

Algorithmique Avancée pour l'Intelligence Artificielle et les graphes (AAIA)

Christine Solnon

INSA de Lyon - 3IF

2025

1

Introduction

- Organisation et objectifs pédagogiques
- Modélisation de problèmes avec des graphes
- Définitions

2

Structures de données pour représenter un graphe

3

Arbres Couvrants Minimaux

4

Parcours de graphes

5

Plus courts chemins

6

Problèmes de planification

7

Problèmes NP-difficiles

8

Conclusion

Positionnement de l'EC AAIA au sein des UE d'IF

Unités d'Enseignement (UE) du département IF :

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

~ Chacune de ces UE est composée d'Éléments Constitutifs (EC)

EC de l'UE DL en 3IF :

- Introduction à l'algo
- Bases de la POO
- POO avancée
- Génie logiciel

EC de l'UE MOM en 3IF :

- Calcul matriciel et synthèse d'images
- Traitement du signal et image
- Probabilités
- **AAIA**

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

~ Implémenter de bons logiciels

Nouvelles compétences :

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

Organisation

9 cours en amphi

- du 10 février au 7 avril

4 TD de 2h et 4 TP de 4h

- du 17 février au 16 mai

Evaluation

- 1 DS (12 juin)
- Questionnaires Moodle (sur les cours et sur les TP)
~ **Attention : les 2 premiers questionnaires ferment dimanche soir !**

1

Introduction

- Organisation et objectifs pédagogiques
- Modélisation de problèmes avec des graphes
- Définitions

2

Structures de données pour représenter un graphe

3

Arbres Couvrants Minimaux

4

Parcours de graphes

5

Plus courts chemins

6

Problèmes de planification

7

Problèmes NP-difficiles

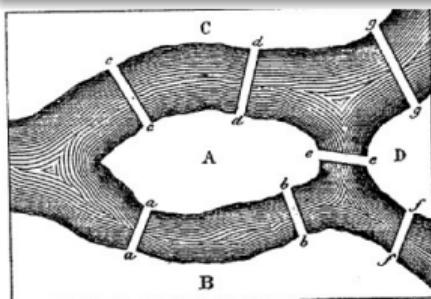
8

Conclusion

Euler 1741 : *Solutio problematis at geometriam situs pertinentis*

où comment résoudre un problème grâce à la “ géométrie de situation ”

“ Outre cette partie de la géométrie qui s’occupe de la grandeur et de la mesure (...), Leibniz a fait mention, pour la première fois, d’une autre partie encore très inconnue actuellement, qu’il a appelée *Geometria Situs* (...). Cette branche s’occupe uniquement de l’ordre et de la situation, indépendamment des rapports de grandeur. ”



Problème :

“ Peut-on arranger son parcours de telle sorte que l’on passe sur chaque pont, et que l’on ne puisse y passer qu’une seule fois ? Cela semble possible, disent les uns ; impossible, disent les autres ; cependant personne n’a la certitude de son sentiment. ”

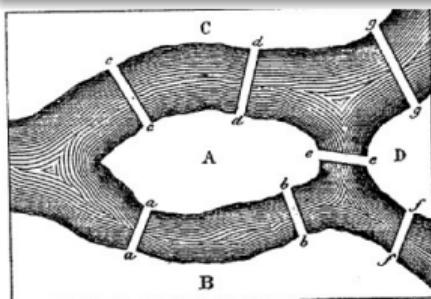
Proposition d'Euler pour résoudre ce problème :

“ Former avec les lettres A, B, C, D une série de 8 lettres dans laquelle ces voisinages (A-C, A-B, A-D, D-C et B-D) apparaissent autant de fois qu'il a été indiqué (2, 2, 1, 1 et 1 fois) ; mais avant de chercher à effectuer une telle disposition, il est bon de se demander si celle-ci est réalisable. (...) Aussi ai-je trouvé une règle qui donne, pour tous les cas, la condition indispensable pour que le problème ne soit pas impossible. ”

Euler 1741 : *Solutio problematis at geometriam situs pertinentis*

où comment résoudre un problème grâce à la “ géométrie de situation ”

“ Outre cette partie de la géométrie qui s’occupe de la grandeur et de la mesure (...), Leibniz a fait mention, pour la première fois, d’une autre partie encore très inconnue actuellement, qu’il a appelée *Geometria Situs* (...). Cette branche s’occupe uniquement de l’ordre et de la situation, indépendamment des rapports de grandeur. ”

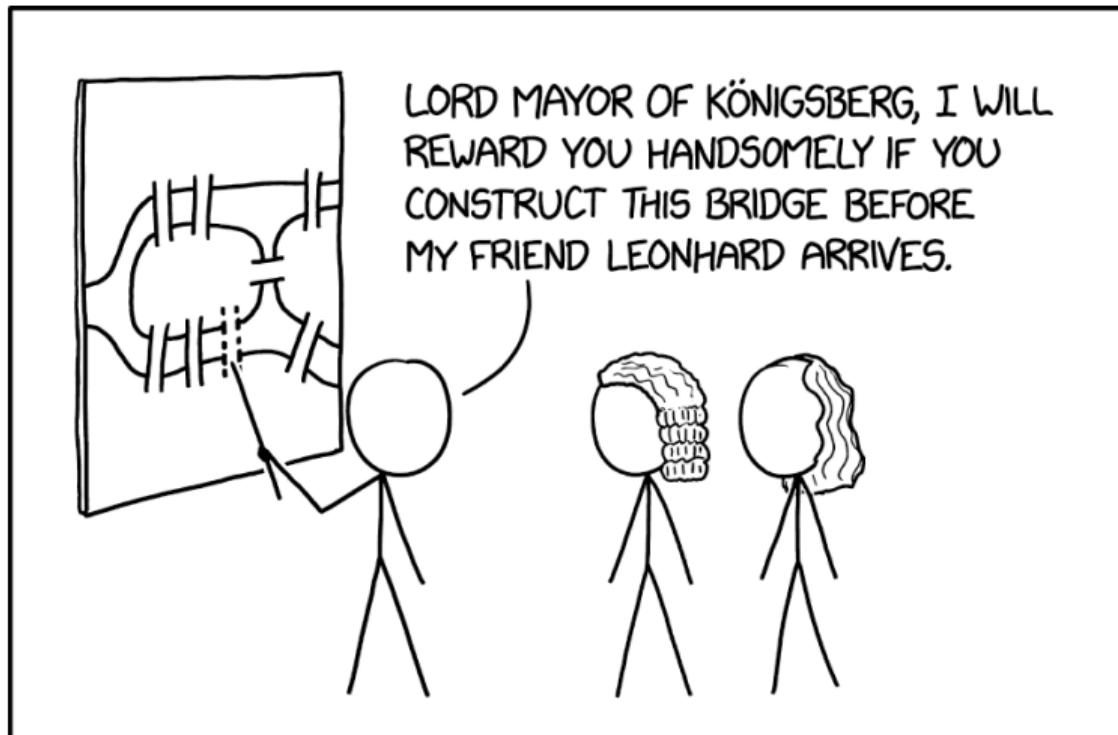


Problème :

“ Peut-on arranger son parcours de telle sorte que l’on passe sur chaque pont, et que l’on ne puisse y passer qu’une seule fois ? Cela semble possible, disent les uns ; impossible, disent les autres ; cependant personne n’a la certitude de son sentiment. ”

Proposition d'Euler pour résoudre ce problème :

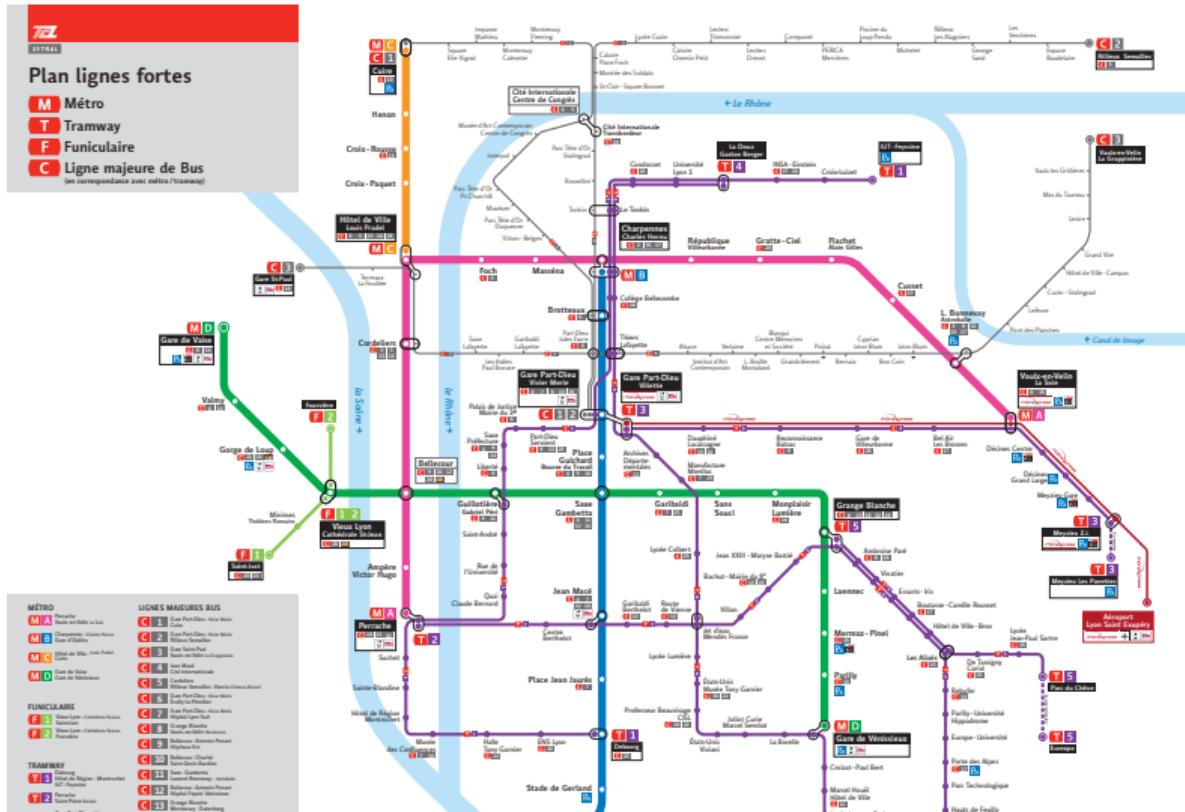
“ Former avec les lettres A, B, C, D une série de 8 lettres dans laquelle ces voisinages (A-C, A-B, A-D, D-C et B-D) apparaissent autant de fois qu'il a été indiqué (2, 2, 1, 1 et 1 fois) ; mais avant de chercher à effectuer une telle disposition, il est bon de se demander si celle-ci est réalisable. (...) Aussi ai-je trouvé une règle qui donne, pour tous les cas, la condition indispensable pour que le problème ne soit pas impossible. ”



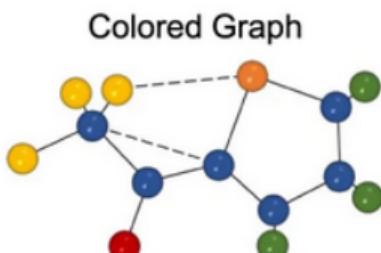
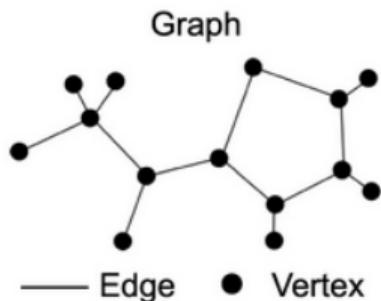
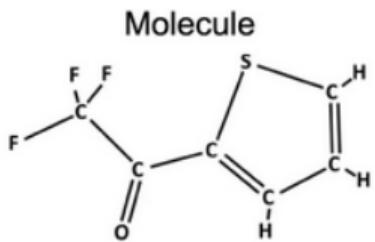
I TRIED TO USE A TIME MACHINE TO CHEAT ON MY ALGORITHMS
FINAL BY PREVENTING GRAPH THEORY FROM BEING INVENTED.

Image from xkcd.com

Exemples de modélisation par des graphes



Exemples de modélisation par des graphes

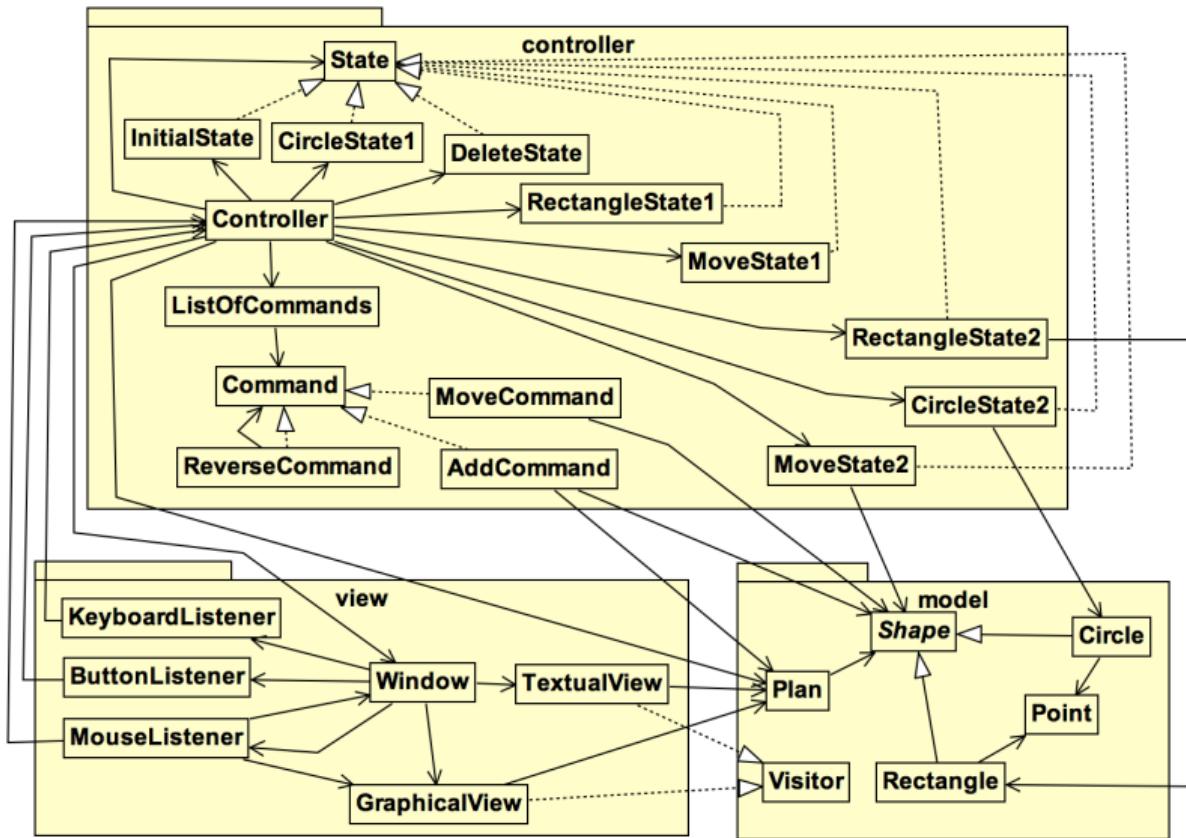


Graphes moléculaires :

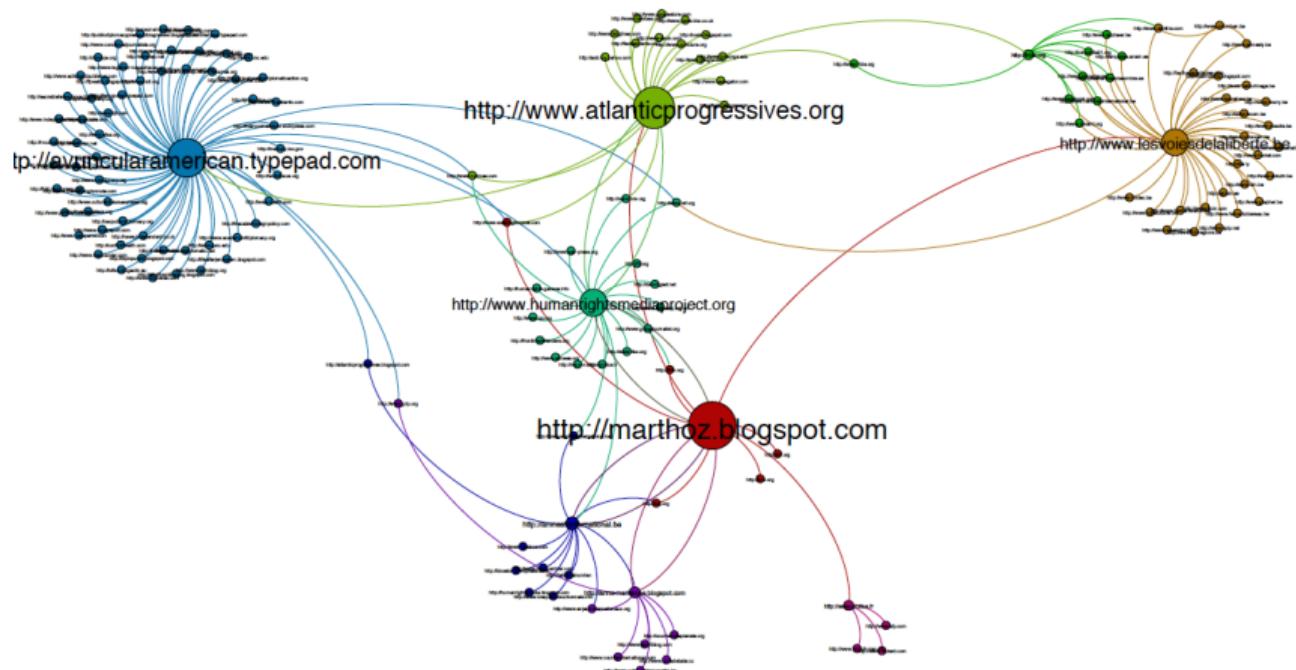
- Sommets = Atomes
- Arêtes = liaisons chimiques

Image : Nature Communications, 2021
(molécule 2-Trifluoroacetyl)

Exemples de modélisation par des graphes



Exemples de modélisation par des graphes



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

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

Exemples de modélisation par des graphes

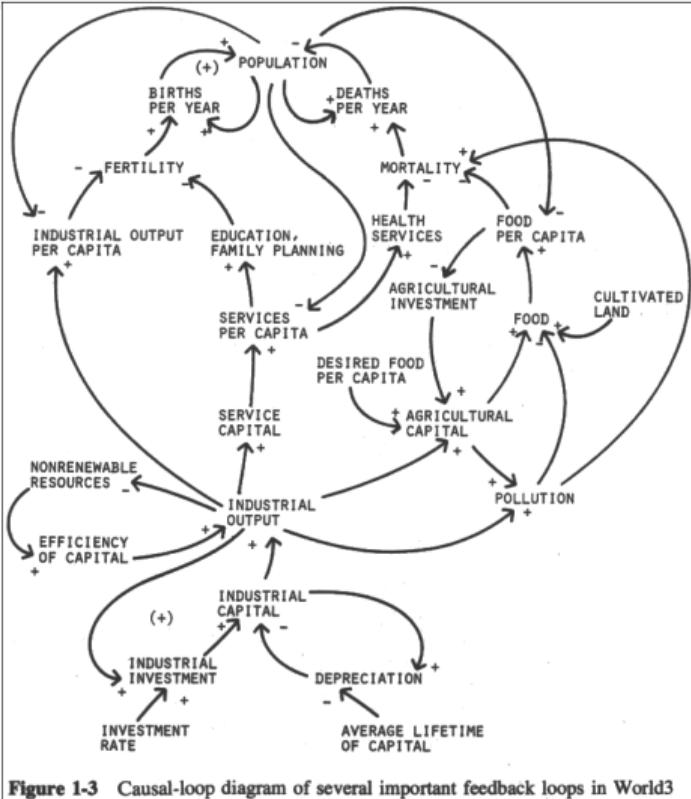
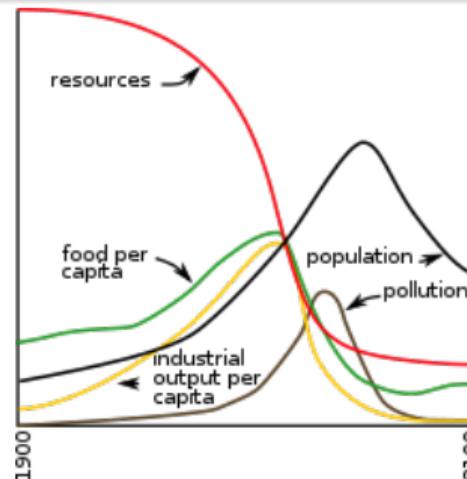


Figure 1-3 Causal-loop diagram of several important feedback loops in World3

The Limits to growth - Meadows et al, 1972

Causal Loop Diagram (World3) :

- Sommets = Variables
- Arêtes = Interactions positives (+) ou négatives (-)



On en verra plus dans l'EC REVE !

1

Introduction

- Organisation et objectifs pédagogiques
- Modélisation de problèmes avec des graphes
- Définitions

2

Structures de données pour représenter un graphe

3

Arbres Couvrants Minimaux

4

Parcours de graphes

5

Plus courts chemins

6

Problèmes de planification

7

Problèmes NP-difficiles

8

Conclusion

Graphes non orientés

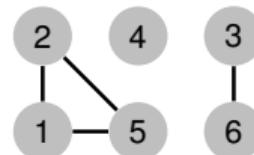
Définition

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

- S est un ensemble de **sommets**
- $A \subseteq S \times S$ est un ensemble d'**arêtes**
 - A définit une relation binaire **symétrique**
 $\sim \forall (s_i, s_j) \in S \times S, (s_i, s_j) \in A \Leftrightarrow (s_j, s_i) \in A$ (noté $\{s_i, s_j\}$)

Exemple :

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



Terminologie :

- s_i est **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 $\forall \{s_i, s_j\} \subseteq S, \{s_i, s_j\} \in A$

Graphes orientés

Définition

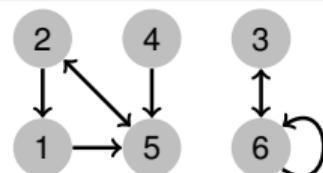
Un graphe orienté est défini par un couple (S, A) tel que

- S est un ensemble de sommets
- $A \subseteq S \times S$ est un ensemble d'**arcs**

~ A définit une relation binaire **non symétrique** : $(s_i, s_j) \in A \not\Rightarrow (s_j, s_i) \in A$

Exemple :

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



Terminologie :

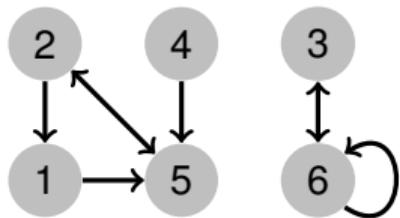
- s_j est **successeur** de s_i si $(s_i, s_j) \in A$: $\text{succ}(s_i) = \{s_j | (s_i, s_j) \in A\}$
- s_j est **prédecesseur** de s_i si $(s_j, s_i) \in A$: $\text{pred}(s_i) = \{s_j | (s_j, s_i) \in A\}$
- **demi-degré extérieur** = nombre de successeurs : $d^{\circ+}(s_i) = \#\text{succ}(s_i)$
- **demi-degré intérieur** = nombre de prédecesseurs : $d^{\circ-}(s_i) = \#\text{pred}(s_i)$

Graphes partiels

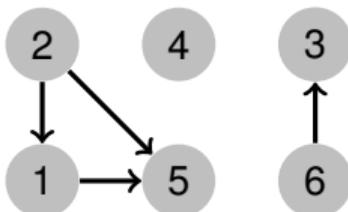
Définition

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

Exemple :



Graphe $G = (S, A)$



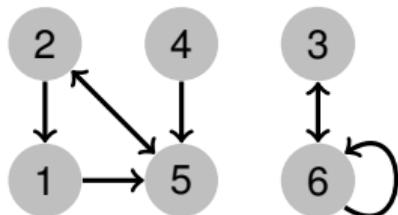
Graphe partiel de G

Sous-graphes

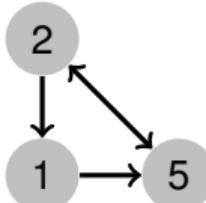
Définition

$G' = (S', A')$ est un sous-graphe de $G = (S, A)$ si $S' \subseteq S$ et $A' = A \cap S' \times S'$
~ $\sim G'$ est le sous-graphe de G **induit** par S'

Exemple :



Graphe $G = (S, A)$

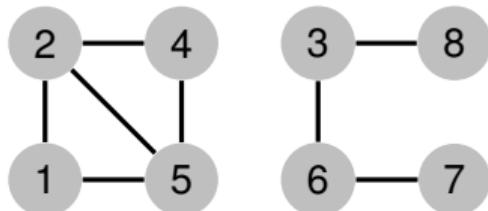


Sous-graphe de G
induit par $\{1, 2, 5\}$

Cheminements et connexités (1/2)

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

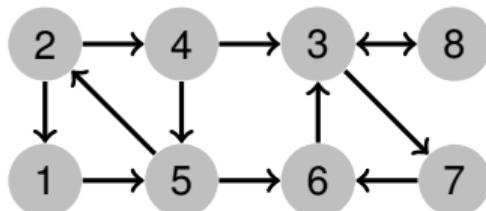
- **Chaîne** = Séquence de sommets $\langle s_0, s_1, s_2, \dots, s_k \rangle$ (notée $s_0 \sim s_k$) telle que $\forall i \in [1, k], \{s_{i-1}, s_i\} \in A$
- **Longueur** d'une chaîne = Nombre d'arêtes dans la chaîne
- **Chaîne élémentaire** = Chaîne dont tous les sommets sont distincts
- **Cycle** = 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 (2/2)

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

- **Chemin** = Séquence de sommets $\langle s_0, s_1, s_2, \dots, s_k \rangle$ (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** = sous-graphe fortement connexe maximal



Arbres et Arborescences

Définition d'un arbre :

Graphe non orienté $G = (S, A)$ vérifiant les 6 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 chaîne entre s_i et s_j

Théorème : 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

- 1 Introduction**
- 2 Structures de données pour représenter un graphe**
- 3 Arbres Couvrants Minimaux**
- 4 Parcours de graphes**
- 5 Plus courts chemins**
- 6 Problèmes de planification**
- 7 Problèmes NP-difficiles**
- 8 Conclusion**

Exemple d'algorithme :

```
1 Fonction entier degré( $g, s_i$ )
    Entrée : Un graphe non orienté  $g$  et un sommet  $s_i$  de  $g$ 
    Sortie : Le degré de  $s_i$ 
    début
        2    $d \leftarrow 0$ 
        3   pour tout sommet  $s_j \in adj(s_i)$  faire
        4       4    $d \leftarrow d + 1$ 
        5   5   retourne  $d$ 
        6
```

Complexité de cet algorithme ?

Exemple d'algorithme :

```
1 Fonction entier degré( $g, s_i$ )
    Entrée : Un graphe non orienté  $g$  et un sommet  $s_i$  de  $g$ 
    Sortie : Le degré de  $s_i$ 
    début
        2    $d \leftarrow 0$ 
        3   pour tout sommet  $s_j \in adj(s_i)$  faire
        4       5    $d \leftarrow d + 1$ 
        6   retourne  $d$ 
```

Complexité de cet algorithme ?

Dépend des structures de données utilisées pour représenter le graphe !

1 **Introduction**

2 **Structures de données pour représenter un graphe**

- Matrice d'adjacence
- Listes d'adjacence

3 **Arbres Couvrants Minimaux**

4 **Parcours de graphes**

5 **Plus courts chemins**

6 **Problèmes de planification**

7 **Problèmes NP-difficiles**

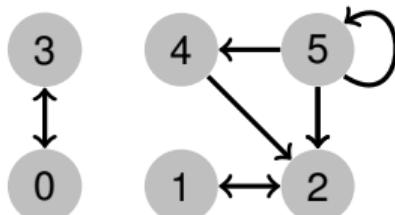
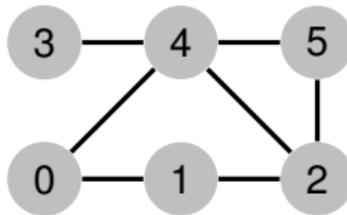
8 **Conclusion**

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

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

Complexité

Complexité en mémoire :

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

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

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

Complexité en temps de $\text{degré}(g, s_i)$:

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

Puissances de la matrice d'adjacence

Définition de M^k :

- $M^1 = M$
- $M^k = M \times M^{k-1}, \forall k > 1$

Théorème : $M^k[s_i][s_j] = \text{nombre de chemins de longueur } k \text{ allant de } s_i \text{ à } s_j$

~ Exercice : démonstration par récurrence sur k

Complexité pour un graphe ayant n sommets ?

- Complexité d'une multiplication de matrices $n \times n$

Puissances de la matrice d'adjacence

Définition de M^k :

- $M^1 = M$
- $M^k = M \times M^{k-1}, \forall k > 1$

Théorème : $M^k[s_i][s_j] = \text{nombre de chemins de longueur } k \text{ allant de } s_i \text{ à } s_j$

~ Exercice : démonstration par récurrence sur k

Complexité pour un graphe ayant n sommets ?

- Complexité d'une multiplication de matrices $n \times n$?

Puissances de la matrice d'adjacence

Définition de M^k :

- $M^1 = M$
- $M^k = M \times M^{k-1}, \forall k > 1$

Théorème : $M^k[s_i][s_j] = \text{nombre de chemins de longueur } k \text{ allant de } s_i \text{ à } s_j$

~ Exercice : démonstration par récurrence sur k

Complexité pour un graphe ayant n sommets ?

- Complexité d'une multiplication de matrices $n \times n$: $\mathcal{O}(n^3)$
 - ~ Peut être optimisé dans le cas de matrices creuses (cf TP1)

Puissances de la matrice d'adjacence

Définition de M^k :

- $M^1 = M$
- $M^k = M \times M^{k-1}, \forall k > 1$

Théorème : $M^k[s_i][s_j] = \text{nombre de chemins de longueur } k \text{ allant de } s_i \text{ à } s_j$

~ Exercice : démonstration par récurrence sur k

Complexité pour un graphe ayant n sommets ?

- Complexité d'une multiplication de matrices $n \times n$: $\mathcal{O}(n^3)$
 - ~ Peut être optimisé dans le cas de matrices creuses (cf TP1)
- Complexité du calcul de M^k ?
 - $\mathcal{O}(kn^3)$ si on fait k multiplications
 - $\mathcal{O}((\log k)n^3)$ en exploitant le fait que $M^{2x} = (M^x)^2$ et $M^{2x+1} = M(M^x)^2$

1 **Introduction**

2 **Structures de données pour représenter un graphe**

- Matrice d'adjacence
- Listes d'adjacence

3 **Arbres Couvrants Minimaux**

4 **Parcours de graphes**

5 **Plus courts chemins**

6 **Problèmes de planification**

7 **Problèmes NP-difficiles**

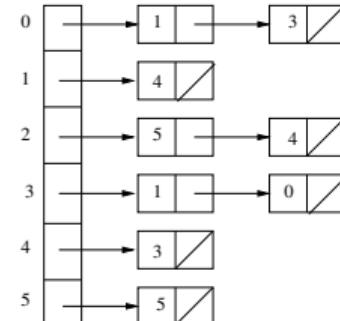
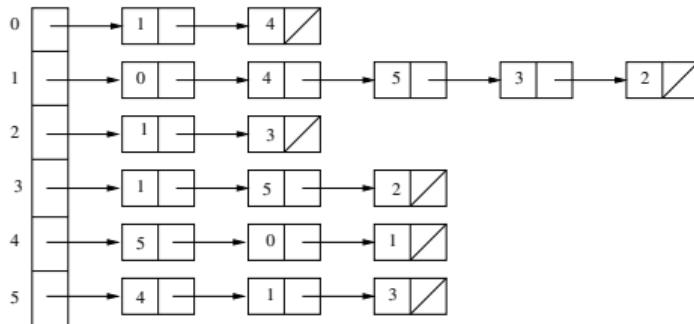
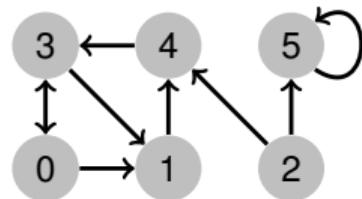
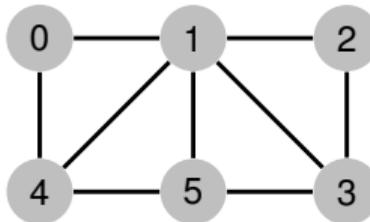
8 **Conclusion**

Listes d'adjacence

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

Tableau succ tel que $\text{succ}[s_i] = \text{liste des successeurs de } s_i$

Exemples :

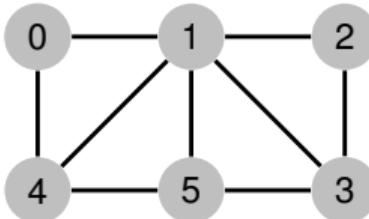


Listes d'adjacence

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

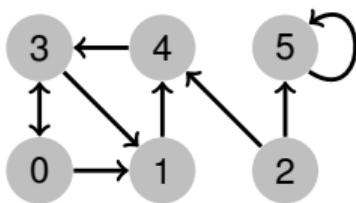
Tableau succ tel que $\text{succ}[s_i] = \text{liste des successeurs de } s_i$

Exemples :



$\text{nbSucc} = \begin{matrix} 2 & 5 & 2 & 3 & 3 & 3 \\ 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$

$\text{succ}[0] =$	1	4			
$\text{succ}[1] =$	0	2	3	4	5
$\text{succ}[2] =$	1	3			
$\text{succ}[3] =$	1	2	5		
$\text{succ}[4] =$	0	1	5		
$\text{succ}[5] =$	1	3	4		



$\text{nbSucc} = \begin{matrix} 2 & 1 & 2 & 2 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$

$\text{succ}[0] =$	1	3
$\text{succ}[1] =$	4	
$\text{succ}[2] =$	5	4
$\text{succ}[3] =$	1	0
$\text{succ}[4] =$	3	
$\text{succ}[5] =$	5	

Complexité

Complexité en mémoire :

~ $\mathcal{O}(n + p)$ 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 :

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

Complexité en temps de $\text{degré}(g, s_i)$:

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

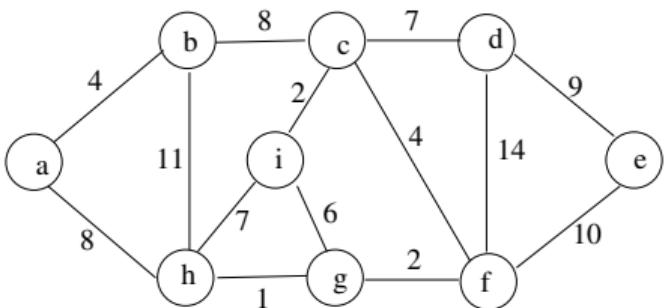
- 1 Introduction**
- 2 Structures de données pour représenter un graphe**
- 3 Arbres Couvrants Minimaux**
- 4 Parcours de graphes**
- 5 Plus courts chemins**
- 6 Problèmes de planification**
- 7 Problèmes NP-difficiles**
- 8 Conclusion**

Arbre Couvrant Minimal (Minimum Spanning Tree / MST)

Le MST d'un graphe $G = (S, A)$ est un graphe partiel $G' = (S, E)$ de G tel que :

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

Exemple :



Spécification du problème MST :

1 Fonction $MST(g, cout)$

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

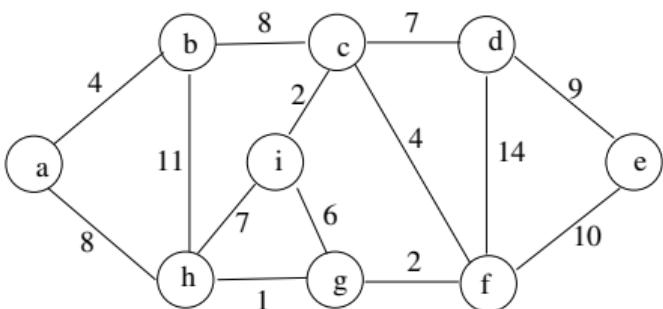
Sortie : Un ensemble d'arêtes $E \subseteq A$ tel que (S, E) est un arbre couvrant minimal de G

Arbre Couvrant Minimal (Minimum Spanning Tree / MST)

Le MST d'un graphe $G = (S, A)$ est un graphe partiel $G' = (S, E)$ de G tel que :

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

Exemple :



Spécification du problème MST :

1 Fonction $MST(g, cout)$

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

Sortie : Un ensemble d'arêtes $E \subseteq A$ tel que (S, E) est un arbre couvrant minimal de G

1 Introduction

2 Structures de données pour représenter un graphe

3 Arbres Couvrants Minimaux

- Algorithme générique
- Algorithme de Kruskal
- Algorithme de Prim

4 Parcours de graphes

5 Plus courts chemins

6 Problèmes de planification

7 Problèmes NP-difficiles

8 Conclusion

Algorithme générique

```

1 Fonction MSTgénérique( $G = (S, A)$ , cout)
2    $E \leftarrow \emptyset$ 
3   tant que  $\#E < \#S - 1$  faire
4     Ajouter dans  $E$  l'arête de coût minimal traversant une coupure  $(P, S \setminus P)$  qui respecte  $E$ 
5   retourne  $E$ 

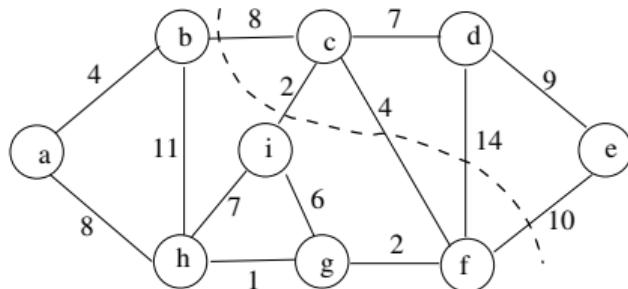
```

Définitions :

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

Exemple :

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



Algorithme générique

```
1 Fonction MSTgénérique( $G = (S, A)$ ,  $\text{cout}$ )
2    $E \leftarrow \emptyset$ 
3   tant que  $\#E < \#S - 1$  faire
4     Ajouter dans  $E$  l'arête de coût minimal traversant une coupure  $(P, S \setminus P)$  qui respecte  $E$ 
5   retourne  $E$ 
```

Preuve de correction

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

Algorithme générique

```

1 Fonction MSTgénérique( $G = (S, A)$ , cout)
2    $E \leftarrow \emptyset$ 
3   tant que  $\#E < \#S - 1$  faire
4     Ajouter dans  $E$  l'arête de coût minimal traversant une coupure  $(P, S \setminus P)$  qui respecte  $E$ 
5   retourne  $E$ 

```

Preuve de correction

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

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

- Algorithme de Kruskal (Kruskal 1956) : (S, E) est une forêt
 $\leadsto (s_i, s_j)$ = arête de coût min connectant 2 arbres différents
 - Algorithme de Prim (Prim 1957) : E est un arbre reliant un sous-ensemble de sommets $P \subseteq S$
 $\leadsto (s_i, s_j)$ = arête de coût min traversant la coupure $(P, S \setminus P)$
- \leadsto Kruskal et Prim sont des algorithmes **gloutons**

