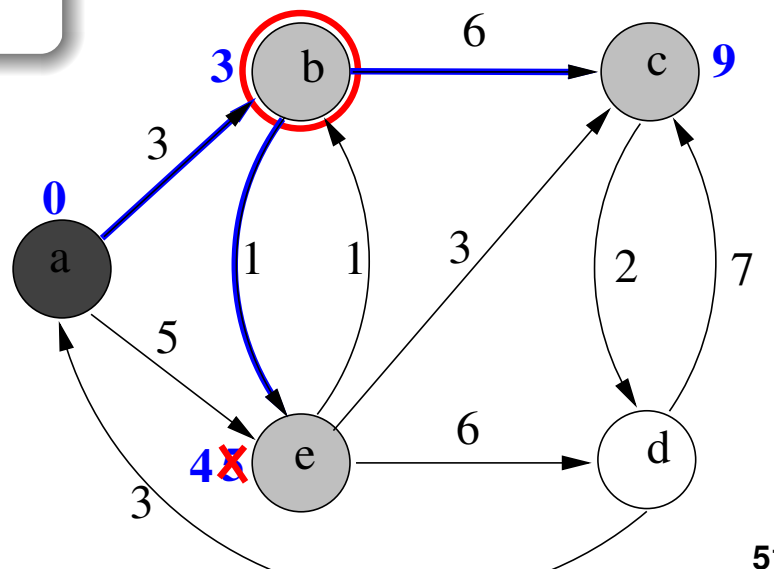
Pour tout sommet s_i,

- 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

Exemple :

$s_i = b$

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



```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

Propriété invariante ligne 5 :

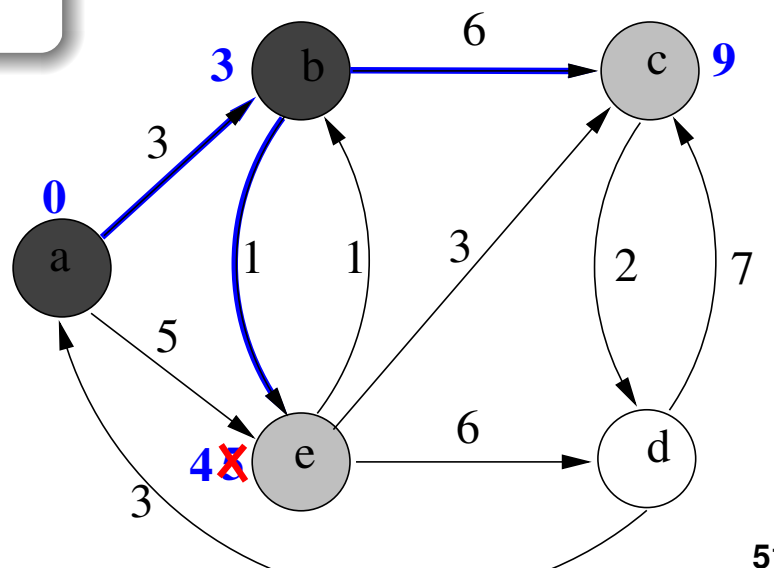
Pour tout sommet s_i,

- 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

Exemple :

$s_i = b$

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



```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

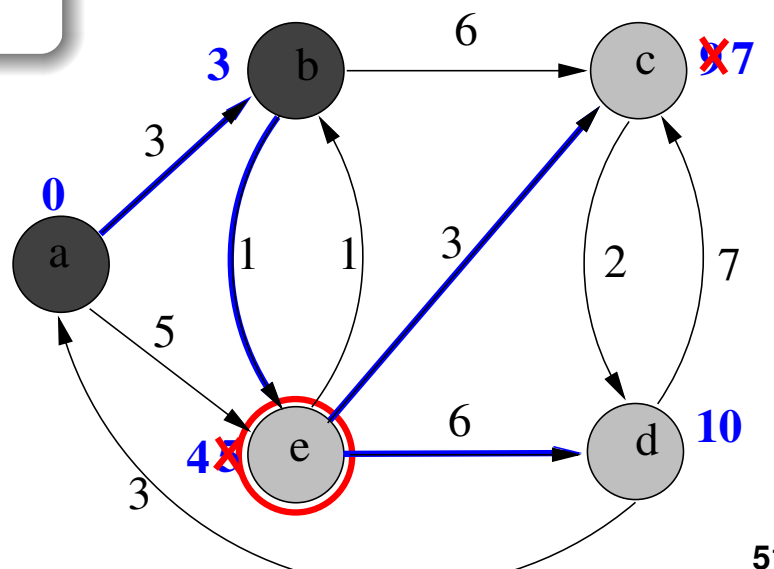
Propriété invariante ligne 5 :

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

Exemple :

$s_i = e$
Arcs relâchés : $(a, b), (a, e), (b, e), (b, c), (e, c), (e, d)$



```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

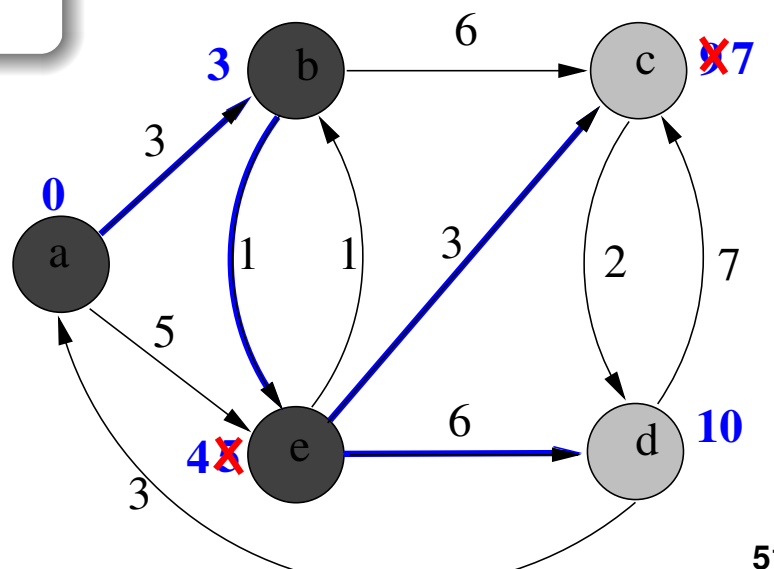
Propriété invariante ligne 5 :

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

Exemple :

$s_i = e$
Arcs relâchés : $(a, b), (a, e), (b, e), (b, c), (e, c), (e, d)$



```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

Propriété invariante ligne 5 :

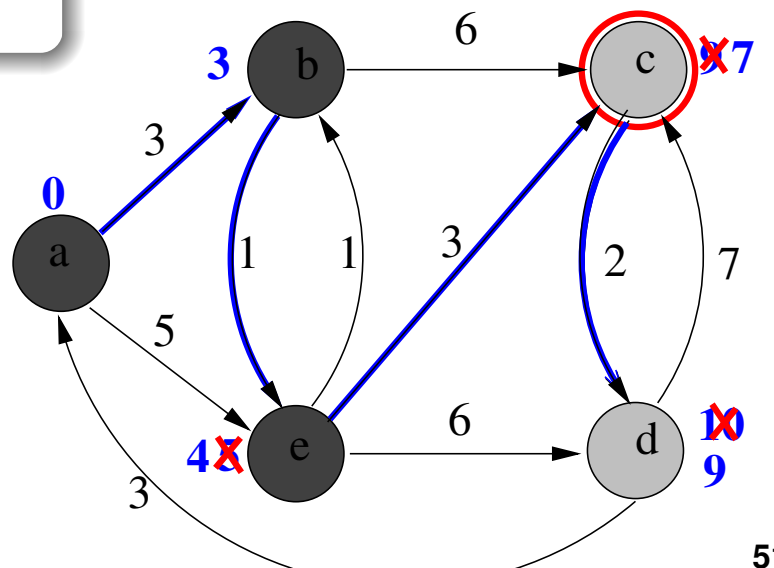
Pour tout sommet s_i,

- 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

Exemple :

$s_i = c$

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



```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

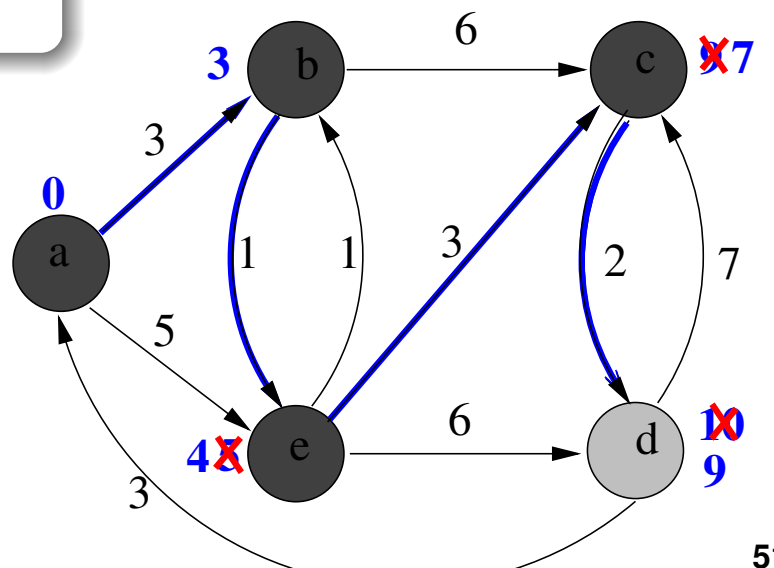
Propriété invariante ligne 5 :

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

Exemple :

$s_i = c$
Arcs relâchés : $(a, b), (a, e), (b, e), (b, c), (e, c), (e, d), (c, d)$



```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

Propriété invariante ligne 5 :

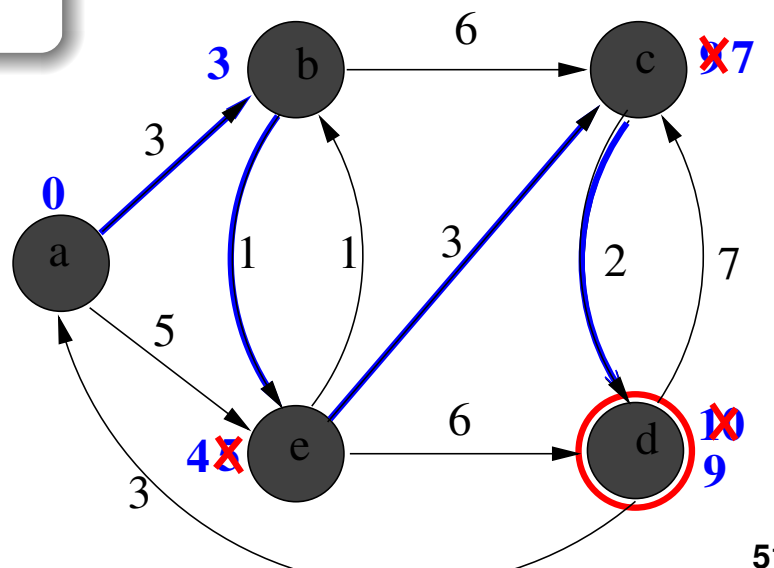
Pour tout sommet s_i,

- 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

Exemple :

$s_i = d$

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



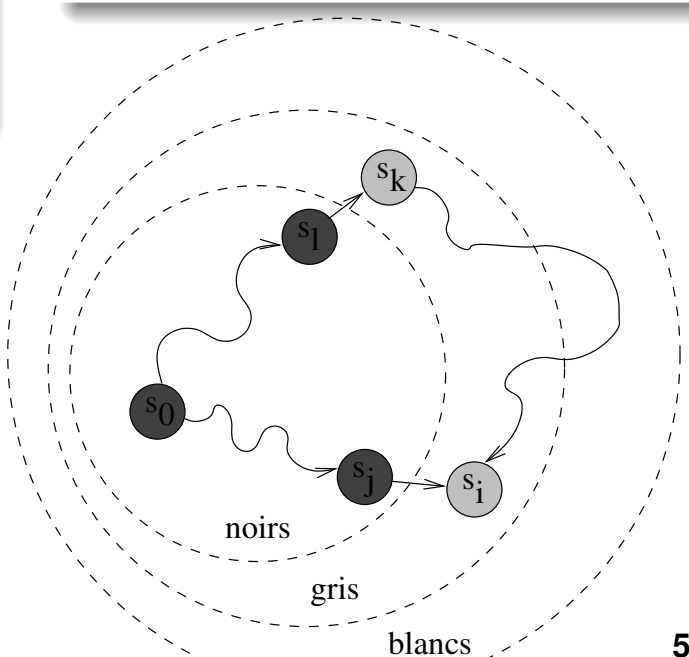
```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

Exemple :

Propriété invariante ligne 5 :

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



```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

Propriété invariante ligne 5 :

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

Complexité pour un graphe ayant n sommets et p arcs ?