1 Introduction

2 Structures de données pour représenter un graphe

3 Arbres Couvrants Minimaux

- Algorithme générique
- Algorithme de Kruskal
- Algorithme de Prim

4 Parcours de graphes

5 Plus courts chemins

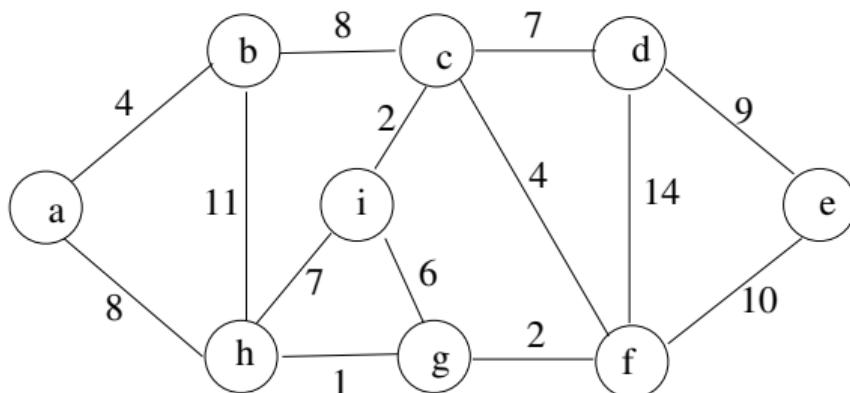
6 Problèmes de planification

7 Problèmes NP-difficiles

8 Conclusion

1 Fonction *Kruskal(g, cout)*2 $E \leftarrow \emptyset$ 3 Trier les arêtes de A par ordre de coût croissant4 **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 7 **retourne** 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    $E \leftarrow \emptyset$ 
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  $E$ 
7   retourne  $E$ 
```

Comment effectuer efficacement le test de la ligne 5 ?

Représenter les composantes connexes de (S, E) par des *disjoint sets*
~ Voir le support de cours sur Moodle pour plus de détails !

Complexité pour un graphe ayant n sommets et p arcs ?

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

1 Introduction

2 Structures de données pour représenter un graphe

3 Arbres Couvrants Minimaux

- Algorithme générique
- Algorithme de Kruskal
- Algorithme de Prim

4 Parcours de graphes

5 Plus courts chemins

6 Problèmes de planification

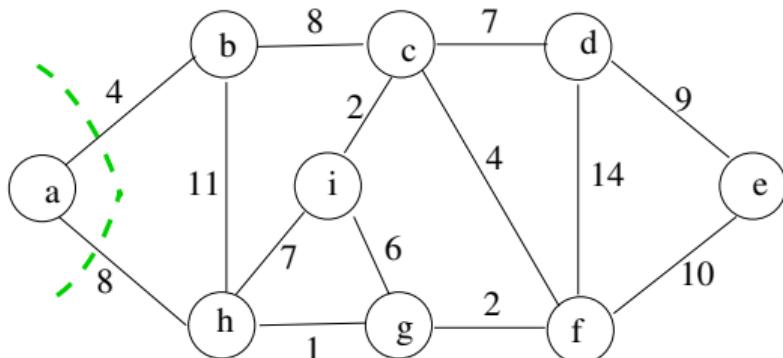
7 Problèmes NP-difficiles

8 Conclusion

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
 12 **retourne** E

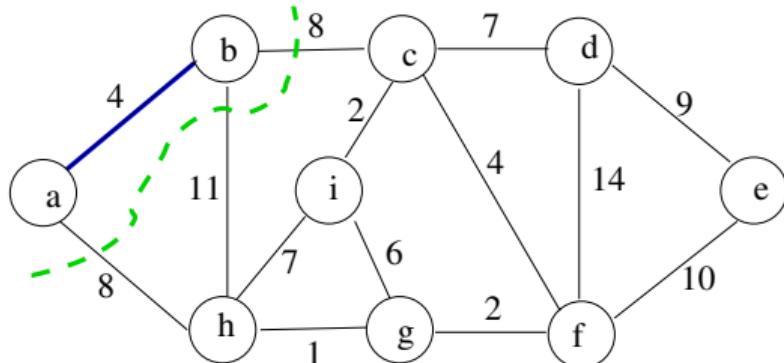
Exemple :



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
 12 **retourne** E

Exemple :



```

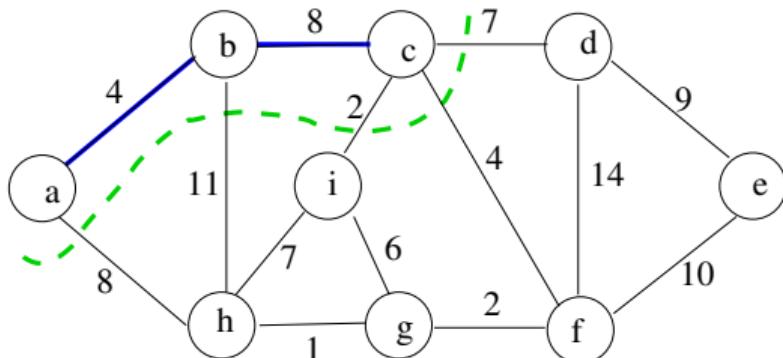
1 Fonction Prim(g, cout)
2   Soit  $s_0$  un sommet de  $S$  choisi arbitrairement
3    $P \leftarrow \{s_0\}$ ;  $E \leftarrow \emptyset$ 

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  $E$ 

12  retourne E

```

Exemple :



```

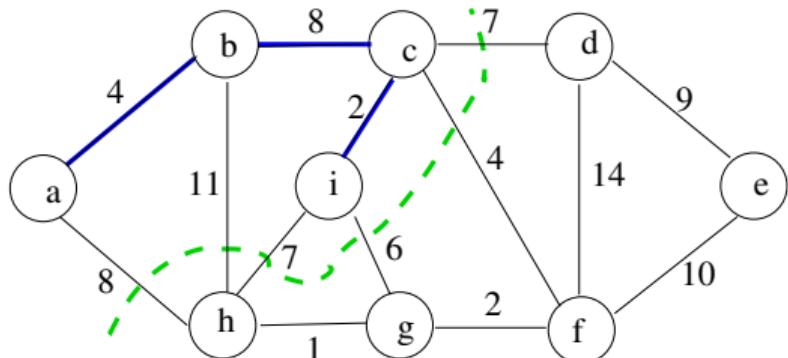
1 Fonction Prim(g, cout)
2   Soit  $s_0$  un sommet de  $S$  choisi arbitrairement
3    $P \leftarrow \{s_0\}$ ;  $E \leftarrow \emptyset$ 

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  $E$ 

12  retourne E

```

Exemple :



```

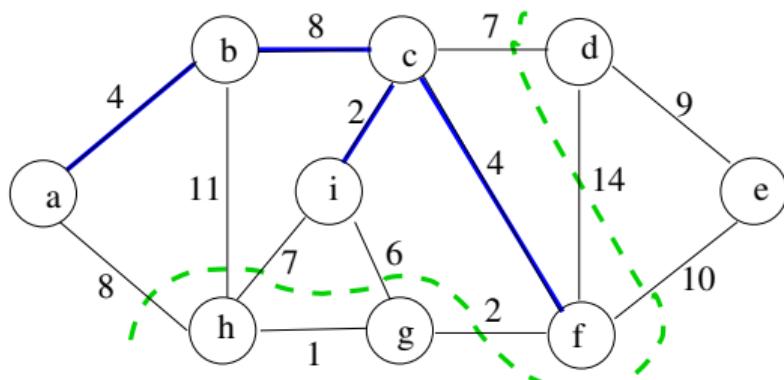
1 Fonction Prim(g, cout)
2   Soit  $s_0$  un sommet de  $S$  choisi arbitrairement
3    $P \leftarrow \{s_0\}$ ;  $E \leftarrow \emptyset$ 

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  $E$ 

12  retourne E

```

Exemple :



```

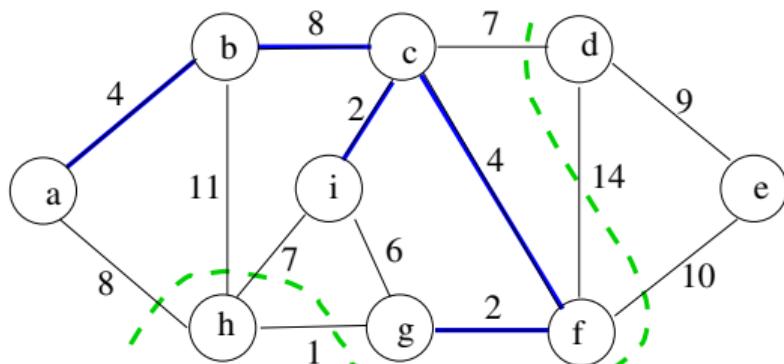
1 Fonction Prim(g, cout)
2   Soit  $s_0$  un sommet de  $S$  choisi arbitrairement
3    $P \leftarrow \{s_0\}$ ;  $E \leftarrow \emptyset$ 

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  $E$ 

12  retourne E

```

Exemple :



```

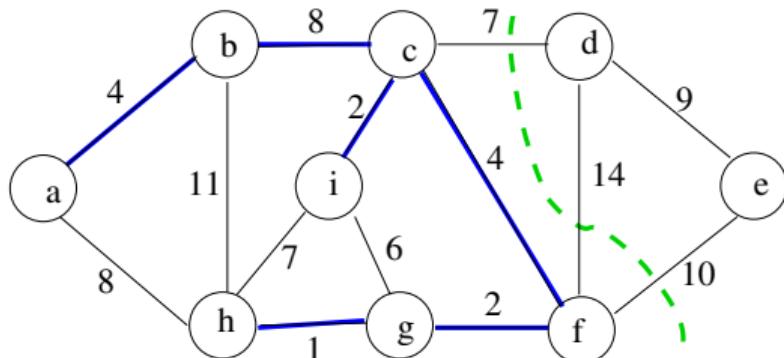
1 Fonction Prim(g, cout)
2   Soit  $s_0$  un sommet de  $S$  choisi arbitrairement
3    $P \leftarrow \{s_0\}$ ;  $E \leftarrow \emptyset$ 

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  $E$ 

12  retourne E

```

Exemple :



1 Fonction $Prim(g, cout)$

2 Soit s_0 un sommet de S choisi arbitrairement

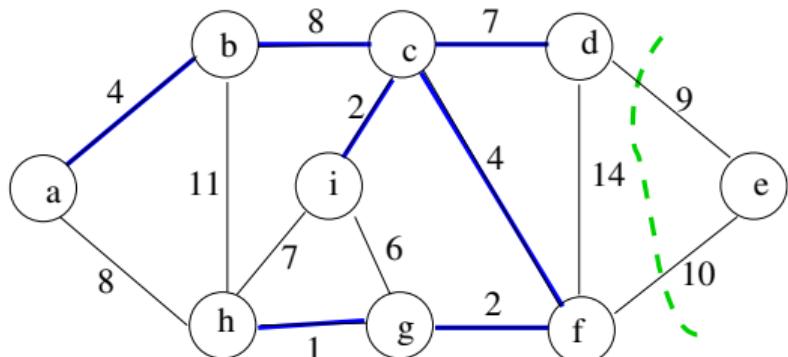
3 $P \leftarrow \{s_0\}; E \leftarrow \emptyset$

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

12 **retourne** E

Exemple :

```

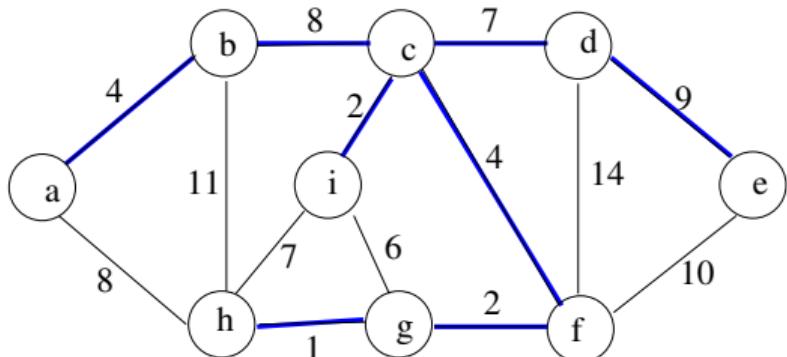
1 Fonction Prim(g, cout)
2   Soit  $s_0$  un sommet de  $S$  choisi arbitrairement
3    $P \leftarrow \{s_0\}$ ;  $E \leftarrow \emptyset$ 

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  $E$ 

12  retourne  $E$ 

```

Exemple :



```
1 Fonction Prim( $g, cout$ )
2   Soit  $s_0$  un sommet de  $S$  choisi arbitrairement
3    $P \leftarrow \{s_0\}; E \leftarrow \emptyset$ 
4
5
6
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  $E$ 
10
11
12  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    $P \leftarrow \{s_0\}; E \leftarrow \emptyset$ 
4
5
6
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  $E$ 
10
11
12  retourne  $E$ 
```

Comment choisir efficacement $\{s_i, s_j\}$ ligne 8 ?

Maintenir les tableaux π et c tels que :

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

```

1 Fonction Prim(g, cout)
2   Soit  $s_0$  un sommet de  $S$  choisi arbitrairement
3    $P \leftarrow \{s_0\}$ ;  $E \leftarrow \emptyset$ 
4   pour chaque sommet  $s_i \in S$  faire
5     si  $s_i \in \text{adj}(s_0)$  alors  $\pi[s_i] \leftarrow s_0$ ;  $c[s_i] \leftarrow \text{cout}(s_0, s_i)$ ;
6     sinon  $\pi[s_i] \leftarrow \text{null}$ ;  $c[s_i] \leftarrow \infty$ ;
7   tant que  $P \neq S$  faire
8     Ajouter dans  $P$  le sommet  $s_i \in S \setminus P$  ayant la plus petite valeur de  $c$ 
9     Ajouter  $(s_i, \pi[s_i])$  à  $E$ 
10    pour chaque sommet  $s_j \in \text{adj}(s_i)$  faire
11      si  $s_j \notin P$  et  $\text{cout}(s_i, s_j) < c[s_j]$  alors  $\pi[s_j] \leftarrow s_i$ ;  $c[s_j] \leftarrow \text{cout}(s_i, s_j)$ ;
12
retourne  $E$ 

```

Comment choisir efficacement $\{s_i, s_j\}$ ligne 8 ?

Maintenir les tableaux π et c tels que :

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

```
1 Fonction Prim( $g, cout$ )
2   Soit  $s_0$  un sommet de  $S$  choisi arbitrairement
3    $P \leftarrow \{s_0\}; E \leftarrow \emptyset$ 
4   pour chaque sommet  $s_i \in S$  faire
5     si  $s_i \in adj(s_0)$  alors  $\pi[s_i] \leftarrow s_0; c[s_i] \leftarrow cout(s_0, s_i);$ 
6     sinon  $\pi[s_i] \leftarrow null; c[s_i] \leftarrow \infty;$ 
7   tant que  $P \neq S$  faire
8     Ajouter dans  $P$  le sommet  $s_i \in S \setminus P$  ayant la plus petite valeur de  $c$ 
9     Ajouter  $(s_i, \pi[s_i])$  à  $E$ 
10    pour chaque sommet  $s_j \in adj(s_i)$  faire
11      si  $s_j \notin P$  et  $cout(s_i, s_j) < c[s_j]$  alors  $\pi[s_j] \leftarrow s_i; c[s_j] \leftarrow cout(s_i, s_j);$ 
12  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    $P \leftarrow \{s_0\}; E \leftarrow \emptyset$ 
4   pour chaque sommet  $s_i \in S$  faire
5     si  $s_i \in adj(s_0)$  alors  $\pi[s_i] \leftarrow s_0; c[s_i] \leftarrow cout(s_0, s_i);$ 
6     sinon  $\pi[s_i] \leftarrow null; c[s_i] \leftarrow \infty;$ 
7   tant que  $P \neq S$  faire
8     Ajouter dans  $P$  le sommet  $s_i \in S \setminus P$  ayant la plus petite valeur de  $c$ 
9     Ajouter  $(s_i, \pi[s_i])$  à  $E$ 
10    pour chaque sommet  $s_j \in adj(s_i)$  faire
11      si  $s_j \notin P$  et  $cout(s_i, s_j) < c[s_j]$  alors  $\pi[s_j] \leftarrow s_i; c[s_j] \leftarrow cout(s_i, s_j);$ 
12  retourne  $E$ 
```

Comment chercher efficacement s_i ligne 8 ?

~ File de priorité implémentée par un tas binaire

```
1 Fonction Prim( $g, cout$ )
2   Soit  $s_0$  un sommet de  $S$  choisi arbitrairement
3    $P \leftarrow \{s_0\}; E \leftarrow \emptyset$ 
4   pour chaque sommet  $s_i \in S$  faire
5     si  $s_i \in adj(s_0)$  alors  $\pi[s_i] \leftarrow s_0; c[s_i] \leftarrow cout(s_0, s_i);$ 
6     sinon  $\pi[s_i] \leftarrow null; c[s_i] \leftarrow \infty;$ 
7   tant que  $P \neq S$  faire
8     Ajouter dans  $P$  le sommet  $s_i \in S \setminus P$  ayant la plus petite valeur de  $c$ 
9     Ajouter  $(s_i, \pi[s_i])$  à  $E$ 
10    pour chaque sommet  $s_j \in adj(s_i)$  faire
11      si  $s_j \notin P$  et  $cout(s_i, s_j) < c[s_j]$  alors  $\pi[s_j] \leftarrow s_i; c[s_j] \leftarrow cout(s_i, s_j);$ 
12  retourne  $E$ 
```

Comment chercher efficacement s_i ligne 8 ?

~ 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    $P \leftarrow \{s_0\}; E \leftarrow \emptyset$ 
4   pour chaque sommet  $s_i \in S$  faire
5     si  $s_i \in adj(s_0)$  alors  $\pi[s_i] \leftarrow s_0; c[s_i] \leftarrow cout(s_0, s_i);$ 
6     sinon  $\pi[s_i] \leftarrow null; c[s_i] \leftarrow \infty;$ 
7   tant que  $P \neq S$  faire
8     Ajouter dans  $P$  le sommet  $s_i \in S \setminus P$  ayant la plus petite valeur de  $c$ 
9     Ajouter  $(s_i, \pi[s_i])$  à  $E$ 
10    pour chaque sommet  $s_j \in adj(s_i)$  faire
11      si  $s_j \notin P$  et  $cout(s_i, s_j) < c[s_j]$  alors  $\pi[s_j] \leftarrow s_i; c[s_j] \leftarrow cout(s_i, s_j);$ 
12
retourne  $E$ 

```

Comment chercher efficacement s_i ligne 8 ?

~ File de priorité implémentée par un tas binaire

Complexité pour un graphe ayant n sommets et p arêtes ?

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

- 1 Introduction**
- 2 Structures de données pour représenter un graphe**
- 3 Arbres Couvrants Minimaux**
- 4 Parcours de graphes**
- 5 Plus courts chemins**
- 6 Problèmes de planification**
- 7 Problèmes NP-difficiles**
- 8 Conclusion**

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
- ~ 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
- ~ 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
- ~ 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 $\text{Parcours}(g, s_0)$

 Entrée : Un graphe g et un sommet s_0 de g

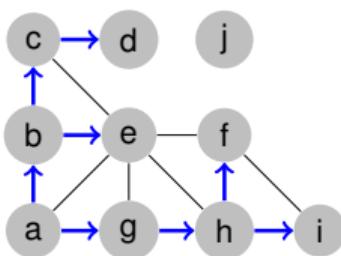
 Sortie : Arborescence π du parcours de g à partir de s_0

Arborescence associée à un parcours :

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

Quelle structure de données pour représenter l'arborescence ?

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



Spécification d'un algorithme de parcours

1 Fonction $\text{Parcours}(g, s_0)$

- Entrée** : Un graphe g et un sommet s_0 de g
- Sortie** : Arborescence π du parcours de g à partir de s_0

Arborescence associée à un parcours :

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

Quelle structure de données pour représenter l'arborescence ?

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

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

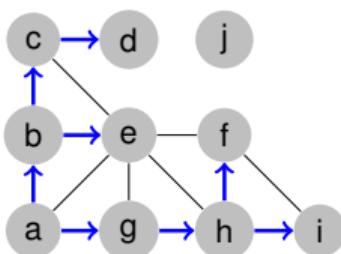


Tableau π correspondant :

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

1 **Introduction**

2 **Structures de données pour représenter un graphe**

3 **Arbres Couvrants Minimaux**

4 **Parcours de graphes**

- Parcours en largeur (BFS)
- Parcours en profondeur (DFS)

5 **Plus courts chemins**

6 **Problèmes de planification**

7 **Problèmes NP-difficiles**

8 **Conclusion**

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 $\pi[s_i] \leftarrow null$

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

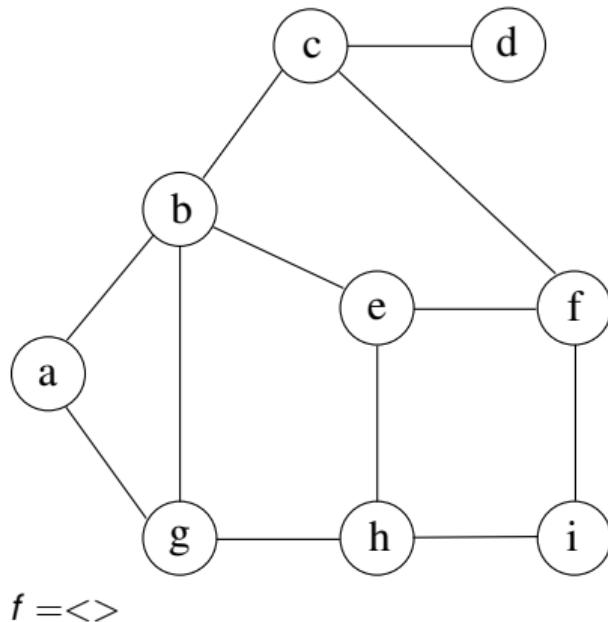
9 pour chaque $s_i \in succ(s_k)$ tq s_i est blanc faire

10 Ajouter s_i dans f et colorier s_i en gris

11 $\pi[s_i] \leftarrow s_k$

12 Enlever s_k de f et colorier s_k en noir

13 retourne π



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 $\pi[s_i] \leftarrow null$

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

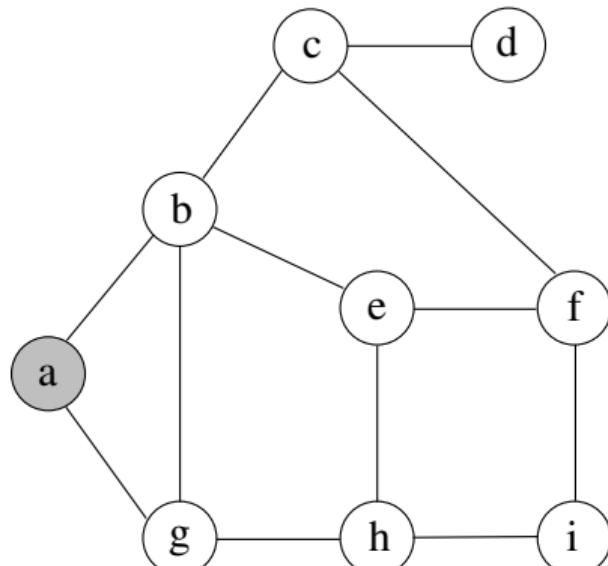
9 pour chaque $s_i \in succ(s_k)$ tq s_i est blanc faire

10 Ajouter s_i dans f et colorier s_i en gris

11 $\pi[s_i] \leftarrow s_k$

12 Enlever s_k de f et colorier s_k en noir

13 retourne π



$$f = \langle a \rangle$$

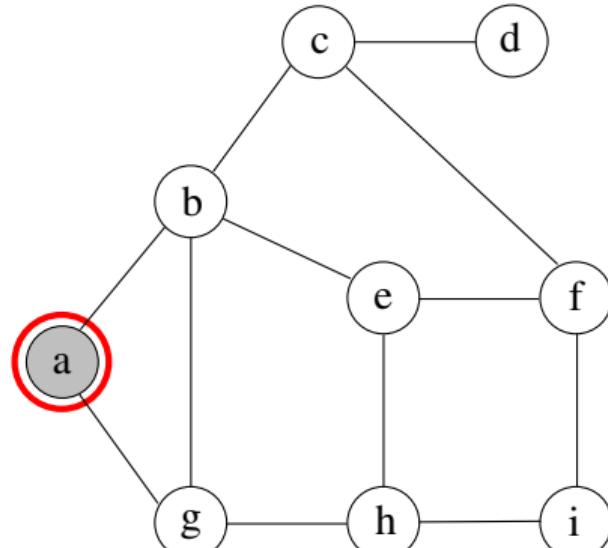
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$f = < a >$$

$$s_k = a$$

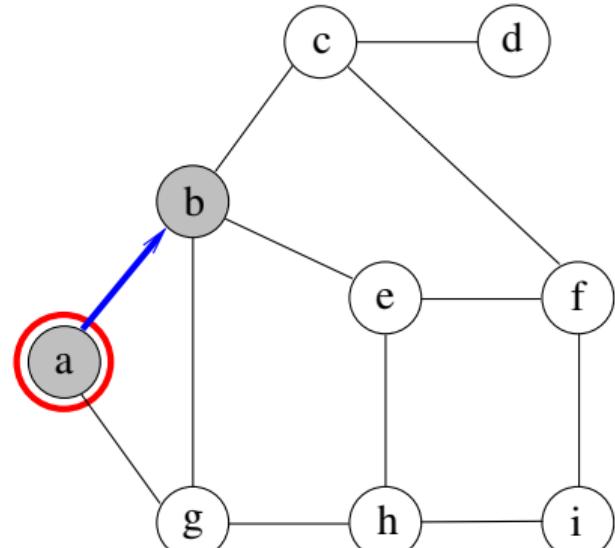
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$f = \langle b, a \rangle$$

$$s_k = a, s_i = b$$

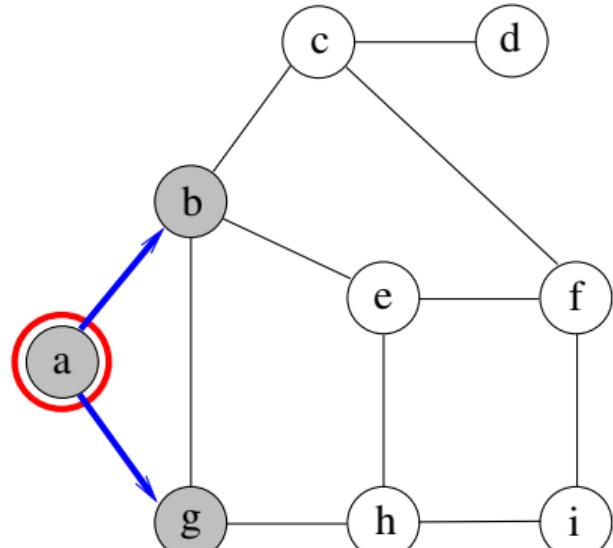
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$f = < g, b, a >$$

$$s_k = a, s_i = g$$

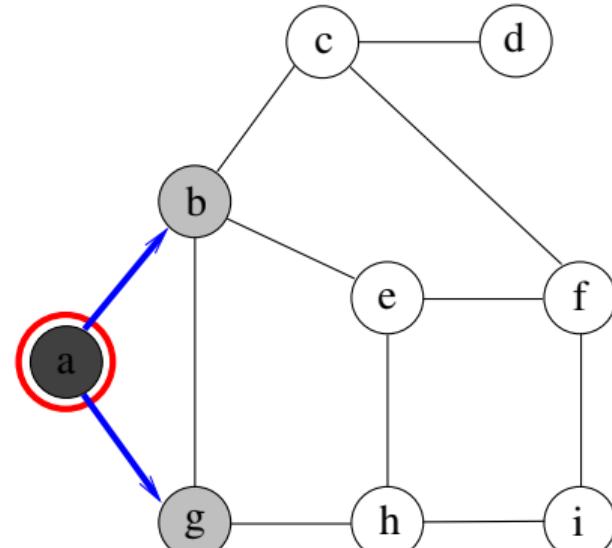
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$f = \langle g, b \rangle$$

$$s_k = a$$

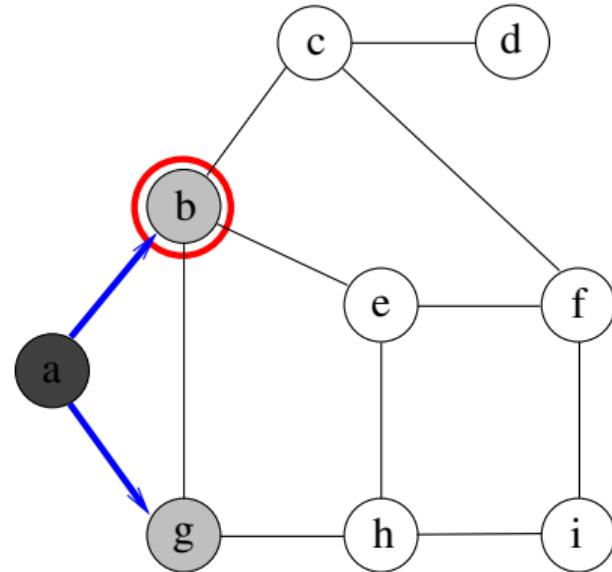
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$\begin{aligned} f &= \langle g, b \rangle \\ s_k &= b \end{aligned}$$

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 $\pi[s_i] \leftarrow null$

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

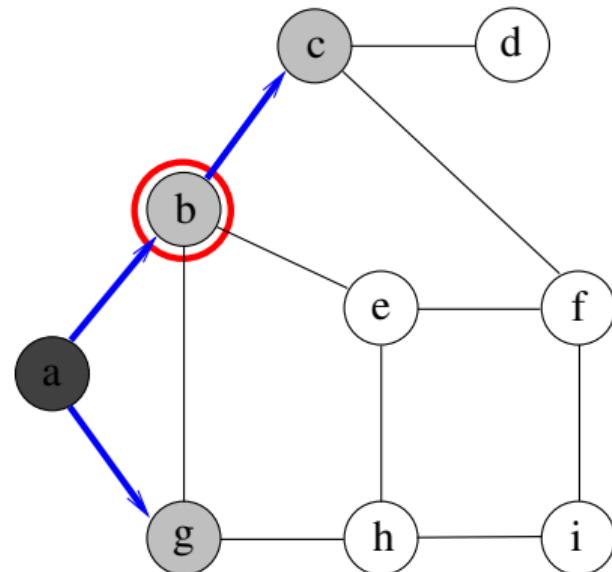
9 pour chaque $s_i \in succ(s_k)$ tq s_i blanc faire

10 Ajouter s_i dans f et colorier s_i en gris

11 $\pi[s_i] \leftarrow s_k$

12 Enlever s_k de f et colorier s_k en noir

13 retourne π



$$f = \langle c, g, b \rangle$$

$$s_k = b, s_i = c$$

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 $\pi[s_i] \leftarrow null$

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

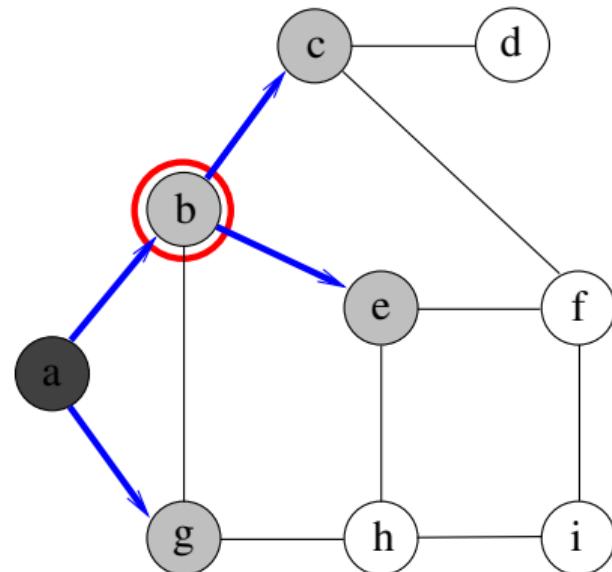
9 pour chaque $s_i \in succ(s_k)$ tq s_i blanc faire

10 Ajouter s_i dans f et colorier s_i en gris

11 $\pi[s_i] \leftarrow s_k$

12 Enlever s_k de f et colorier s_k en noir

13 retourne π



$$f = \langle e, c, g, b \rangle$$

$$s_k = b, s_i = e$$

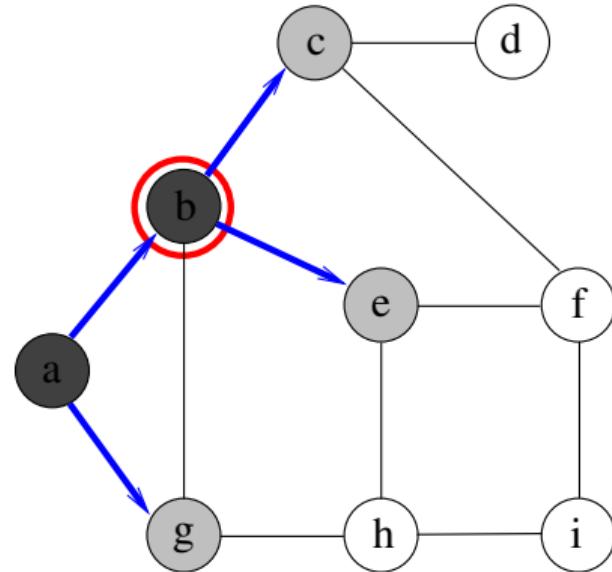
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$f = \langle e, c, g \rangle$$

$$s_k = b$$

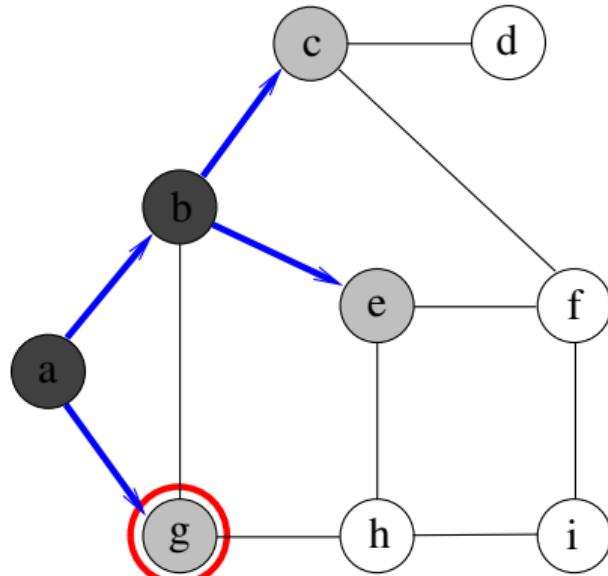
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$\begin{aligned}f &= \langle e, c, g \rangle \\s_k &= g\end{aligned}$$

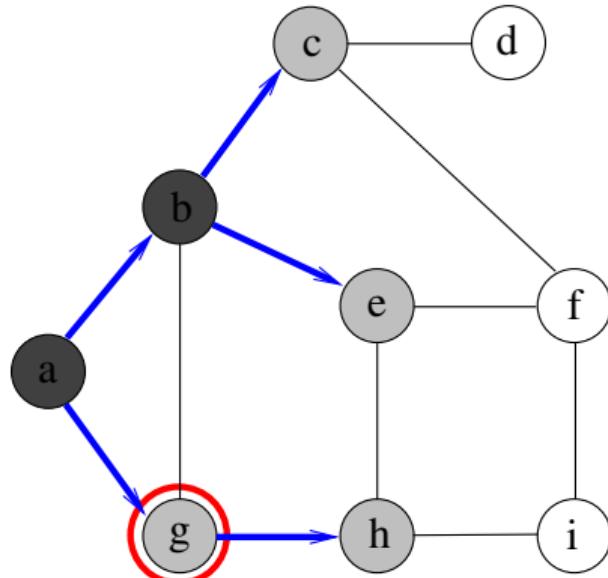
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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      $\pi[s_i] \leftarrow null$ 
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     pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  blanc faire
10       Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11        $\pi[s_i] \leftarrow s_k$ 
12     Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13   retourne  $\pi$ 

```



$$f = \langle h, e, c, g \rangle$$

$$s_k = g, s_i = h$$

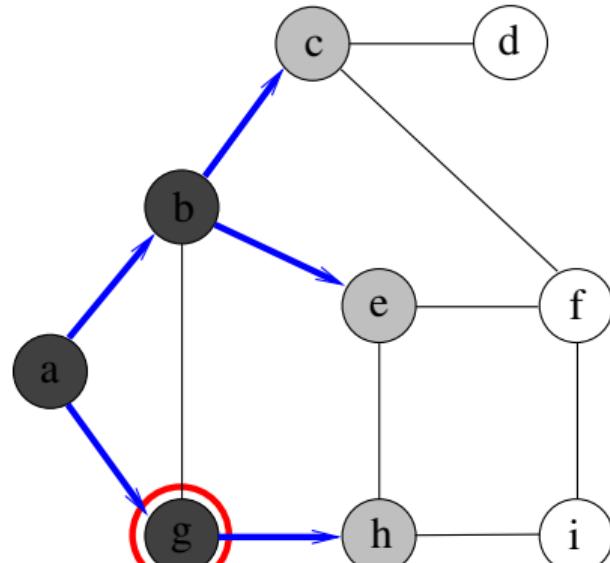
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$f = \langle h, e, c \rangle$$

$$s_k = g$$

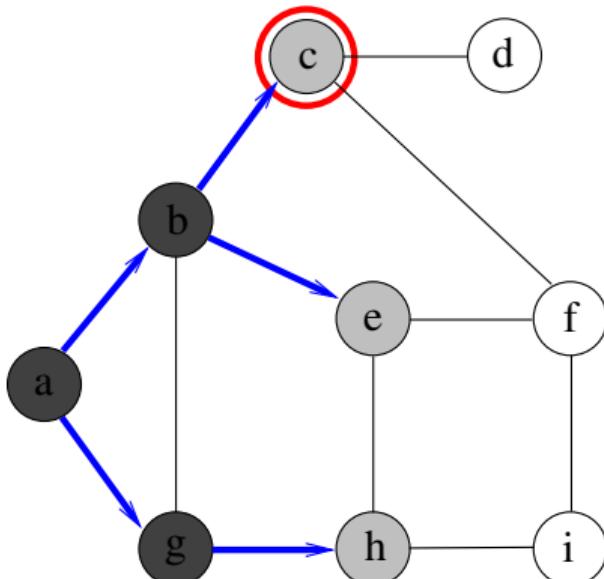
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



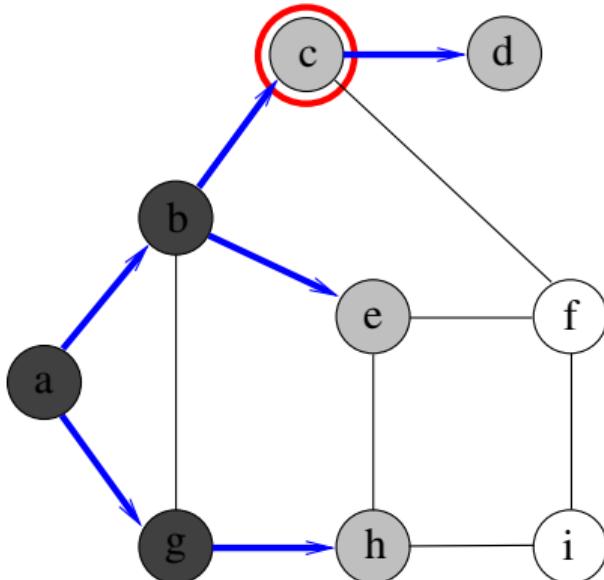
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$f = \langle d, h, e, c \rangle$$

$$s_k = c, s_i = d$$

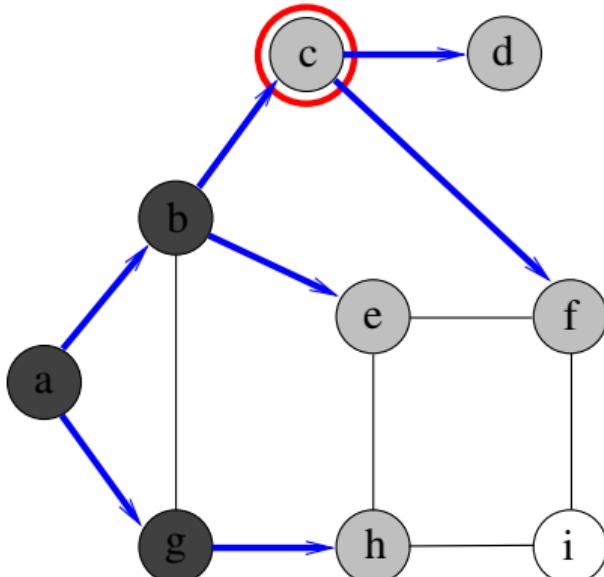
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$\begin{aligned}
 f &= \langle f, d, h, e, c \rangle \\
 s_k &= c, s_i = f
 \end{aligned}$$

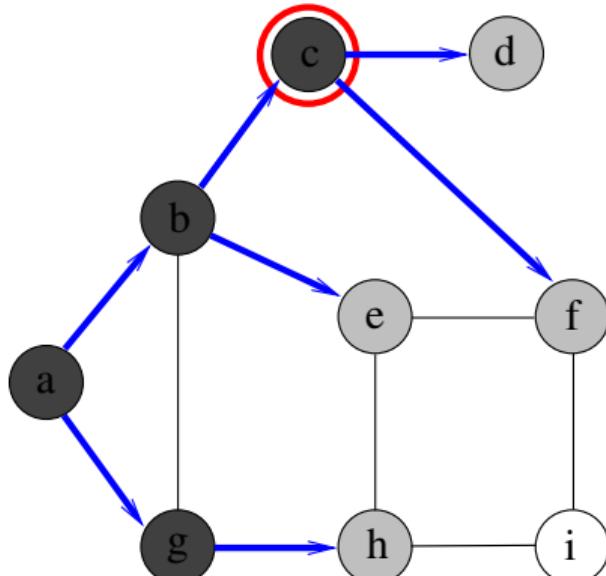
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$f = \langle f, d, h, e \rangle$$

$$s_k = c$$

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        $\pi[s_i] \leftarrow null$ 
5       Colorier  $s_i$  en blanc

```

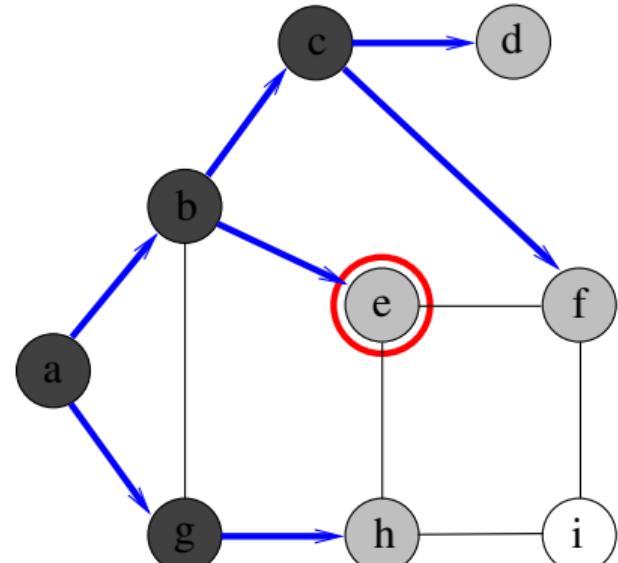
Ajouter s_0 dans f et colorier s_0 en gris

tant que f n'est pas vide faire

```

8   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
9   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10      Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11       $\pi[s_i] \leftarrow s_k$ 
12      Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13
retourne  $\pi$ 

```



$$f = \langle f, d, h, e \rangle$$

$$s_k = e$$

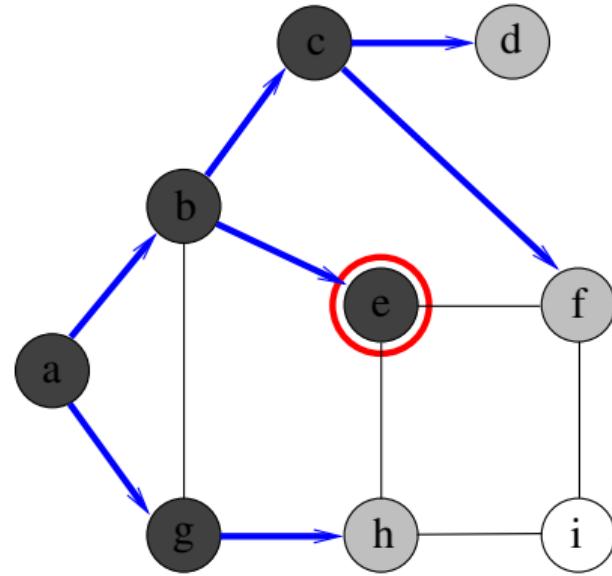
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$\begin{aligned}f &= \langle f, d, h \rangle \\s_k &= e\end{aligned}$$

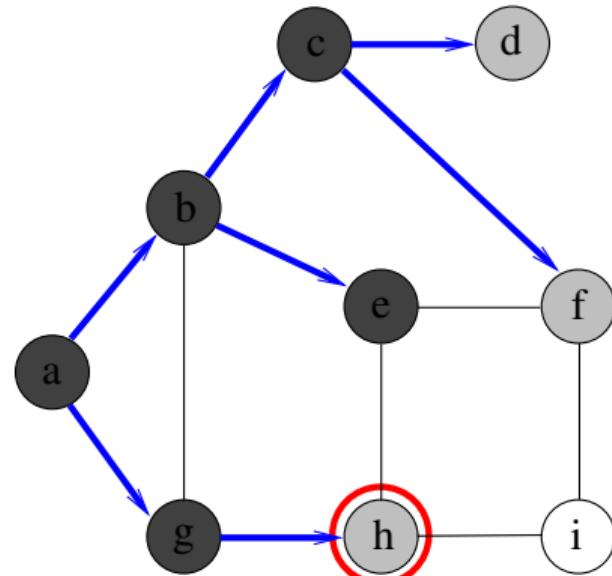
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$f = \langle f, d, h \rangle$$

$$s_k = h$$

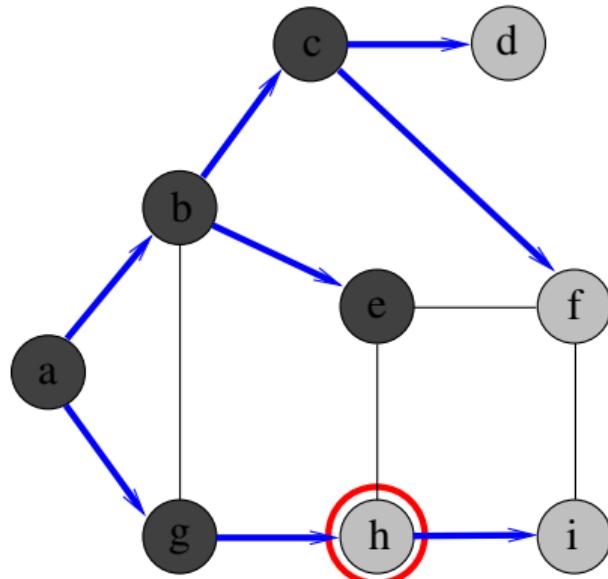
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$\begin{aligned}
 f &= \langle i, f, d, h \rangle \\
 s_k &= h, s_i = i
 \end{aligned}$$

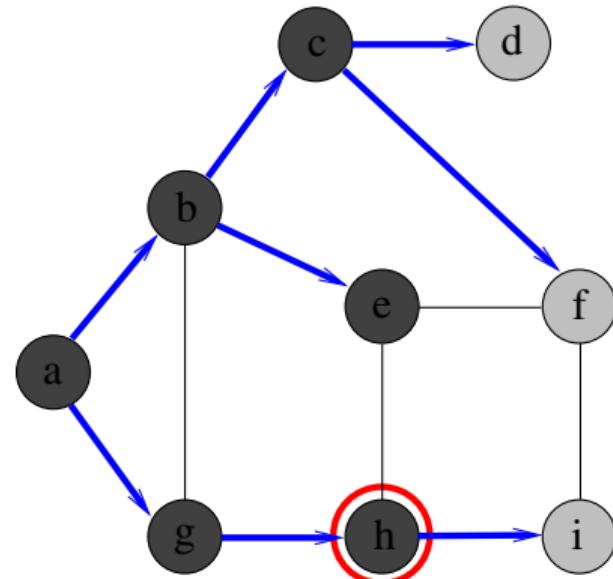
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



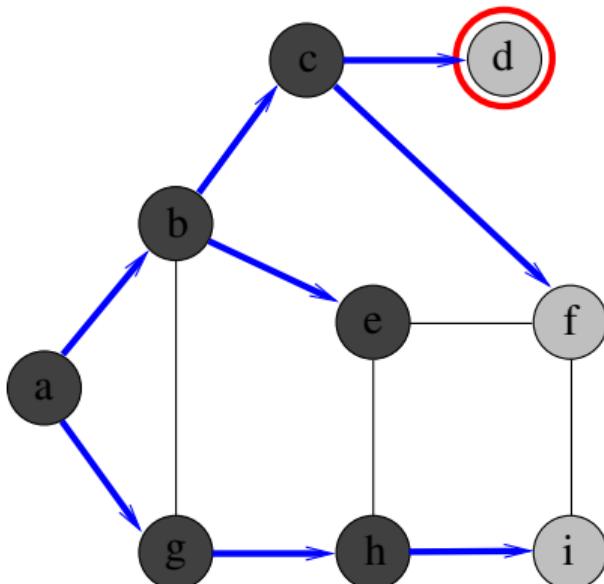
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$\begin{aligned}
 f &= \langle i, f, d \rangle \\
 s_k &= d
 \end{aligned}$$

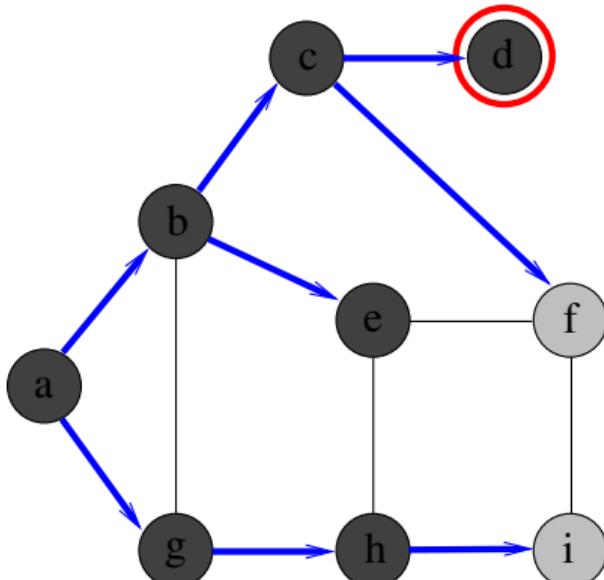
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$\begin{aligned}
 f &= \langle i, f \rangle \\
 s_k &= d
 \end{aligned}$$

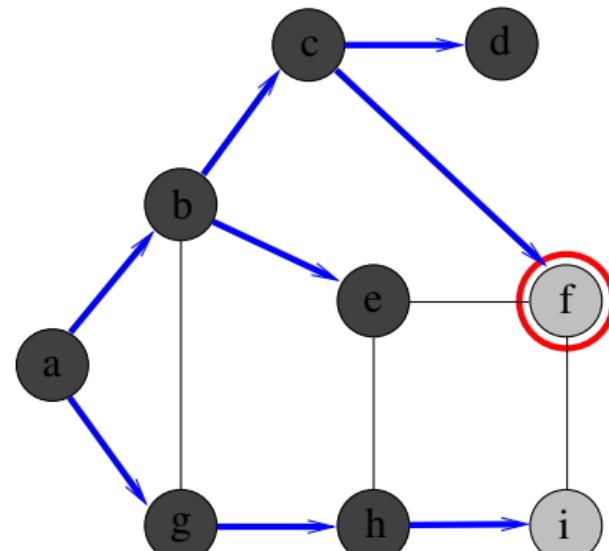
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$f = \langle i, f \rangle$$

$$s_k = f$$

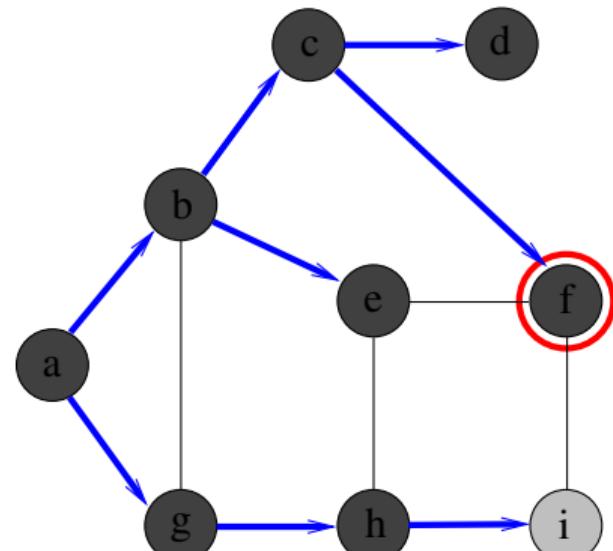
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$f = < i >$$

$$s_k = f$$

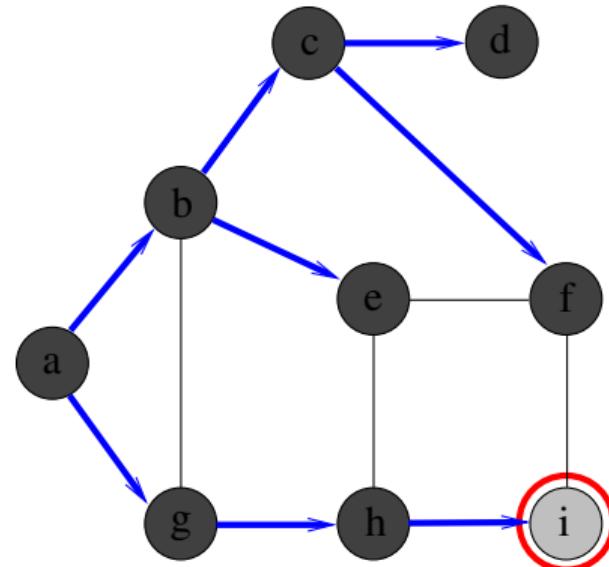
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$\begin{aligned}f &= \langle i \rangle \\s_k &= i\end{aligned}$$

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

Parcours en largeur (Breadth First Search / BFS)

1 Fonction $BFS(g, s_0)$

```

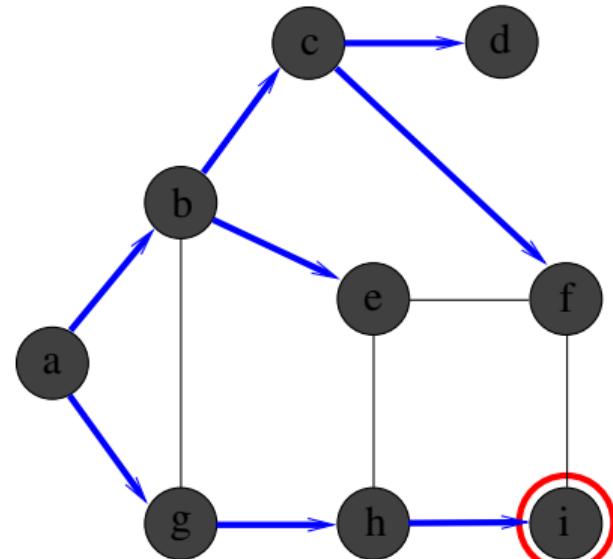
2   Soit  $f$  une file (FIFO) initialisée à vide
3   pour chaque sommet  $s_i$  de  $g$  faire
4        $\pi[s_i] \leftarrow null$ 
5       Colorier  $s_i$  en blanc

```

```

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       pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10          Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11           $\pi[s_i] \leftarrow s_k$ 
12      Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13
retourne  $\pi$ 

```



$$\begin{aligned} f &=<> \\ s_k &= i \end{aligned}$$

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 $\pi[s_i] \leftarrow null$

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

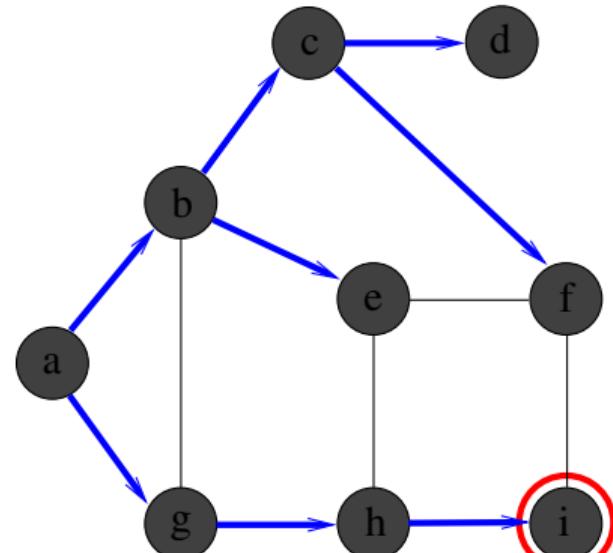
9 pour chaque $s_i \in succ(s_k)$ tq s_i est blanc faire

10 Ajouter s_i dans f et colorier s_i en gris

11 $\pi[s_i] \leftarrow s_k$

12 Enlever s_k de f et colorier s_k en noir

13 retourne π



$$f = <>$$

$$s_k = i$$

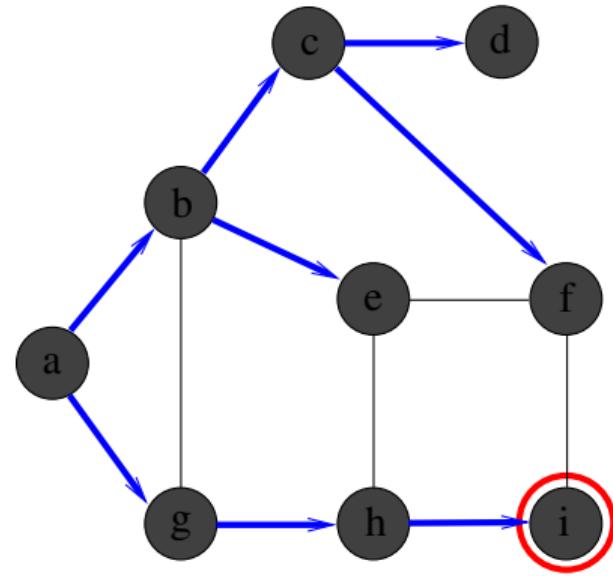
Complexité de BFS pour un graphe ayant n sommets et p arcs ?

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    $\pi[s_i] \leftarrow null$ 
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   pour chaque  $s_i \in succ(s_k)$  tq  $s_i$  est blanc faire
10    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
11     $\pi[s_i] \leftarrow s_k$ 
12 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
13 retourne  $\pi$ 

```



$$\begin{aligned}f &=<> \\ s_k &= i\end{aligned}$$

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

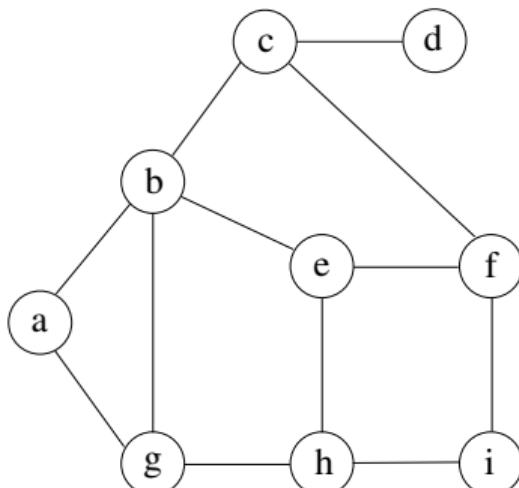
$\sim \mathcal{O}(n + p)$ (sous réserve d'une implémentation par listes d'adjacence)

Utilisation de BFS : Recherche de plus courts chemins

Définition : Soient s_0 et s_i deux sommets

- Plus court chemin entre s_0 et s_i = chemin de longueur minimale
- Distance entre s_0 et s_i = longueur du plus court chemin entre s_0 et s_i , noté $\delta(s_0, s_i)$
~ $\delta(s_0, s_i) = \infty$ s'il n'existe pas de chemin de s_0 jusque s_i

Exemple :

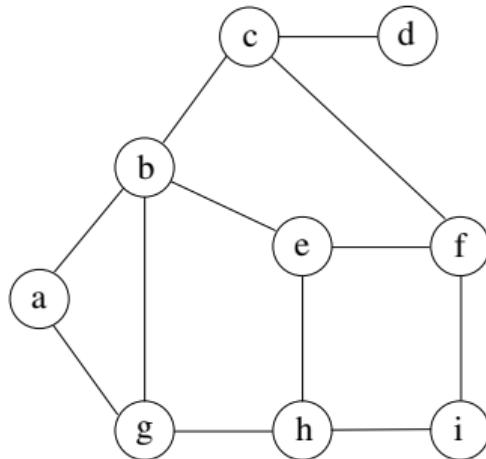


- $\delta(a, a) = 0$
- $\delta(a, b) = \delta(a, g) = 1$
- $\delta(a, c) = \delta(a, e) = \delta(a, h) = 2$
- $\delta(a, d) = \delta(a, f) = \delta(a, i) = 3$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



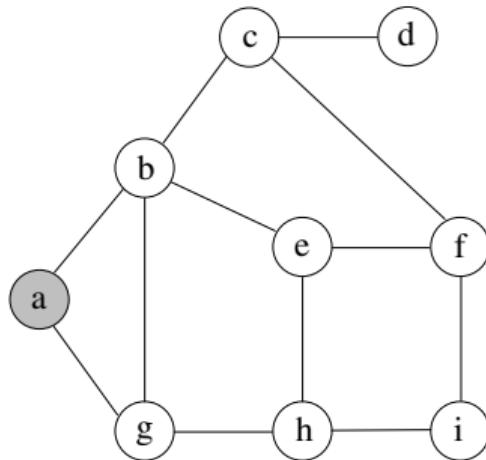
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



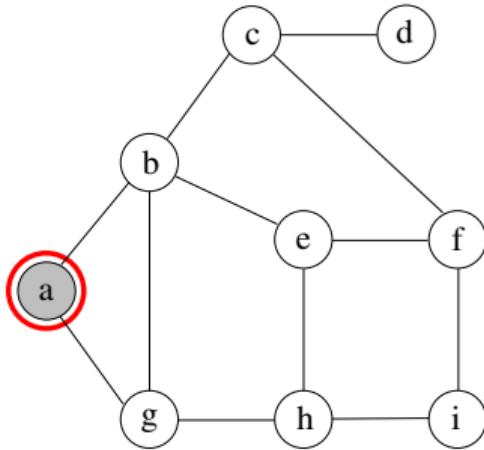
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $< s_1, s_2, \dots, s_k >$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$f = \langle a \rangle \\ d[a] = 0$$

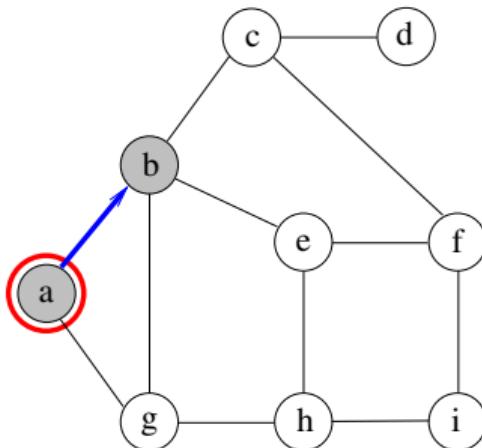
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux : $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$f = \langle b, a \rangle \\ d[a] = 0, d[b] = 1$$

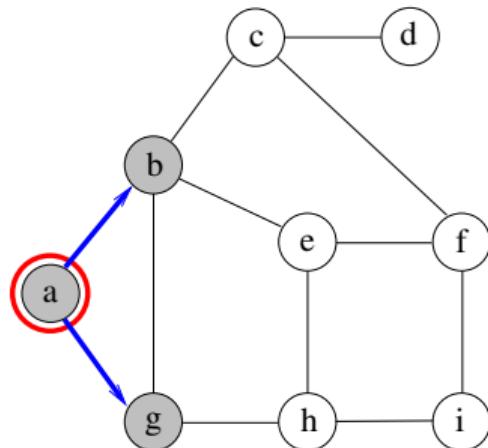
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux : $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$f = \langle g, b, a \rangle$$

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

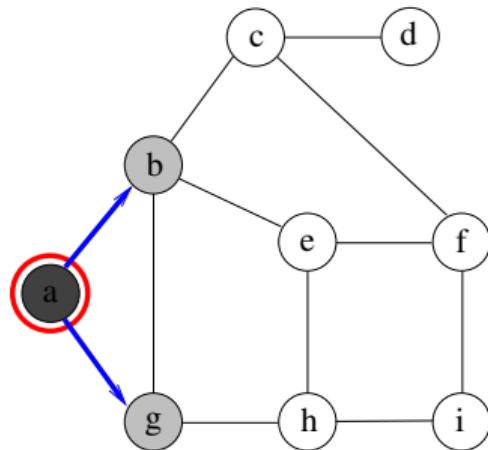
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$f = \langle g, b \rangle \\ d[a] = 0, d[b] = 1, d[g] = 1$$

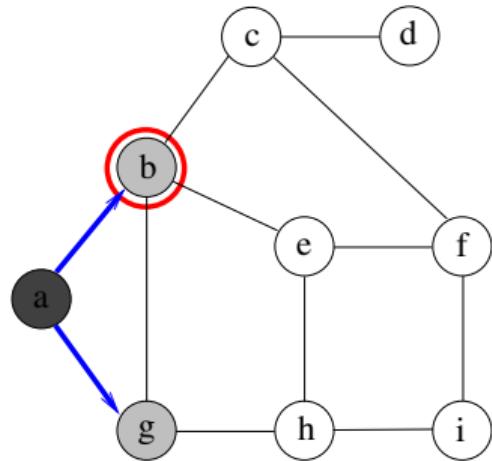
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$f = \langle g, b \rangle \\ d[a] = 0, d[b] = 1, d[g] = 1$$

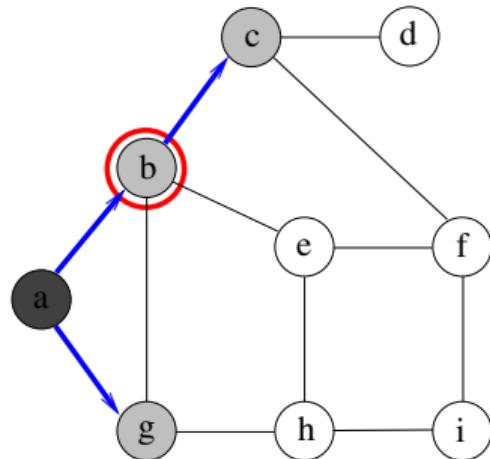
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux : $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$\begin{aligned}
f &= \langle c, g, b \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2
\end{aligned}$$

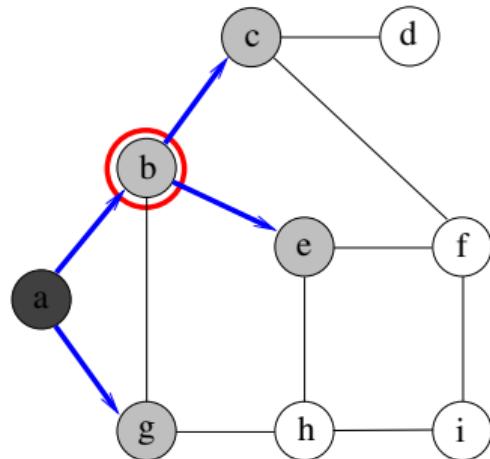
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$f = \langle e, c, g, b \rangle$
 $d[a] = 0, d[b] = 1, d[g] = 1,$
 $d[c] = 2, d[e] = 2$

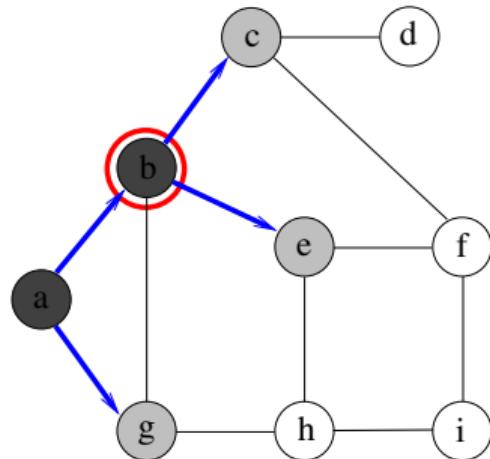
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$\begin{aligned}
f &= \langle e, c, g \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2, d[e] = 2
\end{aligned}$$

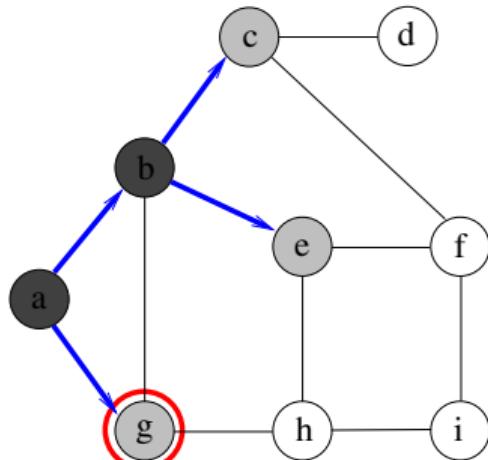
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$f = \langle e, c, g \rangle$
 $d[a] = 0, d[b] = 1, d[g] = 1,$
 $d[c] = 2, d[e] = 2$

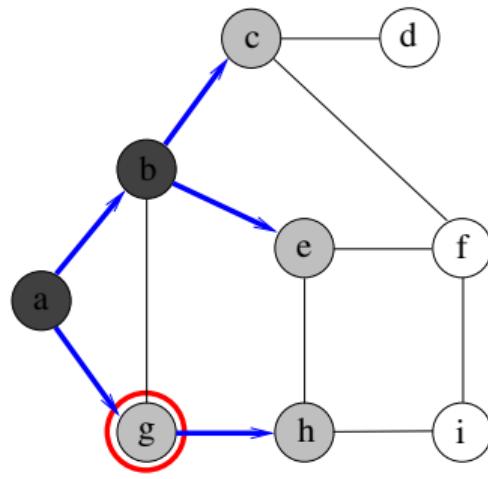
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$\begin{aligned}
f &= \langle h, e, c, g \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2, d[e] = 2, d[h] = 2
\end{aligned}$$

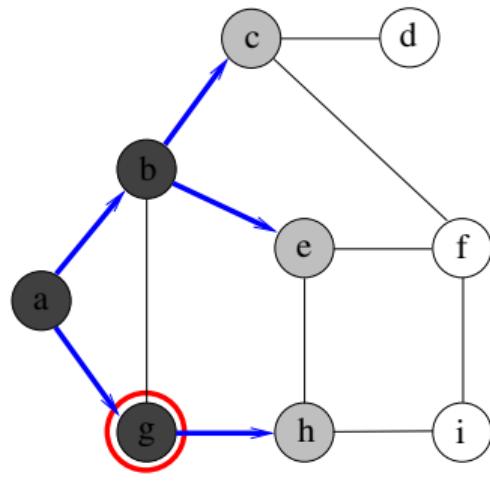
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$\begin{aligned}
f &= \langle h, e, c \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2, d[e] = 2, d[h] = 2
\end{aligned}$$

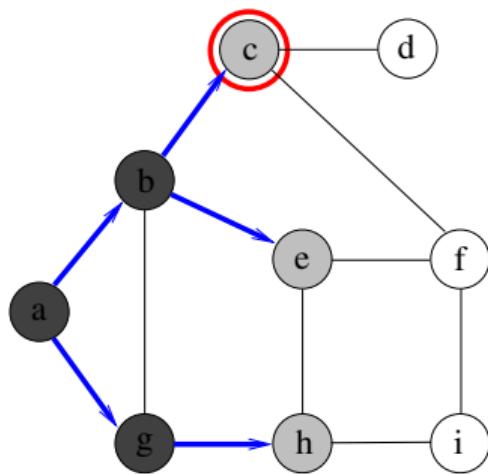
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$\begin{aligned}
f &= \langle h, e, c \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2, d[e] = 2, d[h] = 2
\end{aligned}$$

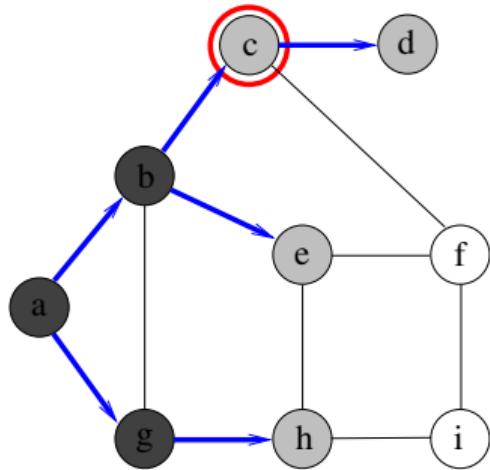
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$\begin{aligned}
f &= \langle d, h, e, c \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2, d[e] = 2, d[h] = 2, \\
d[d] &= 3
\end{aligned}$$

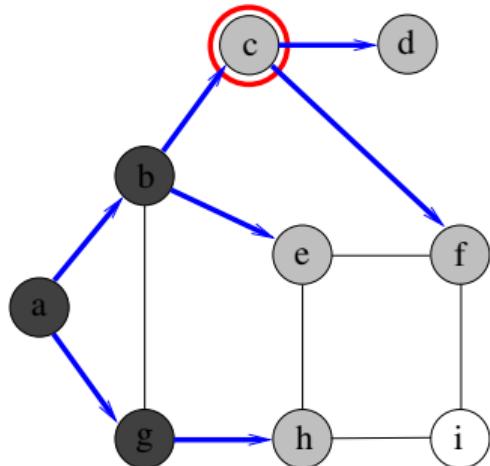
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14  Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15  retourne  $d$ 

```



$$\begin{aligned}
f &= \langle f, d, h, e, c \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2, d[e] = 2, d[h] = 2, \\
d[d] &= 3, d[f] = 3
\end{aligned}$$

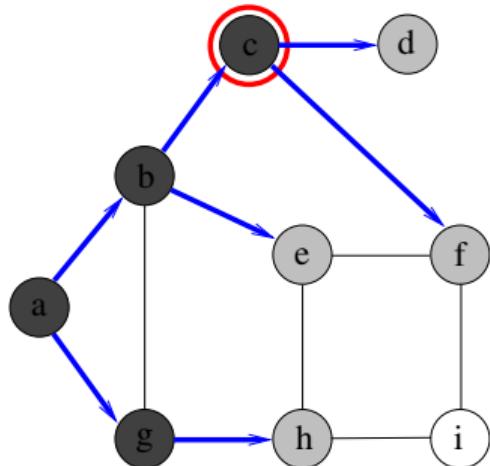
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux : $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14  Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15  retourne  $d$ 

```



$$\begin{aligned}
f &= \langle f, d, h, e \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2, d[e] = 2, d[h] = 2, \\
d[d] &= 3, d[f] = 3
\end{aligned}$$

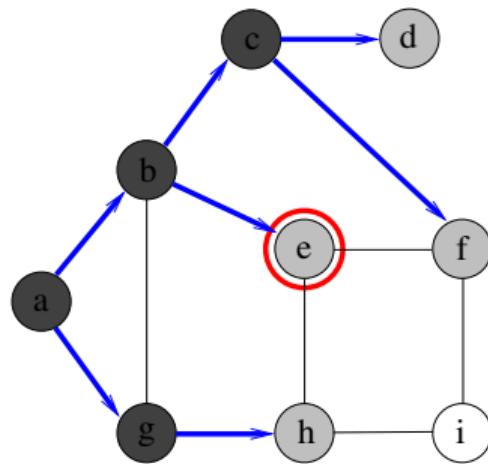
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$\begin{aligned}
f &= \langle f, d, h, e \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2, d[e] = 2, d[h] = 2, \\
d[d] &= 3, d[f] = 3
\end{aligned}$$

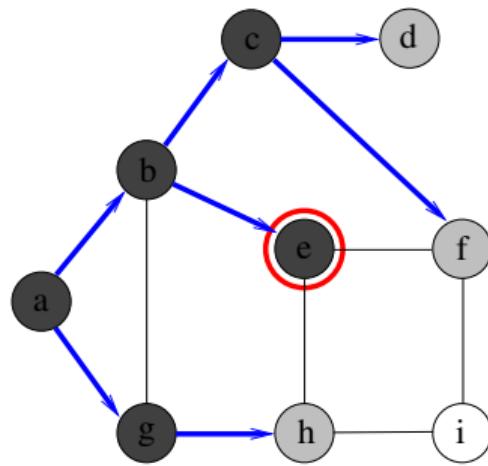
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$\begin{aligned}
f &= \langle f, d, h \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2, d[e] = 2, d[h] = 2, \\
d[d] &= 3, d[f] = 3
\end{aligned}$$

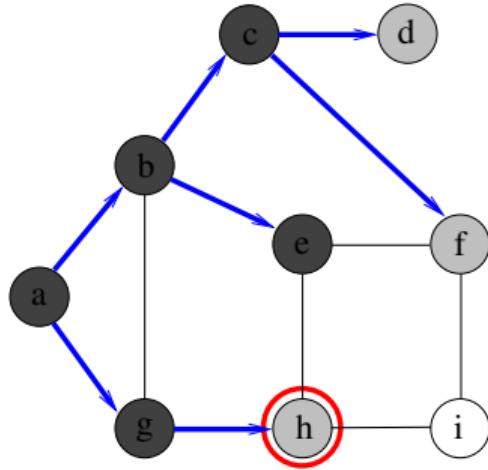
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$\begin{aligned}
f &= \langle f, d, h \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2, d[e] = 2, d[h] = 2, \\
d[d] &= 3, d[f] = 3
\end{aligned}$$

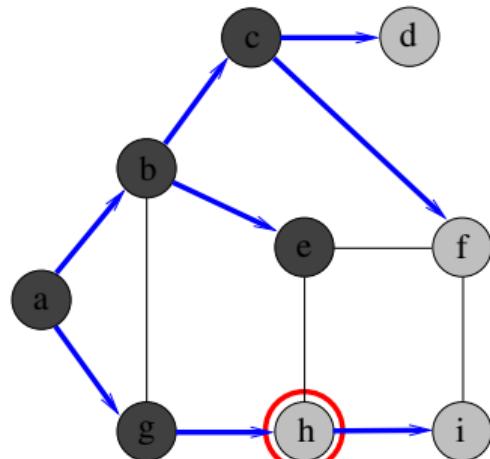
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2   pour chaque sommet  $s_i$  de  $g$  faire
3      $\pi[s_i] \leftarrow \text{null}$ 
4     Colorier  $s_i$  en blanc
5      $d[s_i] \leftarrow \infty$ 
6   Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8   tant que  $f$  n'est pas vide faire
9     Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10    pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11      Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12       $d[s_i] \leftarrow d[s_k] + 1$ 
13       $\pi[s_i] \leftarrow s_k$ 
14    Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15  retourne  $d$ 

```



$$\begin{aligned}
f &= \langle i, f, d, h \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2, d[e] = 2, d[h] = 2, \\
d[d] &= 3, d[f] = 3, d[i] = 3
\end{aligned}$$

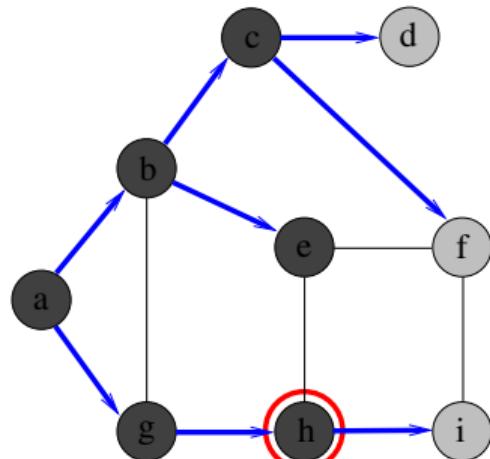
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_1] \leq d[s_2] \leq \dots \leq d[s_k] \leq d[s_1] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2   pour chaque sommet  $s_i$  de  $g$  faire
3      $\pi[s_i] \leftarrow \text{null}$ 
4     Colorier  $s_i$  en blanc
5      $d[s_i] \leftarrow \infty$ 
6   Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8   tant que  $f$  n'est pas vide faire
9     Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10    pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11      Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12       $d[s_i] \leftarrow d[s_k] + 1$ 
13       $\pi[s_i] \leftarrow s_k$ 
14    Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15  retourne  $d$ 

```



$$\begin{aligned}
f &= \langle i, f, d \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2, d[e] = 2, d[h] = 2, \\
d[d] &= 3, d[f] = 3, d[i] = 3
\end{aligned}$$

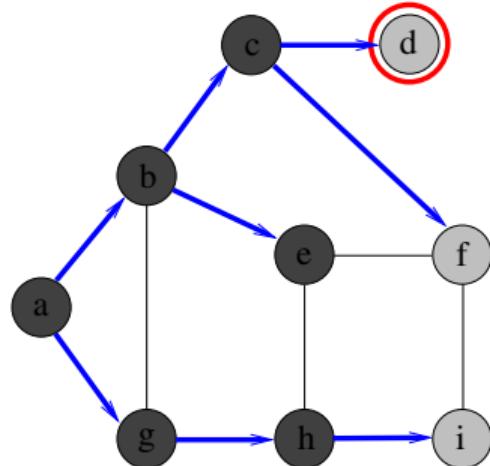
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_1] \leq d[s_2] \leq \dots \leq d[s_k] \leq d[s_1] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2   pour chaque sommet  $s_i$  de  $g$  faire
3      $\pi[s_i] \leftarrow \text{null}$ 
4     Colorier  $s_i$  en blanc
5      $d[s_i] \leftarrow \infty$ 
6   Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8   tant que  $f$  n'est pas vide faire
9     Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10    pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11      Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12       $d[s_i] \leftarrow d[s_k] + 1$ 
13       $\pi[s_i] \leftarrow s_k$ 
14    Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15  retourne  $d$ 