- $\mathcal{O}(n^2)$ si recherche linéaire du sommet gris minimisant d (ligne 6)
- $\mathcal{O}((n+p)log(n))$ si les sommets gris sont stockés dans un tas binaire

```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

Propriété invariante ligne 5 :

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

Complexité pour un graphe ayant n sommets et p arcs ?

- $\mathcal{O}(n^2)$ si recherche linéaire du sommet gris minimisant d (ligne 6)
- $\mathcal{O}((n+p)log(n))$ si les sommets gris sont stockés dans un tas binaire

```
1  Fonction Dijkstra(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞ ; π[s_i] ← null ; Colorier s_i en blanc
4      d[s_0] ← 0 ; Colorier s_0 en gris
5      tant que il existe un sommet gris faire
6          Soit s_i le sommet gris tel que d[s_i] soit minimal
7          pour tout sommet s_j ∈ succ(s_i) faire
8              si s_j est blanc ou gris alors
9                  relacher((s_i, s_j), π, d)
10                 si s_j est blanc alors
11                     Colorier s_j en gris
12         Colorier s_i en noir
13     retourne π et d
```

Propriété invariante ligne 5 :

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

Complexité pour un graphe ayant n sommets et p arcs ?

- $\mathcal{O}(n^2)$ si recherche linéaire du sommet gris minimisant d (ligne 6)
- $\mathcal{O}((n+p)log(n))$ si les sommets gris sont stockés dans un tas binaire

Extension de Dijkstra pour calculer un “meilleur” chemin

Peut-on utiliser Dijkstra pour calculer un meilleur chemin ?

- Coût d’un chemin = fonction (somme, produit, ...) des coûts des arcs
- Recherche d’un chemin minimisant ou maximisant ce coût

Précondition à l’utilisation de Dijkstra :

L’ajout d’un arc (s_i, s_j) à un chemin $s_0 \rightsquigarrow s_i$ ne peut que dégrader son coût :

- Si on cherche un min, alors $cout(s_0 \rightsquigarrow s_i \to s_j) \geq cout(s_0 \rightsquigarrow s_i)$
- Si on cherche un max, alors $cout(s_0 \rightsquigarrow s_i \to s_j) \leq cout(s_0 \rightsquigarrow s_i)$

Exemple d’adaptation de Dijkstra :

Recherche d’un maximum avec :

- Cout des arcs compris entre 0 et 1
- Cout d’un chemin = produit des arcs

Init. d à 0, sauf pour s_0 (init. à 1)

Adapter le relâchement :

```
1 Procédure relacher((s_i, s_j), π, d)
2     si d[s_j] < d[s_i] * cout(s_i, s_j) alors
3         d[s_j] ← d[s_i] * cout(s_i, s_j)
4         π[s_j] ← s_i
```

Extension de Dijkstra pour calculer un “meilleur” chemin

Peut-on utiliser Dijkstra pour calculer un meilleur chemin ?

- Coût d’un chemin = fonction (somme, produit, ...) des coûts des arcs
- Recherche d’un chemin minimisant ou maximisant ce coût

Précondition à l’utilisation de Dijkstra :

L’ajout d’un arc (s_i, s_j) à un chemin $s_0 \rightsquigarrow s_i$ ne peut que dégrader son coût :

- Si on cherche un min, alors $cout(s_0 \rightsquigarrow s_i \rightarrow s_j) \geq cout(s_0 \rightsquigarrow s_i)$
- Si on cherche un max, alors $cout(s_0 \rightsquigarrow s_i \rightarrow s_j) \leq cout(s_0 \rightsquigarrow s_i)$

Exemple d’adaptation de Dijkstra :

Recherche d’un maximum avec :

- Cout des arcs compris entre 0 et 1
- Cout d’un chemin = produit des arcs

Init. d à 0, sauf pour s_0 (init. à 1)

Adapter le relâchement :

```
1 Procédure relacher((s_i, s_j), π, d)
2     si d[s_j] < d[s_i] * cout(s_i, s_j) alors
3         d[s_j] ← d[s_i] * cout(s_i, s_j)
4         π[s_j] ← s_i
```

Extension de Dijkstra pour calculer un “meilleur” chemin

Peut-on utiliser Dijkstra pour calculer un meilleur chemin ?

- Coût d’un chemin = fonction (somme, produit, ...) des coûts des arcs
- Recherche d’un chemin minimisant ou maximisant ce coût

Précondition à l’utilisation de Dijkstra :

L’ajout d’un arc (s_i, s_j) à un chemin $s_0 \rightsquigarrow s_i$ ne peut que dégrader son coût :

- Si on cherche un min, alors $cout(s_0 \rightsquigarrow s_i \rightarrow s_j) \geq cout(s_0 \rightsquigarrow s_i)$
- Si on cherche un max, alors $cout(s_0 \rightsquigarrow s_i \rightarrow s_j) \leq cout(s_0 \rightsquigarrow s_i)$

Exemple d’adaptation de Dijkstra :

Recherche d’un maximum avec :

- Cout des arcs compris entre 0 et 1
- Cout d’un chemin = produit des arcs

Init. d à 0, sauf pour s_0 (init. à 1)

Adapter le relâchement :

```
1 Procédure relacher((s_i, s_j), π, d)
2     si d[s_j] < d[s_i] * cout(s_i, s_j) alors
3         d[s_j] ← d[s_i] * cout(s_i, s_j)
4         π[s_j] ← s_i
```

Principe de l'algorithme

Rappel :

Un DAG est un graphe orienté sans circuit

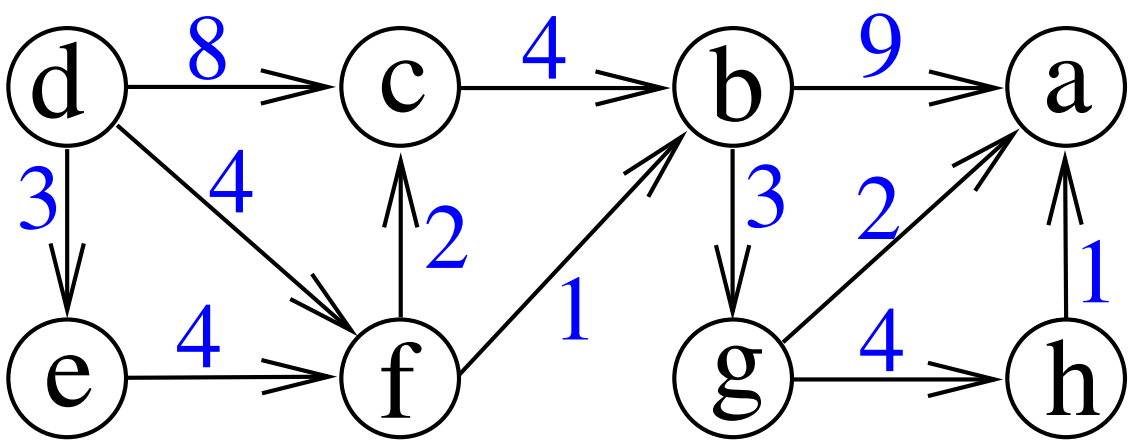
Idée :

- Relâcher les arcs partant de s_i dès que tous les arcs se trouvant sur un chemin entre s_0 et s_i ont déjà été relâchés
- Utiliser un tri topologique pour déterminer l'ordre de relâchement

Algorithme TopoDAG

```
1  Fonction TopoDAG(g, cout, s_0)
       Précondition          : g est un DAG
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞
4          π[s_i] ← null
5      d[s_0] ← 0
6      Trier topologiquement les sommets de g à l'aide d'un parcours en profondeur
7      pour chaque sommet s_i pris selon l'ordre topologique faire
8          pour chaque sommet s_j ∈ succ(s_i) faire
9              relacher((s_i, s_j), π, d)
10     retourne π et d
```

Exercice :



Algorithme TopoDAG

```
1  Fonction TopoDAG(g, cout, s_0)
       Précondition          : g est un DAG
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞
4          π[s_i] ← null
5      d[s_0] ← 0
6      Trier topologiquement les sommets de g à l'aide d'un parcours en profondeur
7      pour chaque sommet s_i pris selon l'ordre topologique faire
8          pour chaque sommet s_j ∈ succ(s_i) faire
9              relacher((s_i, s_j), π, d)
10     retourne π et d
```

Complexité pour un graphe ayant n sommets et p arcs ?

Algorithme TopoDAG

```
1  Fonction TopoDAG(g, cout, s_0)
       Précondition : g est un DAG
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞
4          π[s_i] ← null
5      d[s_0] ← 0
6      Trier topologiquement les sommets de g à l'aide d'un parcours en profondeur
7      pour chaque sommet s_i pris selon l'ordre topologique faire
8          pour chaque sommet s_j ∈ succ(s_i) faire
9              relacher((s_i, s_j), π, d)
10     retourne π et d
```

Complexité pour un graphe ayant n sommets et p arcs ?

$\rightsquigarrow \mathcal{O}(n+p)$

Extension de TopoDAG pour calculer un “meilleur” chemin

Peut-on utiliser TopoDAG pour calculer un meilleur chemin ?

- Coût d’un chemin = fonction (somme, produit, ...) des coûts des arcs
- Recherche d’un chemin minimisant ou maximisant ce coût

Oui !

- Sous réserve que le graphe soit acyclique
- Adapter la procédure de relâchement et l’initialisation de d

Exemple d’adaptation de TopoDAG :

- Recherche d’un plus long chemin quand le coût d’un chemin est égal au coût de son plus petit arc
- Init. d à $-\infty$, sauf pour s_0 $(+\infty)$

```
1 Procédure relacher((s_i, s_j), π, d)
2     si d[s_j] < min(d[s_i], cout(s_i, s_j)) alors
3         d[s_j] ← min(d[s_i], cout(s_i, s_j))
4         π[s_j] ← s_i
```

Extension de TopoDAG pour calculer un “meilleur” chemin

Peut-on utiliser TopoDAG pour calculer un meilleur chemin ?

- Coût d’un chemin = fonction (somme, produit, ...) des coûts des arcs
- Recherche d’un chemin minimisant ou maximisant ce coût

Oui !

- Sous réserve que le graphe soit acyclique
- Adapter la procédure de relâchement et l’initialisation de d

Exemple d’adaptation de TopoDAG :

- Recherche d’un plus long chemin quand le coût d’un chemin est égal au coût de son plus petit arc
- Init. d à $-\infty$, sauf pour s_0 $(+\infty)$

```
1 Procédure relacher((s_i, s_j), π, d)
2     si d[s_j] < min(d[s_i], cout(s_i, s_j)) alors
3         d[s_j] ← min(d[s_i], cout(s_i, s_j))
4         π[s_j] ← s_i
```

Extension de TopoDAG pour calculer un “meilleur” chemin

Peut-on utiliser TopoDAG pour calculer un meilleur chemin ?

- Coût d’un chemin = fonction (somme, produit, ...) des coûts des arcs
- Recherche d’un chemin minimisant ou maximisant ce coût

Oui !

- Sous réserve que le graphe soit acyclique
- Adapter la procédure de relâchement et l’initialisation de d

Exemple d’adaptation de TopoDAG :

- Recherche d’un plus long chemin quand le coût d’un chemin est égal au coût de son plus petit arc
- Init. d à $-\infty$, sauf pour s_0 $(+\infty)$