```



$$\begin{aligned}
f &= \langle i, f, d \rangle \\
d[a] &= 0, d[b] = 1, d[g] = 1, \\
d[c] &= 2, d[e] = 2, d[h] = 2, \\
d[d] &= 3, d[f] = 3, d[i] = 3
\end{aligned}$$

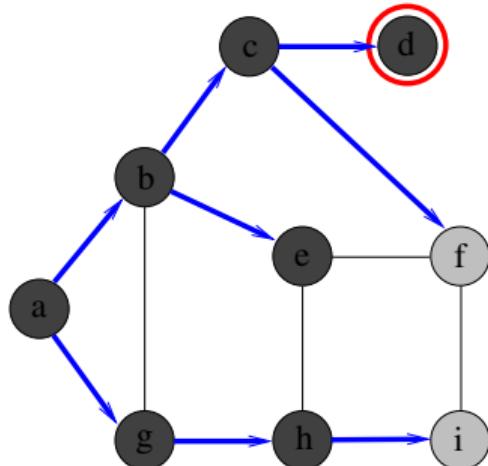
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_1] \leq d[s_2] \leq \dots \leq d[s_k] \leq d[s_1] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$f = \langle i, f \rangle$$

$$\begin{aligned}d[a] &= 0, d[b] = 1, d[g] = 1, \\d[c] &= 2, d[e] = 2, d[h] = 2, \\d[d] &= 3, d[f] = 3, d[i] = 3\end{aligned}$$

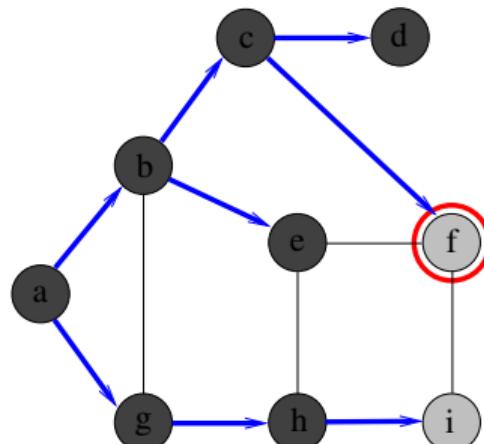
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux : $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14  Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15  retourne  $d$ 

```



$$f = \langle i, f \rangle$$

$$\begin{aligned}d[a] &= 0, d[b] = 1, d[g] = 1, \\d[c] &= 2, d[e] = 2, d[h] = 2, \\d[d] &= 3, d[f] = 3, d[i] = 3\end{aligned}$$

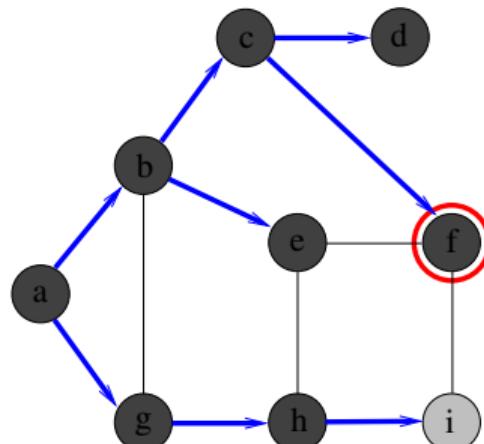
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14  Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15  retourne  $d$ 

```



$$f = \langle i \rangle$$

$$\begin{aligned}d[a] &= 0, d[b] = 1, d[g] = 1, \\d[c] &= 2, d[e] = 2, d[h] = 2, \\d[d] &= 3, d[f] = 3, d[i] = 3\end{aligned}$$

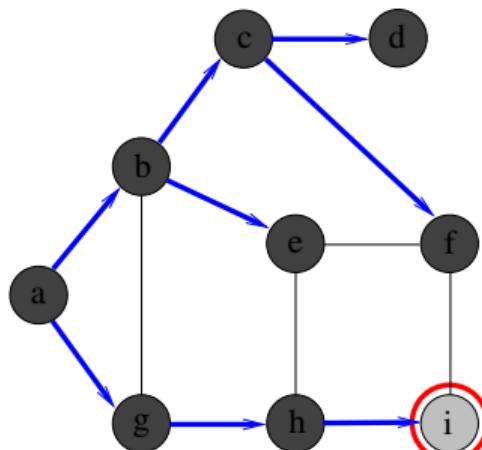
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$f = \langle i \rangle$$

$$\begin{aligned}d[a] &= 0, d[b] = 1, d[g] = 1, \\d[c] &= 2, d[e] = 2, d[h] = 2, \\d[d] &= 3, d[f] = 3, d[i] = 3\end{aligned}$$

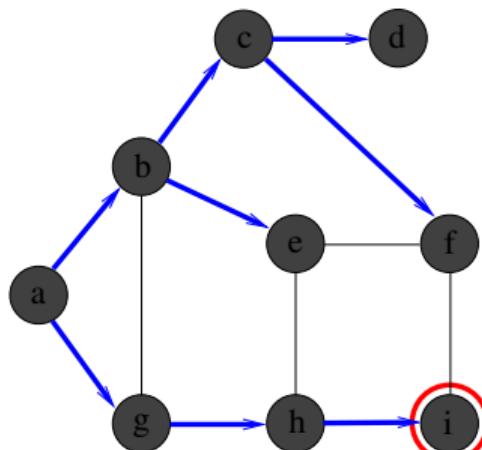
Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

```

1 Fonction calculeDistances( $g, s_0$ )
2 pour chaque sommet  $s_i$  de  $g$  faire
3    $\pi[s_i] \leftarrow \text{null}$ 
4   Colorier  $s_i$  en blanc
5    $d[s_i] \leftarrow \infty$ 
6 Ajouter  $s_0$  dans  $f$  et colorier  $s_0$  en gris
7    $d[s_0] \leftarrow 0$ 
8 tant que  $f$  n'est pas vide faire
9   Soit  $s_k$  le sommet le plus ancien dans  $f$ 
10  pour chaque  $s_i \in \text{succ}(s_k)$  tq  $s_i$  est blanc faire
11    Ajouter  $s_i$  dans  $f$  et colorier  $s_i$  en gris
12     $d[s_i] \leftarrow d[s_k] + 1$ 
13     $\pi[s_i] \leftarrow s_k$ 
14 Enlever  $s_k$  de  $f$  et colorier  $s_k$  en noir
15 retourne  $d$ 

```



$$f = \langle \rangle$$

$$\begin{aligned}d[a] &= 0, d[b] = 1, d[g] = 1, \\d[c] &= 2, d[e] = 2, d[h] = 2, \\d[d] &= 3, d[f] = 3, d[i] = 3\end{aligned}$$

Preuve : propriétés invariantes à la ligne 9

- ➊ Aucun successeur d'un sommet noir n'est blanc
- ➋ Pour tout sommet s_i gris ou noir, $d[s_i] = \delta(s_0, s_i)$
- ➌ Soit $\langle s_1, s_2, \dots, s_k \rangle$ les sommets de f , du plus récent au plus vieux :
 $d[s_k] \leq d[s_{k-1}] \leq \dots \leq d[s_1] \leq d[s_k] + 1$

Affichage du plus court chemin

1 Proc *plusCourtChemin*(s_0, s_j, π)

Entrée : 2 sommets s_0 et s_j , et une arborescence π

Précond. : $\pi = \text{arborescence retournée par } \text{calculeDistance}(g, s_0)$

Postcond. : 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 $\pi[s_j] = \text{null}$ alors afficher("Pas de chemin!");

4 sinon

5 plusCourtChemin($s_0, \pi[s_j], \pi$)

6 afficher(" suivi de ", s_j)

Exemple :

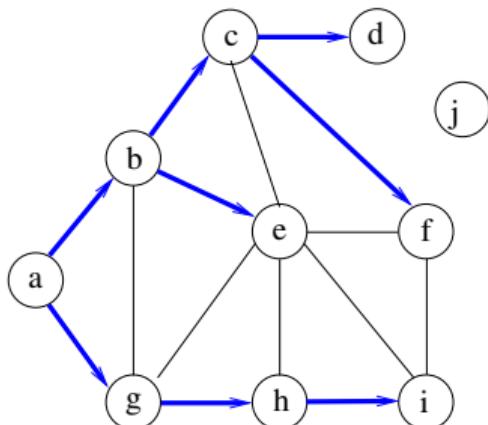


Tableau π correspondant :

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

1 Introduction

2 Structures de données pour représenter un graphe

3 Arbres Couvrants Minimaux

4 Parcours de graphes

- Parcours en largeur (BFS)
- Parcours en profondeur (DFS)

5 Plus courts chemins

6 Problèmes de planification

7 Problèmes NP-difficiles

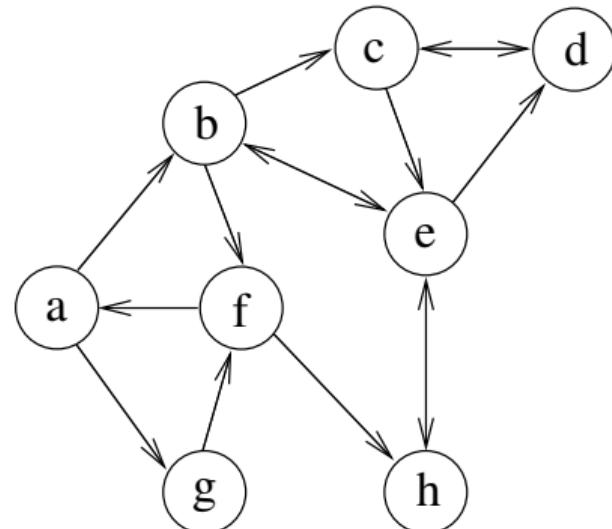
8 Conclusion

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



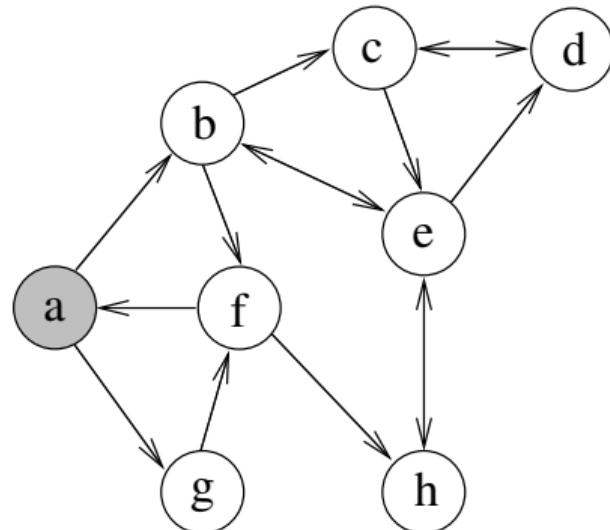
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    | Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11    |  $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    | Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



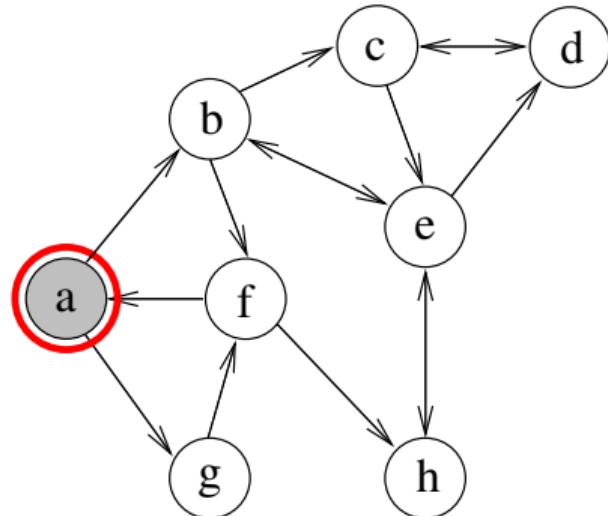
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



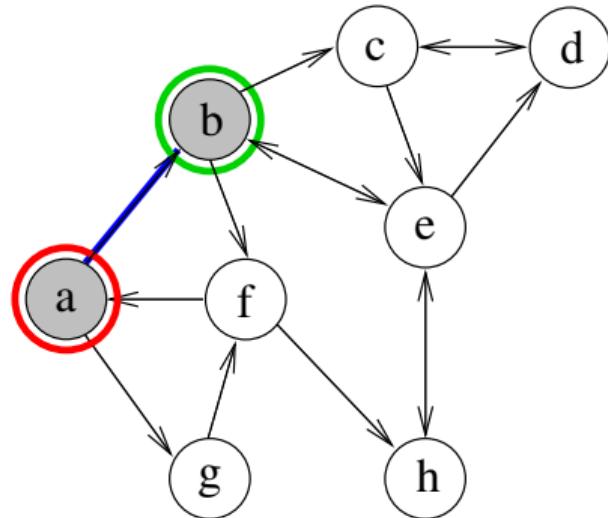
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$p = \langle a, b \rangle$$

$$s_i = a, s_j = b$$

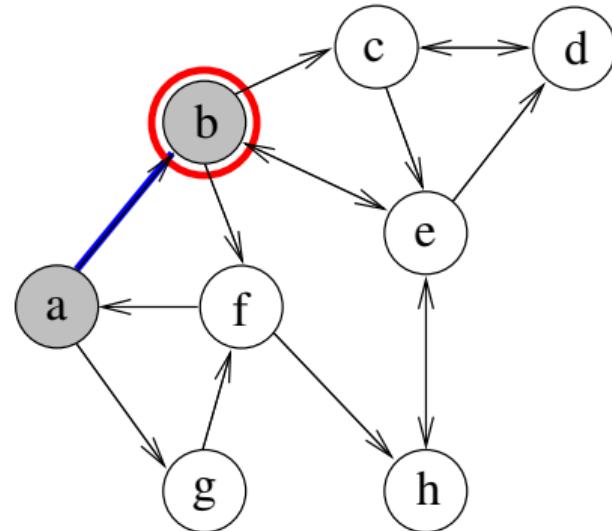
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$p = \langle a, b \rangle$$

$$s_i = b$$

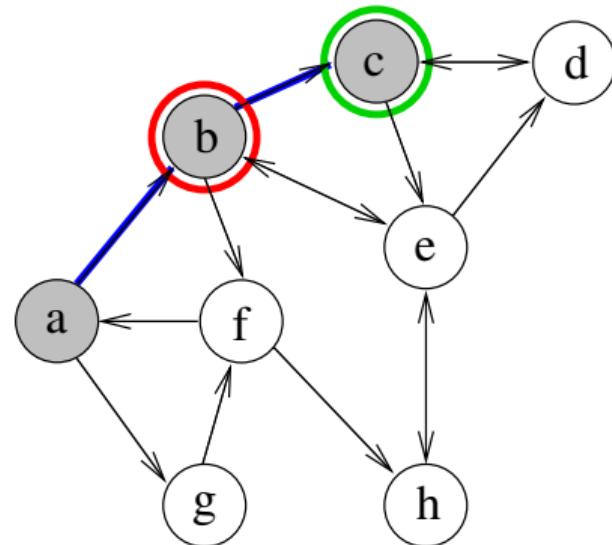
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    | Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11    |  $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    | Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



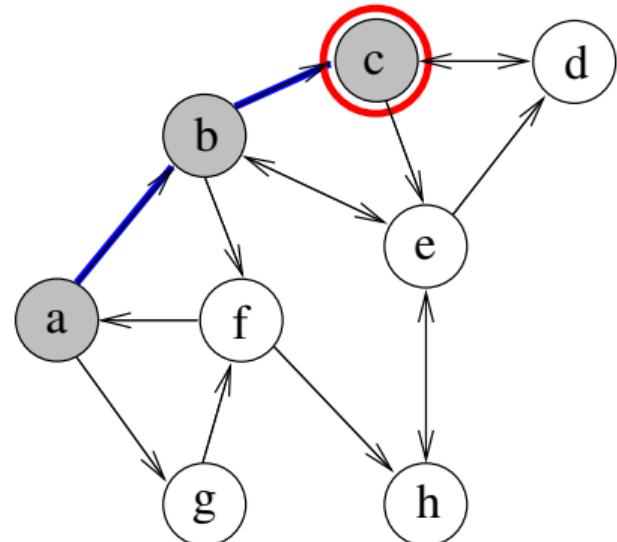
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$p = \langle a, b, c \rangle$$

$$s_i = c$$

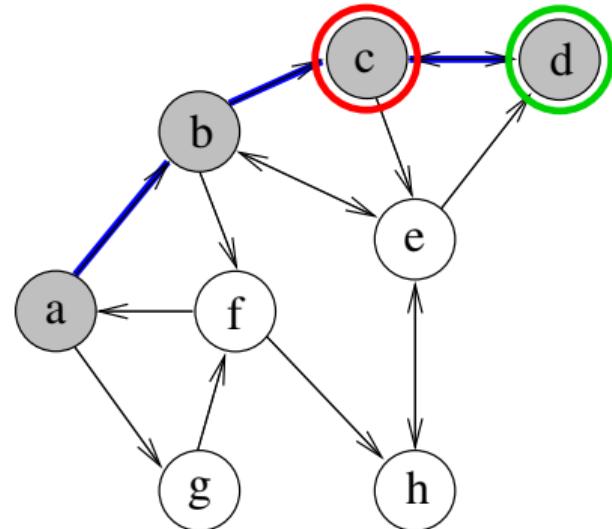
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$\begin{aligned}
p &= \langle a, b, c, d \rangle \\
s_i &= c, s_j = d
\end{aligned}$$

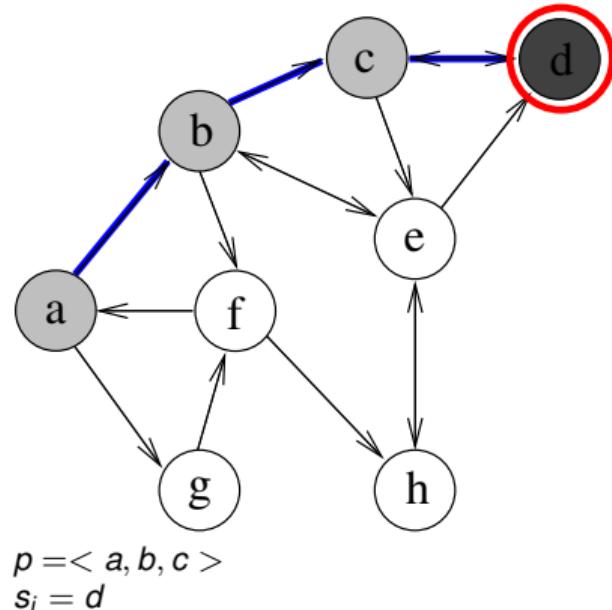
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



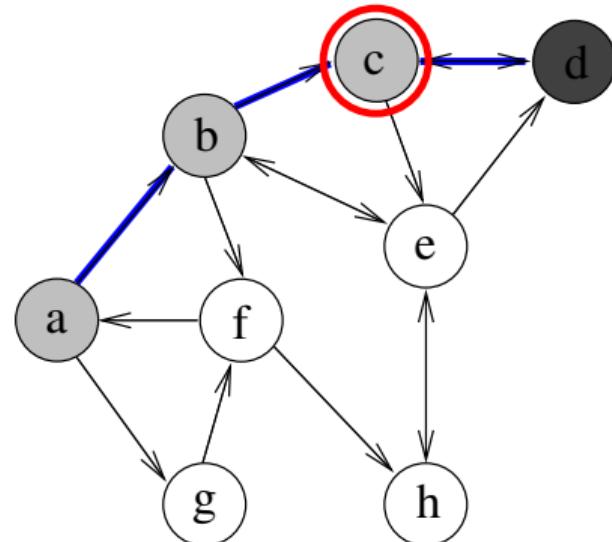
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$p = \langle a, b, c \rangle$$

$$s_i = c$$

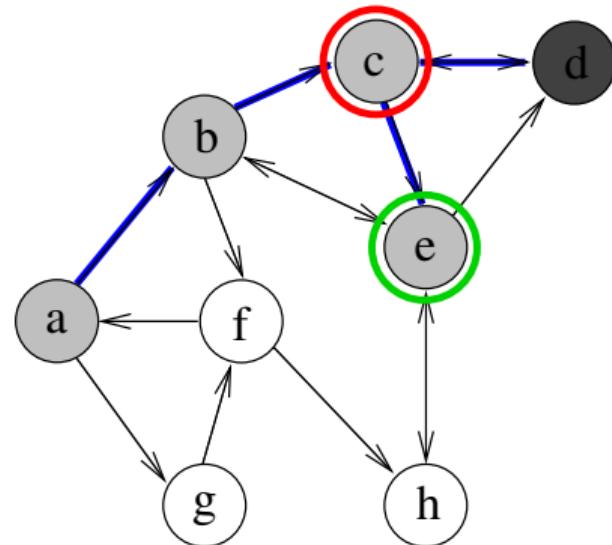
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    | Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11    |  $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    | Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$\begin{aligned} p &= \langle a, b, c, e \rangle \\ s_i &= c, s_j = e \end{aligned}$$

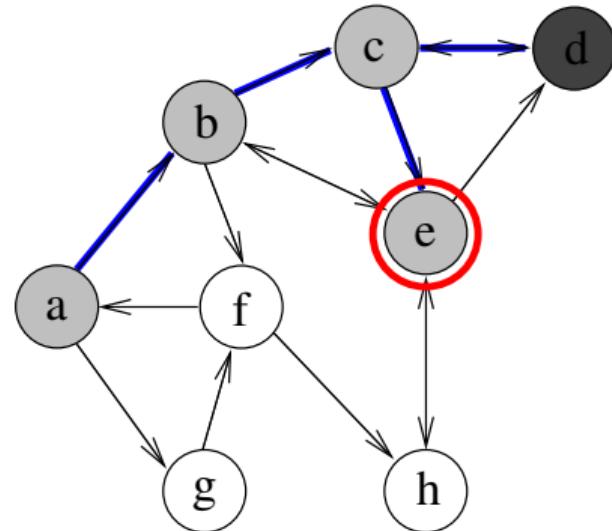
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$p = \langle a, b, c, e \rangle$$

$$s_i = e$$

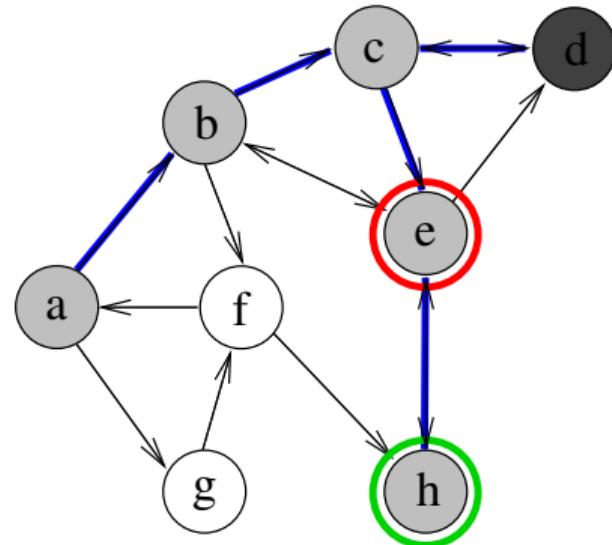
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10      Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11       $\pi[s_j] \leftarrow s_i$ 
12    sinon
13      Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



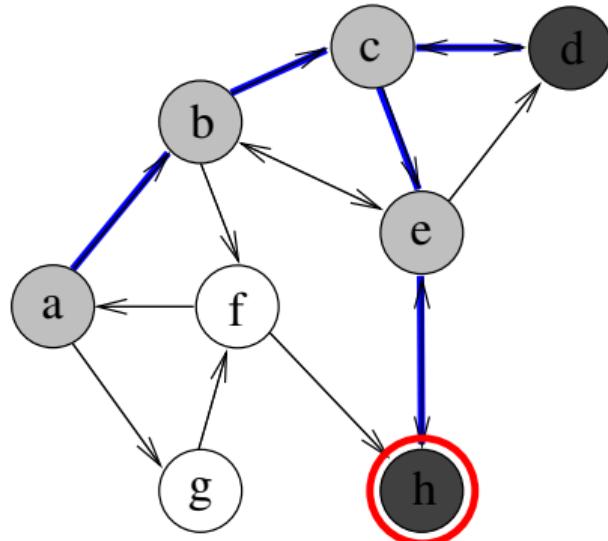
$$\begin{aligned} p &= \langle a, b, c, e, h \rangle \\ s_i &= e, s_j = h \end{aligned}$$

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10       Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11        $\pi[s_j] \leftarrow s_i$ 
12     sinon
13       Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 
```



$$\begin{aligned} p &= \langle a, b, c, e \rangle \\ s_i &= h \end{aligned}$$

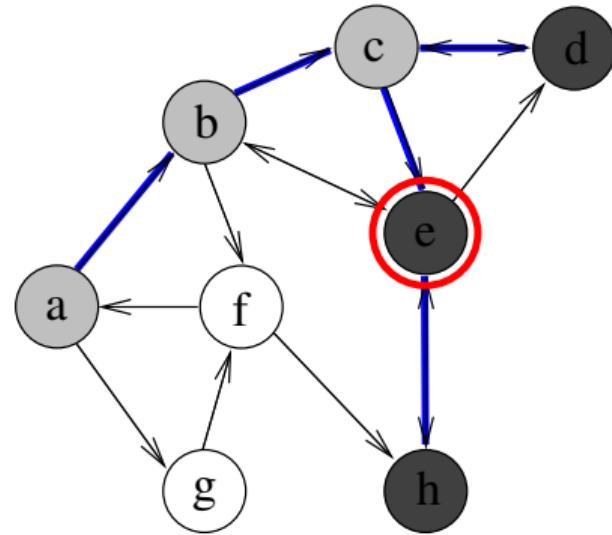
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$p = \langle a, b, c \rangle$$

$$s_i = e$$

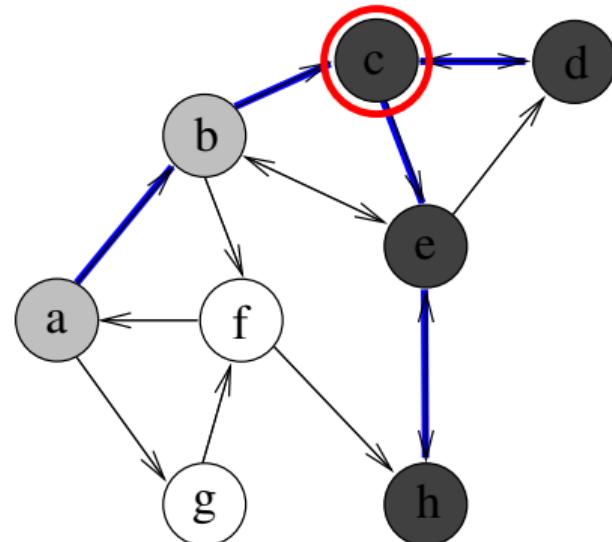
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$p = \langle a, b \rangle$$

$$s_i = c$$

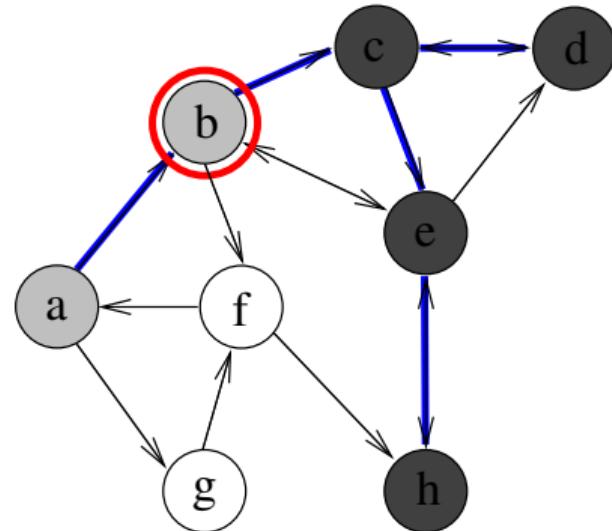
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$p = \langle a, b \rangle$$

$$s_i = b$$

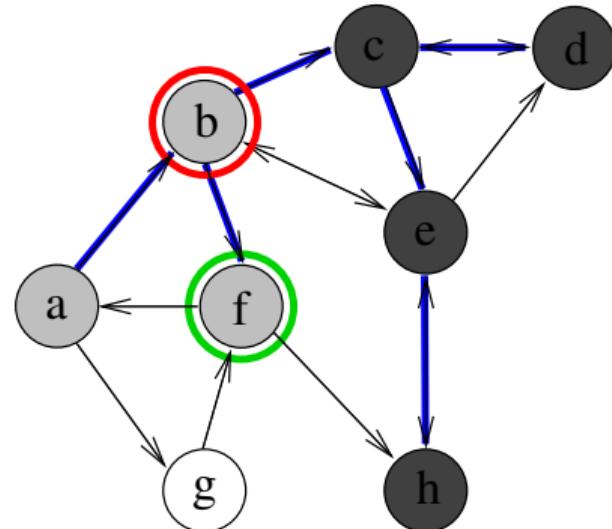
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



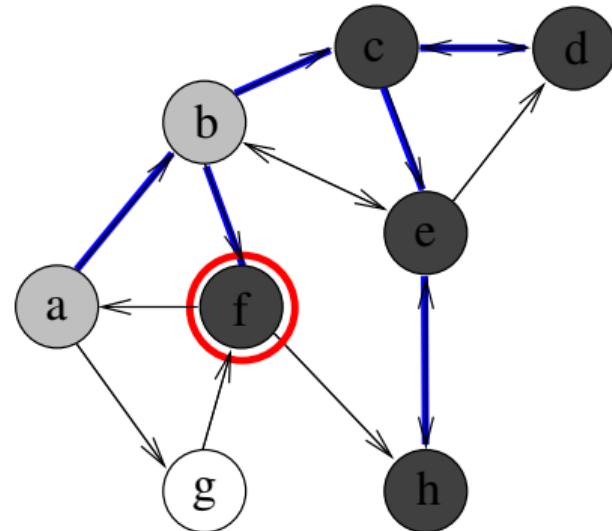
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$p = \langle a, b \rangle$$

$$s_i = f$$

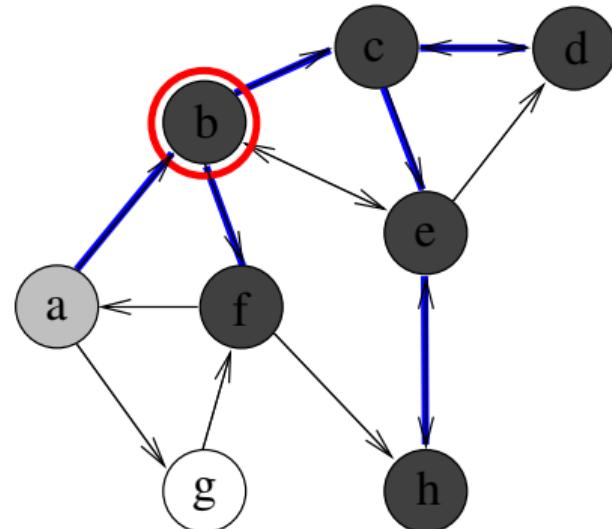
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$p = \langle a \rangle$$

$$s_i = b$$

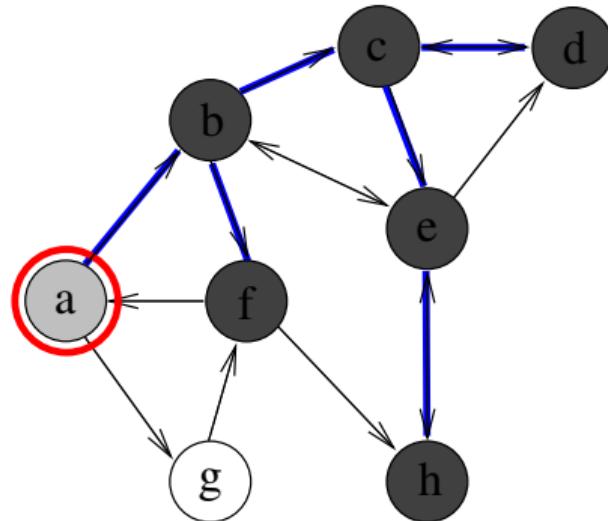
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$\begin{aligned} p &=< a > \\ s_i &= a \end{aligned}$$

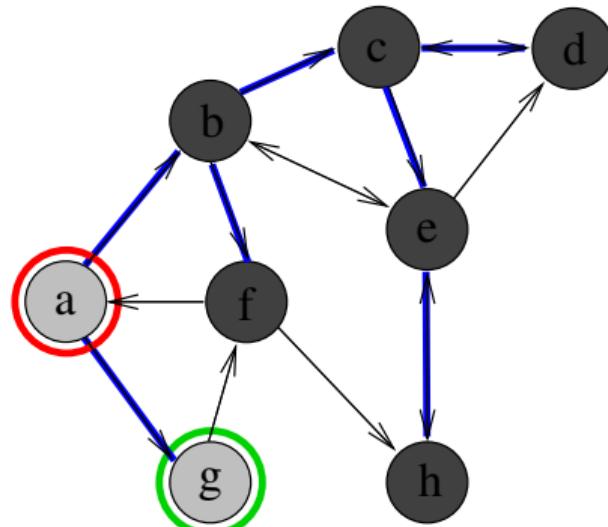
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



$$\begin{aligned}
p &= \langle a, g \rangle \\
s_i &= a, s_j = g
\end{aligned}$$

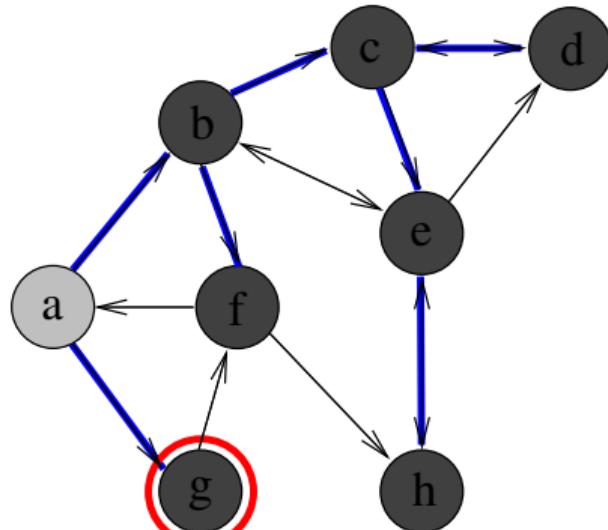
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10       Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11        $\pi[s_j] \leftarrow s_i$ 
12     sinon
13       Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



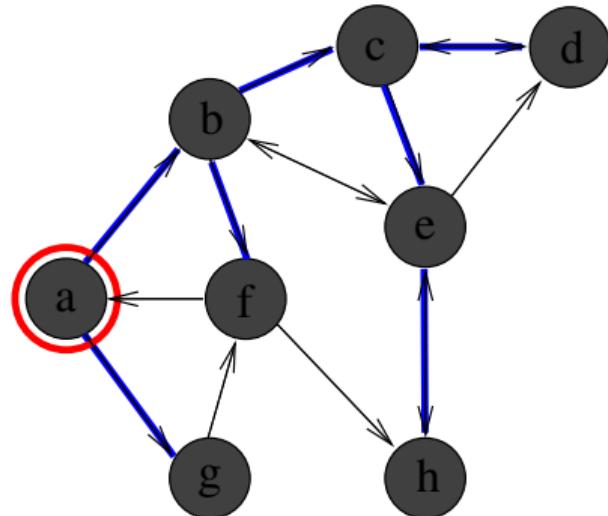
$$\begin{aligned} p &=< a > \\ s_i &= g \end{aligned}$$

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 
```



$$\begin{aligned} p &=<> \\ s_i &= a \end{aligned}$$

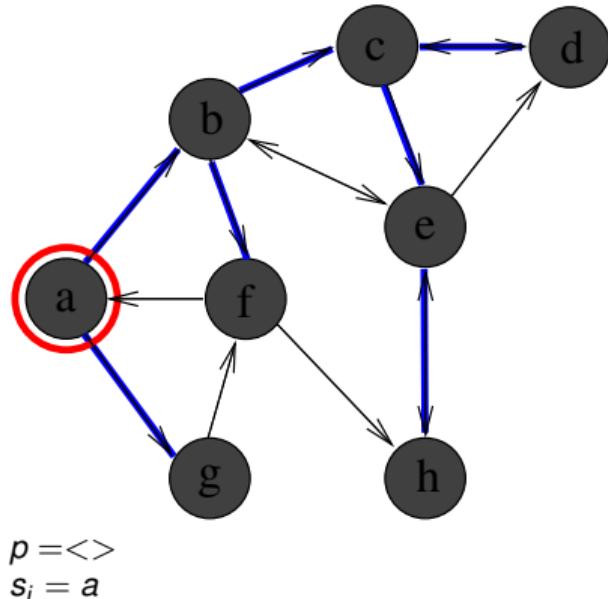
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



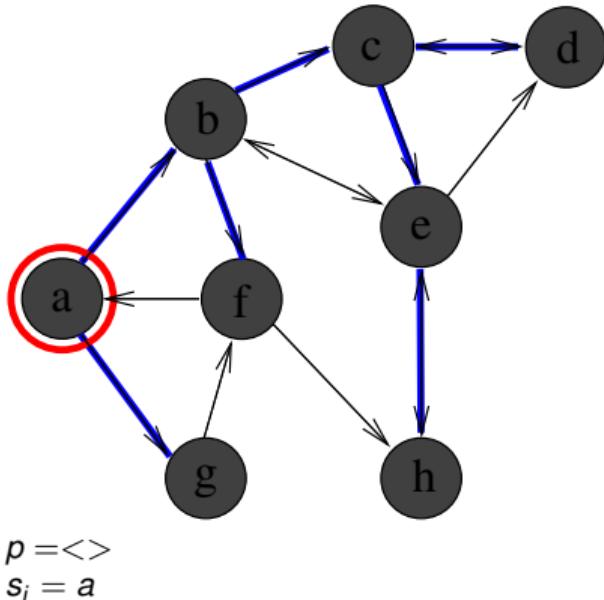
Complexité de DFS pour un graphe ayant n sommets et p arcs ?

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  $s_i \in S$  faire
4    $\pi[s_i] \leftarrow null$ 
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  $\exists s_j \in succ(s_i)$  tel que  $s_j$  soit blanc alors
10    Empiler  $s_j$  dans  $p$  et colorier  $s_j$  en gris
11     $\pi[s_j] \leftarrow s_i$ 
12  sinon
13    Dépiler  $s_i$  de  $p$  et colorier  $s_i$  en noir
14
retourne  $\pi$ 

```



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

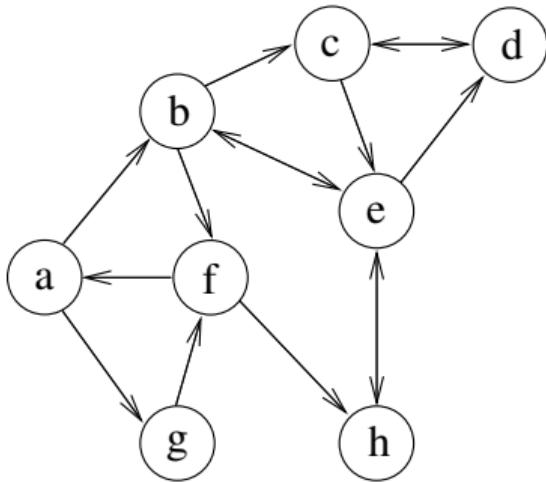
~ $\mathcal{O}(n + p)$ (sous réserve d'une implémentation par listes d'adjacence)

Version récursive de DFS

```

1 Proc DFSRec( $g, s_0$ )
Entrée : Un graphe  $g$  et un sommet  $s_0$ 
Précond. :  $s_0$  est blanc
début
3   Colorier  $s_0$  en gris
4   pour tout  $s_j \in succ(s_0)$  faire
5     si  $s_j$  est blanc alors
6        $\pi[s_j] \leftarrow s_0$ 
7       DFSRec( $g, s_j$ )
8   Colorier  $s_0$  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

Détection de circuits

1 Fonction booléen $\text{possèdeCircuit}(g, s_0)$

Entrée : Un graphe g et un sommet s_0

Sortie : Vrai si g possède un circuit ; faux sinon

Précond. : s_0 est blanc

début

Colorier s_0 en gris

pour tout $s_j \in \text{succ}(s_0)$ faire

 si s_j est gris alors retourne vrai;

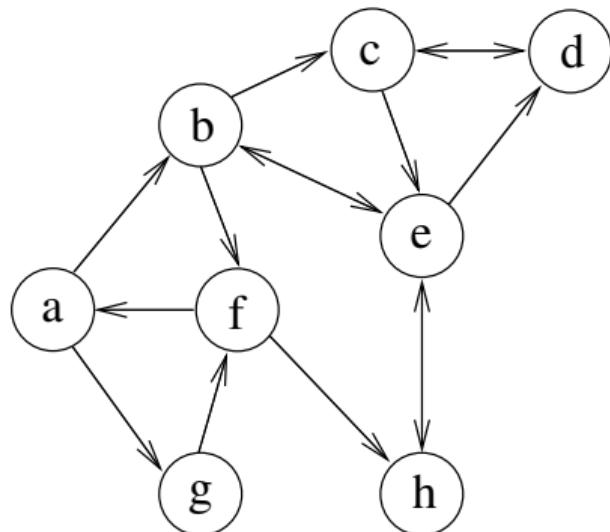
 sinon si s_j est blanc alors

 si $\text{possèdeCircuit}(g, s_j)$ alors

 retourne vrai

Colorier s_0 en noir

retourne faux



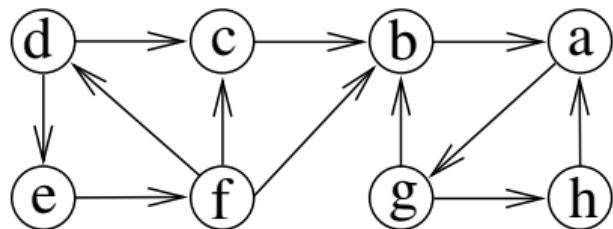
Numérotation des sommets

```

1 Proc DFSnum(g)
2   cpt  $\leftarrow$  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 Proc DFSrec(g,  $s_0$ )
7   début
8     Colorier  $s_0$  en gris
9     pour tout  $s_j \in \text{succ}(s_0)$  faire
10       si  $s_j$  est blanc alors
11          $\pi[s_j] \leftarrow s_0$ 
12         DFSrec(g,  $s_j$ )
13     Colorier  $s_0$  en noir
14     num[ $s_0$ ]  $\leftarrow$  cpt
15     cpt  $\leftarrow$  cpt + 1

```

\leadsto num = Numéro d'ordre de coloriage en noir



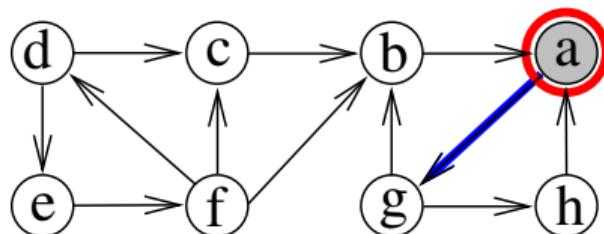
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
7 Proc DFSrec( $g, s_0$ )
8   début
9     Colorier  $s_0$  en gris
10    pour tout  $s_j \in succ(s_0)$  faire
11      si  $s_j$  est blanc alors
12         $\pi[s_j] \leftarrow s_0$ 
13        DFSrec( $g, s_j$ )
14
15      Colorier  $s_0$  en noir
16       $num[s_0] \leftarrow cpt$ 
17       $cpt \leftarrow cpt + 1$ 

```

~ num = Numéro d'ordre de coloriage en noir



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

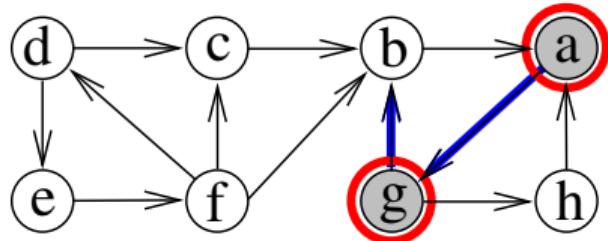
Numérotation des sommets

```

1 Proc DFSnum(g)
2   cpt  $\leftarrow$  1
3   Colorier tous les sommets en blanc
4   pour chaque sommet si de g faire
5     si si est blanc alors DFSrec(g, si);
6 Proc DFSrec(g, s0)
7   début
8     Colorier s0 en gris
9     pour tout sj  $\in$  succ(s0) faire
10    si sj est blanc alors
11       $\pi[s_j] \leftarrow s_0$ 
12      DFSrec(g, sj)
13
14    Colorier s0 en noir
15    num[s0]  $\leftarrow$  cpt
      cpt  $\leftarrow$  cpt + 1

```

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



Pile = $< a, g >$
cpt = 1
s₀ = *g*; *s_j* = *b*

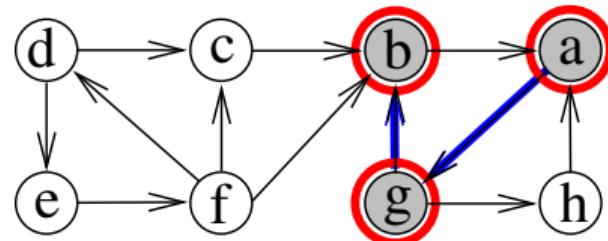
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
7 Proc DFSrec( $g, s_0$ )
8   début
9     Colorier  $s_0$  en gris
10    pour tout  $s_j \in succ(s_0)$  faire
11      si  $s_j$  est blanc alors
12         $\pi[s_j] \leftarrow s_0$ 
13        DFSrec( $g, s_j$ )
14
15      Colorier  $s_0$  en noir
16       $num[s_0] \leftarrow cpt$ 
17       $cpt \leftarrow cpt + 1$ 

```

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



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

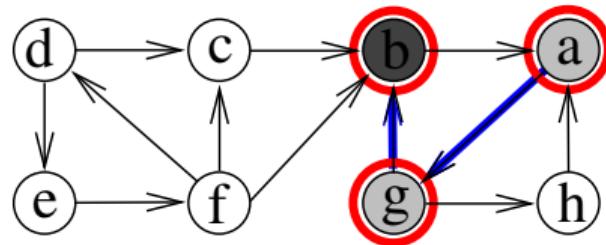
Numérotation des sommets

```

1 Proc DFSnum(g)
2   cpt  $\leftarrow$  1
3   Colorier tous les sommets en blanc
4   pour chaque sommet si de g faire
5     si si est blanc alors DFSrec(g, si);
6 Proc DFSrec(g, s0)
7   début
8     Colorier s0 en gris
9     pour tout sj  $\in$  succ(s0) faire
10    si sj est blanc alors
11       $\pi[s_j] \leftarrow s_0$ 
12      DFSrec(g, sj)
13
14    Colorier s0 en noir
15    num[s0]  $\leftarrow$  cpt
      cpt  $\leftarrow$  cpt + 1

```

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



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

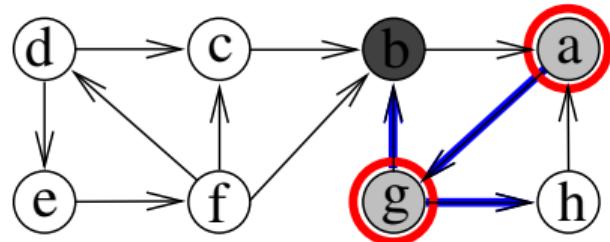
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
7 Proc DFSrec( $g, s_0$ )
8   début
9     Colorier  $s_0$  en gris
10    pour tout  $s_j \in succ(s_0)$  faire
11      si  $s_j$  est blanc alors
12         $\pi[s_j] \leftarrow s_0$ 
13        DFSrec( $g, s_j$ )
14
15      Colorier  $s_0$  en noir
16       $num[s_0] \leftarrow cpt$ 
17       $cpt \leftarrow cpt + 1$ 

```

~ num = Numéro d'ordre de coloriage en noir



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

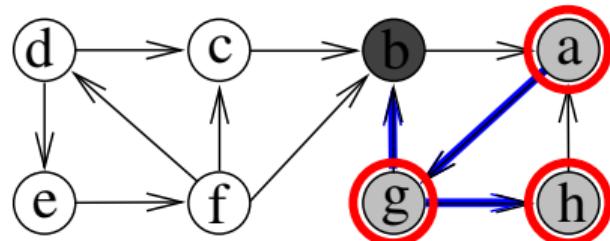
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
7 Proc DFSrec( $g, s_0$ )
8   début
9     Colorier  $s_0$  en gris
10    pour tout  $s_j \in succ(s_0)$  faire
11      si  $s_j$  est blanc alors
12         $\pi[s_j] \leftarrow s_0$ 
13        DFSrec( $g, s_j$ )
14
15      Colorier  $s_0$  en noir
16       $num[s_0] \leftarrow cpt$ 
17       $cpt \leftarrow cpt + 1$ 

```

~ num = Numéro d'ordre de coloriage en noir



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

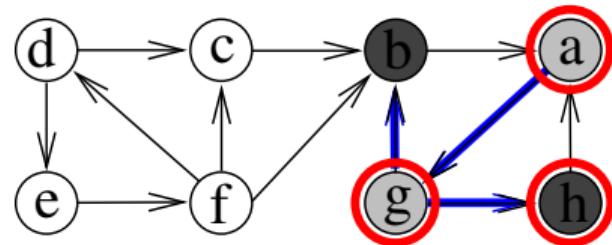
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
7 Proc DFSrec( $g, s_0$ )
8   début
9     Colorier  $s_0$  en gris
10    pour tout  $s_j \in succ(s_0)$  faire
11      si  $s_j$  est blanc alors
12         $\pi[s_j] \leftarrow s_0$ 
13        DFSrec( $g, s_j$ )
14
15      Colorier  $s_0$  en noir
16       $num[s_0] \leftarrow cpt$ 
17       $cpt \leftarrow cpt + 1$ 

```

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



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

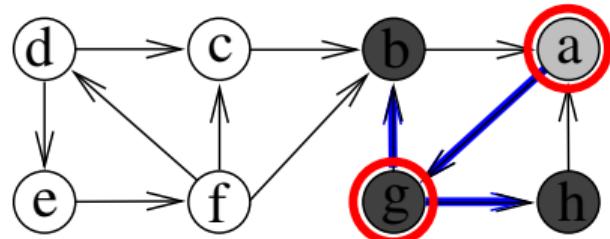
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
7 Proc DFSrec( $g, s_0$ )
8   début
9     Colorier  $s_0$  en gris
10    pour tout  $s_j \in succ(s_0)$  faire
11      si  $s_j$  est blanc alors
12         $\pi[s_j] \leftarrow s_0$ 
13        DFSrec( $g, s_j$ )
14
15      Colorier  $s_0$  en noir
16       $num[s_0] \leftarrow cpt$ 
17       $cpt \leftarrow cpt + 1$ 

```

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



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

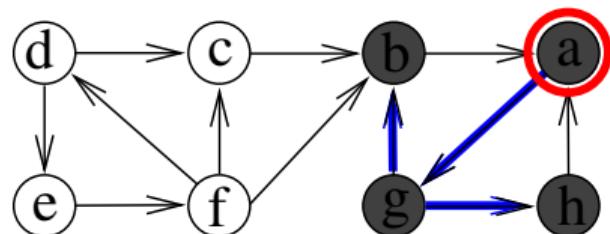
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
6 Proc DFSrec( $g, s_0$ )
7   début
8     Colorier  $s_0$  en gris
9     pour tout  $s_j \in succ(s_0)$  faire
10    si  $s_j$  est blanc alors
11       $\pi[s_j] \leftarrow s_0$ 
12      DFSrec( $g, s_j$ )
13
13   Colorier  $s_0$  en noir
14    $num[s_0] \leftarrow cpt$ 
15    $cpt \leftarrow cpt + 1$ 

```

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



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

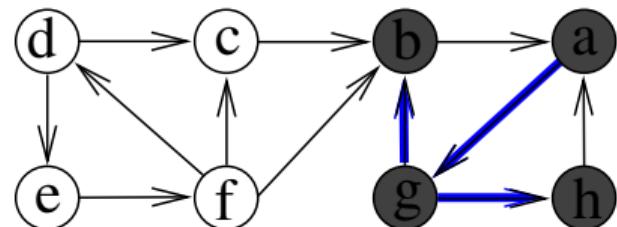
Numérotation des sommets

```

1 Proc DFSnum(g)
2   cpt  $\leftarrow$  1
3   Colorier tous les sommets en blanc
4   pour chaque sommet si de g faire
5     si si est blanc alors DFSrec(g, si);
6 Proc DFSrec(g, s0)
7   début
8     Colorier s0 en gris
9     pour tout sj  $\in$  succ(s0) faire
10    si sj est blanc alors
11       $\pi[s_j] \leftarrow s_0$ 
12      DFSrec(g, sj)
13    Colorier s0 en noir
14    num[s0]  $\leftarrow$  cpt
15    cpt  $\leftarrow$  cpt + 1

```

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



Pile = <>
cpt = 5

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

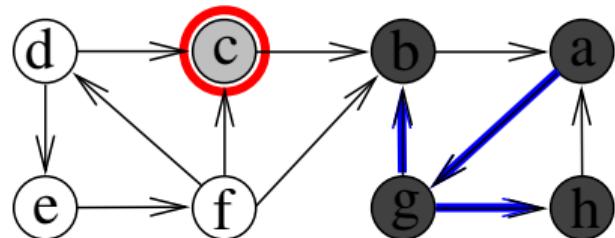
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
7 Proc DFSrec( $g, s_0$ )
8   début
9     Colorier  $s_0$  en gris
10    pour tout  $s_j \in succ(s_0)$  faire
11      si  $s_j$  est blanc alors
12         $\pi[s_j] \leftarrow s_0$ 
13        DFSrec( $g, s_j$ )
14
15      Colorier  $s_0$  en noir
16       $num[s_0] \leftarrow cpt$ 
17       $cpt \leftarrow cpt + 1$ 

```

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



Pile = < c >

$cpt = 5$

$s_0 = c$

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

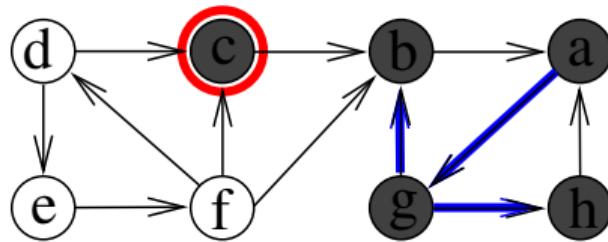
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
7 Proc DFSrec( $g, s_0$ )
8   début
9     Colorier  $s_0$  en gris
10    pour tout  $s_j \in succ(s_0)$  faire
11      si  $s_j$  est blanc alors
12         $\pi[s_j] \leftarrow s_0$ 
13        DFSrec( $g, s_j$ )
14
15      Colorier  $s_0$  en noir
16       $num[s_0] \leftarrow cpt$ 
17       $cpt \leftarrow cpt + 1$ 

```

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



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

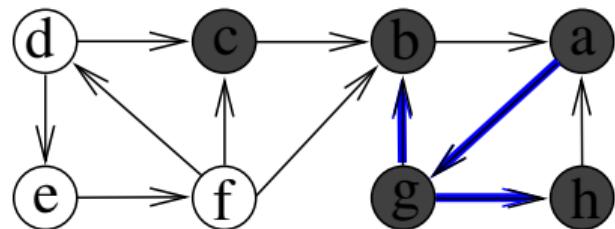
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
6 Proc DFSrec( $g, s_0$ )
7   début
8     Colorier  $s_0$  en gris
9     pour tout  $s_j \in succ(s_0)$  faire
10    si  $s_j$  est blanc alors
11       $\pi[s_j] \leftarrow s_0$ 
12      DFSrec( $g, s_j$ )
13
14    Colorier  $s_0$  en noir
15     $num[s_0] \leftarrow cpt$ 
       $cpt \leftarrow cpt + 1$ 

```

~ num = Numéro d'ordre de coloriage en noir



Pile = <>
cpt = 6

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

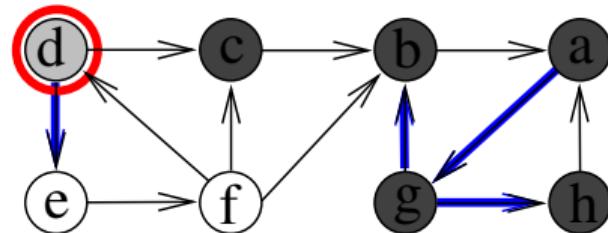
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
6 Proc DFSrec( $g, s_0$ )
7   début
8     Colorier  $s_0$  en gris
9     pour tout  $s_j \in succ(s_0)$  faire
10    si  $s_j$  est blanc alors
11       $\pi[s_j] \leftarrow s_0$ 
12      DFSrec( $g, s_j$ )
13
14    Colorier  $s_0$  en noir
15     $num[s_0] \leftarrow cpt$ 
       $cpt \leftarrow cpt + 1$ 

```

~ num = Numéro d'ordre de coloriage en noir



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

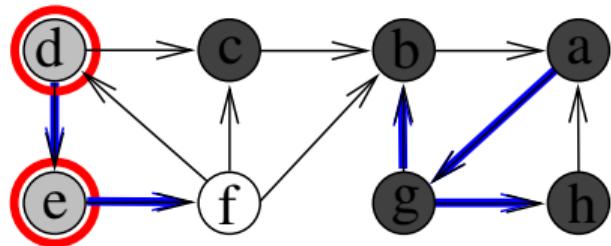
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
6 Proc DFSrec( $g, s_0$ )
7   début
8     Colorier  $s_0$  en gris
9     pour tout  $s_j \in succ(s_0)$  faire
10    si  $s_j$  est blanc alors
11       $\pi[s_j] \leftarrow s_0$ 
12      DFSrec( $g, s_j$ )
13
14    Colorier  $s_0$  en noir
15     $num[s_0] \leftarrow cpt$ 
       $cpt \leftarrow cpt + 1$ 

```

~ num = Numéro d'ordre de coloriage en noir



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

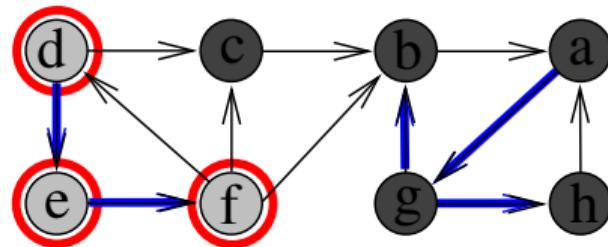
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
7 Proc DFSrec( $g, s_0$ )
8   début
9     Colorier  $s_0$  en gris
10    pour tout  $s_j \in succ(s_0)$  faire
11      si  $s_j$  est blanc alors
12         $\pi[s_j] \leftarrow s_0$ 
13        DFSrec( $g, s_j$ )
14
15      Colorier  $s_0$  en noir
16       $num[s_0] \leftarrow cpt$ 
17       $cpt \leftarrow cpt + 1$ 

```

~ num = Numéro d'ordre de coloriage en noir



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

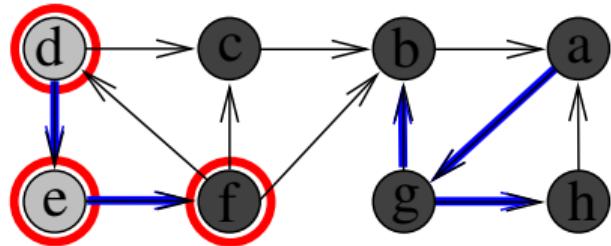
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
7 Proc DFSrec( $g, s_0$ )
8   début
9     Colorier  $s_0$  en gris
10    pour tout  $s_j \in succ(s_0)$  faire
11      si  $s_j$  est blanc alors
12         $\pi[s_j] \leftarrow s_0$ 
13        DFSrec( $g, s_j$ )
14
15      Colorier  $s_0$  en noir
16       $num[s_0] \leftarrow cpt$ 
17       $cpt \leftarrow cpt + 1$ 

```

~ num = Numéro d'ordre de coloriage en noir



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

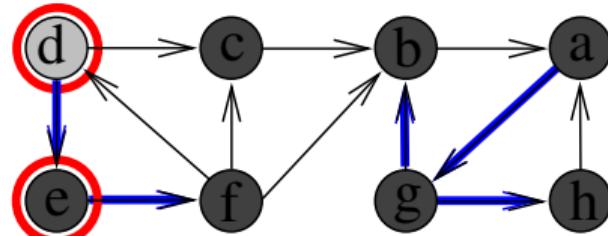
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
6 Proc DFSrec( $g, s_0$ )
7   début
8     Colorier  $s_0$  en gris
9     pour tout  $s_j \in succ(s_0)$  faire
10    si  $s_j$  est blanc alors
11       $\pi[s_j] \leftarrow s_0$ 
12      DFSrec( $g, s_j$ )
13
13   Colorier  $s_0$  en noir
14    $num[s_0] \leftarrow cpt$ 
15    $cpt \leftarrow cpt + 1$ 

```

~ num = Numéro d'ordre de coloriage en noir



Pile = $< d, e >$

$cpt = 8$

$s_0 = e$

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

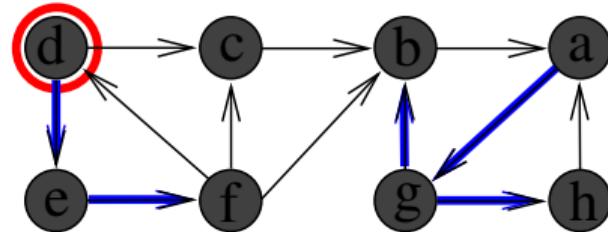
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
6 Proc DFSrec( $g, s_0$ )
7   début
8     Colorier  $s_0$  en gris
9     pour tout  $s_j \in succ(s_0)$  faire
10    si  $s_j$  est blanc alors
11       $\pi[s_j] \leftarrow s_0$ 
12      DFSrec( $g, s_j$ )
13
13   Colorier  $s_0$  en noir
14    $num[s_0] \leftarrow cpt$ 
15    $cpt \leftarrow cpt + 1$ 

```

~ num = Numéro d'ordre de coloriage en noir



Pile = < d >

$cpt = 9$

$s_0 = d$

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

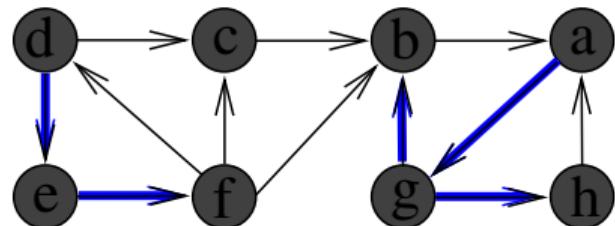
Numérotation des sommets

```

1 Proc DFSnum( $g$ )
2    $cpt \leftarrow 1$ 
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
6 Proc DFSrec( $g, s_0$ )
7   début
8     Colorier  $s_0$  en gris
9     pour tout  $s_j \in succ(s_0)$  faire
10    si  $s_j$  est blanc alors
11       $\pi[s_j] \leftarrow s_0$ 
12      DFSrec( $g, s_j$ )
13
14    Colorier  $s_0$  en noir
15     $num[s_0] \leftarrow cpt$ 
       $cpt \leftarrow cpt + 1$ 

```

~ num = Numéro d'ordre de coloriage en noir



Pile = <>
cpt = 9

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

Tri topologique d'un DAG

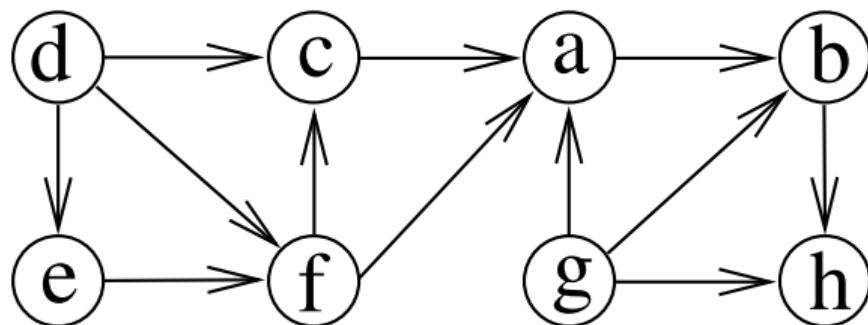
Définitions :

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

Théorème :

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

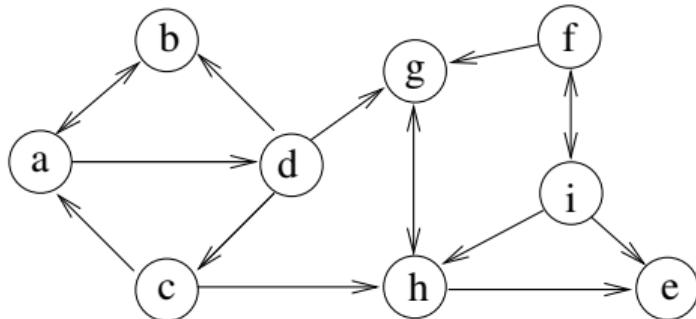
Exercice : Tri topologique du graphe suivant



Recherche des composantes fortement connexes

```
1 Proc SCC(g)
2   DFSnum(g)
3   Construire le graphe  $g^t = (S, A^t)$  tel que  $A^t = \{(s_i, s_j) \mid (s_j, s_i) \in A\}$ 
4   Colorier tous les sommets de  $g^t$  en blanc
5   pour chaque sommet  $s_i$  pris par ordre de num décroissant faire
6     si  $s_i$  est blanc alors
7       DFSrec( $g^t, s_i$ )
8       Afficher l'ensemble des sommets coloriés en noir lors du dernier appel à DFSrec
```

Exercice :



Recherche des composantes fortement connexes

```

1 Proc SCC(g)
2   DFSnum(g)
3   Construire le graphe  $g^t = (S, A^t)$  tel que  $A^t = \{(s_i, s_j) \mid (s_j, s_i) \in A\}$ 
4   Colorier tous les sommets de  $g^t$  en blanc
5   pour chaque sommet  $s_i$  pris par ordre de num décroissant faire
6     si  $s_i$  est blanc alors
7       DFSrec( $g^t, s_i$ )
8       Afficher l'ensemble des sommets coloriés en noir lors du dernier appel à DFSrec

```

Preuve de correction : Propriété vérifiée après la ligne 3

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

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

Recherche des composantes fortement connexes

```
1 Proc SCC(g)
2   DFSnum(g)
3   Construire le graphe  $g^t = (S, A^t)$  tel que  $A^t = \{(s_i, s_j) \mid (s_j, s_i) \in A\}$ 
4   Colorier tous les sommets de  $g^t$  en blanc
5   pour chaque sommet  $s_i$  pris par ordre de num décroissant faire
6     si  $s_i$  est blanc alors
7       DFSrec( $g^t, s_i$ )
8       Afficher l'ensemble des sommets coloriés en noir lors du dernier appel à DFSrec
```

Complexité pour un graphe ayant n sommets et p arcs :

- $\mathcal{O}(n + p)$ si implémentation par listes d'adjacences
- Algorithme de (Tarjan 1972) plus efficace en pratique... mais plus compliqué aussi !

Recherche des composantes fortement connexes

```
1 Proc SCC(g)
2   DFSnum(g)
3   Construire le graphe  $g^t = (S, A^t)$  tel que  $A^t = \{(s_i, s_j) \mid (s_j, s_i) \in A\}$ 
4   Colorier tous les sommets de  $g^t$  en blanc
5   pour chaque sommet  $s_i$  pris par ordre de num décroissant faire
6     si  $s_i$  est blanc alors
7       DFSrec( $g^t, s_i$ )
8       Afficher l'ensemble des sommets coloriés en noir lors du dernier appel à DFSrec
```

Complexité pour un graphe ayant n sommets et p arcs :

- $\mathcal{O}(n + p)$ si implémentation par listes d'adjacences
- Algorithme de (Tarjan 1972) plus efficace en pratique... mais plus compliqué aussi !

Recherche des composantes fortement connexes

```
1 Proc SCC(g)
2   DFSnum(g)
3   Construire le graphe  $g^t = (S, A^t)$  tel que  $A^t = \{(s_i, s_j) \mid (s_j, s_i) \in A\}$ 
4   Colorier tous les sommets de  $g^t$  en blanc
5   pour chaque sommet  $s_i$  pris par ordre de num décroissant faire
6     si  $s_i$  est blanc alors
7       DFSrec( $g^t, s_i$ )
8       Afficher l'ensemble des sommets coloriés en noir lors du dernier appel à DFSrec
```

Complexité pour un graphe ayant n sommets et p arcs :

- $\mathcal{O}(n + p)$ si implémentation par listes d'adjacences
- Algorithme de (Tarjan 1972) plus efficace en pratique... mais plus compliqué aussi !

1 Introduction

2 Structures de données pour représenter un graphe

3 Arbres Couvrants Minimaux

4 Parcours de graphes

5 Plus courts chemins

- Plus courts chemins à origine unique
- Plus courts chemins pour tout couple de sommets
- Généralisation à la recherche de meilleurs chemins

6 Problèmes de planification

7 Problèmes NP-difficiles

8 Conclusion

Définitions :

Soit $G = (S, A)$ un graphe (orienté ou non) et une fonction coût $c : A \rightarrow \mathbb{R}$

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

$$\delta(s_i, s_j) = +\infty$$

$$\delta(s_i, s_j) = -\infty$$

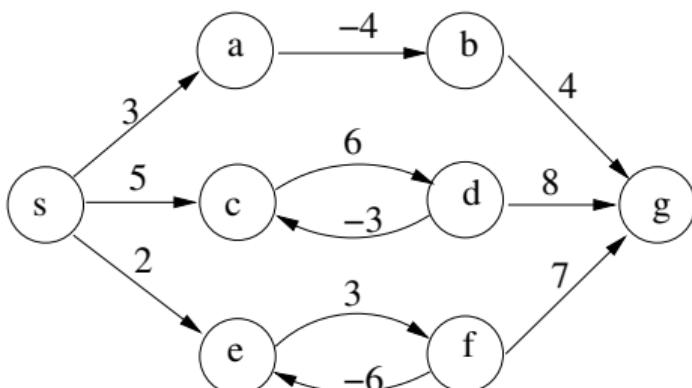
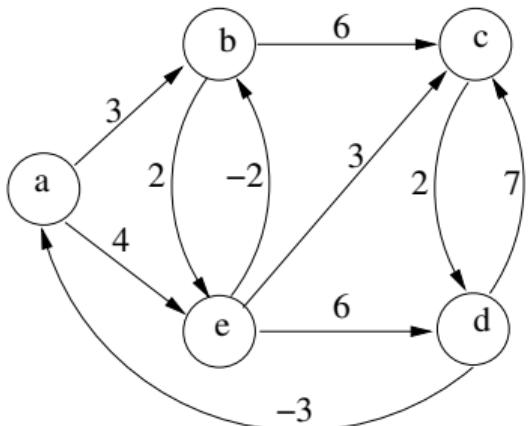
$$\delta(s_i, s_j) = \min\{c(p) | p = \text{chemin de } s_i \text{ à } s_j\}$$

s'il n'existe pas de chemin de s_i vers s_j

s'il existe un circuit absorbant

sinon

Exemples :



Principe d'optimalité d'une sous-structure

Tout sous-chemin d'un plus court chemin est un plus court chemin :

Soit $p = \langle s_0, s_1, \dots, s_k \rangle$ un plus court chemin

$\forall i, j$ tel que $0 \leq i < j \leq k : p_{ij} = \langle s_i, s_{i+1}, \dots, s_j \rangle$ est un plus court chemin

Exercice :

Démonstration...

Exploitation par des algorithmes de programmation dynamique :

- Dijkstra, TopoDAG, Bellman-Ford :

$$\delta(s_i, s_j) = \min_{s_k \in \text{pred}(s_j)} \delta(s_i, s_k) + c(s_k, s_j)$$

- Floyd-Warshall :

$$\delta(s_i, s_j) = \min_{s_k \in S} \delta(s_i, s_k) + \delta(s_k, s_j)$$

1 Introduction

2 Structures de données pour représenter un graphe

3 Arbres Couvrants Minimaux

4 Parcours de graphes

5 Plus courts chemins

- Plus courts chemins à origine unique
- Plus courts chemins pour tout couple de sommets
- Généralisation à la recherche de meilleurs chemins

6 Problèmes de planification

7 Problèmes NP-difficiles

8 Conclusion

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

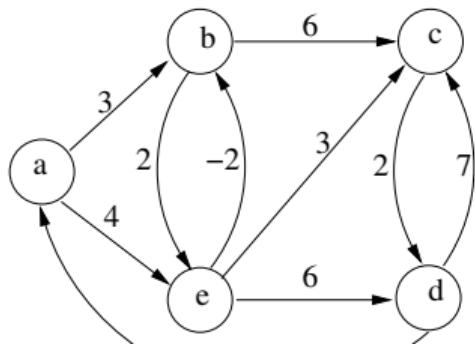
1 Fonction $PlusCourtsChemins(g, c, s_0)$

- Entrée** : Un graphe (orienté ou non) $g = (S, A)$ muni d'une fonction de coût $c : A \rightarrow \mathbb{R}$
 Un sommet de départ $s_0 \in S$
- Sortie** : Une arborescence des plus courts chemins partant de s_0
 Un tableau d tel que $\forall s_i \in S, d[s_i] = \delta(s_0, s_i)$

Arborescence des plus courts chemins :

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

Exemple :



$$d = \begin{array}{|c|c|c|c|c|} \hline 0 & 2 & 7 & 9 & 4 \\ \hline \end{array}$$

$$\pi = \begin{array}{|c|c|c|c|c|} \hline - & e & e & c & a \\ \hline a & b & c & d & e \\ \hline \end{array}$$

Principe commun à Dijkstra, TopoDAG et Bellman-Ford :

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

1 **Proc** *relacher*((s_i, s_j), π, d)

Entrée : Un arc (s_i, s_j)

Entrée/Sortie : Les tableaux π et d

Précond. : $d[s_i] \geq \delta(s_0, s_i)$ et $d[s_j] \geq \delta(s_0, s_j)$

Postcond. : $\delta(s_0, s_j) \leq d[s_j] \leq d[s_i] + c(s_i, s_j)$

2 **début**

3 **si** $d[s_j] > d[s_i] + c(s_i, s_j)$ **alors**

4 $d[s_j] \leftarrow d[s_i] + c(s_i, s_j)$

5 $\pi[s_j] \leftarrow s_i$

Principe commun à Dijkstra, TopoDAG et Bellman-Ford :

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

Dans quel ordre relâcher les arcs ?

- Dijkstra relâche les arcs partant du sommet minimisant d
 \sim Chaque arc est relâché exactement une fois
 \sim **Ne marche que si tous les coûts sont positifs**
- TopoDAG relâche un arc si tous ses prédécesseurs ont été relâchés
 \sim Chaque arc est relâché exactement une fois
 \sim **Ne marche que si le graphe est acyclique**
- Bellman-Ford relâche tous les arcs à chaque itér., jusqu'à convergence
 \sim Chaque arc est relâché plusieurs fois
 \sim **Marche dans tous les cas**

Principe de l'algorithme de Dijkstra (Dijkstra 1959)

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
 $\leadsto d[s_i] = +\infty$
 - s_i est gris s'il a été découvert et sa borne peut encore diminuer
 $\leadsto \delta(s_0, s_i) \leq d[s_i] < +\infty$
 - s_i est noir si sa borne ne peut plus diminuer
 $\leadsto d[s_i] = \delta(s_0, s_i)$
 \leadsto Tous les arcs partant de s_i peuvent être relâchés
- Les sommets gris sont stockés dans une file de priorité (au lieu de FIFO) : à chaque itération, on relâche les arcs partant du sommet gris minimisant d , et on le colorie en noir
- **Ne marche que si tous les coûts sont positifs**
 \leadsto Précondition à Dijkstra : Pour tout arc $(s_i, s_j) \in A$, $\text{cout}(s_i, s_j) \geq 0$

1 Fonction $Dijkstra(g, c, s_0)$

```

1   pour chaque sommet  $s_i \in S$  faire
2        $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow null$ ; Colorier  $s_i$  en blanc
3
4        $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
5   tant que il existe un sommet gris faire
6       Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
7       pour tout sommet  $s_j \in succ(s_i)$  faire
8           si  $s_j$  est blanc ou gris alors
9               relacher( $(s_i, s_j)$ ,  $\pi$ ,  $d$ )
10            si  $s_j$  est blanc alors colorier  $s_j$  en gris;
11
12   Colorier  $s_i$  en noir
13
14   retourne  $\pi$  et  $d$ 

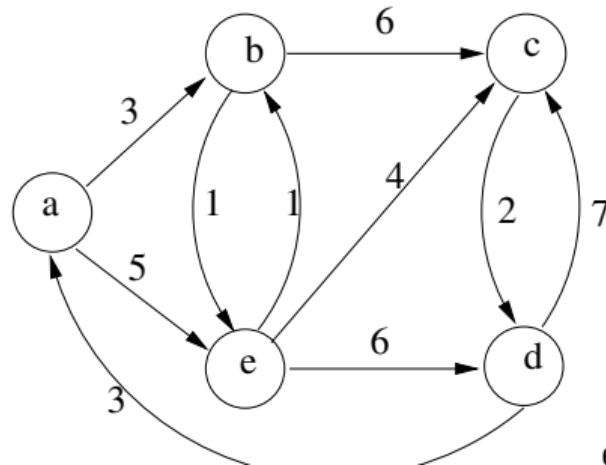
```

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 $d[s_i] = \delta(s_0, s_i)$
- Les succ. d'un sommet noir sont gris ou noirs



1 Fonction $Dijkstra(g, c, s_0)$

```

1   pour chaque sommet  $s_i \in S$  faire
2      $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow null$ ; Colorier  $s_i$  en blanc
3
4    $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
5   tant que il existe un sommet gris faire
6     Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
7     pour tout sommet  $s_j \in succ(s_i)$  faire
8       si  $s_j$  est blanc ou gris alors
9         relacher( $(s_i, s_j)$ ,  $\pi$ ,  $d$ )
10        si  $s_j$  est blanc alors colorier  $s_j$  en gris;
11
12   Colorier  $s_i$  en noir
13
14   retourne  $\pi$  et  $d$ 

```

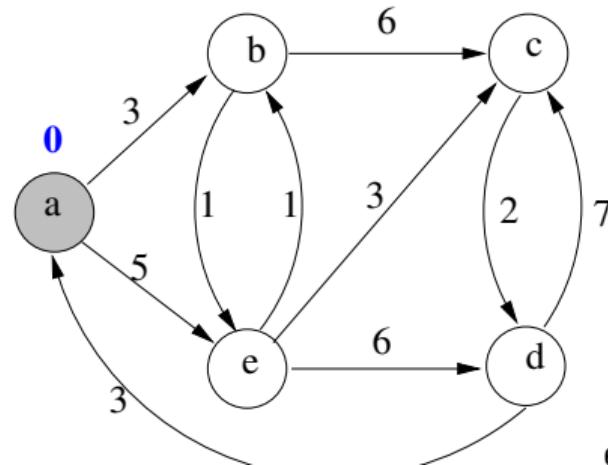
Exemple :

$$s_0 = a$$

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 $d[s_i] = \delta(s_0, s_i)$
- Les succ. d'un sommet noir sont gris ou noirs



1 Fonction $Dijkstra(g, c, s_0)$

```

1   pour chaque sommet  $s_i \in S$  faire
2      $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow null$ ; Colorier  $s_i$  en blanc
3      $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
4   tant que il existe un sommet gris faire
5     Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
6     pour tout sommet  $s_j \in succ(s_i)$  faire
7       si  $s_j$  est blanc ou gris alors
8         relacher( $(s_i, s_j)$ ,  $\pi$ ,  $d$ )
9       si  $s_j$  est blanc alors colorier  $s_j$  en gris;
10
11   Colorier  $s_i$  en noir
12   retourne  $\pi$  et  $d$ 

```

Exemple :

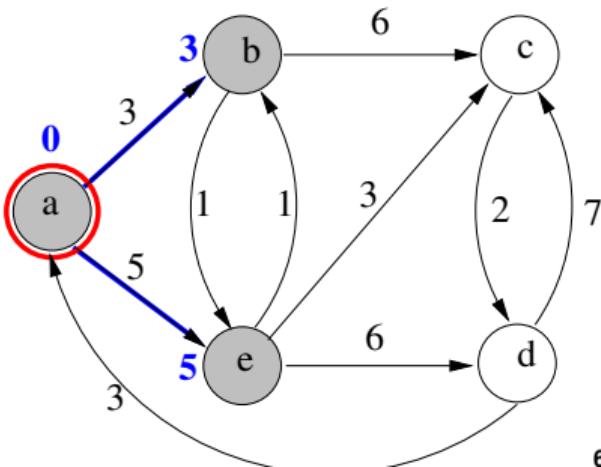
$$s_i = a$$

Arcs relâchés : $(a, b), (a, e)$

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 $d[s_i] = \delta(s_0, s_i)$
- Les succ. d'un sommet noir sont gris ou noirs