```
1 Procédure relacher((s_i, s_j), π, d)
2     si d[s_j] < min(d[s_i], cout(s_i, s_j)) alors
3         d[s_j] ← min(d[s_i], cout(s_i, s_j))
4         π[s_j] ← s_i
```

Extension de TopoDAG pour calculer un “meilleur” chemin

Peut-on utiliser TopoDAG pour calculer un meilleur chemin ?

- Coût d’un chemin = fonction (somme, produit, ...) des coûts des arcs
- Recherche d’un chemin minimisant ou maximisant ce coût

Oui !

- Sous réserve que le graphe soit acyclique
- Adapter la procédure de relâchement et l’initialisation de d

Exemple d’adaptation de TopoDAG :

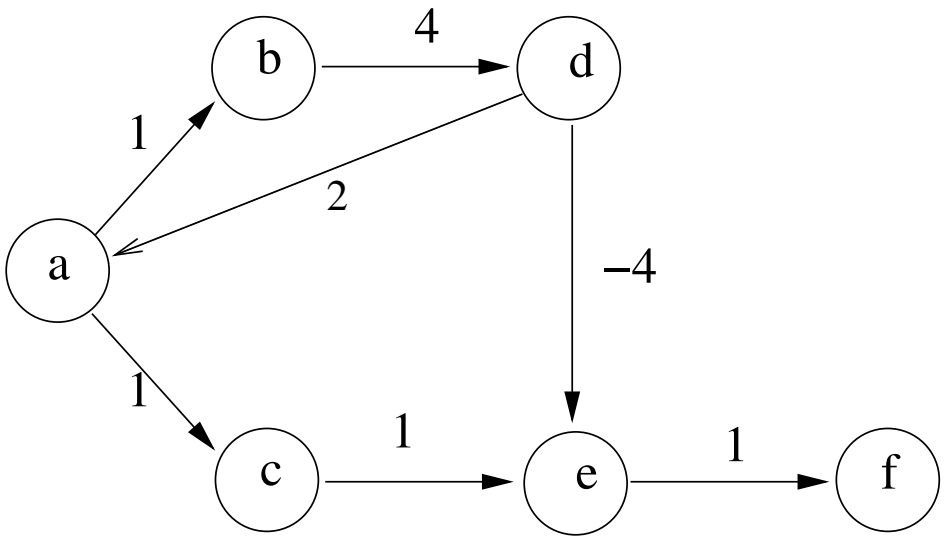
- Recherche d’un plus long chemin quand le coût d’un chemin est égal au coût de son plus petit arc
- Init. d à $-\infty$, sauf pour s_0 $(+\infty)$

```
1 Procédure relacher((s_i, s_j), π, d)
2     si d[s_j] < min(d[s_i], cout(s_i, s_j)) alors
3         d[s_j] ← min(d[s_i], cout(s_i, s_j))
4         π[s_j] ← s_i
```

Aurait-on pu utiliser Dijkstra dans ce cas ?

Principe de l'algorithme

Exemple de graphe pour lequel Dijkstra et TopoDAG ne marchent pas :



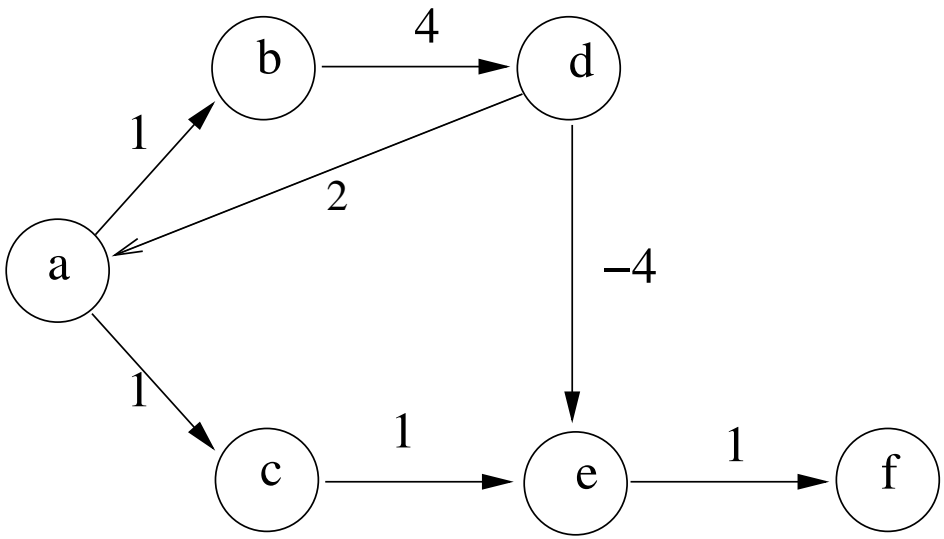
Idée de l'algorithme de Bellman-Ford :

- Répéter :
  - Relâcher tous les arcs
- Jusqu'à avoir trouvé tous les plus courts chemins

Quand sait-on qu'on a trouvé tous les plus courts chemins ?

Principe de l'algorithme

Exemple de graphe pour lequel Dijkstra et TopoDAG ne marchent pas :



Idée de l'algorithme de Bellman-Ford :

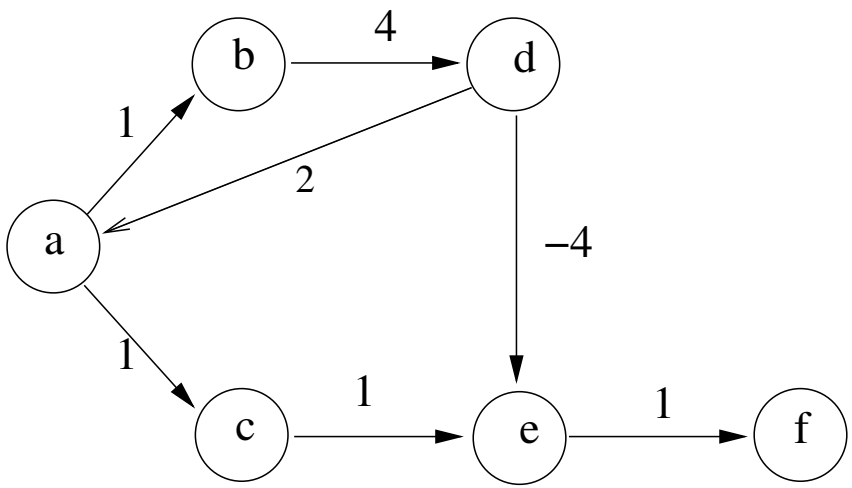
- Répéter :
  - Relâcher tous les arcs
- Jusqu'à avoir trouvé tous les plus courts chemins

Quand sait-on qu'on a trouvé tous les plus courts chemins ?

Algorithme Bellman-Ford

```
1  Fonction Bellman-Ford(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞
4          π[s_i] ← null
5      d[s_0] ← 0
6      pour ?????? faire
7          pour chaque arc (s_i, s_j) ∈ A faire
8              relacher((s_i, s_j), π, d)
11     retourne π et d
```

Exemple :



Algorithme Bellman-Ford

```
1  Fonction Bellman-Ford(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞
4          π[s_i] ← null
5      d[s_0] ← 0
6      pour ????? faire
7          pour chaque arc (s_i, s_j) ∈ A faire
8              relacher((s_i, s_j), π, d)
11     retourne π et d
```

Propriété vérifiée après k passages boucle 6-8, pour tout sommet s_i :

$d[s_i]$ = longueur du plus court chemin de s_0 à s_i ayant au plus k arcs

- Arrêter après $|S| - 1$ passages
- Détecter les circuits absorbants

Algorithme Bellman-Ford

```
1  Fonction Bellman-Ford(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞
4          π[s_i] ← null
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              relacher((s_i, s_j), π, d)
11     retourne π et d
```

Propriété vérifiée après k passages boucle 6-8, pour tout sommet s_i :

$d[s_i]$ = longueur du plus court chemin de s_0 à s_i ayant au plus k arcs

- Arrêter après $|S| - 1$ passages
- Détecter les circuits absorbants

Algorithme Bellman-Ford

```
1  Fonction Bellman-Ford(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞
4          π[s_i] ← null
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              relacher((s_i, s_j), π, d)
9      si ∃ un arc (s_i, s_j) ∈ A tel que d[s_j] > d[s_i] + cout(s_i, s_j) alors
10         afficher("Le graphe contient un circuit absorbant")
11     retourne π et d
```

Propriété vérifiée après k passages boucle 6-8, pour tout sommet s_i :

$d[s_i]$ = longueur du plus court chemin de s_0 à s_i ayant au plus k arcs

- Arrêter après $|S| - 1$ passages
- Détecter les circuits absorbants

Algorithme Bellman-Ford

```
1  Fonction Bellman-Ford(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞
4          π[s_i] ← null
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              relacher((s_i, s_j), π, d)
9      si ∃ un arc (s_i, s_j) ∈ A tel que d[s_j] > d[s_i] + cout(s_i, s_j) alors
10         afficher("Le graphe contient un circuit absorbant")
11     retourne π et d
```

Complexité pour un graphe ayant n sommets et p arcs ?

Algorithme Bellman-Ford

```
1  Fonction Bellman-Ford(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞
4          π[s_i] ← null
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              relacher((s_i, s_j), π, d)
9      si ∃ un arc (s_i, s_j) ∈ A tel que d[s_j] > d[s_i] + cout(s_i, s_j) alors
10         afficher("Le graphe contient un circuit absorbant")
11     retourne π et d
```

Complexité pour un graphe ayant n sommets et p arcs ?

- $\mathcal{O}(np)$
- Possibilité d'améliorer les performances (sans changer la complexité) :
  - Arrêter dès que d n'est plus modifié
  - Ne relâcher que les arcs (s_i, s_j) pour lesquels $d[s_i]$ a été modifié

Algorithme Bellman-Ford