1 Fonction $Dijkstra(g, c, s_0)$

```

2   pour chaque sommet  $s_i \in S$  faire
3      $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow null$ ; Colorier  $s_i$  en blanc
4      $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
5   tant que il existe un sommet gris faire
6     Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
7     pour tout sommet  $s_j \in succ(s_i)$  faire
8       si  $s_j$  est blanc ou gris alors
9         relacher( $(s_i, s_j)$ ,  $\pi$ ,  $d$ )
10        si  $s_j$  est blanc alors colorier  $s_j$  en gris;
11
12   Colorier  $s_i$  en noir
13
14   retourne  $\pi$  et  $d$ 

```

Exemple :

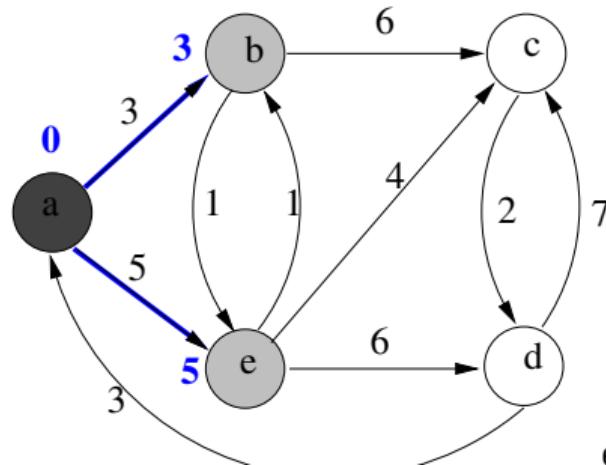
$$s_i = a$$

Arcs relâchés : $(a, b), (a, e)$

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 $d[s_i] = \delta(s_0, s_i)$
- Les succ. d'un sommet noir sont gris ou noirs



1 Fonction $Dijkstra(g, c, s_0)$

```

2   pour chaque sommet  $s_i \in S$  faire
3      $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow null$ ; Colorier  $s_i$  en blanc
4      $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
5   tant que il existe un sommet gris faire
6     Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
7     pour tout sommet  $s_j \in succ(s_i)$  faire
8       si  $s_j$  est blanc ou gris alors
9         relacher( $(s_i, s_j), \pi, d$ )
10        si  $s_j$  est blanc alors colorier  $s_j$  en gris;
11   Colorier  $s_i$  en noir
12   retourne  $\pi$  et  $d$ 

```

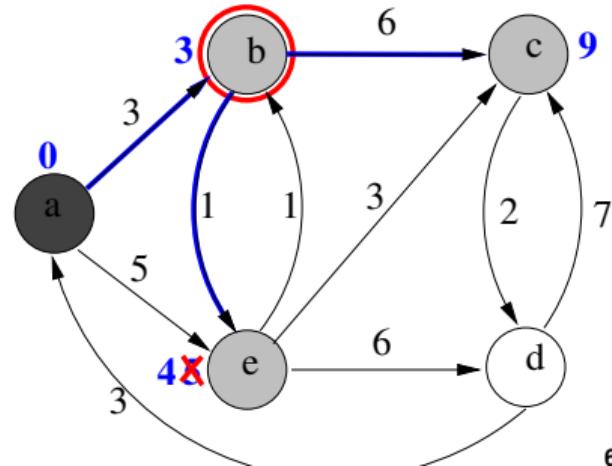
Exemple :

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

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 $d[s_i] = \delta(s_0, s_i)$
- Les succ. d'un sommet noir sont gris ou noirs



1 Fonction Dijkstra(g, c, s_0)

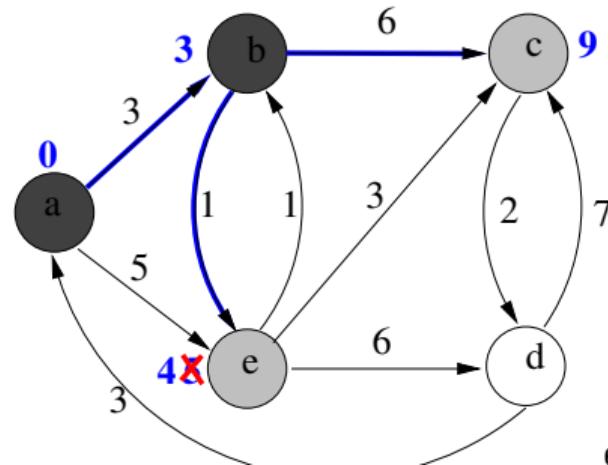
```

1   pour chaque sommet  $s_i \in S$  faire
2      $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow \text{null}$ ; Colorier  $s_i$  en blanc
3      $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
4   tant que il existe un sommet gris faire
5     Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
6     pour tout sommet  $s_j \in \text{succ}(s_i)$  faire
7       si  $s_j$  est blanc ou gris alors
8         relacher( $(s_i, s_j), \pi, d$ )
9       si  $s_j$  est blanc alors colorier  $s_j$  en gris;
10
11   Colorier  $s_i$  en noir
12   retourne  $\pi$  et  $d$ 

```

Exemple : $s_i = b$ Arcs relâchés : $(a, b), (a, e), (b, e), (b, c)$ **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 $d[s_i] = \delta(s_0, s_i)$
- Les succ. d'un sommet noir sont gris ou noirs



1 Fonction $Dijkstra(g, c, s_0)$

```

1   pour chaque sommet  $s_i \in S$  faire
2      $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow null$ ; Colorier  $s_i$  en blanc
3
4      $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
5   tant que il existe un sommet gris faire
6     Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
7     pour tout sommet  $s_j \in succ(s_i)$  faire
8       si  $s_j$  est blanc ou gris alors
9         relacher( $(s_i, s_j)$ ,  $\pi$ ,  $d$ )
10        si  $s_j$  est blanc alors colorier  $s_j$  en gris;
11
12   Colorier  $s_i$  en noir
13
14   retourne  $\pi$  et  $d$ 

```

Exemple :

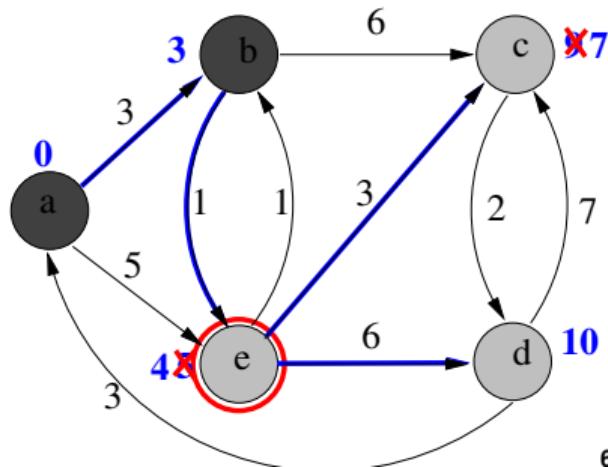
$$s_i = e$$

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

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 $d[s_i] = \delta(s_0, s_i)$
- Les succ. d'un sommet noir sont gris ou noirs



1 Fonction Dijkstra(g, c, s_0)

```

1   pour chaque sommet  $s_i \in S$  faire
2      $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow \text{null}$ ; Colorier  $s_i$  en blanc
3      $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
4   tant que il existe un sommet gris faire
5     Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
6     pour tout sommet  $s_j \in \text{succ}(s_i)$  faire
7       si  $s_j$  est blanc ou gris alors
8         relacher( $(s_i, s_j), \pi, d$ )
9       si  $s_j$  est blanc alors colorier  $s_j$  en gris;
10
11   Colorier  $s_i$  en noir
12   retourne  $\pi$  et  $d$ 

```

Exemple :

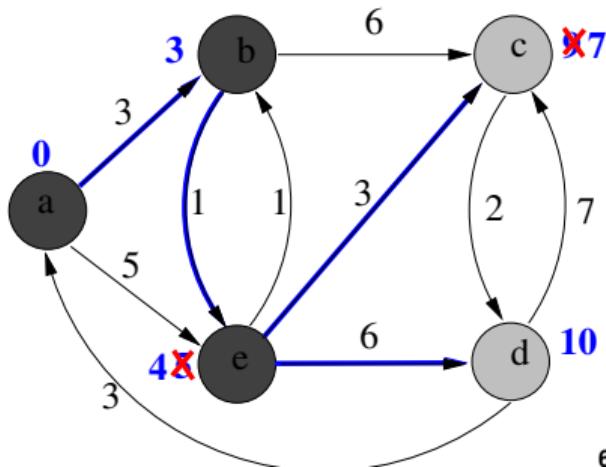
$$s_i = e$$

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

Propriété invariante ligne 5 :

Pour tout sommet s_i ,

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



1 Fonction $Dijkstra(g, c, s_0)$

```

1   pour chaque sommet  $s_i \in S$  faire
2      $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow null$ ; Colorier  $s_i$  en blanc
3
4      $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
5     tant que il existe un sommet gris faire
6       Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
7       pour tout sommet  $s_j \in succ(s_i)$  faire
8         si  $s_j$  est blanc ou gris alors
9           relacher( $(s_i, s_j)$ ,  $\pi$ ,  $d$ )
10          si  $s_j$  est blanc alors colorier  $s_j$  en gris;
11
12       Colorier  $s_i$  en noir
13
14   retourne  $\pi$  et  $d$ 

```

Exemple :

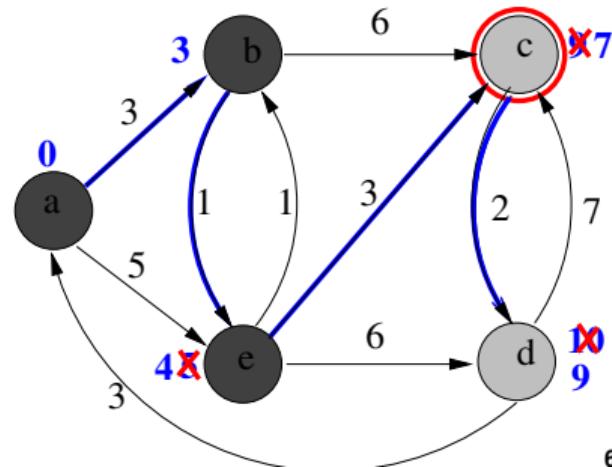
$$s_i = c$$

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

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 $d[s_i] = \delta(s_0, s_i)$
- Les succ. d'un sommet noir sont gris ou noirs



1 Fonction $Dijkstra(g, c, s_0)$

```

2   pour chaque sommet  $s_i \in S$  faire
3      $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow null$ ; Colorier  $s_i$  en blanc
4      $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
5   tant que il existe un sommet gris faire
6     Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
7     pour tout sommet  $s_j \in succ(s_i)$  faire
8       si  $s_j$  est blanc ou gris alors
9         relacher( $(s_i, s_j)$ ,  $\pi$ ,  $d$ )
10        si  $s_j$  est blanc alors colorier  $s_j$  en gris;
11
12   Colorier  $s_i$  en noir
13
14   retourne  $\pi$  et  $d$ 

```

Exemple :

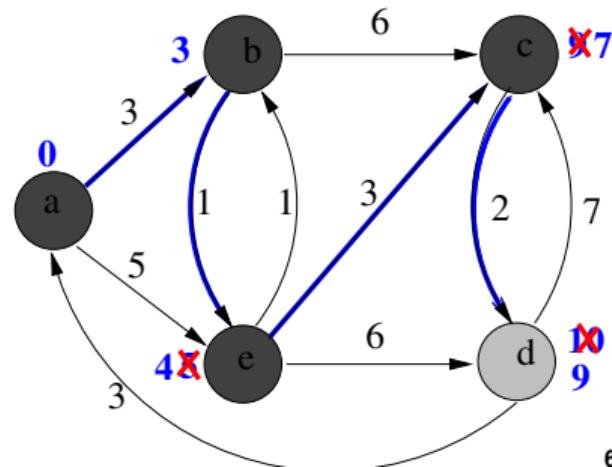
$$s_i = c$$

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

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 $d[s_i] = \delta(s_0, s_i)$
- Les succ. d'un sommet noir sont gris ou noirs



1 Fonction $Dijkstra(g, c, s_0)$

```

2   pour chaque sommet  $s_i \in S$  faire
3      $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow null$ ; Colorier  $s_i$  en blanc
4      $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
5   tant que il existe un sommet gris faire
6     Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
7     pour tout sommet  $s_j \in succ(s_i)$  faire
8       si  $s_j$  est blanc ou gris alors
9         relacher( $(s_i, s_j)$ ,  $\pi$ ,  $d$ )
10        si  $s_j$  est blanc alors colorier  $s_j$  en gris;
11   Colorier  $s_i$  en noir
12   retourne  $\pi$  et  $d$ 

```

Exemple :

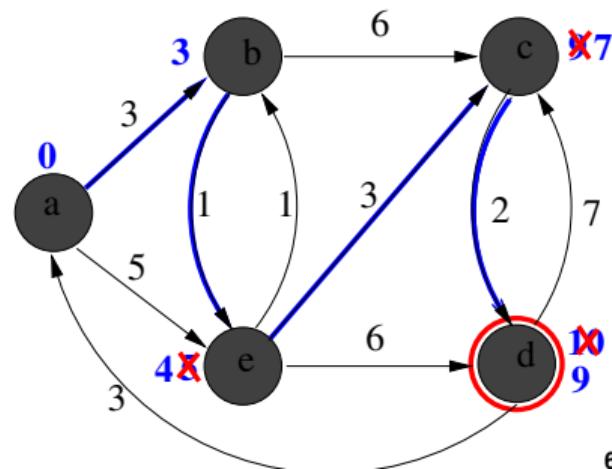
$$s_i = d$$

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

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 $d[s_i] = \delta(s_0, s_i)$
- Les succ. d'un sommet noir sont gris ou noirs



1 Fonction *Dijkstra*(g, c, s_0)

```

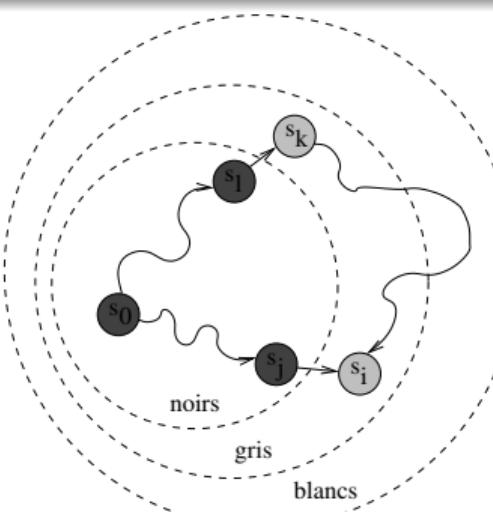
1   pour chaque sommet  $s_i \in S$  faire
2        $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow \text{null}$ ; Colorier  $s_i$  en blanc
3        $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
4   tant que il existe un sommet gris faire
5       Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
6       pour tout sommet  $s_j \in \text{succ}(s_i)$  faire
7           si  $s_j$  est blanc ou gris alors
8               relacher( $(s_i, s_j), \pi, d$ )
9           si  $s_j$  est blanc alors colorier  $s_j$  en gris;
10
11      Colorier  $s_i$  en noir
12
retourne  $\pi$  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 $d[s_i] = \delta(s_0, s_i)$
- Les succ. d'un sommet noir sont gris ou noirs



1 Fonction $Dijkstra(g, c, s_0)$

```

2   pour chaque sommet  $s_i \in S$  faire
3      $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow null$ ; Colorier  $s_i$  en blanc
4    $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
5   tant que il existe un sommet gris faire
6     Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
7     pour tout sommet  $s_j \in succ(s_i)$  faire
8       si  $s_j$  est blanc ou gris alors
9         relacher( $(s_i, s_j), \pi, d$ )
10        si  $s_j$  est blanc alors colorier  $s_j$  en gris;
11    Colorier  $s_i$  en noir
12  retourne  $\pi$  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 $d[s_i] = \delta(s_0, s_i)$
- 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 $Dijkstra(g, c, s_0)$

```

2   pour chaque sommet  $s_i \in S$  faire
3      $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow null$ ; Colorier  $s_i$  en blanc
4      $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
5   tant que il existe un sommet gris faire
6     Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
7     pour tout sommet  $s_j \in succ(s_i)$  faire
8       si  $s_j$  est blanc ou gris alors
9         relacher( $(s_i, s_j), \pi, d$ )
10        si  $s_j$  est blanc alors colorier  $s_j$  en gris;
11      Colorier  $s_i$  en noir
12  retourne  $\pi$  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 $d[s_i] = \delta(s_0, s_i)$
- 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 $Dijkstra(g, c, s_0)$

```

2   pour chaque sommet  $s_i \in S$  faire
3      $d[s_i] \leftarrow +\infty$ ;  $\pi[s_i] \leftarrow null$ ; Colorier  $s_i$  en blanc
4    $d[s_0] \leftarrow 0$ ; Colorier  $s_0$  en gris
5   tant que il existe un sommet gris faire
6     Soit  $s_i$  le sommet gris tq  $d[s_i]$  soit minimal
7     pour tout sommet  $s_j \in succ(s_i)$  faire
8       si  $s_j$  est blanc ou gris alors
9         relacher( $(s_i, s_j), \pi, d$ )
10        si  $s_j$  est blanc alors colorier  $s_j$  en gris;
11      Colorier  $s_i$  en noir
12  retourne  $\pi$  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 $d[s_i] = \delta(s_0, s_i)$
- 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

Principe de l'algorithme TopoDAG

Rappel :

Un DAG est un graphe orienté sans circuit

Idée :

- Relâcher les arcs partant de s_i seulement si tous les arcs se trouvant sur un chemin entre s_0 et s_i ont déjà été relâchés
 - ~ Dans ce cas, on a la garantie que $d[s_i] = \delta(s_0, s_i)$
- Utiliser un tri topologique pour déterminer l'ordre de relâchement

Algorithm TopoDAG

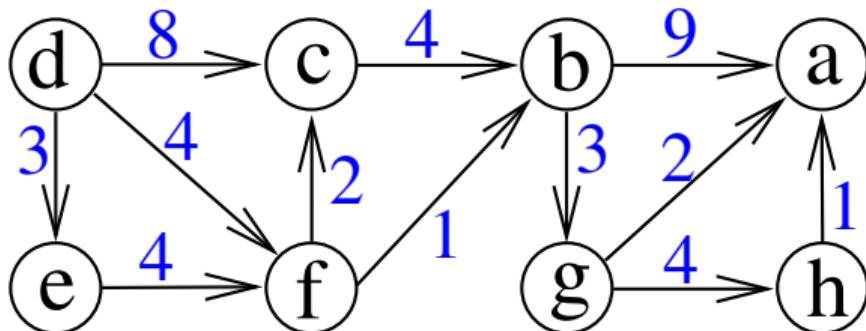
```

1 Fonction TopoDAG( $g, c, s_0$ )
    Précond. :  $g$  est un DAG
    pour chaque sommet  $s_i \in S$  faire
         $d[s_i] \leftarrow +\infty$ 
         $\pi[s_i] \leftarrow \text{null}$ 
     $d[s_0] \leftarrow 0$ 
    Trier topologiquement les sommets de  $g$  à l'aide d'un DFS partant de  $s_0$ 
    pour chaque sommet  $s_i$  pris selon l'ordre topologique faire
        pour chaque sommet  $s_j \in \text{succ}(s_i)$  faire relacher( $(s_i, s_j), \pi, d$ );
    retourne  $\pi$  et  $d$ 

```

Exercice :

$$s_0 = d$$



Algorithm TopoDAG

```
1 Fonction TopoDAG( $g, c, s_0$ )
    Précond. :  $g$  est un DAG
    pour chaque sommet  $s_i \in S$  faire
         $d[s_i] \leftarrow +\infty$ 
         $\pi[s_i] \leftarrow \text{null}$ 
    5  $d[s_0] \leftarrow 0$ 
    Trier topologiquement les sommets de  $g$  à l'aide d'un DFS partant de  $s_0$ 
    pour chaque sommet  $s_i$  pris selon l'ordre topologique faire
        pour chaque sommet  $s_j \in \text{succ}(s_i)$  faire relacher( $(s_i, s_j), \pi, d$ );
    9 retourne  $\pi$  et  $d$ 
```

Complexité pour un graphe ayant n sommets et p arcs ?

Algorithm TopoDAG

```
1 Fonction TopoDAG( $g, c, s_0$ )
    Précond. :  $g$  est un DAG
    pour chaque sommet  $s_i \in S$  faire
         $d[s_i] \leftarrow +\infty$ 
         $\pi[s_i] \leftarrow \text{null}$ 
    5  $d[s_0] \leftarrow 0$ 
    6 Trier topologiquement les sommets de  $g$  à l'aide d'un DFS partant de  $s_0$ 
    7 pour chaque sommet  $s_i$  pris selon l'ordre topologique faire
        pour chaque sommet  $s_j \in \text{succ}(s_i)$  faire relacher( $(s_i, s_j), \pi, d$ );
    9 retourne  $\pi$  et  $d$ 
```

Complexité pour un graphe ayant n sommets et p arcs ?

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

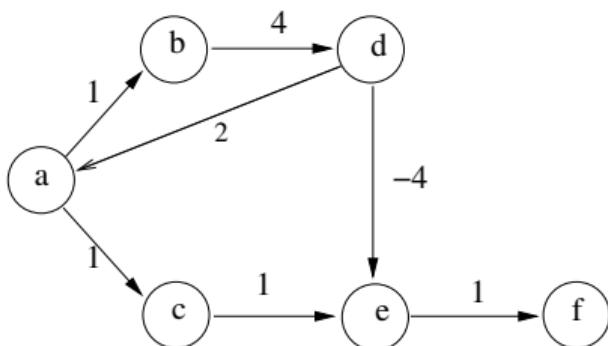
Points communs et limitations de Dijkstra et TopoDAG

- Relâchement des arcs partant de s_i quand $d[s_i] = \delta(s_0, s_i)$
 - ~ Principe d'optimalité des sous-chemins : $\delta(s_0, s_j) = \min_{s_i \in \text{pred}(s_j)} \delta(s_0, s_i) + c(s_i, s_j)$
- Principe **glouton** pour choisir s_i :
 - Dijkstra : Choisir le sommet s_i gris qui minimise $d[s_i]$
 - ~ Si tous les coûts sont positifs, $d[s_i]$ ne pourra plus diminuer
 - TopoDAG : Choisir s_i selon un ordre topologique
 - ~ Si le graphe est un DAG, $d[s_i]$ ne pourra plus diminuer

Points communs et limitations de Dijkstra et TopoDAG

- Relâchement des arcs partant de s_i quand $d[s_i] = \delta(s_0, s_i)$
 ~ Principe d'optimalité des sous-chemins : $\delta(s_0, s_j) = \min_{s_i \in \text{pred}(s_j)} \delta(s_0, s_i) + c(s_i, s_j)$
- Principe **glouton** pour choisir s_i :
 - Dijkstra : Choisir le sommet s_i gris qui minimise $d[s_i]$
 ~ Si tous les coûts sont positifs, $d[s_i]$ ne pourra plus diminuer
 - TopoDAG : Choisir s_i selon un ordre topologique
 ~ Si le graphe est un DAG, $d[s_i]$ ne pourra plus diminuer

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



Programmation dynamique pour le calcul de plus courts chemins

Principe des approches "Diviser pour Régner" :

- Décomposer le problème à résoudre en sous-problèmes
- Résoudre chaque sous-problème
- Calculer la solution du problème initial à partir des solutions des sous-problèmes

Exemples : Quicksort, MergeSort, Branch-and-Bound, ..

~ Tous les sous-problèmes sont différents

Programmation dynamique ?

Approche "Diviser pour régner" où un même sous-problème peut apparaître plusieurs fois

Etapes pour résoudre un problème en programmation dynamique

Etape 1 : Définir ce qu'est un sous-problème

Etape 2 : Définir les équations de Bellman

- Identifier les sous-problèmes de base, où la solution est connue sans calcul
- Définir récursivement la solution des sous-problèmes plus compliqués

Basées sur le principe **d'optimalité des sous-structures**

Etape 3 : Analyse des performances

Combien de sous-problèmes différents doivent être résolus ?

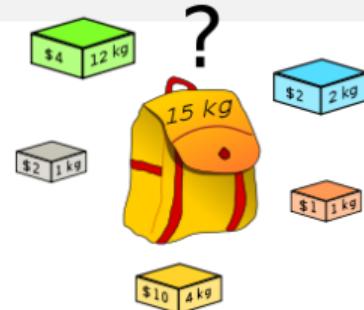
Etape 4 : Choisir une approche pour éviter de recalculer des solutions

- Top-Down : Implémentation récursive avec mémoïsation
 - ~ Facile à programmer (et à prouver correct !)
- Bottom-Up : Implémentation itérative, en remplissant un tableau
 - ~ Parfois plus économique en mémoire

Rappelez-vous, au S1... le sac-à-dos !

Description du problème :

- Entrée : un poids max p_{max} et un ensemble d'objets $O = \{1, \dots, n\}$
 ↳ Chaque objet i a un poids p_i et une utilité u_i
- Sortie : le sous-ensemble $S \subseteq O$ maximisant $\sum_{i \in S} u_i$ et tel que
 $\sum_{i \in S} p_i \leq p_{max}$



Etape 1 : Définir ce qu'est un sous-problème

$\forall i \in [1, n], \forall p \in [1, p_{max}] : m(i, p) = \text{utilité max qd } O = \{1, \dots, i\} \text{ et } p_{max} = p$

Etape 2 : Equations de Bellman

- $m(1, p) = 0$ si $p < p_1$ et $m(1, p) = u_1$ sinon
- $\forall i \in [2, n], \forall p \in [0, p_i - 1] : m(i, p) = m(i - 1, p)$
- $\forall i \in [2, n], \forall p \in [p_i, p_{max}] : m(i, p) = \max\{u_i + m(i - 1, p - p_i), m(i - 1, p)\}$

Etape 3 : Analyse des performances

Combien de sous-problèmes différents ?

Programmation dynamique pour le calcul de plus courts chemins

Etape 1 : Définir ce qu'est un sous-problème

Pour tout entier k et pour tout sommet $i \in S$:

Sous-problème $d(k, i)$ = longueur du + court chemin de s_0 jusque i **n'utilisant pas plus de k arcs**

Etape 2 : Equations de Bellman

- $k = 0 : d(0, i) = 0$ si $i = s_0$ et $d(0, i) = +\infty$ sinon
- $k > 1 : \text{Soit } x = \min_{j \in \text{pred}(i)} d(k - 1, j) + c(j, i)$
si $x < d(k - 1, i)$ alors $d(k, i) = x$
sinon $d(k, i) = d(k - 1, i)$

Question : Pour quelle valeur de k est-on sûr d'avoir $d(k, i) = \delta(s_0, i)$?

Etape 3 : Analyse des performances :

- Combien de sous-problèmes différents pour un graphe de n sommets ?
- Que doit-on faire pour chaque sous-problème ?

Etape 4 : Calcul récursif Top-down

Rappel des équations de Bellman :

- $k = 0 : d(0, i) = 0$ si $i = s_0$ et $d(0, i) = +\infty$ sinon
- $k > 1 : \text{Soit } x = \min_{j \in \text{pred}(i)} d(k - 1, j) + c(j, i)$
si $x < d(k - 1, i)$ alors $d(k, i) = x$ sinon $d(k, i) = d(k - 1, i)$

```

1 Fonction Calcul-récuratif( $g, c, s_0, k, i$ )
2   si  $k = 0$  alors
3     si  $i = s_0$  alors retourne 0;
4     retourne  $\infty$ 
5    $x \leftarrow \infty$ 
6   pour chaque sommet  $j \in \text{pred}(i)$  faire
7      $x \leftarrow \min(x, \text{Calcul-récuratif}(g, c, s_0, k - 1, j) + c(j, i))$ 
8   retourne  $\min(x, \text{Calcul-récuratif}(g, c, s_0, k - 1, i))$ 

```

Appel initial pour calculer $\delta(s_0, i)$: Calcul-récuratif($g, c, s_0, n - 1, i$)

Complexité de cet algorithme ?

Etape 4 : Calcul récursif Top-down

Rappel des équations de Bellman :

- $k = 0 : d(0, i) = 0$ si $i = s_0$ et $d(0, i) = +\infty$ sinon
- $k > 1 : \text{Soit } x = \min_{j \in \text{pred}(i)} d(k - 1, j) + c(j, i)$
si $x < d(k - 1, i)$ alors $d(k, i) = x$ sinon $d(k, i) = d(k - 1, i)$

```

1 Fonction Calcul-récuratif( $g, c, s_0, k, i$ )
2   si  $k = 0$  alors
3     si  $i = s_0$  alors retourne 0;
4     retourne  $\infty$ 
5    $x \leftarrow \infty$ 
6   pour chaque sommet  $j \in \text{pred}(i)$  faire
7      $x \leftarrow \min(x, \text{Calcul-récuratif}(g, c, s_0, k - 1, j) + c(j, i))$ 
8   retourne  $\min(x, \text{Calcul-récuratif}(g, c, s_0, k - 1, i))$ 

```

Appel initial pour calculer $\delta(s_0, i)$: Calcul-récuratif($g, c, s_0, n - 1, i$)

Complexité de cet algorithme ?

Il faut mémoiser pour ne pas calculer plusieurs fois $d(k, i)$!

Etape 4 : Calcul récursif Top-down avec mémoïsation

Rappel des équations de Bellman :

- $k = 0 : d(0, i) = 0$ si $i = s_0$ et $d(0, i) = +\infty$ sinon
- $k > 1 : \text{Soit } x = \min_{j \in \text{pred}(i)} d(k - 1, j) + c(j, i)$
si $x < d(k - 1, i)$ alors $d(k, i) = x$ sinon $d(k, i) = d(k - 1, i)$

Utilisation d'une variable globale d pour mémoiser les valeurs calculées :

```

1 Fonction Calcul-réc-mémo( $g, c, s_0, k, i$ )
2   si  $k = 0$  alors
3     si  $i = s_0$  alors retourne 0 sinon retourne  $\infty$ ;
4   si  $d[k][i] = null$  alors
5      $d[k][i] \leftarrow$  Calcul-réc-mémo( $g, c, s_0, k - 1, i$ )
6     pour chaque sommet  $j \in \text{pred}(i)$  faire
7        $d[k][i] \leftarrow \min(d[k][i], \text{Calcul-réc-mémo}(g, c, s_0, k - 1, j) + c(j, i))$ 
8   retourne  $d[k][i]$ 

```

Etape 4 : Calcul itératif, de bas en haut

Rappel des équations de Bellman :

- $k = 0 : d(0, i) = 0$ si $i = s_0$ et $d(0, i) = +\infty$ sinon
- $k > 1 : \text{Soit } x = \min_{j \in \text{pred}(i)} d(k - 1, j) + c(j, i)$
 $\quad \text{si } x < d(k - 1, i) \text{ alors } d(k, i) = x \text{ sinon } d(k, i) = d(k - 1, i)$

```

1 Fonction Calcul-itératif( $g, c, s_0$ )
2   pour chaque sommet  $i \in S$  faire  $d[0][i] \leftarrow +\infty;$ 
3    $d[0][s_0] \leftarrow 0$ 
4   pour  $k$  variant de 1 à  $\#S - 1$  faire
5     pour chaque sommet  $i \in S$  faire
6        $d[k][i] \leftarrow d[k - 1][i]$ 
7       pour chaque sommet  $j \in \text{pred}(i)$  faire
8          $d[k][i] \leftarrow \min(d[k][i], d[k - 1][j] + c(j, i))$ 
9   retourne  $d[\#S - 1]$ 

```

Complexité de cet algorithme ?

Etape 4 : Calcul itératif, de bas en haut

Rappel des équations de Bellman :

- $k = 0 : d(0, i) = 0$ si $i = s_0$ et $d(0, i) = +\infty$ sinon
- $k > 1 : \text{Soit } x = \min_{j \in \text{pred}(i)} d(k - 1, j) + c(j, i)$
 $\quad \text{si } x < d(k - 1, i) \text{ alors } d(k, i) = x \text{ sinon } d(k, i) = d(k - 1, i)$

```

1 Fonction Calcul-itératif( $g, c, s_0$ )
2   pour chaque sommet  $i \in S$  faire  $d[0][i] \leftarrow +\infty;$ 
3    $d[0][s_0] \leftarrow 0$ 
4   pour  $k$  variant de 1 à  $\#S - 1$  faire
5     pour chaque sommet  $i \in S$  faire
6        $d[k][i] \leftarrow d[k - 1][i]$ 
7       pour chaque sommet  $j \in \text{pred}(i)$  faire
8          $d[k][i] \leftarrow \min(d[k][i], d[k - 1][j] + c(j, i))$ 
9   retourne  $d[\#S - 1]$ 

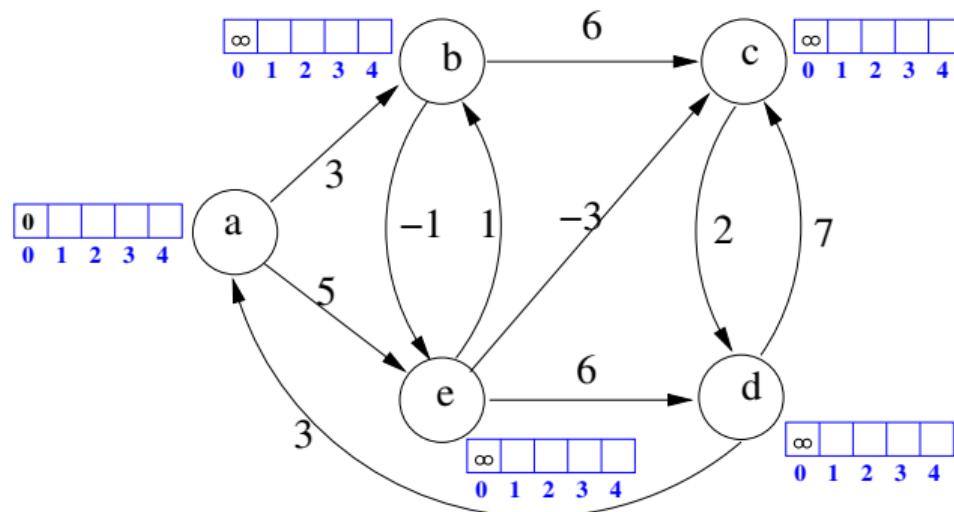
```

Complexité de cet algorithme ? $\mathcal{O}(np)$

```

1 Fonction Calcul-itératif( $g, c, s_0$ )
2   pour chaque sommet  $i \in S$  faire  $d[0][i] \leftarrow +\infty$ ;
3      $d[0][s_0] \leftarrow 0$ 
4   pour  $k$  variant de 1 à  $\#S - 1$  faire
5     pour chaque sommet  $i \in S$  faire
6        $d[k][i] \leftarrow d[k - 1][i]$ 
7       pour chaque sommet  $j \in pred(i)$  faire
8          $d[k][i] \leftarrow \min(d[k][i], d[k - 1][j] + c(j, i))$ 
9   retourne  $d[\#S - 1]$ 

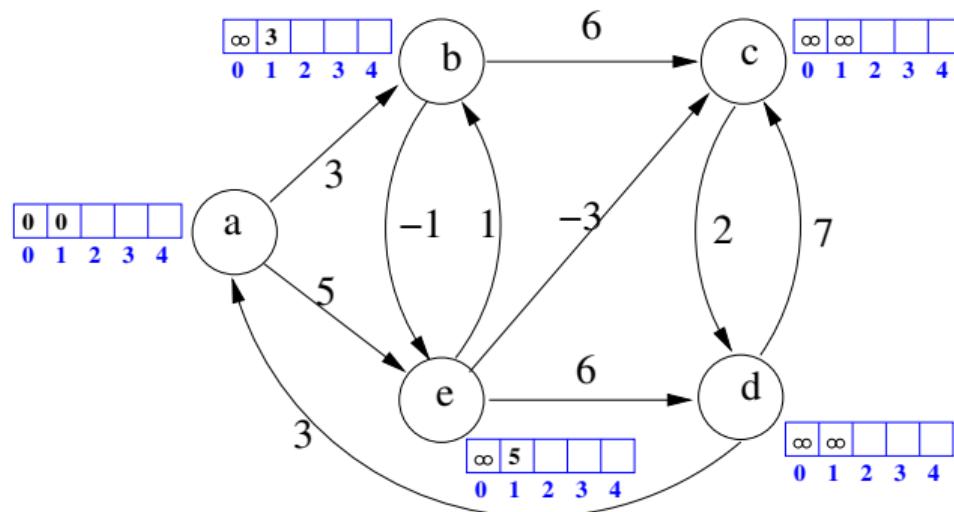
```



```

1 Fonction Calcul-itératif( $g, c, s_0$ )
2   pour chaque sommet  $i \in S$  faire  $d[0][i] \leftarrow +\infty$ ;
3    $d[0][s_0] \leftarrow 0$ 
4   pour  $k$  variant de 1 à  $\#S - 1$  faire
5     pour chaque sommet  $i \in S$  faire
6        $d[k][i] \leftarrow d[k - 1][i]$ 
7       pour chaque sommet  $j \in pred(i)$  faire
8          $d[k][i] \leftarrow \min(d[k][i], d[k - 1][j] + c(j, i))$ 
9   retourne  $d[\#S - 1]$ 

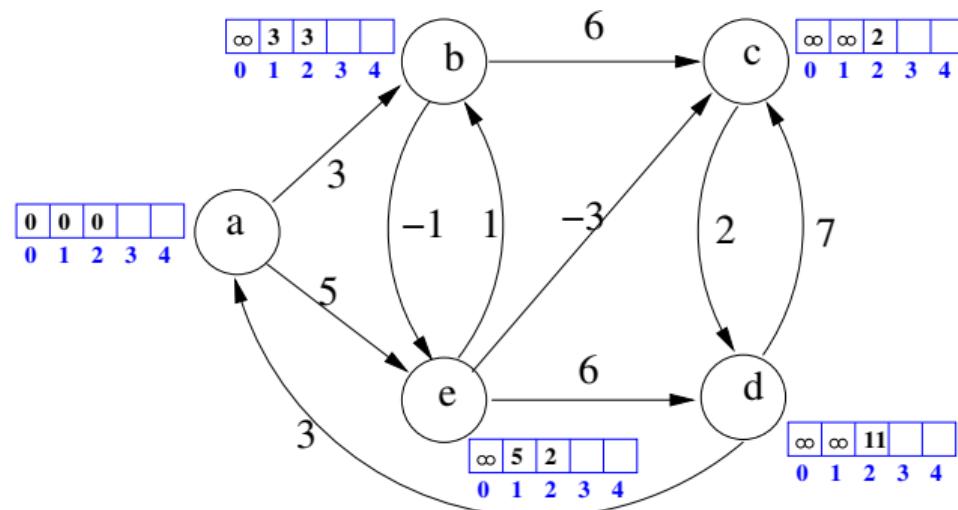
```



```

1 Fonction Calcul-itératif( $g, c, s_0$ )
2   pour chaque sommet  $i \in S$  faire  $d[0][i] \leftarrow +\infty$ ;
3      $d[0][s_0] \leftarrow 0$ 
4   pour  $k$  variant de 1 à  $\#S - 1$  faire
5     pour chaque sommet  $i \in S$  faire
6        $d[k][i] \leftarrow d[k - 1][i]$ 
7       pour chaque sommet  $j \in pred(i)$  faire
8          $d[k][i] \leftarrow \min(d[k][i], d[k - 1][j] + c(j, i))$ 
9   retourne  $d[\#S - 1]$ 

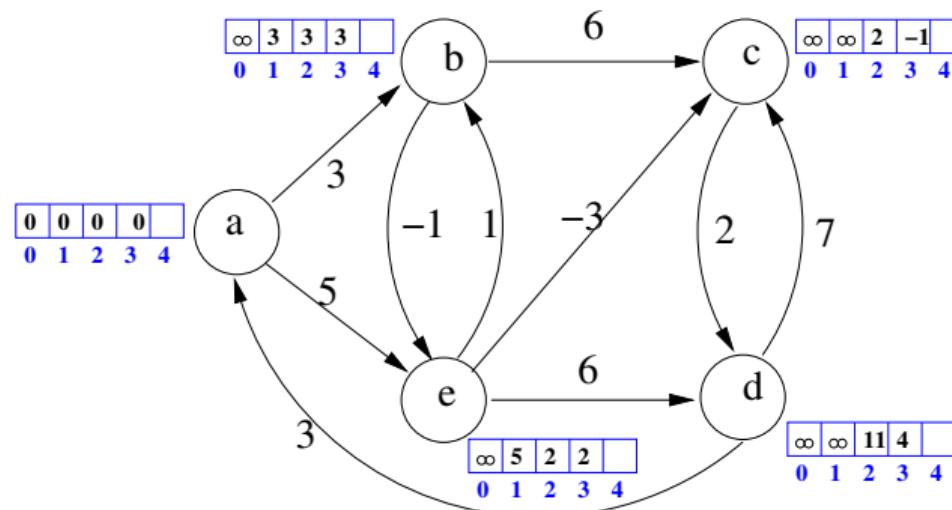
```



```

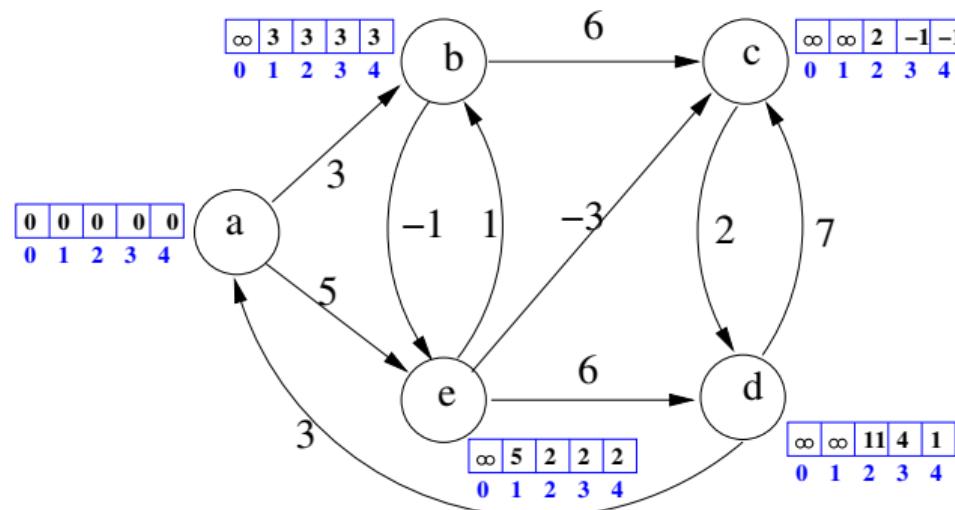
1 Fonction Calcul-itératif( $g, c, s_0$ )
2   pour chaque sommet  $i \in S$  faire  $d[0][i] \leftarrow +\infty$ ;
3      $d[0][s_0] \leftarrow 0$ 
4   pour  $k$  variant de 1 à  $\#S - 1$  faire
5     pour chaque sommet  $i \in S$  faire
6        $d[k][i] \leftarrow d[k - 1][i]$ 
7       pour chaque sommet  $j \in pred(i)$  faire
8          $d[k][i] \leftarrow \min(d[k][i], d[k - 1][j] + c(j, i))$ 
9   retourne  $d[\#S - 1]$ 