```
1  Fonction Bellman-Ford(g, cout, s_0)
2      pour chaque sommet s_i ∈ S faire
3          d[s_i] ← +∞
4          π[s_i] ← null
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              relacher((s_i, s_j), π, d)
9      si ∃ un arc (s_i, s_j) ∈ A tel que d[s_j] > d[s_i] + cout(s_i, s_j) alors
10         afficher("Le graphe contient un circuit absorbant")
11     retourne π et d
```

Complexité pour un graphe ayant n sommets et p arcs ?

- $\mathcal{O}(np)$
- Possibilité d'améliorer les performances (sans changer la complexité) :
  - Arrêter dès que d n'est plus modifié
  - Ne relâcher que les arcs (s_i, s_j) pour lesquels $d[s_i]$ a été modifié

Extensions de Bellman-Ford

Peut-on utiliser Bellman-Ford pour calculer un meilleur chemin ?

- Coût d'un chemin = fonction (somme, produit, ...) des coûts des arcs
- Recherche d'un chemin minimisant ou maximisant ce coût

Extensions de Bellman-Ford

Peut-on utiliser Bellman-Ford pour calculer un meilleur chemin ?

- Coût d'un chemin = fonction (somme, produit, ...) des coûts des arcs
- Recherche d'un chemin minimisant ou maximisant ce coût

Oui !

- Détection des circuits absorbants
- Il faut adapter la procédure de relâchement et l'initialisation de d

Extensions de Bellman-Ford

Peut-on utiliser Bellman-Ford pour calculer un meilleur chemin ?

- Coût d'un chemin = fonction (somme, produit, ...) des coûts des arcs
- Recherche d'un chemin minimisant ou maximisant ce coût

Oui !

- Détection des circuits absorbants
- Il faut adapter la procédure de relâchement et l'initialisation de d

Précondition commune aux 3 algorithmes que nous avons vus :

- Tout sous-chemin d'un plus court chemin doit être un plus court chemin
- Propriété non vérifiée si on ajoute des contraintes
  - Exemple : minimiser le coût tout en assurant que durée $\leq$ borne

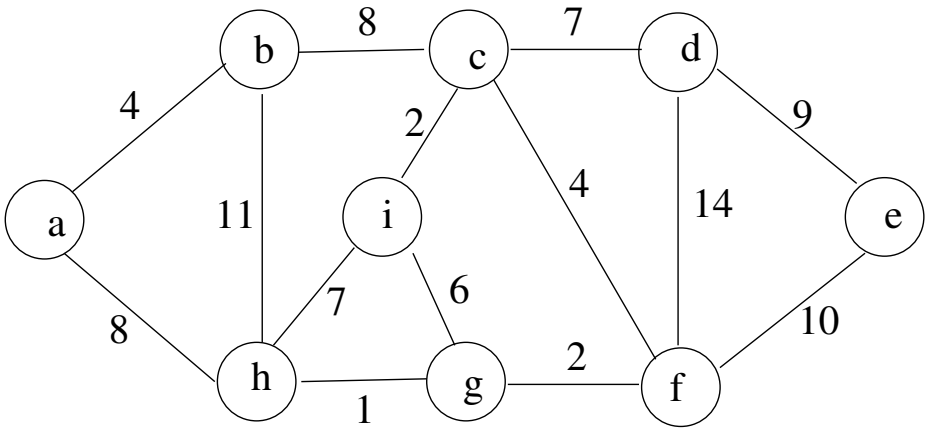
$\rightsquigarrow$ Problème NP-difficile !

Définition d'un arbre couvrant minimal (Minimum Spanning Tree) :

Graphe partiel $G' = (S, A')$ de G tel que

- G' est un arbre couvrant (G' est connexe et sans cycle)
- la somme des coûts des arêtes de A' est minimale

Exemple :



Spécification du problème MST :

1 **Fonction** *MST(g, cout)*

Entrée : Un graphe non orienté $g = (S, A)$
Une fonction $cout : A \to \mathbb{R}$

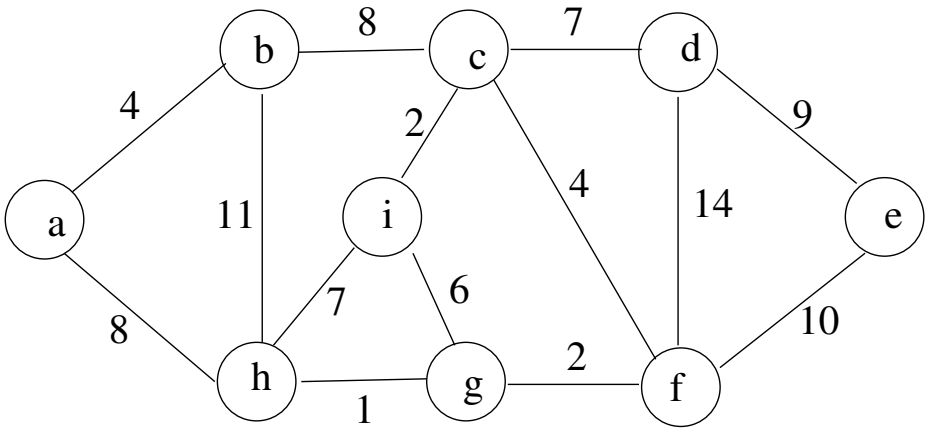
Postcondition : Retourne un ensemble d'arêtes $E \subseteq A$ tel que (S, E) est un arbre couvrant minimal de g

Définition d'un arbre couvrant minimal (Minimum Spanning Tree) :

Graphe partiel $G' = (S, A')$ de G tel que

- G' est un arbre couvrant (G' est connexe et sans cycle)
- la somme des coûts des arêtes de A' est minimale

Exemple :



Spécification du problème MST :

1 **Fonction** *MST(g, cout)*

Entrée	: Un graphe non orienté $g = (S, A)$ Une fonction $cout : A \to \mathbb{R}$
Postcondition	: Retourne un ensemble d'arêtes $E \subseteq A$ tel que (S, E) est un arbre couvrant minimal de g

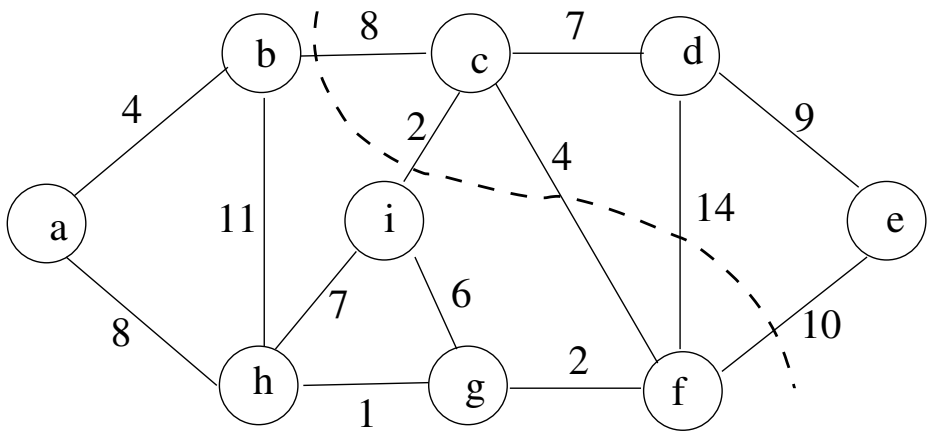
```
1 Fonction MSTgénérique(g, cout)
2     E ← ∅
3     tant que | E |<| S | −1 faire
4         Soit (P, S \ P) une coupure qui respecte E
5         Ajouter dans E l'arête de coût minimal traversant (P, S \ P)
6     retourne E
```

Définitions :

- Coupure d'un graphe $G = (S, A)$: Partition de S en 2 parties $(P, S \setminus P)$
- (s_i, s_j) traverse une coupure $(P, S \setminus P)$ si $|P \cap \{s_i, s_j\}| = 1$
- Une coupure respecte $E \subseteq A$ si aucune arête de E n'est traversée

Exemple :

$P = \{a, b, h, i, g, f\}$
$S \setminus P = \{c, d, e\}$



```
1 Fonction MSTgénérique(g, cout)
2     E ← ∅
3     tant que | E |<| S | −1 faire
4         Soit (P, S \ P) une coupure qui respecte E
5         Ajouter dans E l'arête de coût minimal traversant (P, S \ P)
6     retourne E
```

Preuve de correction

Propriété invariante à la ligne 3 : $\exists$ MST $G' = (S, E')$ tq $E \subseteq E'$

```
1 Fonction MSTgénérique(g, cout)
2     E ← ∅
3     tant que | E |<| S | −1 faire
4         Soit (P, S \ P) une coupure qui respecte E
5         Ajouter dans E l'arête de coût minimal traversant (P, S \ P)
6     retourne E
```

Preuve de correction

Propriété invariante à la ligne 3 : $\exists$ MST $G' = (S, E')$ tq $E \subseteq E'$

Comment trouver l'arête (s_i, s_j) traversant la coupure (ligne 5) ?

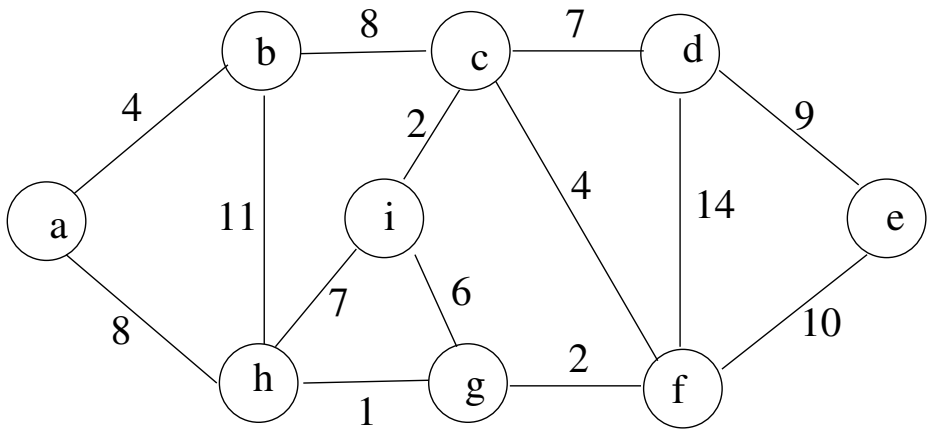
- Algorithme de Kruskal :
  - (S, E) est une forêt
  - (s_i, s_j) = arête de coût min connectant 2 arbres différents
- Algorithme de Prim :
  - E est un arbre reliant un sous-ensemble de sommets $P \subseteq S$
  - (s_i, s_j) = arête de coût min traversant la coupure $(P, S \setminus P)$

$\rightsquigarrow$ Kruskal et Prim sont des algorithmes gloutons

```
1 Fonction Kruskal(g, cout)
2     E ← ∅
3     Trier les arêtes de A par ordre de coût croissant
4     pour chaque arête (s_i, s_j) prise par ordre de coût croissant faire
5         si s_i et s_j sont dans 2 composantes connexes différentes de (S, E) alors
6             Ajouter (s_i, s_j) dans E
```

exemple :

Arêtes triées par coût croissant :
(h,g), (i,c), (g,f), (a,b), (c,f), (i,g), (h,i), (c,d), (a,h), (b,c), (d,e), (f,e), (b,h), (d,f)



```
1 Fonction Kruskal(g, cout)
2     E ← ∅
3     Trier les arêtes de A par ordre de coût croissant
4     pour chaque arête (s_i, s_j) prise par ordre de coût croissant faire
5         si s_i et s_j sont dans 2 composantes connexes différentes de (S, E) alors
6             Ajouter (s_i, s_j) dans E
```

Comment effectuer efficacement le test de la ligne 5 ?

```
1 Fonction Kruskal(g, cout)
2     E ← ∅
3     Trier les arêtes de A par ordre de coût croissant
4     pour chaque arête (s_i, s_j) prise par ordre de coût croissant faire
5         si s_i et s_j sont dans 2 composantes connexes différentes de (S, E) alors
6             Ajouter (s_i, s_j) dans E
```

Comment effectuer efficacement le test de la ligne 5 ?

Représenter les composantes connexes de (S, E) par des *disjoint sets* :

- Composantes connexes de (S, E) représentées par une forêt
  $\rightsquigarrow$ Vecteur π tq $\pi[s_i] = \mathit{null}$ si s_i est racine et $\pi[s_i]$ = père de s_i sinon
- Pour déterminer si s_i et s_j sont dans la même composante :
  $\rightsquigarrow$ Remonter jusqu'aux racines et comparer les racines
- Pour fusionner deux composantes (ligne 6) :
  $\rightsquigarrow$ Racine d'un arbre devient fils de la racine de l'autre arbre
  $\rightsquigarrow$ Opt1 : Aplatir les arbres lors de la recherche des racines
  $\rightsquigarrow$ Opt2 : Rattacher l'arbre le moins profond sous le plus profond

$\rightsquigarrow$ Voir le livre de Cormen et al pour plus de détails !