```



```

1 Fonction Calcul-itératif( $g, c, s_0$ )
2   pour chaque sommet  $i \in S$  faire  $d[0][i] \leftarrow +\infty$ ;
3    $d[0][s_0] \leftarrow 0$ 
4   pour  $k$  variant de 1 à  $\#S - 1$  faire
5     pour chaque sommet  $i \in S$  faire
6        $d[k][i] \leftarrow d[k - 1][i]$ 
7       pour chaque sommet  $j \in pred(i)$  faire
8          $d[k][i] \leftarrow \min(d[k][i], d[k - 1][j] + c(j, i))$ 
9   retourne  $d[\#S - 1]$ 
```



```

1 Fonction Calcul-itératif( $g, c, s_0$ )
2   pour chaque sommet  $i \in S$  faire  $d[0][i] \leftarrow +\infty$ ;
3      $d[0][s_0] \leftarrow 0$ 
4     pour  $k$  variant de 1 à  $\#S - 1$  faire
5       pour chaque sommet  $i \in S$  faire
6          $d[k][i] \leftarrow d[k - 1][i]$ 
7         pour chaque sommet  $j \in pred(i)$  faire
8            $d[k][i] \leftarrow \min(d[k][i], d[k - 1][j] + c(j, i))$ 
9   retourne  $d[\#S - 1]$ 

```

Déterminer les circuits absorbants :

Ligne 9 : Tester si $\exists (i, j) \in A$ tq $d[\#S - 1][j] > d[\#S - 1][i] + c(i, j)$

Possibilité d'améliorer les performances (sans changer la complexité) :

Sortir de la boucle (4-8) si $d[k] = d[k - 1]$

```

1 Fonction Calcul-itératif( $g, c, s_0$ )
2   pour chaque sommet  $i \in S$  faire  $d[0][i] \leftarrow +\infty$ ;
3      $d[0][s_0] \leftarrow 0$ 
4     pour  $k$  variant de 1 à  $\#S - 1$  faire
5       pour chaque sommet  $i \in S$  faire
6          $d[k][i] \leftarrow d[k - 1][i]$ 
7         pour chaque sommet  $j \in pred(i)$  faire
8            $d[k][i] \leftarrow \min(d[k][i], d[k - 1][j] + c(j, i))$ 
9   retourne  $d[\#S - 1]$ 

```

Déterminer les circuits absorbants :

Ligne 9 : Tester si $\exists (i, j) \in A$ tq $d[\#S - 1][j] > d[\#S - 1][i] + c(i, j)$

Possibilité d'améliorer les performances (sans changer la complexité) :

Sortir de la boucle (4-8) si $d[k] = d[k - 1]$

Etape 4 : Bellman-Ford (avec procédure de relâchement)

```
1 Fonction Bellman-Ford( $g, c, s_0$ )
2   pour chaque sommet  $i \in S$  faire
3      $d[i] \leftarrow +\infty$ 
4      $\pi[i] \leftarrow \text{null}$ 
5    $d[s_0] \leftarrow 0$ 
6   pour  $k$  variant de 1 à  $\#S - 1$  faire
7     pour chaque sommet  $i \in S$  faire
8       pour chaque sommet  $j \in \text{pred}(i)$  faire relâcher( $(i, j), \pi, d$ );
9   retourne  $\pi$  et  $d$ 
```

Que change le fait d'utiliser le même d pour toutes les itérations ?

Complexité pour un graphe ayant n sommets et p arcs : $\mathcal{O}(np)$

Possibilité d'améliorer les performances (sans changer la complexité) :

- Sortir de la boucle (6-9) si d n'est pas modifié
- Ne relâcher que les arcs (i, j) pour lesquels $d[i]$ a été modifié

Etape 4 : Bellman-Ford (avec procédure de relâchement)

```

1 Fonction Bellman-Ford( $g, c, s_0$ )
2   pour chaque sommet  $i \in S$  faire
3      $d[i] \leftarrow +\infty$ 
4      $\pi[i] \leftarrow \text{null}$ 
5    $d[s_0] \leftarrow 0$ 
6   pour  $k$  variant de 1 à  $\#S - 1$  faire
7     pour chaque sommet  $i \in S$  faire
8       pour chaque sommet  $j \in \text{pred}(i)$  faire relâcher( $(i, j), \pi, d$ );
9   retourne  $\pi$  et  $d$ 

```

Que change le fait d'utiliser le même d pour toutes les itérations ?

Complexité pour un graphe ayant n sommets et p arcs : $\mathcal{O}(np)$

Possibilité d'améliorer les performances (sans changer la complexité) :

- Sortir de la boucle (6-9) si d n'est pas modifié
- Ne relâcher que les arcs (i, j) pour lesquels $d[i]$ a été modifié

- 1 Introduction
- 2 Structures de données pour représenter un graphe
- 3 Arbres Couvrants Minimaux
- 4 Parcours de graphes
- 5 Plus courts chemins
 - Plus courts chemins à origine unique
 - Plus courts chemins pour tout couple de sommets
 - Généralisation à la recherche de meilleurs chemins
- 6 Problèmes de planification
- 7 Problèmes NP-difficiles
- 8 Conclusion

Spécification du problème de plus court chemin pour tout couple de sommets

1 Fonction $PlusCourtsChemins(g, c)$

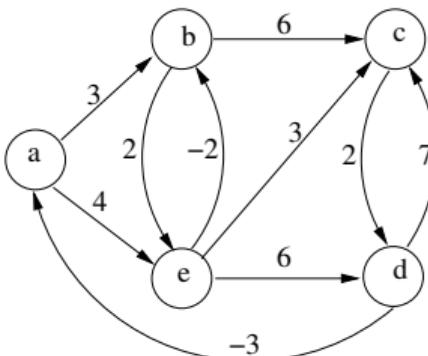
Entrée : Un graphe $g = (S, A)$ et une fct coût $c : A \rightarrow \mathbb{R}$

Postcond. : Retourne une matrice de liaisons π et un tableau d tq $\forall i, j \in S, d[i][j] = \delta(i, j)$

Matrice de liaison : Tableau $\pi : S \times S \rightarrow S \cup \{\text{null}\}$ tel que

- Si $i = j$ ou $i \not\sim j$, alors $\pi[i][j] = \text{null}$
- Sinon : $\pi[i][j] = \text{prédécesseur de } j \text{ dans un plus court chemin de } i \text{ à } j$

Exemple :



Résolution par programmation dynamique

Etape 1 : Définir ce qu'est un sous-problème

Pour tout $k \in [0, \#S]$ et pour tout couple de sommets $(i, j) \in S^2$:

sous-problème $d(k, i, j)$ = longueur du plus court chemin de i jusque j **dont chaque sommet intermédiaire appartient à $\{1, \dots, k\}$**

Etape 2 : Equations de Bellman

- $k = 0 : d(0, i, j) = c(i, j)$ si $(i, j) \in A$ et $d(0, i, j) = +\infty$ sinon
- $k > 1 : d(k, i, j) = \min\{d(k - 1, i, j), d(k - 1, i, k) + d(k - 1, k, j)\}$

Etape 3 : Analyse des performances :

- Combien de sous-problèmes différents pour un graphe de n sommets ?
- Que doit-on faire pour chaque sous-problème ?

Algorithme de Floyd-Warshall

```
1 Fonction Floyd-Warshall( $g, c$ )
2   pour chaque couple de sommets  $(i, j) \in S \times S$  faire
3     si  $(i, j) \in A$  alors  $d[0][i][j] \leftarrow c[i][j];$ 
4     sinon  $d[0][i][j] \leftarrow +\infty;$ 
5   pour  $k$  variant de 1 à  $n$  faire
6     pour chaque couple de sommets  $(i, j) \in S \times S$  faire
7        $d[k][i][j] \leftarrow \min(d[k - 1][i][j], d[k - 1][i][k] + d[k - 1][k][j])$ 
8   retourne  $d[n]$ 
```

Complexité de Floyd-Warshall ?

Calcul de la matrice de liaisons

- Calcul d'une matrice $\pi[k]$ pour chaque $k \in [0, n]$:
 - $\pi[0]$ correspond à la matrice d'adjacence de g
 - $\pi[k][i][j] = \pi[k - 1][i][j]$ ou $\pi[k - 1][i][k] + \pi[k - 1][k][j]$ selon le min (ligne 7)

Algorithme de Floyd-Warshall

```

1 Fonction Floyd-Warshall( $g, c$ )
2   pour chaque couple de sommets  $(i, j) \in S \times S$  faire
3     si  $(i, j) \in A$  alors  $d[0][i][j] \leftarrow c[i][j];$ 
4     sinon  $d[0][i][j] \leftarrow +\infty;$ 
5   pour  $k$  variant de 1 à  $n$  faire
6     pour chaque couple de sommets  $(i, j) \in S \times S$  faire
7        $d[k][i][j] \leftarrow \min(d[k - 1][i][j], d[k - 1][i][k] + d[k - 1][k][j])$ 
8   retourne  $d[n]$ 
```

Complexité de Floyd-Warshall : $\mathcal{O}(n^3)$

Calcul de la matrice de liaisons

- Calcul d'une matrice $\pi[k]$ pour chaque $k \in [0, n]$:
 - $\pi[0]$ correspond à la matrice d'adjacence de g
 - $\pi[k][i][j] = \pi[k - 1][i][j]$ ou $\pi[k - 1][i][j] + \pi[k - 1][i][k]$ selon le min (ligne 7)

Algorithme de Floyd-Warshall

```
1 Fonction Floyd-Warshall( $g, c$ )
2   pour chaque couple de sommets  $(i, j) \in S \times S$  faire
3     si  $(i, j) \in A$  alors  $d[0][i][j] \leftarrow c[i][j]$ ;
4     sinon  $d[0][i][j] \leftarrow +\infty$ ;
5   pour  $k$  variant de 1 à  $n$  faire
6     pour chaque couple de sommets  $(i, j) \in S \times S$  faire
7        $d[k][i][j] \leftarrow \min(d[k - 1][i][j], d[k - 1][i][k] + d[k - 1][k][j])$ 
8   retourne  $d[n]$ 
```

Complexité de Floyd-Warshall : $\mathcal{O}(n^3)$

Calcul de la matrice de liaisons

- Calcul d'une matrice $\pi[k]$ pour chaque $k \in [0, n]$:
 - $\pi[0]$ correspond à la matrice d'adjacence de g
 - $\pi[k][i][j] = \pi[k - 1][i][j]$ ou $\pi[k - 1][k][j]$ selon le min (ligne 7)

- 1 Introduction
- 2 Structures de données pour représenter un graphe
- 3 Arbres Couvrants Minimaux
- 4 Parcours de graphes
- 5 Plus courts chemins
 - Plus courts chemins à origine unique
 - Plus courts chemins pour tout couple de sommets
 - Généralisation à la recherche de meilleurs chemins
- 6 Problèmes de planification
- 7 Problèmes NP-difficiles
- 8 Conclusion

Problèmes de recherche de meilleurs chemins

Exemple 1 : Chemin le plus long

- Coût d'un chemin π = somme des coûts des arcs de π
- Objectif = Chercher un chemin de coût maximal

Exemple 2 : Chemin le plus probable

- Coût d'un chemin π = produit des probabilités des arcs de π
- Objectif = Chercher un chemin de coût maximal

Exemple 3 : Chemin le plus court avec une borne b sur la probabilité

- Coût d'un chemin π :
 - Si le produit des probabilités des arcs de π est inférieur à b alors coût de $\pi = \infty$
 - Sinon coût de π = somme des coûts des arcs de π
- Objectif = Chercher un chemin de coût minimal

Peut-on adapter les algorithmes de calcul de plus court chemins ?

Préconditions communes à tous les algorithmes vus :

- Pas de circuits absorbants
- Optimalité des sous-structures : Tout sous-chemin d'un chemin optimal est optimal

L'optimalité des sous-structures est-elle vérifiée pour les 3 exemples ?

- Chemin le plus long ?
- Chemin le plus probable ?
- Chemin le plus court avec une borne sur la probabilité ?

Que faire si l'optimalité des sous-structures n'est pas vérifiée ?

Le problème devient \mathcal{NP} -difficile

~ On y reviendra plus tard !

Extension de Dijkstra pour calculer un “meilleur” chemin

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

Extension de Dijkstra pour calculer un “meilleur” chemin

Précondition à l'utilisation de Dijkstra :

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

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

Exemple : Chemin le plus probable

- OK car $\text{cout}(s_0 \leadsto s_i \rightarrow s_j) = \text{cout}(s_0 \leadsto s_i) * p(s_i, s_j) \leq \text{cout}(s_0 \leadsto s_i)$ (car $0 \leq p(s_i, s_j) \leq 1$)
- Initialiser d à 0, sauf pour $d[s_0]$ qui est initialisé à 1
- Procédure de relâchement :
 - 1 **Proc** *relacher* $((s_i, s_j), \pi, d)$
 - 2 **si** $d[s_j] < d[s_i] * p(s_i, s_j)$ **alors**
 - 3 $d[s_j] \leftarrow d[s_i] * p(s_i, s_j)$
 - 4 $\pi[s_j] \leftarrow s_i$

Extension de Dijkstra pour calculer un “meilleur” chemin

Précondition à l'utilisation de Dijkstra :

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

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

Exemple : Chemin maximisant le coût de son plus petit arc

- OK car $\text{cout}(s_0 \leadsto s_i \rightarrow s_j) = \min(\text{cout}(s_0 \leadsto s_i), \text{cout}(s_i, s_j)) \leq \text{cout}(s_0 \leadsto s_i)$
- Initialiser d à 0, sauf pour $d[s_0]$ qui est initialisé à ∞
- Procédure de relâchement :
 - 1 **Proc** *relacher* $((s_i, s_j), \pi, d)$
 - 2 si $d[s_j] < \min(d[s_i], \text{cout}(s_i, s_j))$ alors
 - 3 $d[s_j] \leftarrow \min(d[s_i], \text{cout}(s_i, s_j))$
 - 4 $\pi[s_j] \leftarrow s_i$

Extension de TopoDAG pour calculer un “meilleur” chemin

Précondition à l'utilisation de TopoDAG :

Le graphe ne doit pas avoir de circuit

Exemple d'adaptation de TopoDAG :

Recherche d'un plus long chemin

- Initialiser d à $-\infty$, sauf pour $d[s_0]$ qui est initialisé à 0

- Procédure de relâchement :

```
1 Proc relacher(( $s_i, s_j$ ),  $\pi$ ,  $d$ )
2   si  $d[s_j] < d[s_i] + cout(s_i, s_j)$  alors
3      $d[s_j] \leftarrow d[s_i] + cout(s_i, s_j)$ 
4      $\pi[s_j] \leftarrow s_i$ 
```

Extension de TopoDAG pour calculer un “meilleur” chemin

Précondition à l'utilisation de TopoDAG :

Le graphe ne doit pas avoir de circuit

Exemple d'adaptation de TopoDAG :

Recherche d'un plus long chemin

- Initialiser d à $-\infty$, sauf pour $d[s_0]$ qui est initialisé à 0

- Procédure de relâchement :

```
1 Proc relacher(( $s_i, s_j$ ),  $\pi$ ,  $d$ )
2   si  $d[s_j] < d[s_i] + cout(s_i, s_j)$  alors
3      $d[s_j] \leftarrow d[s_i] + cout(s_i, s_j)$ 
4      $\pi[s_j] \leftarrow s_i$ 
```

Aurait-on pu utiliser Dijkstra dans ce cas ?

Extensions de Bellman-Ford et Floyd-Warshall

Préconditions à l'utilisation de Bellman-Ford ou Floyd-Warshall

- Pas de circuit absorbant
- Optimalité des sous-structures

Exemples :

- Chemin maximisant la somme des coûts :
 - ~ Pas de circuit dont la somme des coûts est positive
- Chemin maximisant le produit des coûts :
 - ~ Pas de circuit dont le produit des coûts est supérieur à 1

1 **Introduction**

2 **Structures de données pour représenter un graphe**

3 **Arbres Couvrants Minimaux**

4 **Parcours de graphes**

5 **Plus courts chemins**

6 **Problèmes de planification**

- Définition des problèmes de planification
- Résolution d'un problème de planification avec A*

7 **Problèmes NP-difficiles**

8 **Conclusion**

Définition d'un problème de planification

Données en entrée du problème :

- Un ensemble (éventuellement infini) d'états E
- Un état initial $e_{init} \in E$ et un ensemble d'états finaux $F \subseteq E$
- Un ensemble d'actions A et une fonction $actions : E \rightarrow \mathcal{P}(A)$
 $\leadsto actions(e)$ = ensemble des actions pouvant être appliquées à l'état e
- Une fonction de transition $t : E \times A \rightarrow E$
 \leadsto si $a \in actions(e)$ alors $t(e, a)$ = état obtenu quand on applique a sur e
- Eventuellement : une fonction de coût $c : E \times A \rightarrow \mathbb{R}$

Données en sortie :

- Un plan d'action = $e_1 \xrightarrow{a_1} e_2 \xrightarrow{a_2} e_3 \xrightarrow{a_3} \dots e_n \xrightarrow{a_n} e_{n+1}$ tel que
 - $e_1 = e_{init}$
 - $e_{n+1} \in F$
 - $\forall i \in [1, n], a_i \in actions(e_i)$ et $e_{i+1} = t(e_i, a_i)$
- Variantes : Trouver le plan minimisant le nombre d'actions ou la somme des coûts des actions

Exemple 1 : Le compte est bon

Objectif du problème :

Trouver comment calculer un nombre x à partir d'un ensemble N de nombres

Formalisation du problème :

- Chaque état est un ensemble de nombres
- L'état initial est N et F est l'ensemble des états contenant x
- Une action est l'application d'une opération ($+, -, *$ ou $/$) à deux nombres
- $actions(e) = \{x \ op \ y \mid \{x, y\} \subseteq e, op \in \{+, -, *, /\}\}$
- $t(e, x \ op \ y) = e \setminus \{x, y\} \cup \{x \ op \ y\}$

Plan pour $x = 321$ et $N = \{1, 3, 4, 5, 7, 10, 25\}$:

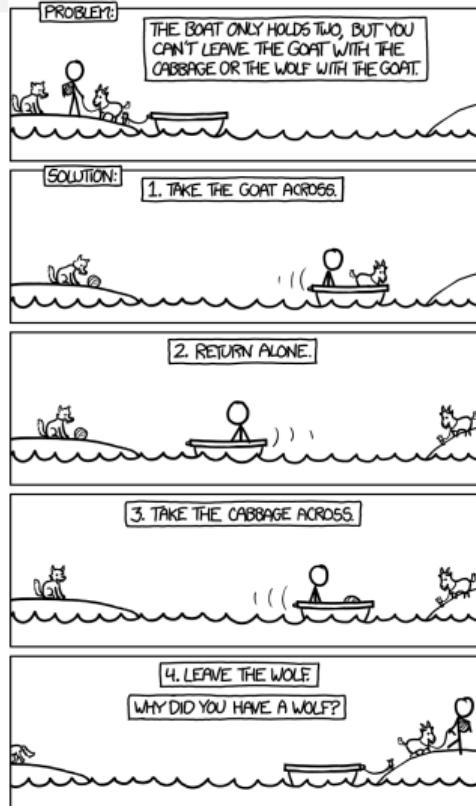
- $e_1 = \{1, 3, 4, 5, 7, 10, 25\}$ et $a_1 = 25 * 3$
- $e_2 = \{1, 4, 5, 7, 10, 75\}$ et $a_2 = 75 + 5$
- $e_3 = \{1, 4, 7, 10, 80\}$ et $a_3 = 80 * 4$
- $e_4 = \{1, 7, 10, 320\}$ et $a_4 = 320 + 1$
- $e_5 = \{7, 10, 321\}$

Ex. 2 : Traversées de rivières (ou de ponts, etc) (1/2)

Objectif du problème :

Faire traverser une rivière à un groupe en respectant des contraintes

Image : <https://xkcd.com/1134/>



Exemple 2 : Traversées de rivières (ou de ponts, etc) (2/2)

Formalisation du problème pour un ensemble P de personnes :

- E = ensemble des partitions de P en 2 parties (une de chaque côté de la rivière)
- L'état initial est (P, \emptyset) et $F = \{(\emptyset, P)\}$
- Une action est le passage de personnes d'un côté à l'autre
- $actions(P_1, P_2) = \{(x, 1)|x \subseteq P_1\} \cup \{(x, 2)|x \subseteq P_2\}$ tel que contraintes satisfaites
- $t((P_1, P_2), (x, 1)) = (P_1 \setminus x, P_2 \cup x)$ et $t((P_1, P_2), (x, 2)) = (P_1 \cup x, P_2 \setminus x)$

Plan pour faire traverser un chou, une brebis et un loup par un passeur :

$P = \{C, B, L, P\}$ et contraintes = ne pas laisser C et B seuls, ni B et L seuls

- | | |
|---|--|
| • $e_1 = (\{C, B, L, P\}, \emptyset)$, $a_1 = (\{B, P\}, 1)$ | • $e_5 = (\{L, P, B\}, \{C\})$, $a_5 = (\{L, P\}, 1)$ |
| • $e_2 = (\{C, L\}, \{B, P\})$, $a_2 = (\{P\}, 2)$ | • $e_6 = (\{B\}, \{C, L, P\})$, $a_6 = (\{P\}, 2)$ |
| • $e_3 = (\{C, L, P\}, \{B\})$, $a_3 = (\{C, P\}, 1)$ | • $e_7 = (\{B, P\}, \{C, L\})$, $a_7 = (\{P, B\}, 1)$ |
| • $e_4 = (\{L\}, \{B, C, P\})$, $a_4 = (\{P, B\}, 2)$ | • $e_8 = (\emptyset, \{P, B, C, L\})$ |

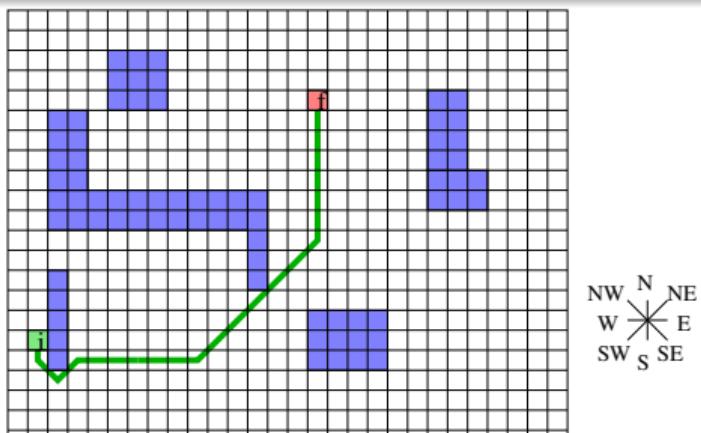
Exemple 3 : Recherche de chemin dans une grille

Objectif du problème :

Aller de i à f dans une grille tq pour chaque cellule c_i , $\text{free}(c_i) = \text{vrai}$ si c_i n'est pas un obstacle

Formalisation du problème :

- $E = \{c_i | \text{free}(c_i)\}$
- $\text{actions}(c_i) = \{d \in \{N, NW, W, SW, S, SE, E, NE\} | \text{free}(t(c_i, d))\}$
où $t(c_i, d)$ = cellule voisine de c_i dans la direction d



Exemple de plan :

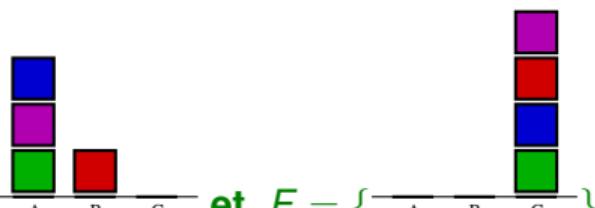
- $a_1 = S$
- $a_2 = SE$
- $a_3 = NE$
- $a_4 = \dots = a_9 = E$
- $a_{10} = \dots = a_{15} = NE$
- $a_{16} = \dots = a_{21} = N$

Exemple 4 : Le monde des blocs

Objectif : Programmer un robot pour qu'il change la configuration de blocs

Formalisation du problème :

- E = ensemble des configurations de x blocs sur y piles
- L'état initial et l'état final sont 2 configurations données
- Action = faire passer un bloc d'une pile à une autre
- $actions(c) = \{(p_1, p_2) | p_1 \text{ est une pile contenant au moins 1 bloc}\}$
- $t((p_1, p_2), c) = \text{faire passer le bloc au sommet de } p_1 \text{ sur } p_2$



Exemple de plan pour $E_{init} =$ et $F = \{$

Actions du plan : $(A, B), (A, B), (A, C), (B, A), (B, C), (B, C), (A, C)$

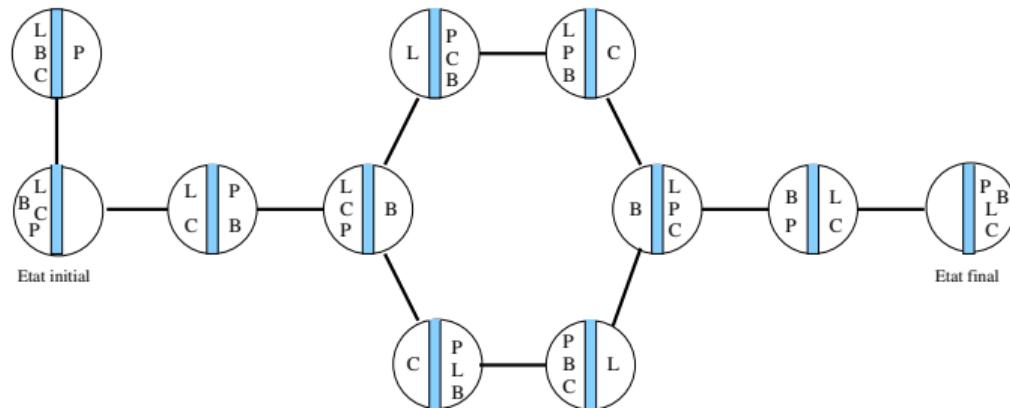
Modéliser un problème de planification à l'aide d'un graphe

Définition du graphe d'états (aussi appelé graphe états-transitions)

- Les sommets du graphe sont les états
- Les arcs correspondent aux transitions possibles entre états
- Les arcs sont étiquetés par les actions

Graphe orienté ou non selon que les transitions sont symétriques ou non

Exemple : Graphe d'états pour la traversée de rivière



Résoudre un problème de planification modélisé par un graphe d'états

Quels algorithmes utiliser pour...

- Déterminer s'il existe un plan d'actions ?
- Chercher le plan d'action minimisant le nombre d'actions ?
- Chercher le plan d'action minimisant la somme des coûts ?
- Compter le nombre de plans d'actions différents possibles ?

Quelles sont les complexités de ces algorithmes ?

- Par rapport au graphe d'états ?
- Par rapport au problème de planification ?

En général, le graphe d'états est trop gros pour pouvoir être construit

~ Construire le graphe au fur et à mesure de la recherche du chemin

Résoudre un problème de planification modélisé par un graphe d'états

Quels algorithmes utiliser pour...

- Déterminer s'il existe un plan d'actions ?
- Chercher le plan d'action minimisant le nombre d'actions ?
- Chercher le plan d'action minimisant la somme des coûts ?
- Compter le nombre de plans d'actions différents possibles ?

Quelles sont les complexités de ces algorithmes ?

- Par rapport au graphe d'états ?
- Par rapport au problème de planification ?

En général, le graphe d'états est trop gros pour pouvoir être construit

~ Construire le graphe au fur et à mesure de la recherche du chemin

- 1 Introduction
- 2 Structures de données pour représenter un graphe
- 3 Arbres Couvrants Minimaux
- 4 Parcours de graphes
- 5 Plus courts chemins
- 6 Problèmes de planification
 - Définition des problèmes de planification
 - Résolution d'un problème de planification avec A*
- 7 Problèmes NP-difficiles
- 8 Conclusion

Adaptons Dijkstra pour chercher le plan d'action le moins coûteux

```

1 Fonction DijkstraPlanif( $E, F, A, e_0, actions, t$ )
2   noirs  $\leftarrow \emptyset$ 
3   gris  $\leftarrow \{e_0\}$ 
4    $d[e_0] \leftarrow 0$ 
5   tant que  $gris \neq \emptyset$  faire
6     Soit  $e_i$  l'état de  $gris$  minimisant  $d[e_i]$ 
7     si  $e_i \in F$  alors retourne  $(d[e_i], \pi)$ ;
8     pour tout  $a \in actions(e_i)$  faire
9        $e_j \leftarrow t(e_i, a)$ 
10      si  $e_j \in noirs$  alors continue;
11      si  $e_j \notin gris$  ou  $d[e_i] + cout(a) < d[e_j]$  alors
12         $d[e_j] \leftarrow d[e_i] + cout(a)$ 
13         $\pi[e_j] \leftarrow e_i$ 
14        si  $e_j \notin gris$  alors Ajouter  $e_j$  dans  $gris$ ;
15      Retirer  $e_i$  de  $gris$  et l'ajouter dans  $noirs$ 
16    retourne Pas de plan!

```

Rappel du code couleur :

- e_i est gris s'il est découvert, mais il a des succ. non explorés
 $\leadsto d[e_i] \leq \delta(e_{init}, e_i)$
- e_i est noir si tous ses succ sont gris ou noirs
 $\leadsto d[e_i] = \delta(e_{init}, e_i)$

Adaptons Dijkstra pour chercher le plan d'action le moins coûteux

```

1 Fonction DijkstraPlanif( $E, F, A, e_0, actions, t$ )
2   noirs  $\leftarrow \emptyset$ 
3   gris  $\leftarrow \{e_0\}$ 
4    $d[e_0] \leftarrow 0$ 
5   tant que gris  $\neq \emptyset$  faire
6     Soit  $e_i$  l'état de gris minimisant  $d[e_i]$ 
7     si  $e_i \in F$  alors retourne ( $d[e_i], \pi$ );
8     pour tout  $a \in actions(e_i)$  faire
9        $e_j \leftarrow t(e_i, a)$ 
10      si  $e_j \in noirs$  alors continue;
11      si  $e_j \notin gris$  ou  $d[e_i] + cout(a) < d[e_j]$  alors
12         $d[e_j] \leftarrow d[e_i] + cout(a)$ 
13         $\pi[e_j] \leftarrow e_i$ 
14        si  $e_j \notin gris$  alors Ajouter  $e_j$  dans gris;
15
16      Retirer  $e_i$  de gris et l'ajouter dans noirs
17
18  retourne Pas de plan!

```

```

void dijkstra(State s0, StateGraph &g){
    map[s0].d = 0; map[s0].isGrey = true; q.push({s0, 0});
    while (!q.empty()){
        State s = q.top().first; q.pop();
        if (!map[s].isGrey) continue;
        if (s.isFinal()){
            printSolution(g, s, map); return;
        }
        int nbActions = g.searchActions(s);
        for (int i=0; i<nbActions; i++){
            State ss = g.transition(s, i);
            if (map.count(ss)==0 ||
                map[s].d + g.getCost(s, i) < map[ss].d){
                map[ss].d = map[s].d + g.getCost(s, i);
                map[ss].pred = s;
                q.push( {ss, map[ss].d} );
            }
        }
        map[s].isGrey = false;
    }
    printf("The problem has no solution\n");
}

```

Implémentation (cf archive [dijkstra.tgz](#) sur Moodle) :

- map associe à chaque état e_i découvert les valeurs de $d[e_i]$ et $\pi[e_i]$, et sa couleur (Booléen isGrey)
- q contient les sommets gris
~ Attention : quand $map[ss].d$ diminue, ss est remis dans q

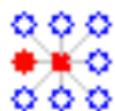
Graphe initial :

Quel est le problème avec Dijkstra ?



Itération 1 :

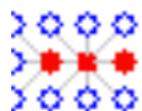
Quel est le problème avec Dijkstra ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 2 :

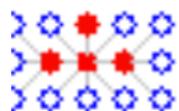
Quel est le problème avec Dijkstra ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 3 :

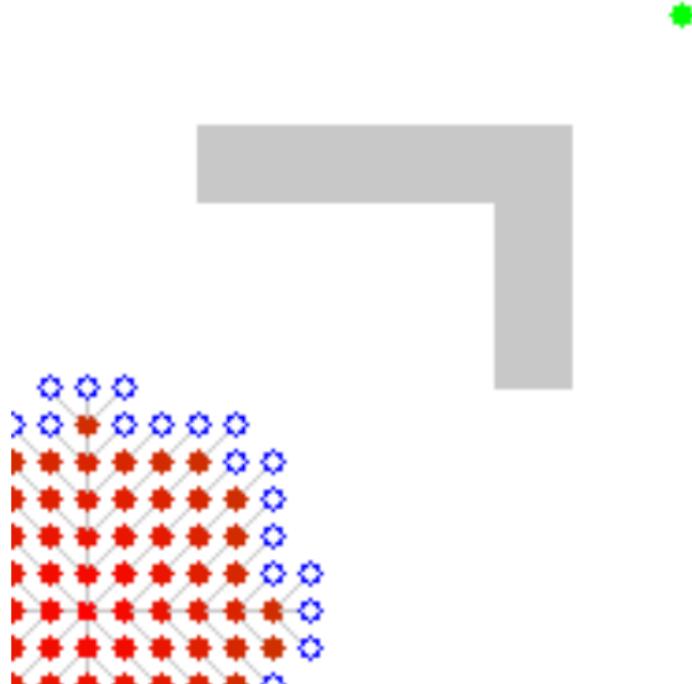
Quel est le problème avec Dijkstra ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 50 :

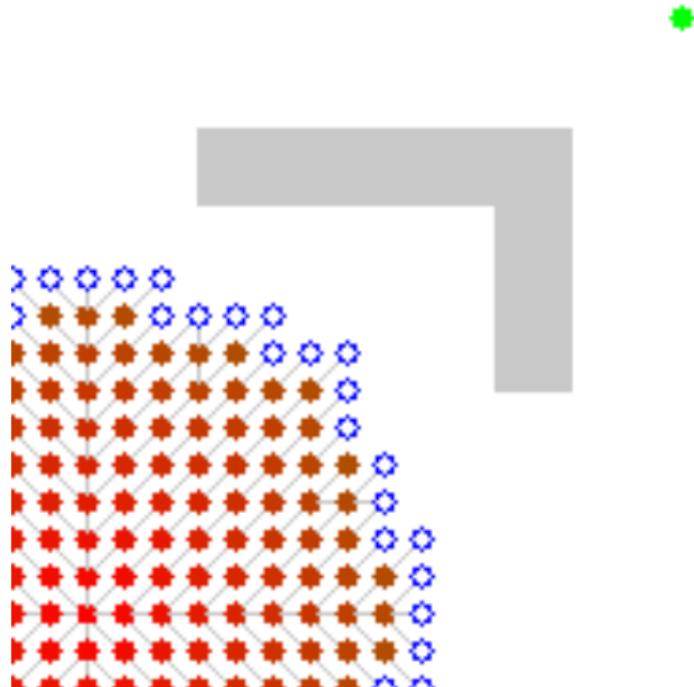
Quel est le problème avec Dijkstra ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 100 :

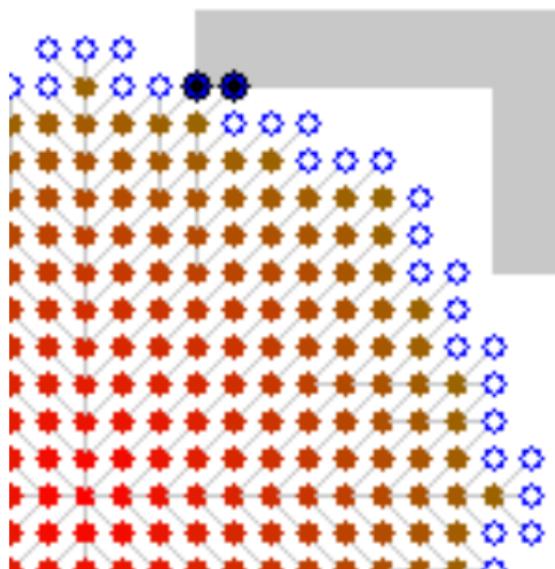
Quel est le problème avec Dijkstra ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 150 :

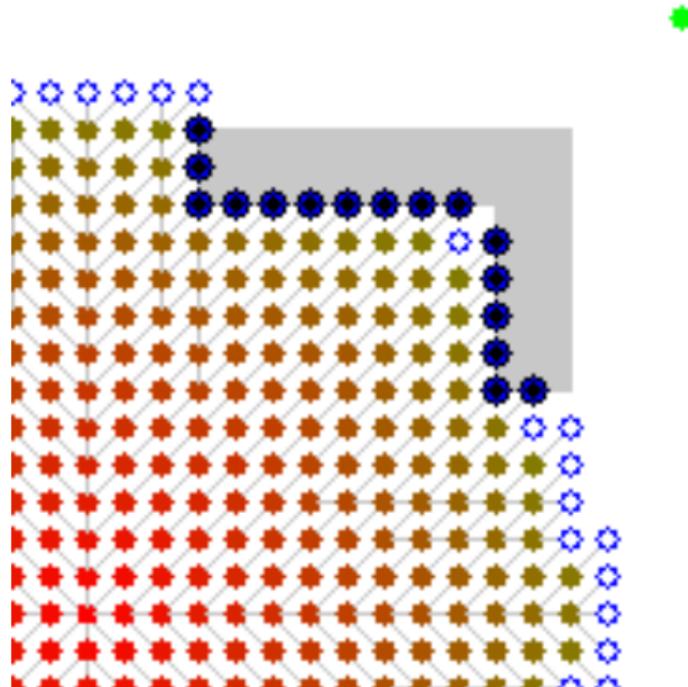
Quel est le problème avec Dijkstra ?