```
1 Fonction Kruskal(g, cout)
2     E ← ∅
3     Trier les arêtes de A par ordre de coût croissant
4     pour chaque arête (s_i, s_j) prise par ordre de coût croissant faire
5         si s_i et s_j sont dans 2 composantes connexes différentes de (S, E) alors
6             Ajouter (s_i, s_j) dans E
```

Complexité pour un graphe ayant n sommets et p arcs ?

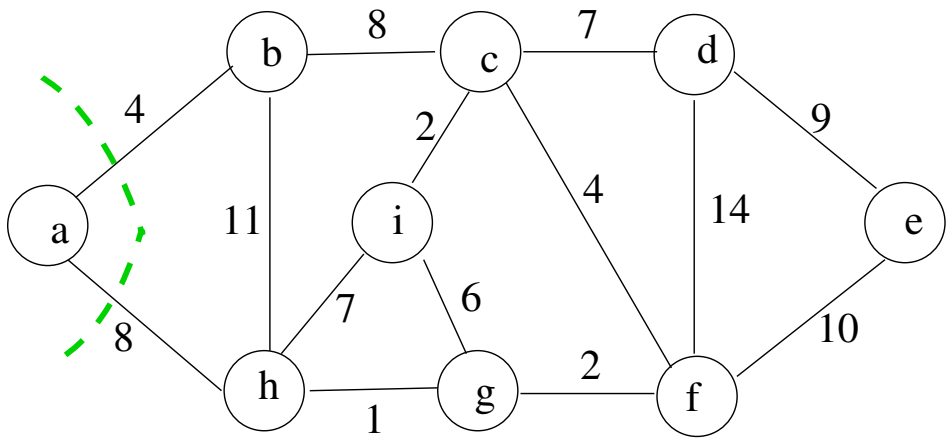
```
1 Fonction Kruskal(g, cout)
2     E ← ∅
3     Trier les arêtes de A par ordre de coût croissant
4     pour chaque arête (s_i, s_j) prise par ordre de coût croissant faire
5         si s_i et s_j sont dans 2 composantes connexes différentes de (S, E) alors
6             Ajouter (s_i, s_j) dans E
```

Complexité pour un graphe ayant n sommets et p arcs ?

$\rightsquigarrow \mathcal{O}(p \log p)$

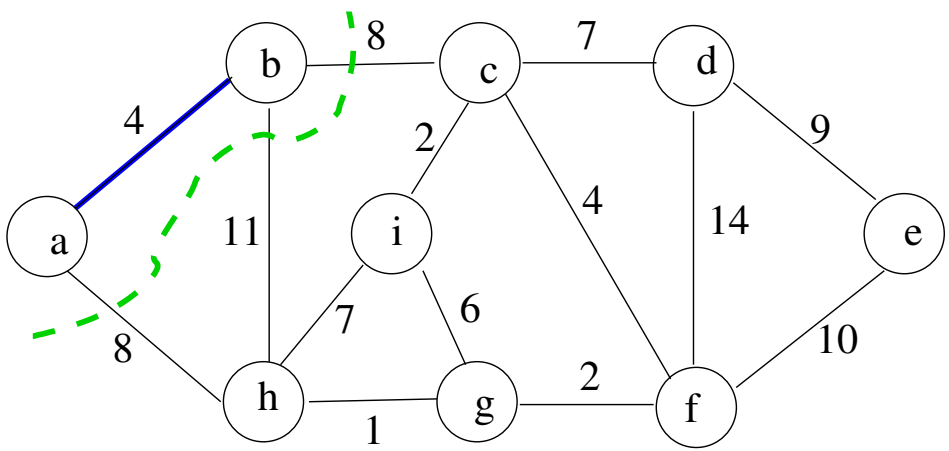
```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
7      tant que P ≠ S faire
8          Soit (s_i, s_j) l'arête min traversant (P, S \ P) avec s_i ∈ P
9          Ajouter s_j dans P et ajouter (s_i, s_j) à E
13     retourne E
```

exemple :



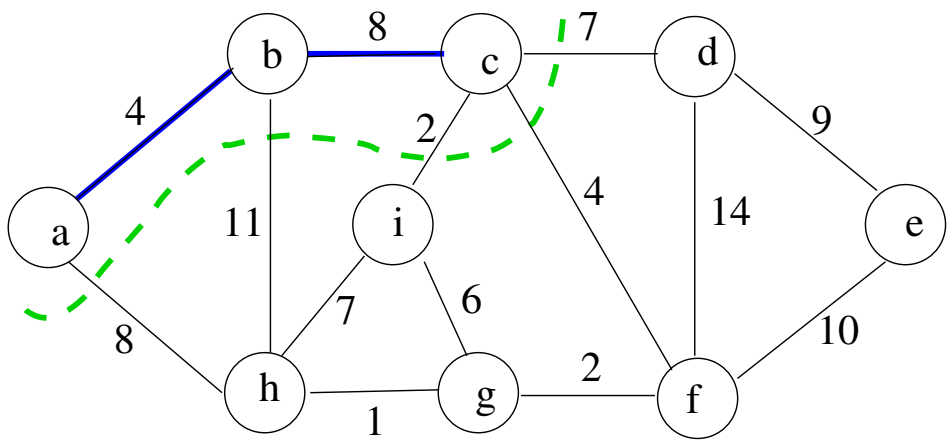
1 **Fonction** *Prim(g, cout)*
2 Soit s_0 un sommet de S choisi arbitrairement
3 $P \leftarrow \{s_0\}$; $E \leftarrow \emptyset$
7 **tant que** $P \neq S$ **faire**
8 Soit (s_i, s_j) l'arête min traversant $(P, S \setminus P)$ avec $s_i \in P$
9 Ajouter s_j dans P et ajouter (s_i, s_j) à E
13 **retourne** E

exemple :



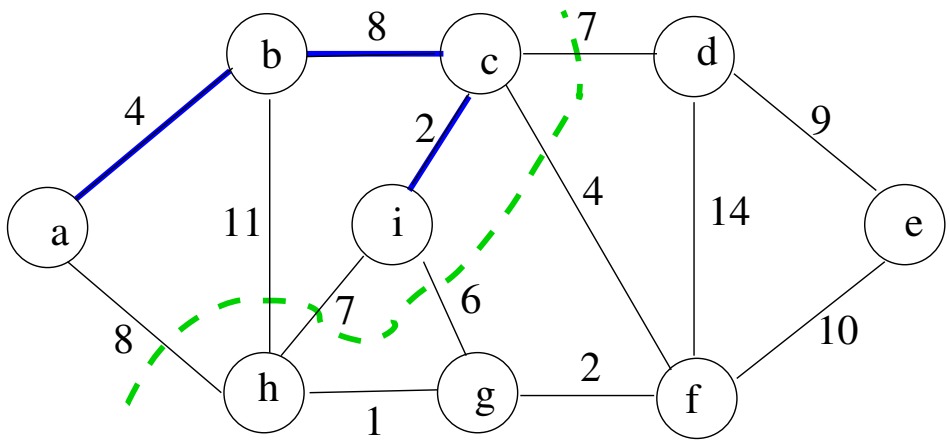
```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
7      tant que P ≠ S faire
8          Soit (s_i, s_j) l'arête min traversant (P, S \ P) avec s_i ∈ P
9          Ajouter s_j dans P et ajouter (s_i, s_j) à E
13     retourne E
```

exemple :



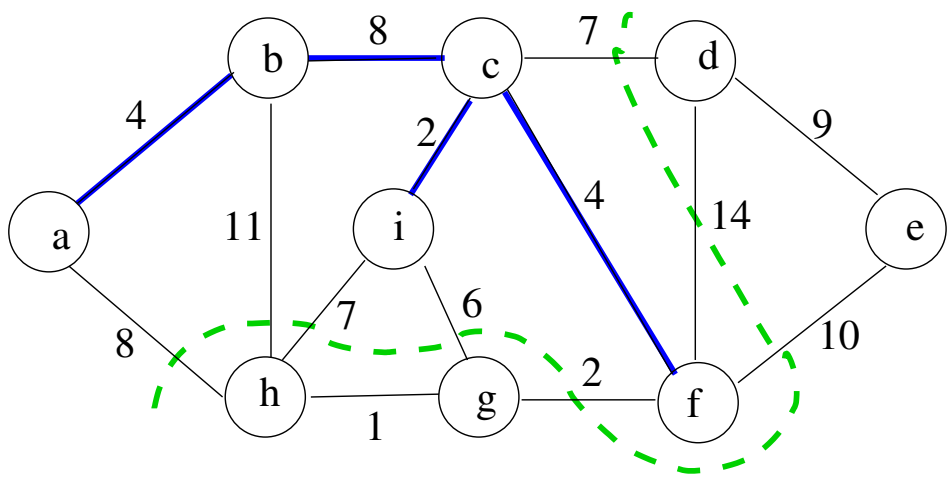
```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
7      tant que P ≠ S faire
8          Soit (s_i, s_j) l'arête min traversant (P, S \ P) avec s_i ∈ P
9          Ajouter s_j dans P et ajouter (s_i, s_j) à E
13     retourne E
```

exemple :



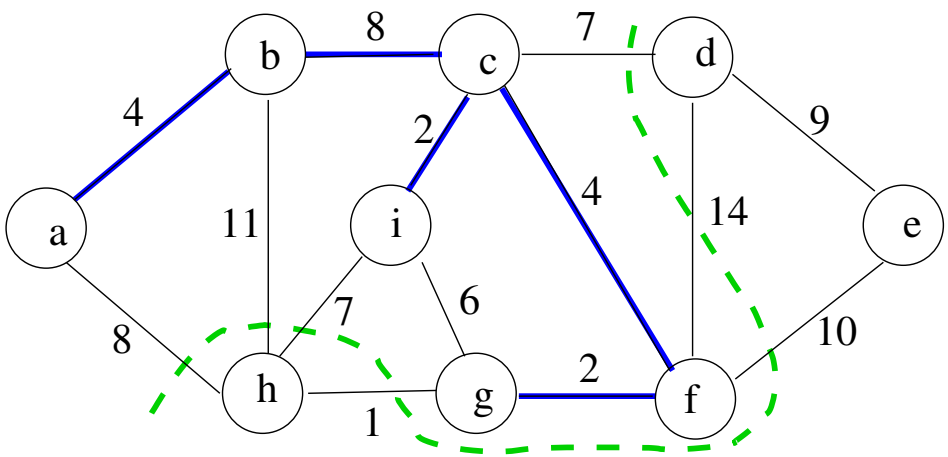
```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
7      tant que P ≠ S faire
8          Soit (s_i, s_j) l'arête min traversant (P, S \ P) avec s_i ∈ P
9          Ajouter s_j dans P et ajouter (s_i, s_j) à E
13     retourne E
```

exemple :



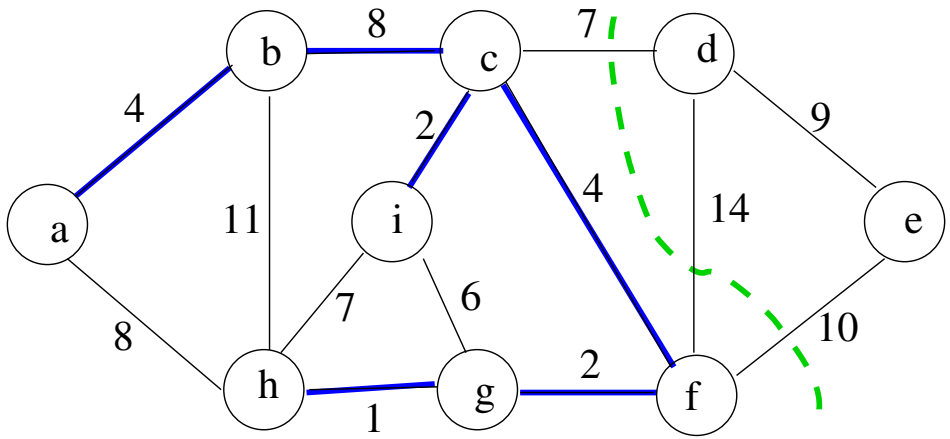
```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
7      tant que P ≠ S faire
8          Soit (s_i, s_j) l'arête min traversant (P, S \ P) avec s_i ∈ P
9          Ajouter s_j dans P et ajouter (s_i, s_j) à E
13     retourne E
```

exemple :



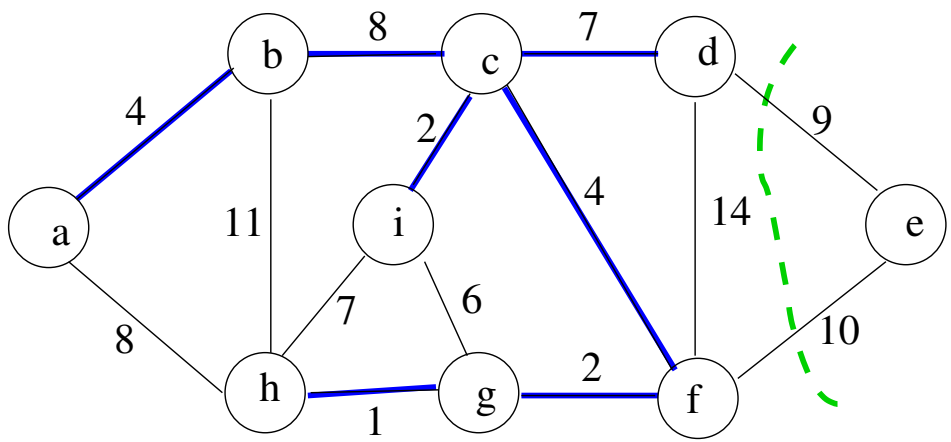
```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
7      tant que P ≠ S faire
8          Soit (s_i, s_j) l'arête min traversant (P, S \ P) avec s_i ∈ P
9          Ajouter s_j dans P et ajouter (s_i, s_j) à E
13     retourne E
```

exemple :



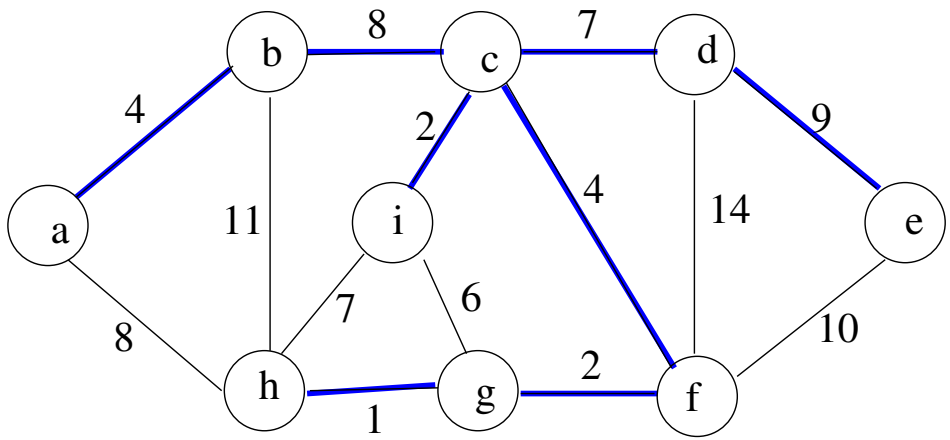
```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
7      tant que P ≠ S faire
8          Soit (s_i, s_j) l'arête min traversant (P, S \ P) avec s_i ∈ P
9          Ajouter s_j dans P et ajouter (s_i, s_j) à E
13     retourne E
```

exemple :



```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
7      tant que P ≠ S faire
8          Soit (s_i, s_j) l'arête min traversant (P, S \ P) avec s_i ∈ P
9          Ajouter s_j dans P et ajouter (s_i, s_j) à E
13     retourne E
```

exemple :



```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
7      tant que P ≠ S faire
8          Soit (s_i, s_j) l'arête min traversant (P, S \ P) avec s_i ∈ P
9          Ajouter s_j dans P et ajouter (s_i, s_j) à E
13     retourne E
```

Comment choisir efficacement (s_i, s_j) ligne 8 ?

```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
7      tant que P ≠ S faire
8          Soit (s_i, s_j) l'arête min traversant (P, S \ P) avec s_i ∈ P
9          Ajouter s_j dans P et ajouter (s_i, s_j) à E
13     retourne E
```

Comment choisir efficacement (s_i, s_j) ligne 8 ?

Maintenir les tableaux π et c tels que :

- $\forall s_i \in P$:
  $(s_i, \pi[s_i])$ = plus petite arête partant de s_i et traversant $(P, S \setminus P)$
- $c[s_i] = cout(s_i, \pi[s_i])$

```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
4      pour chaque sommet s_i ∈ S faire
5          si s_i ∈ adj(s_0) alors π[s_i] ← s_0 ; c[s_i] ← cout(s_0, s_i) ;
6          sinon π[s_i] ← null ; c[s_i] ← ∞ ;
7      tant que P ≠ S faire
8          Ajouter dans P le sommet s_i ∈ S \ P ayant la plus petite valeur de c
9          Ajouter (s_i, π[s_i]) à E
10         pour chaque sommet s_j ∈ adj(s_i) faire
11             si s_j ∉ P et cout(s_i, s_j) < c[s_j] alors
12                 π[s_j] ← s_i ; c[s_j] ← cout(s_i, s_j)
13     retourne E
```

Comment choisir efficacement (s_i, s_j) ligne 8 ?

Maintenir les tableaux π et c tels que :

- $\forall s_i \in P$:
  $(s_i, \pi[s_i])$ = plus petite arête partant de s_i et traversant $(P, S \setminus P)$
- $c[s_i] = cout(s_i, \pi[s_i])$

```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
4      pour chaque sommet s_i ∈ S faire
5          si s_i ∈ adj(s_0) alors π[s_i] ← s_0 ; c[s_i] ← cout(s_0, s_i) ;
6          sinon π[s_i] ← null ; c[s_i] ← ∞ ;
7      tant que P ≠ S faire
8          Ajouter dans P le sommet s_i ∈ S \ P ayant la plus petite valeur de c
9          Ajouter (s_i, π[s_i]) à E
10         pour chaque sommet s_j ∈ adj(s_i) faire
11             si s_j ∉ P et cout(s_i, s_j) < c[s_j] alors
12                 π[s_j] ← s_i ; c[s_j] ← cout(s_i, s_j)
13     retourne E
```

Comment chercher efficacement s_i ligne 8 ?

```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
4      pour chaque sommet s_i ∈ S faire
5          si s_i ∈ adj(s_0) alors π[s_i] ← s_0 ; c[s_i] ← cout(s_0, s_i) ;
6          sinon π[s_i] ← null ; c[s_i] ← ∞ ;
7      tant que P ≠ S faire
8          Ajouter dans P le sommet s_i ∈ S \ P ayant la plus petite valeur de c
9          Ajouter (s_i, π[s_i]) à E
10         pour chaque sommet s_j ∈ adj(s_i) faire
11             si s_j ∉ P et cout(s_i, s_j) < c[s_j] alors
12                 π[s_j] ← s_i ; c[s_j] ← cout(s_i, s_j)
13     retourne E
```

Comment chercher efficacement s_i ligne 8 ?

$\rightsquigarrow$ File de priorité implémentée par un tas binaire

```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
4      pour chaque sommet s_i ∈ S faire
5          si s_i ∈ adj(s_0) alors π[s_i] ← s_0 ; c[s_i] ← cout(s_0, s_i) ;
6          sinon π[s_i] ← null ; c[s_i] ← ∞ ;
7      tant que P ≠ S faire
8          Ajouter dans P le sommet s_i ∈ S \ P ayant la plus petite valeur de c
9          Ajouter (s_i, π[s_i]) à E
10         pour chaque sommet s_j ∈ adj(s_i) faire
11             si s_j ∉ P et cout(s_i, s_j) < c[s_j] alors
12                 π[s_j] ← s_i ; c[s_j] ← cout(s_i, s_j)
13     retourne E
```

Comment chercher efficacement s_i ligne 8 ?

$\rightsquigarrow$ File de priorité implémentée par un tas binaire

Complexité pour un graphe ayant n sommets et p arêtes ?

```
1  Fonction Prim(g, cout)
2      Soit s_0 un sommet de S choisi arbitrairement
3      P ← {s_0} ; E ← ∅
4      pour chaque sommet s_i ∈ S faire
5          si s_i ∈ adj(s_0) alors π[s_i] ← s_0 ; c[s_i] ← cout(s_0, s_i) ;
6          sinon π[s_i] ← null ; c[s_i] ← ∞ ;
7      tant que P ≠ S faire
8          Ajouter dans P le sommet s_i ∈ S \ P ayant la plus petite valeur de c
9          Ajouter (s_i, π[s_i]) à E
10         pour chaque sommet s_j ∈ adj(s_i) faire
11             si s_j ∉ P et cout(s_i, s_j) < c[s_j] alors
12                 π[s_j] ← s_i ; c[s_j] ← cout(s_i, s_j)
13     retourne E
```

Comment chercher efficacement s_i ligne 8 ?

$\rightsquigarrow$ File de priorité implémentée par un tas binaire

Complexité pour un graphe ayant n sommets et p arêtes ?

$\rightsquigarrow \mathcal{O}(p \log n)$

Problèmes, instances et algorithmes (rappels)

Spécification d'un problème :

- Paramètres en entrée et en sortie
- Eventuellement : Préconditions sur les paramètres en entrée
- Postrelation entre les valeurs des paramètres en entrée et en sortie

Instance d'un problème :

Valuation des paramètres en entrée satisfaisant les préconditions

Algorithme pour un problème P :

Séquence d'instructions élémentaires permettant de calculer les valeurs des paramètres en sortie à partir des valeurs des paramètres en entrée, pour toute instance de P

Exemple 1 : Recherche d'un élément dans un tableau trié

Spécification du problème :

- Entrées :
  - un tableau *tab* comportant n entiers indicés de 0 à $n-1$
  - une valeur entière e
- Sortie : un entier i
- Précondition : les éléments de *tab* sont triés par ordre croissant
- Postrelation :
  - si $\forall j \in [0, n-1], tab[j] \neq e$ alors $i = n$
  - sinon $i \in [0, n-1]$ et $tab[i] = e$

Exemples d'instances :

- Entrées : $e = 8$ et $tab =$

4	4	7	8	10	11	12

  $\leadsto$ Sortie : $i = 3$

- Entrées : $e = 9$ et $tab =$

4	4	7	8	10	11	12

  $\leadsto$ Sortie : $i = 7$

Exemple 2 : Satisfiabilité d'une formule booléenne (SAT)

Spécification du problème :

- Entrées : une formule booléenne F définie sur un ens. X de n variables
- Sortie : une valuation $V : X \to \{vrai, faux\}$ des variables de F
- Précondition : F est sous forme normale conjonctive (CNF)
- Postrelation : Si F est satisfiable alors V satisfait F

Exemple d'instance

- Entrées : $X = \{a, b, c, d, e\}$,
  $F = (a \vee \neg b) \wedge (\neg a \vee c \vee \neg d) \wedge (\neg b \vee \neg c \vee \neg e) \wedge (a \vee b \vee d \vee e)$
  $\rightsquigarrow$ Sortie : $V = \{a = vrai, b = faux, c = vrai, d = vrai, e = faux\}$

Complexité d'un algorithme (rappel)

Estimation des ressources nécessaires à l'exécution d'un algorithme :

- Temps = estimation du nombre d'instructions élémentaires
- Espace = estimation de l'espace mémoire utilisé

$\rightsquigarrow$ Estimation dépendante de la taille n des paramètres en entrée

Ordre de grandeur d'une fonction $f(n)$:

$\mathcal{O}(g(n)) \rightsquigarrow \exists c, n_0$ tel que $\forall n > n_0, f(n) < c.g(n)$

- $\mathcal{O}(log_k(n))$: logarithmique
- $\mathcal{O}(n)$: linéaire
- $\mathcal{O}(n^k)$: polynomial
- $\mathcal{O}(k^n)$: exponentiel

Complexité des problèmes de décision

Problèmes de décision :

La sortie et la postrelation sont remplacées par une question binaire sur les paramètres en entrée ($\rightsquigarrow$ Réponse $\in \{vrai, faux\}$)

Exemple : Description du problème de décision *Recherche*

- Entrées = un tableau *tab* contenant n entiers et un entier e
- Question = Existe-t'il un élément de *tab* qui soit égal à e ?

Complexité d'un problème X :

- Complexité du meilleur algo (pas nécessairement connu) résolvant X :
  - Chaque algorithme résolvant X fournit une borne supérieure
  - On peut trouver des bornes inférieures en analysant le problème
- Si plus grande borne inférieure = plus petite borne supérieure
  Alors la complexité de X est connue ; Sinon la complexité est ouverte...

La classe $\mathcal{P}$

Appartenance d'un problème de décision X à la classe $\mathcal{P}$:

- $X \in \mathcal{P}$ s'il existe un algorithme Polynomial pour résoudre X
  $\rightsquigarrow$ Complexité en $\mathcal{O}(n^k)$ avec
  - n = taille des données en entrée de l'instance
  - k = constante indépendante de l'instance
- $\mathcal{P}$ est la classe des problèmes traitables en un temps raisonnable
  $\rightsquigarrow$ *Tractable problems*

Exemples de problèmes de décision appartenant à $\mathcal{P}$:

- Déterminer si un entier appartient à un tableau (trié ou pas)
- Déterminer s'il existe un chemin entre 2 sommets d'un graphe
- Déterminer s'il existe un arbre couvrant de coût borné dans un graphe
- ...
- Déterminer si un nombre est premier
  $\rightsquigarrow$ Prime is in $\mathcal{P}$ [Agrawal - Kayal - Saxena 2002] !

La classe $\mathcal{NP}$

Appartenance d'un problème de décision X à la classe $\mathcal{NP}$:

- $X \in \mathcal{NP}$ s'il existe un algorithme Polynomial pour une machine de Turing Non déterministe
- Autrement dit : $X \in \mathcal{NP}$ si pour toute instance I de X telle que réponse(I) = vrai, il existe un certificat $c(I)$ permettant de vérifier en temps polynomial que réponse(I) = vrai

  $\rightsquigarrow$ Il est facile de vérifier qu'une solution de X est correcte

Exemple : SAT $\in \mathcal{NP}$

- Description du problème SAT (rappel) :
  - Entrées = une formule booléenne F portant sur un ensemble X de n variables booléennes
  - Question = Existe-t'il une valuation des var. de X qui satisfait F ?
- Certificat : une valuation $V : X \rightarrow \{vrai, faux\}$ qui satisfait F

Relation entre $\mathcal{P}$ et $\mathcal{NP}$:

- $\mathcal{P} \subseteq \mathcal{NP}$
- Conjecture : $\mathcal{P} \neq \mathcal{NP}$

  1 million de dollars à gagner (cf www.claymath.org/millennium-problems)

Problèmes $\mathcal{NP}$-complets :

- Les problèmes les plus difficiles de la classe $\mathcal{NP}$:
  $\rightsquigarrow$ X est $\mathcal{NP}$-complet si ($X \in \mathcal{NP}$) et ($X \in \mathcal{P} \Rightarrow \mathcal{P}$=$\mathcal{NP}$)
- Théorème de [Cook 1971] : SAT est $\mathcal{NP}$-complet
- Depuis 1971, des centaines de problèmes montrés $\mathcal{NP}$-complets
  $\rightsquigarrow$ Voir par exemple [Garey et Johnson 1979]

Démonstration de $\mathcal{NP}$-complétude :

- Montrer que le problème X appartient à $\mathcal{NP}$
- Trouver une réduction polynomiale pour transformer un problème $\mathcal{NP}$-complet connu en X

Exercice

Description du problème Clique :

- Entrées : un graphe $G = (V, E)$ et un entier positif k
- Question : Existe-t'il $S \subseteq V$ tel que $|S| = k$ et $\forall i, j \in S, i \neq j \Rightarrow (i, j) \in E$

Montrer que Clique $\in \mathcal{NP}$:

$\rightsquigarrow$ Certificat ?

Montrer que Clique est $\mathcal{NP}$-complet :

$\rightsquigarrow$ Réduction de SAT :

- Donner un algorithme polynomial permettant de transformer n'importe quelle instance I_1 de SAT en une instance I_2 de Clique
- Montrer que réponse(I_1) = réponse(I_2)

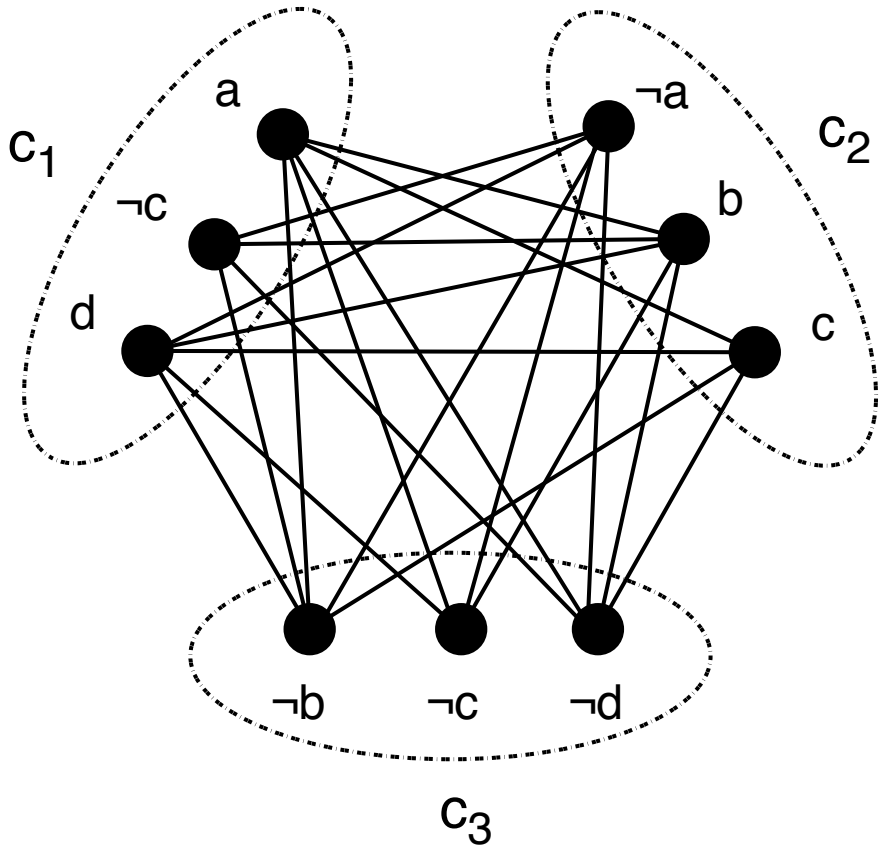
Solution

Graphe non orienté $G = (S, A)$ associé à une formule F :

- S associe un sommet à chaque littéral de chaque clause de F
  $\rightsquigarrow$ $c(u)$ et $l(u)$ = clause et littéral correspondant au sommet u
- $A = \{(u, v) \in S \times S \mid c(u) \neq c(v) \text{ et } l(u) \neq \neg l(v)\}$

Exemple :

Formule F :
$(a \vee \neg c \vee d) \wedge$
$(\neg a \vee b \vee c) \wedge$
$(\neg b \vee \neg c \vee \neg d)$



Problèmes $\mathcal{NP}$-difficiles

Problèmes au moins aussi difficiles que ceux de $\mathcal{NP}$:

- X est $\mathcal{NP}$-difficile si : $X \in \mathcal{P} \Rightarrow \mathcal{P} = \mathcal{NP}$
  $\rightsquigarrow$ Vérifier qu'une solution de X est correcte peut être un pb difficile
- $\mathcal{NP}$-complet $\subset \mathcal{NP}$-difficile

Exemple : Problème de la clique exacte

- Entrées : un graphe $G = (V, E)$ et un entier positif k
- Question : La plus grande clique de G est-elle de taille k ?

Ce problème appartient-il à $\mathcal{NP}$?

Complexité des problèmes d'optimisation :

- Déterminée en fonction du problème de décision associé
- $\mathcal{NP}$-difficile si le problème de décision est $\mathcal{NP}$-complet

Problèmes indécidables

$\rightsquigarrow$ **Problèmes pour lesquels il n'existe pas d'algo pour une machine de Turing**

Exemple 1 : Problème de l'arrêt

- Entrée : Un programme P (une suite de caractères)
- Question : Est-ce que l'exécution de P se termine en un temp fini ?

Exemple 2 : Problème de Post

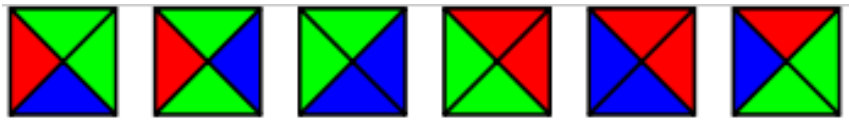
- Entrée : 2 listes finies $\alpha_1, \alpha_2, \ldots, \alpha_n$ et $\beta_1, \beta_2, \ldots, \beta_n$ de mots
- Question : $\exists k$ indices $i_1, i_2, \ldots, i_k$ tq $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_n} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_n}$?
- Exemple d'instance : Entrée =

α_1	α_2	α_3
a	ab	bba

β_1	β_2	β_3
baa	aa	bb

Exemple 3 : Problème de pavage

- Entrée : Un ensemble fini S de carrés aux arêtes coloriées
- Question : Peut-on paver n'importe quel cadre $n \times n$ avec des copies de carrés de S de sorte que 2 arêtes adjacentes soient de même couleur ?
- Exemple d'instance : Entrée =



Enumération de toutes les cliques d'un graphe

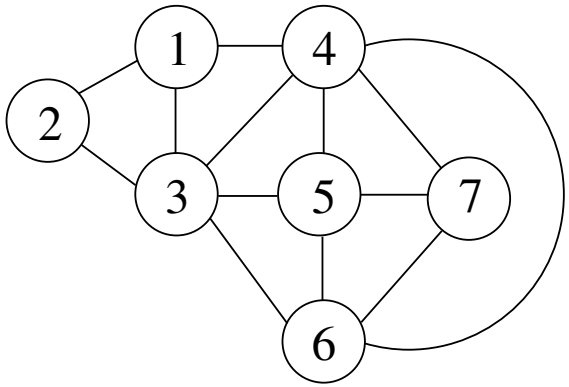
1 **Fonction** *enumClique*(g, c)
  Entrée : Un graphe $g = (S, A)$ et un ens. de sommets $c \subseteq S$
  Précondition : c est une clique de g
  Postcondition : Retourne l'ensemble des cliques de g qui sont des sur-ens. de c
2 $S \leftarrow \{c\}$
3 $cand \leftarrow \{s_i \in S \mid \forall s_j \in c, (s_i, s_j) \in A \text{ et } s_i > s_j\}$
4 **pour** *chaque sommet* $s_i \in cand$ **faire**
5 $S \leftarrow S \cup \text{enumClique}(g, c \cup \{s_i\}, k)$
6 **retourne** S

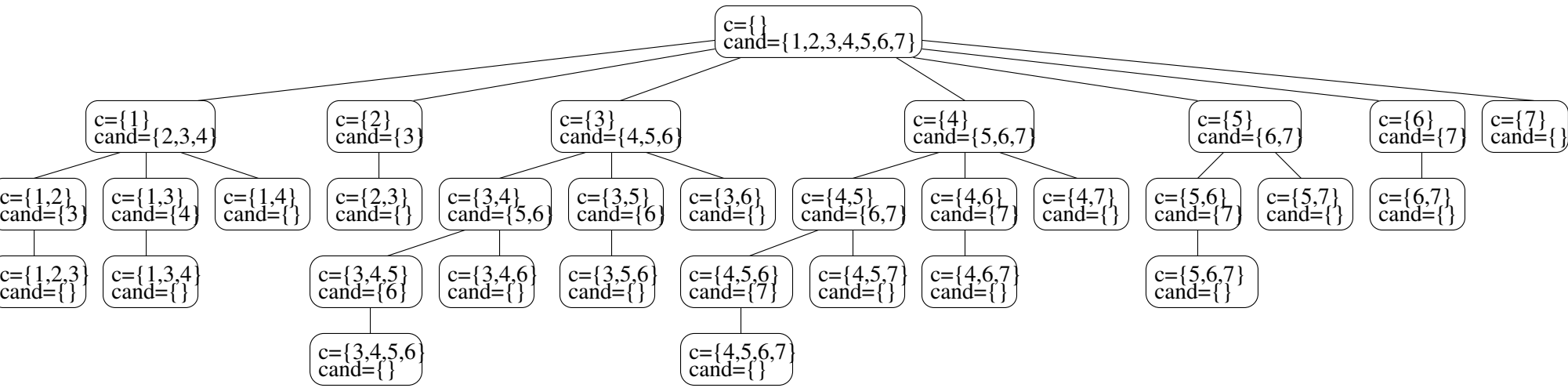
Arbre de recherche associé à une exécution de *enumClique* :

- Chaque nœud de l'arbre = 1 clique
- Racine = clique vide
- c est le père de c' si *enumClique*(g, c) appelle *enumClique*(g, c')

Exploration de l'arbre en profondeur d'abord (retour-arrière chronologique)

Exemple d'arbre de recherche :





1 **Fonction** *enumClique*(g, c)
  Entrée : Un graphe $g = (S, A)$ et un ens. de sommets $c \subseteq S$
  Précondition : c est une clique de g
  Postcondition : Retourne l'ensemble des cliques de g qui sont des sur-ens. de c
2 $S \leftarrow \{c\}$
3 $cand \leftarrow \{s_i \in S \mid \forall s_j \in c, (s_i, s_j) \in A \text{ et } s_i > s_j\}$
4 **pour** *chaque sommet* $s_i \in cand$ **faire**
5 $S \leftarrow S \cup \text{enumClique}(g, c \cup \{s_i\}, k)$
6 **retourne** S

Complexité de *enumClique* si $|S| = n$ et si g contient k cliques :

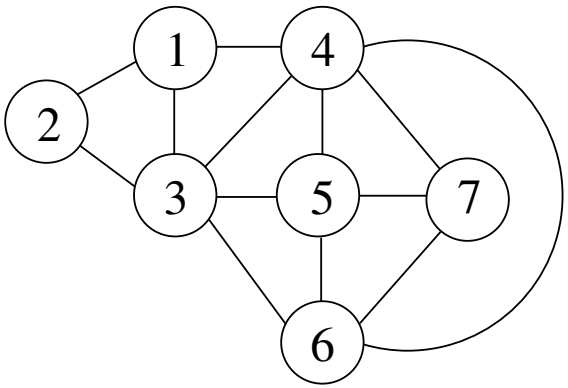
- Nombre d'appels à *enumclique* = k :
  - A chaque appel, le paramètre c en entrée est une clique
  - Si c' est une clique de g alors il y aura exactement 1 appel à *enumClique* pour lequel $c = c'$
- A chaque appel, construction de *cand* (ou maintien incrémental)

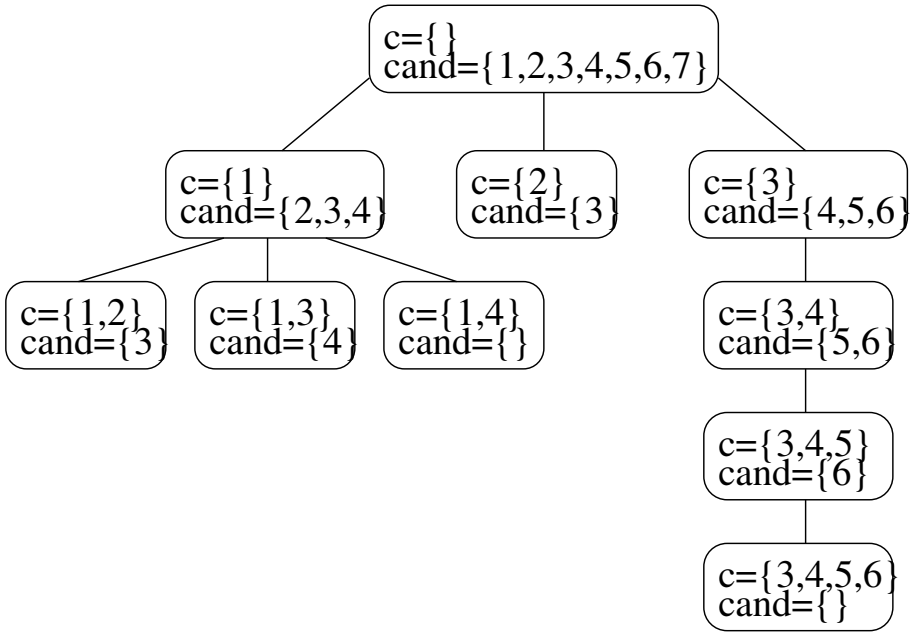
$\rightsquigarrow$ Complexité = $\mathcal{O}(nk)$ (*incremental polynomial time*)

Recherche d'une clique d'ordre k

1 **Fonction** *chercheClique*(g, c, k)
  Entrée : graphe $g = (S, A)$, $c \subseteq S$ et entier k
  Précondition : c est une clique de taille inférieure ou égale à k
  Postcondition : Retourne vrai si $\exists$ une clique c' de g tq $|c'| = k$ et $c \subseteq c'$; faux sinon
2 **si** $|c| = k$ **alors retourne** *vrai*;
3 $cand \leftarrow \{s_i \in S \mid \forall s_j \in c, (s_i, s_j) \in A \wedge s_i > s_j\}$
4 **si** $|c| + |cand| < k$ **alors retourne** *faux*;
5 **pour** *chaque sommet* $s \in cand$ **faire**
6 **si** *chercheClique*$(g, c \cup \{s\}, k)$ **alors retourne** *vrai*;
7 **retourne** *faux*

Arbre de recherche pour $k = 4$:

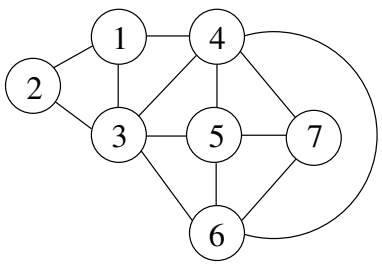


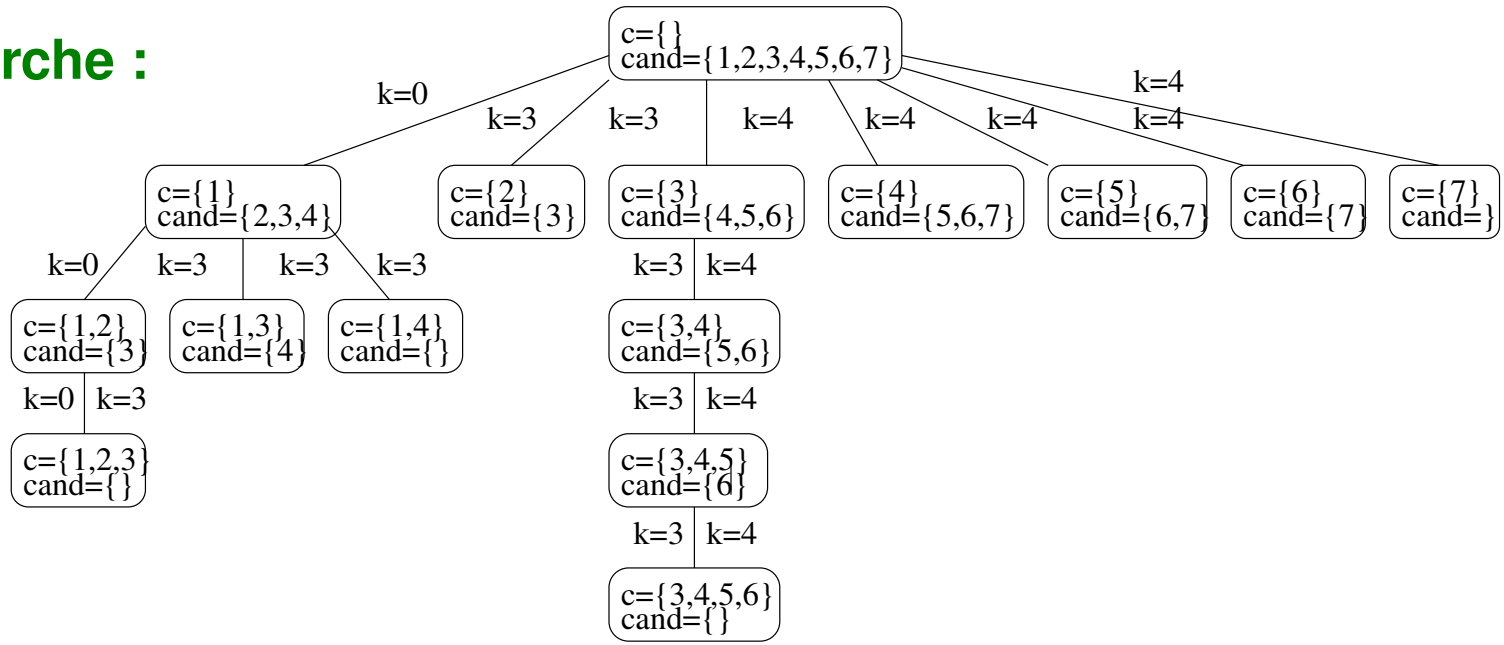


Recherche d'une clique maximum

1 **Fonction** *chercheCliqueMax*(g, c, k)
  Entrée : graphe $g = (S, A)$, $c \subseteq S$ et entier k
  Précondition : c est une clique
  Postcondition : ret. $\max(k, k')$ où k' = taille de la + grande clique de g contenant c
2 $cand \leftarrow \{s_i \in S \mid \forall s_j \in c, (s_i, s_j) \in A \wedge s_i > s_j\}$
3 **si** $cand = \emptyset$ **alors retourne** $max(|c|, k)$;
4 **pour** *chaque sommet* $s \in cand$ **et tant que** $|c| + |cand| > k$ **faire**
5 $k \leftarrow$chercheCliqueMax$(g, c \cup \{s\}, k)$
6 **retourne** k

Arbre de recherche :

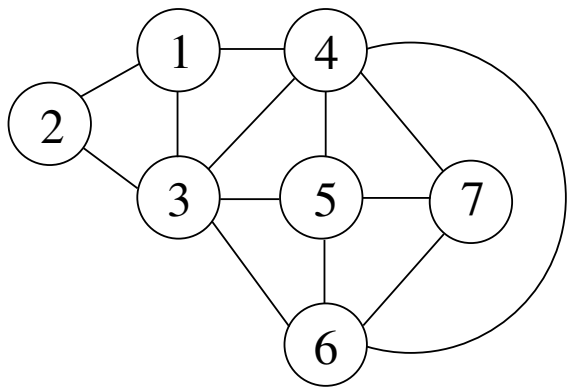




Construction gloutonne d'une clique

```
1 Fonction chercheCliqueGlouton(g)
      Entrée          : Un graphe g = (S, A)
      Postcondition   : retourne une clique de g
2     cand ← S
3     c ← ∅
4     tant que cand ≠ ∅ faire
5         Soit s_i le sommet de cand maximisant |cand ∩ adj(s_i)|
6         c ← c ∪ {s_i}
7         cand ← cand ∩ adj(s_i)
8     retourne c
```

Exercice :



Construction gloutonne d'une clique

```
1 Fonction chercheCliqueGlouton(g)
      Entrée          : Un graphe g = (S, A)
      Postcondition   : retourne une clique de g
2     cand ← S
3     c ← ∅
4     tant que cand ≠ ∅ faire
5         Soit s_i le sommet de cand maximisant |cand ∩ adj(s_i)|
6         c ← c ∪ {s_i}
7         cand ← cand ∩ adj(s_i)
8     retourne c
```

Complexité ?

Coloriage de graphes

Définitions pour un graphe non orienté $G = (S, A)$:

- Coloriage de $G =$ fonction $c : S \to \mathbb{N}$ tq $\forall (s_i, s_j) \in A, c(s_i) \neq c(s_j)$
- Nombre chromatique de $G = \chi(G) = min_c(max_{s_i \in S}(c(s_i)))$

Complexité :

- Décider si $\chi(G)$ est inférieur à une borne k donnée est $\mathcal{NP}$-complet
- Déterminer $\chi(G)$ est $\mathcal{NP}$-difficile

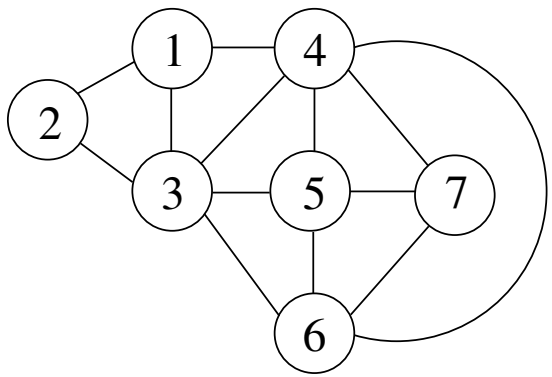
Relation entre coloriage et cliques :

Pour tout graphe G, $\chi(G) \geq$ nombre de sommets de la clique maximum de G

Algorithme glouton de Brélaz (DSATUR)

```
1  Fonction brélaz(g)
       Postcondition : retourne une borne supérieure de χ(g)
2      borne_χ ← 0
3      tant que tous les sommets ne sont pas coloriés faire
4          Choisir un sommet s_i non colorié tel que :
5          - s_i = sommet ayant le plus de voisins coloriés avec des valeurs différentes
6          - en cas d'ex æquo, s_i = sommet ayant le plus de voisins non coloriés
7          si ∀k ∈ [1, borne_χ], ∃s_j ∈ adj(s_i) tel que s_j est colorié avec k alors
8              borne_χ ← borne_χ + 1
9          k ← plus petite couleur ∈ [1, borne_χ] non prise par un voisin de s_i
10         Colorier s_i avec k
11     retourne borne_χ
```

Exercice :



Algorithme glouton de Brélaz (DSATUR)

```
1  Fonction brélaz(g)
       Postcondition : retourne une borne supérieure de χ(g)
2      borne_χ ← 0
3      tant que tous les sommets ne sont pas coloriés faire
4          Choisir un sommet s_i non colorié tel que :
5          - s_i = sommet ayant le plus de voisins coloriés avec des valeurs différentes
6          - en cas d'ex æquo, s_i = sommet ayant le plus de voisins non coloriés
7          si ∀k ∈ [1, borne_χ], ∃s_j ∈ adj(s_i) tel que s_j est colorié avec k alors
8              borne_χ ← borne_χ + 1
9          k ← plus petite couleur ∈ [1, borne_χ] non prise par un voisin de s_i
10         Colorier s_i avec k
11     retourne borne_χ
```

Complexité ?

Au delà des cliques et du coloriage

Il existe bien d'autres problèmes $\mathcal{NP}$-difficiles

- Recherche de circuits hamiltoniens, Voyageur de commerce
- Recherche de plus courts chemins sous contraintes
- Partitionnement de graphes, Coupure minimale sous contraintes
- ...

Ces problèmes sont rencontrés dans de très nombreuses applications

- Optimisation de tournées de livraison
- Recherche du chemin le plus rapide comportant moins de k changements dans un réseau de transports en commun
- Segmentation d'images
- ...

Besoin de concevoir des algorithmes qui passent à l'échelle !

Recherche opérationnelle, Programmation par contraintes, Méta-heuristiques, ...