Subh83, CC BY 3.0, via Wikimedia Commons

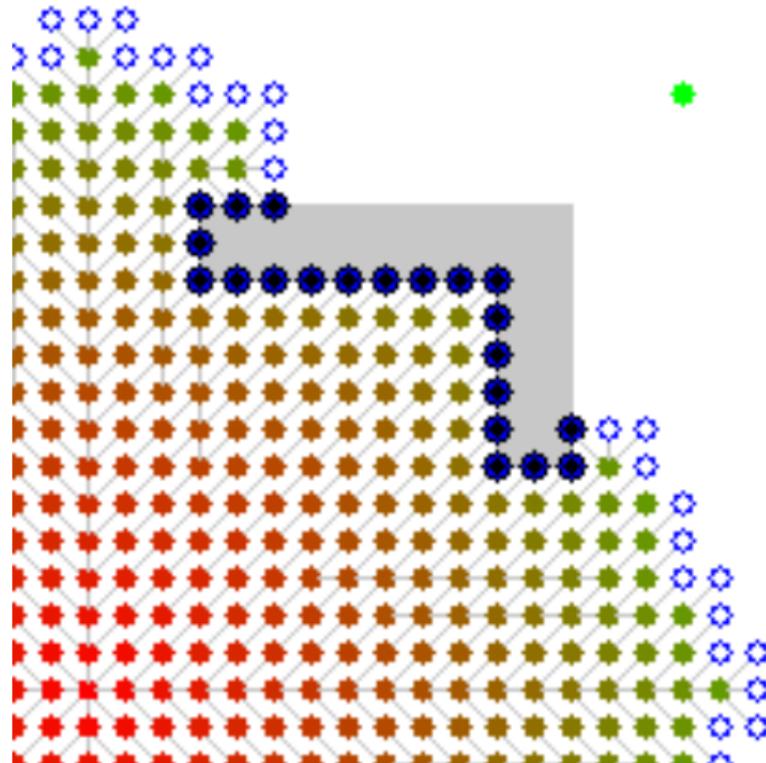
Itération 200 :

Quel est le problème avec Dijkstra ?



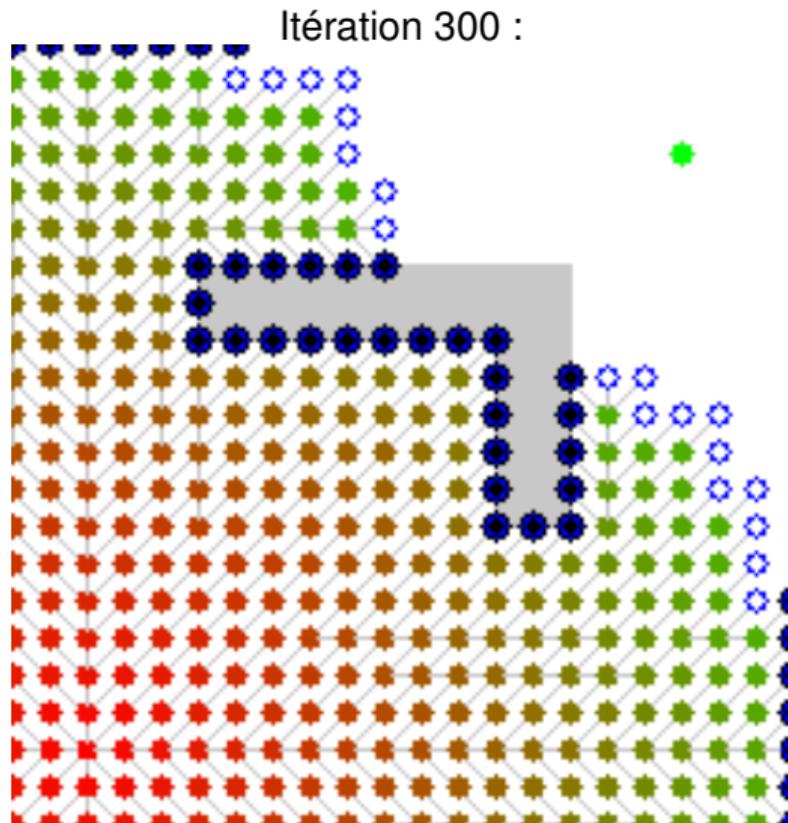
Subh83, CC BY 3.0, via Wikimedia Commons

Itération 250 :



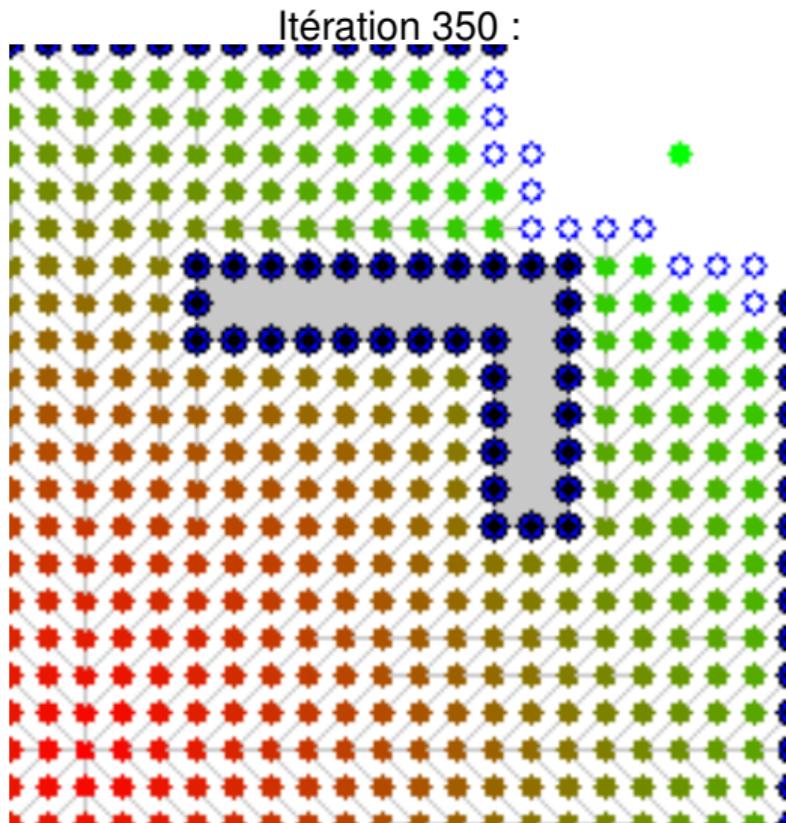
Quel est le problème avec Dijkstra ?

Quel est le problème avec Dijkstra ?



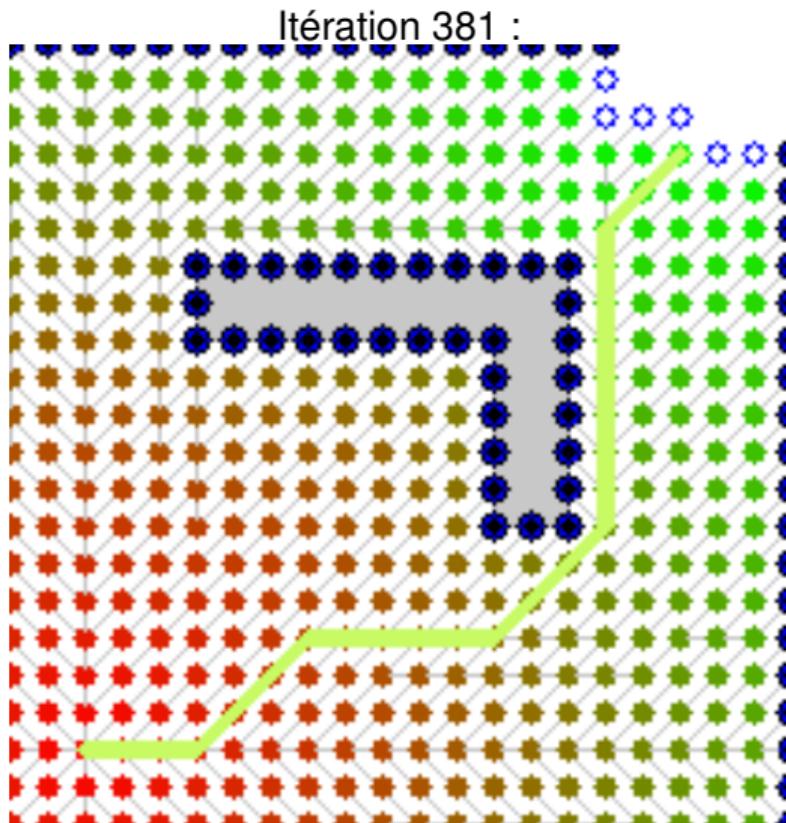
Subh83, CC BY 3.0, via Wikimedia Commons

Quel est le problème avec Dijkstra ?



Subh83, CC BY 3.0, via Wikimedia Commons

Quel est le problème avec Dijkstra ?

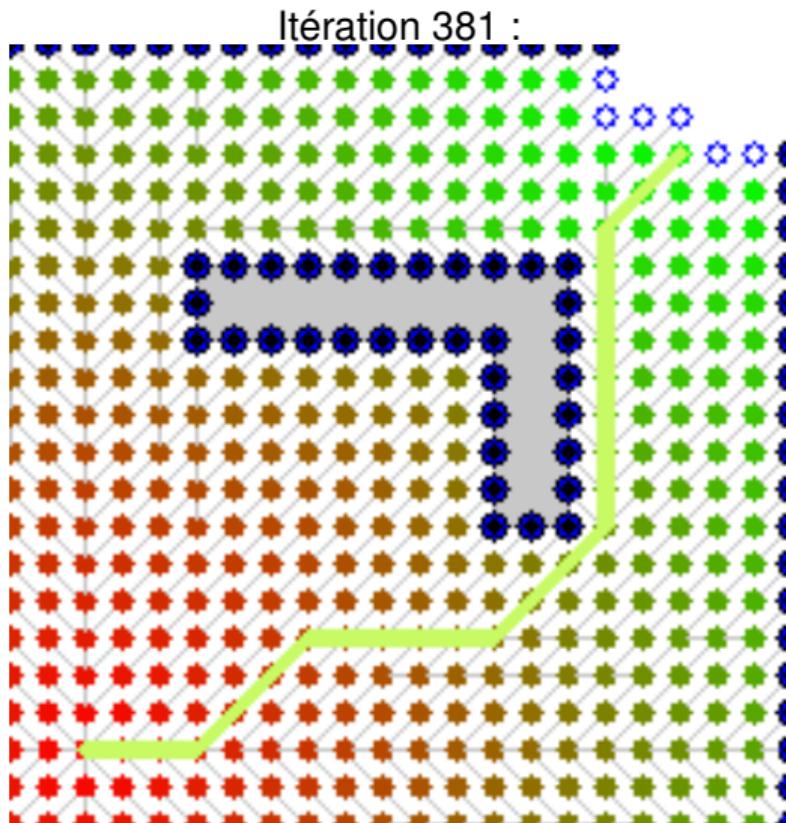


Subh83, CC BY 3.0, via Wikimedia Commons

Quel est le problème avec Dijkstra ?

Il part tous azimuts...

Introduisons une heuristique pour donner le cap !



Subh83, CC BY 3.0, via Wikimedia Commons

Introduction d'une heuristique



Qu'est-ce qu'une heuristique ?

Règle empirique, permettant de résoudre “approximativement” un problème
~ sans garantie sur la qualité de la solution, mais rapide

2001, l'Odyssée de l'espace (1968)

Heuristically programmed ALgorithmic computer

Comment utiliser une heuristique pour guider Dijkstra ?

A chaque itération, choisir l'état gris e minimisant $f(e) = g(e) + h(e)$ où

- $g(e)$ = coût du meilleur chemin connu allant de e_{init} à e
 ~ Correspond au tableau d utilisé dans Dijkstra
- $h(e)$ = estimation heuristique de la distance de e à un état final

Ex : Estimation de la distance entre (x, y) et (x_f, y_f) dans une grille

- Distance Euclidienne : $h_e(x, y) = \sqrt{(x - x_f)^2 + (y - y_f)^2}$
 ~ Ignore les obstacles + déplacement en ligne droite
- Distance de Manhattan : $h_m(x, y) = |x - x_f| + |y - y_f|$
 ~ Ignore les obstacles + déplacements horizontaux et verticaux
- Distance Octile : $h_o(x, y) = \sqrt{2} \min(|x - x_f|, |y - y_f|) + ||x - x_f| - |y - y_f||$
 ~ Ignore les obstacles + déplacements horizontaux, verticaux ou en diagonale

Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Changement de terminologie entre Dijkstra et A* :

- Sommets gris \sim Etats ouverts (*open, fringe, frontier*)
- Sommets noirs \sim Etats fermés (*closed*)
- Sommets blancs \sim Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14
15 retourne Pas de plan !

```

Graphe initial :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 1 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 2 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

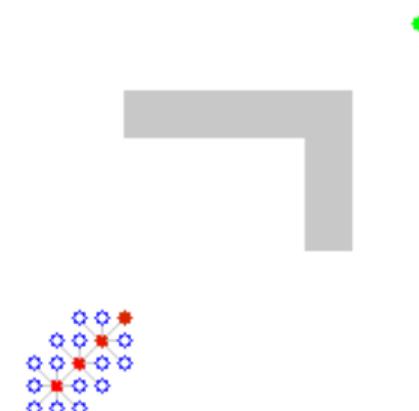
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 3 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

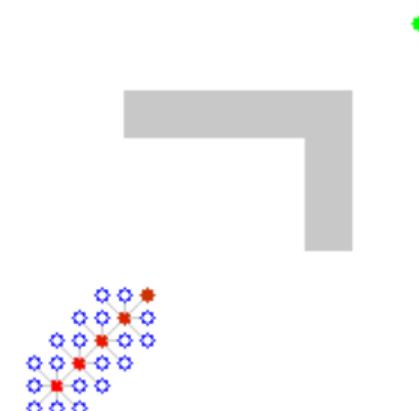
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 4 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

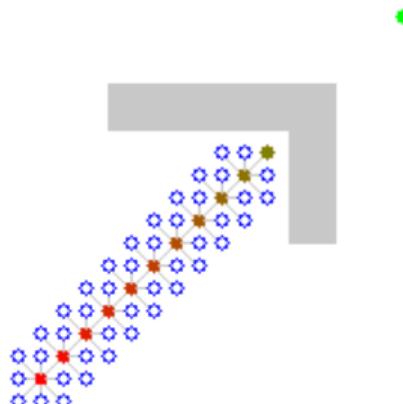
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 10 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

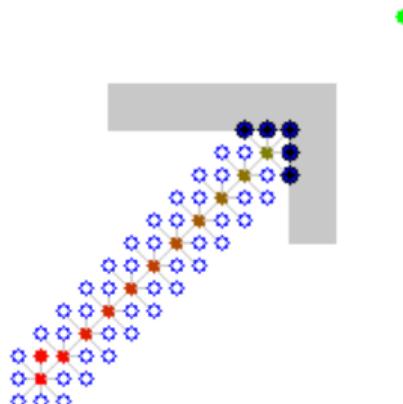
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 11 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

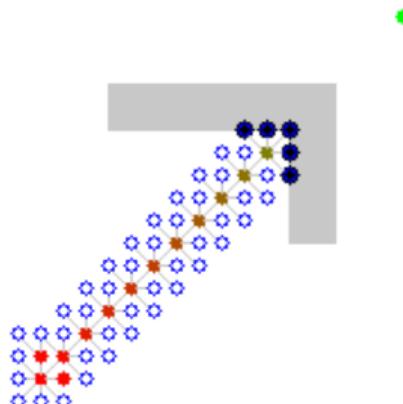
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 12 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

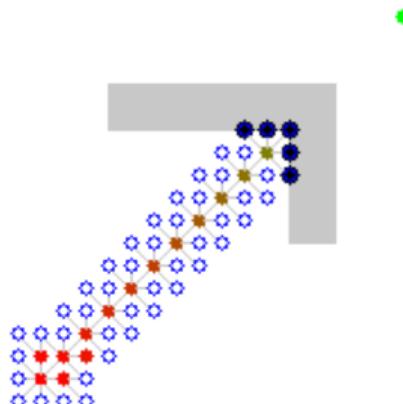
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 13 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

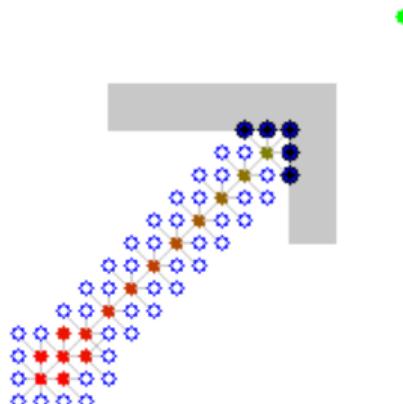
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 14 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

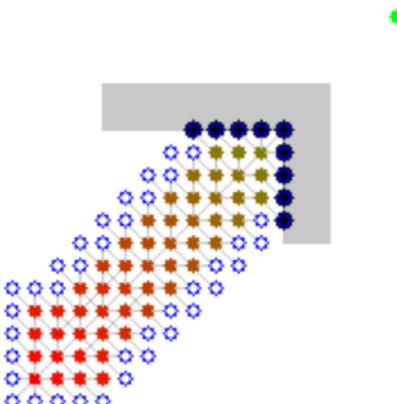
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 50 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

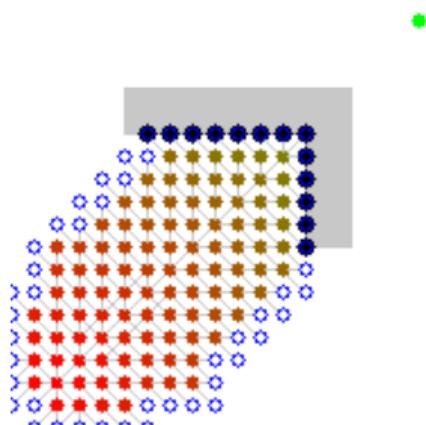
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 100 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

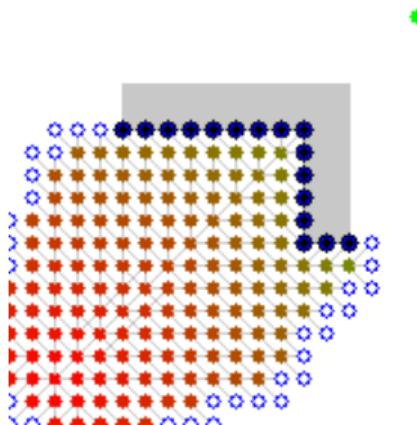
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 150 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

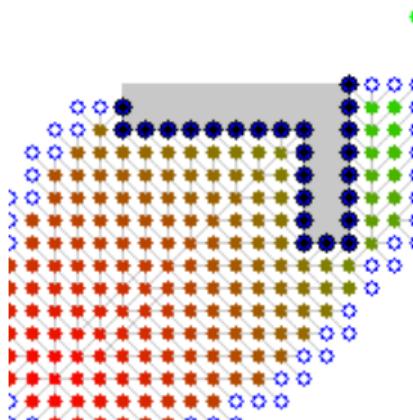
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 170 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

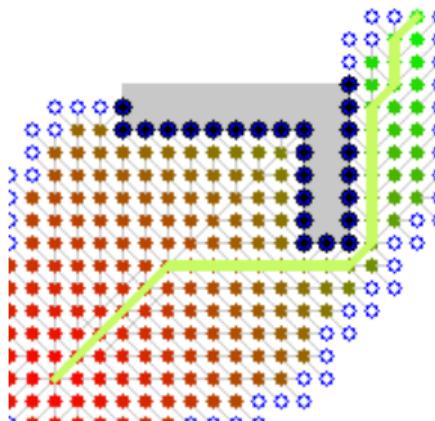
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 194 :



Subh83, CC BY 3.0, via Wikimedia Commons

Changement de terminologie entre Dijkstra et A* :

- Sommets gris ~ Etats ouverts (*open, fringe, frontier*)
- Sommets noirs ~ Etats fermés (*closed*)
- Sommets blancs ~ Etats ni ouverts ni fermés ($g[e_i] = \infty$ pour tout état e_i ni ouvert ni fermé)

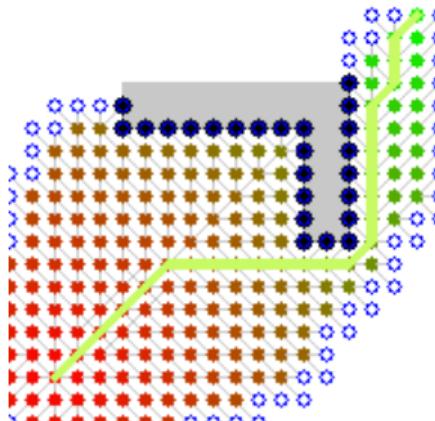
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne ( $g[e_i]$ ,  $\pi$ );
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 194 :



Subh83, CC BY 3.0, via Wikimedia Commons

Condition pour que A* retourne la solution optimale :

- h doit être **admissible**, i.e., $\forall e \in E, h(e) \leq \min_{f \in F} \delta(e, f)$
- Les distances euclidienne, de manhattan et octile sont-elles admissibles ?

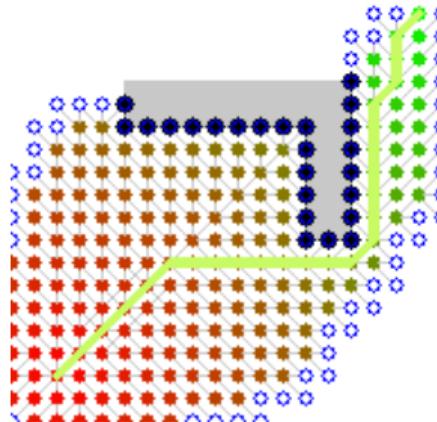
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

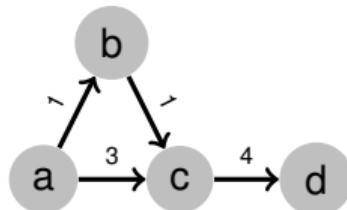
```

Itération 194 :



Subh83, CC BY 3.0, via Wikimedia Commons

A* peut relâcher plusieurs fois un même arc :



Exécuter A* avec $e_{init} = a$ en supposant que :

- $h[a] = 2$
- $h[b] = 4$
- $h[c] = 1$

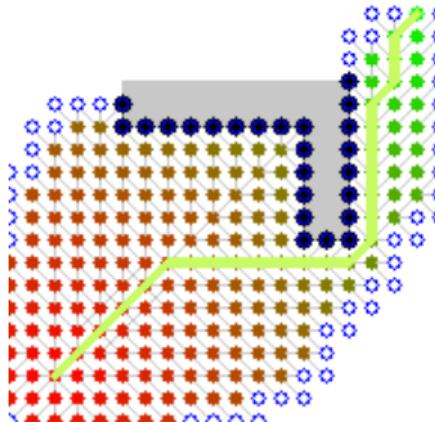
Algorithm A* (Hart, Nilsson, Raphael 1968)

```

1 fermés ← ∅; ouverts ← { $e_{init}$ } ;  $g[e_{init}] \leftarrow 0$ 
2 tant que  $ouverts \neq \emptyset$  faire
3   Soit  $e_i$  l'état de  $ouverts$  minimisant  $g[e_i] + h[e_i]$ 
4   si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
5   pour tout  $a \in actions(e_i)$  faire
6      $e_j \leftarrow t(e_i, a)$ 
7     /* Ne pas ignorer les états fermés ! */
8     si  $g[e_i] + cout(a) < g[e_j]$  alors
9        $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10       $\pi[e_j] \leftarrow e_i$ 
11      si  $e_j \notin ouverts$  alors Ajouter  $e_j$  ds  $ouverts$ ;
12
13 Retirer  $e_i$  de  $ouverts$  et l'ajouter dans  $fermés$ 
14 retourne Pas de plan !

```

Itération 194 :



Subh83, CC BY 3.0, via Wikimedia Commons

Condition pour que chaque arc soit relâché au plus une fois :

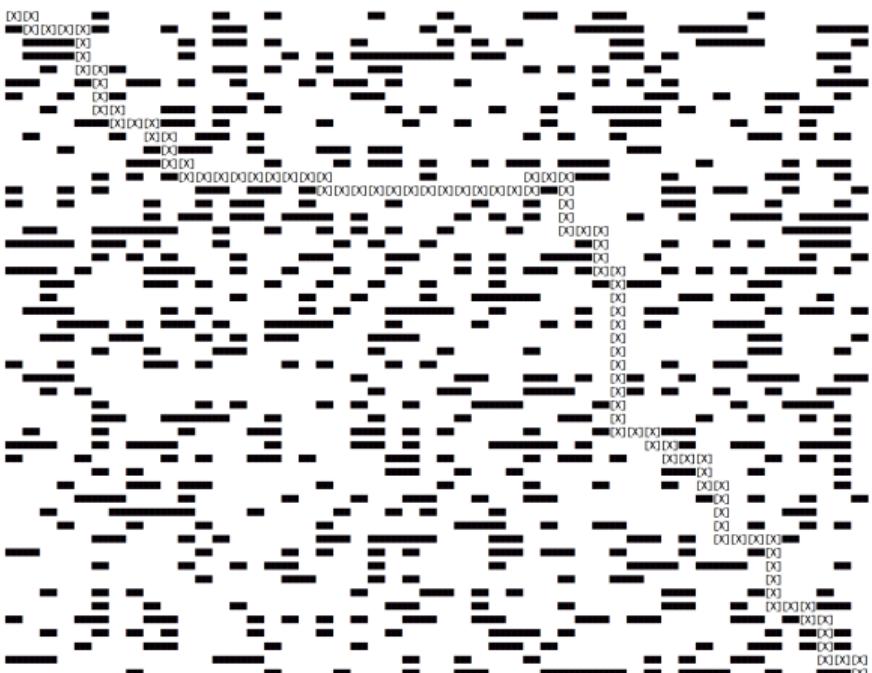
- h doit être cohérente
 $\sim \forall e_i \in E, \forall a \in actions(e_i) : h(e_i) \leq cout(a) + h(t(e_i, a))$
- Montrer que si h est cohérente, alors h est admissible

Influence de l'heuristique h sur A*

Exemple : Chemin de (0, 0) jusque (99, 99) dans une grille 100×100

- $h = 0$ (Dijkstra) :
1510 itérations
- $h =$ distance Euclidienne :
875 itérations
- $h =$ distance de Manhattan :
370 itérations

~ cf aStar.tgz sur Moodle et TP2



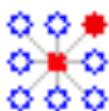
Graphe initial :

Que se passe-t-il si h n'est pas admissible ?



Itération 1 :

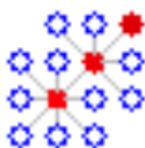
Que se passe-t-il si h n'est pas admissible ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 2 :

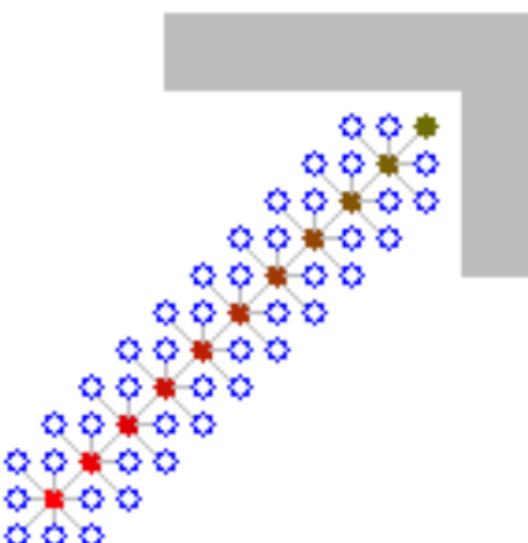
Que se passe-t-il si h n'est pas admissible ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 10 :

Que se passe-t-il si h n'est pas admissible ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 11 :

Que se passe-t-il si h n'est pas admissible ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 12 :

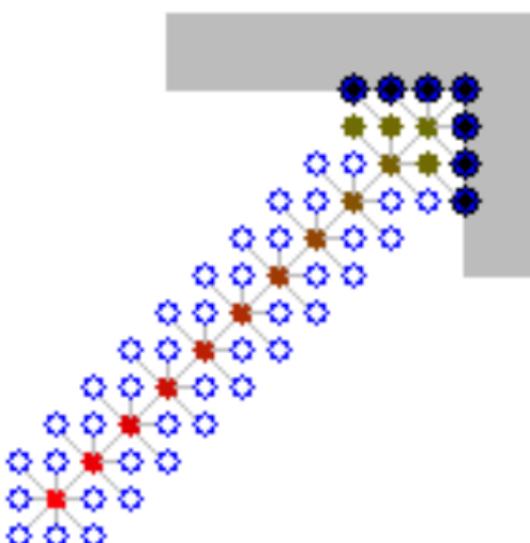
Que se passe-t-il si h n'est pas admissible ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 13 :

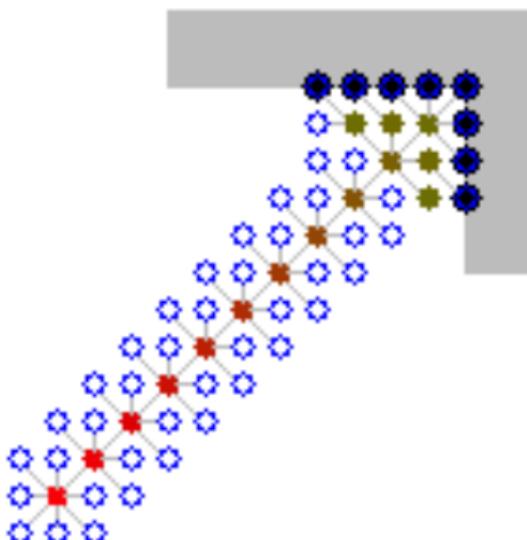
Que se passe-t-il si h n'est pas admissible ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 14 :

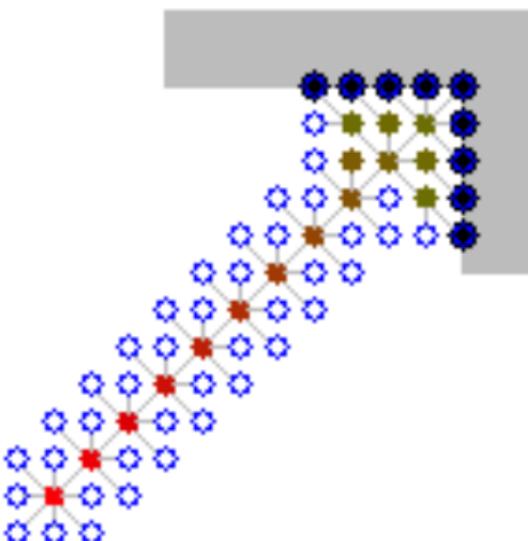
Que se passe-t-il si h n'est pas admissible ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 15 :

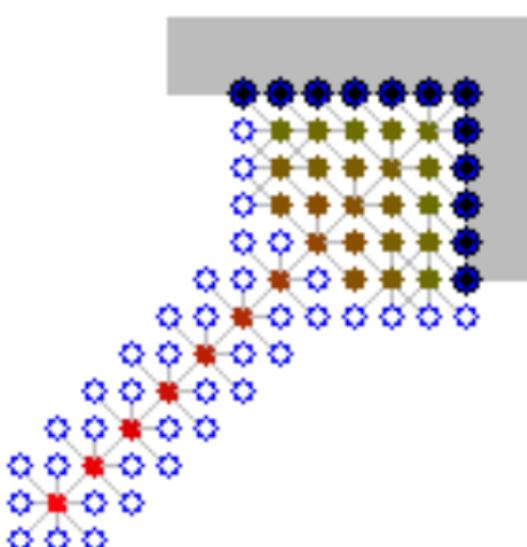
Que se passe-t-il si h n'est pas admissible ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 28 :

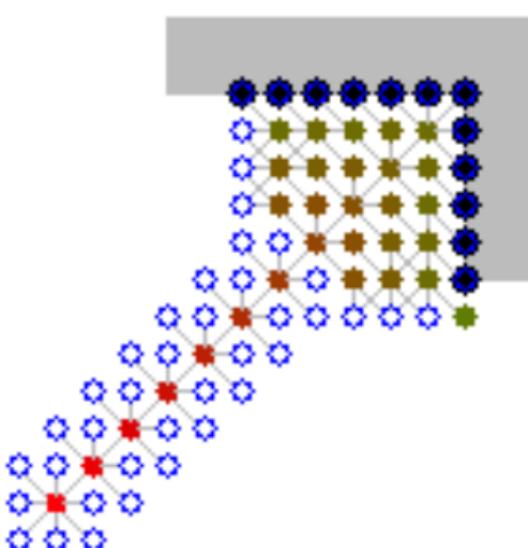
Que se passe-t-il si h n'est pas admissible ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 29 :

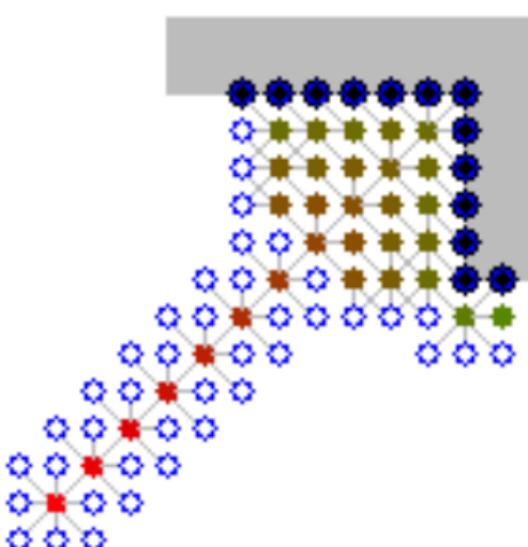
Que se passe-t-il si h n'est pas admissible ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 30 :

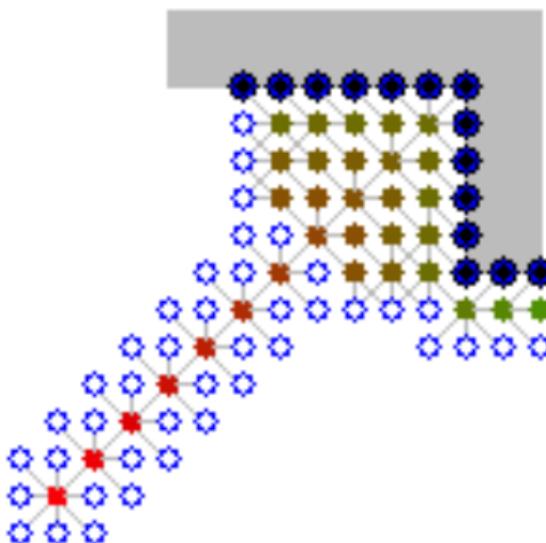
Que se passe-t-il si h n'est pas admissible ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 31 :

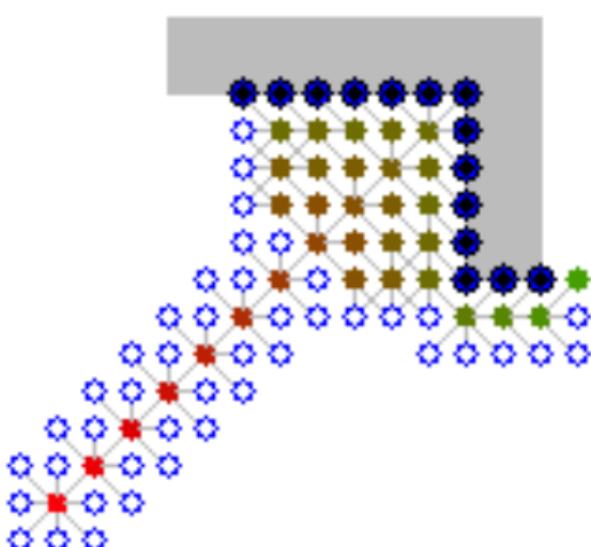
Que se passe-t-il si h n'est pas admissible ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 32 :

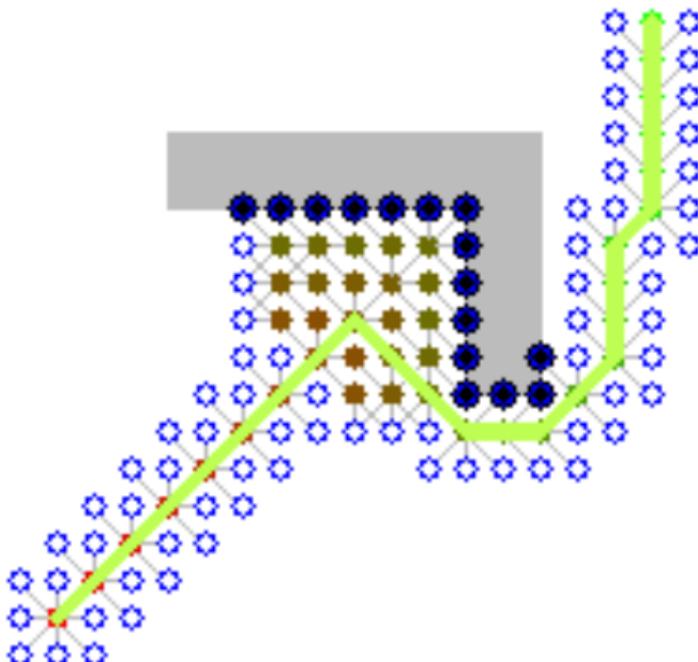
Que se passe-t-il si h n'est pas admissible ?



Subh83, CC BY 3.0, via Wikimedia Commons

Itération 42 :

Que se passe-t-il si h n'est pas admissible ?

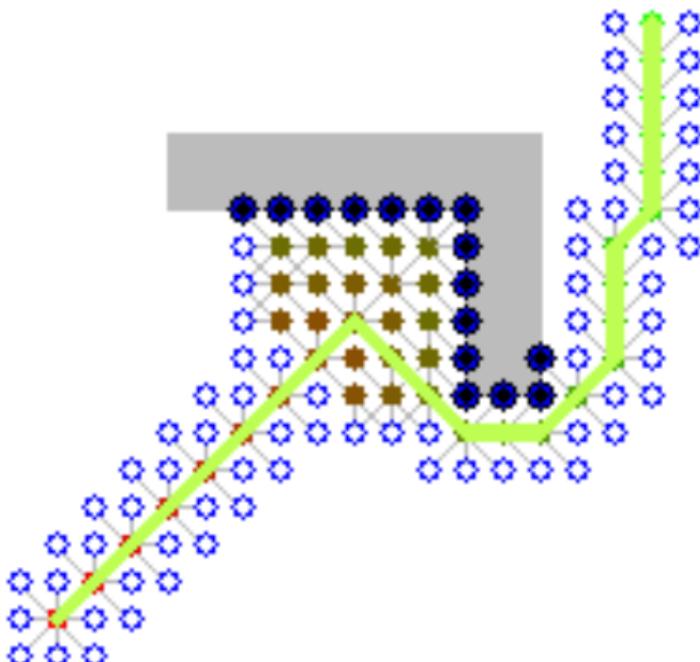


Subh83, CC BY 3.0, via Wikimedia Commons

Itération 42 :

Que se passe-t-il si h n'est pas admissible ?

- A* trouve un chemin sous-optimal en développant moins d'états
- Exploitons cela pour construire un algorithme heuristique !



Subh83, CC BY 3.0, via Wikimedia Commons

Weighted A* (WA*)

```

1 Fonction  $WA^*(E, F, A, e_{init}, actions, t, h, w)$ 
2   fermés  $\leftarrow \emptyset$ ; ouverts  $\leftarrow \{e_{init}\}$ ;  $g[e_{init}] \leftarrow 0$ 
3   tant que gris  $\neq \emptyset$  faire
4     Soit  $e_i$  l'état de ouverts minimisant  $g[e_i] + w * h[e_i]$ 
5     si  $e_i \in F$  alors retourne  $(g[e_i], \pi)$ ;
6     pour tout  $a \in actions(e_i)$  faire
7        $e_j \leftarrow t(e_i, a)$ 
8       si  $g[e_i] + cout(a) < g[e_j]$  alors
9          $g[e_j] \leftarrow g[e_i] + cout(a)$ 
10         $\pi[e_j] \leftarrow e_i$ 
11        si  $e_j \notin ouverts$  alors Ajouter  $e_j$  dans ouverts;
12
13   Retirer  $e_i$  de ouverts et l'ajouter dans fermés
14
15   retourne Pas de plan !

```

Le paramètre w permet de choisir un compromis entre qualité et durée

- Augmenter w accélère l'exécution, mais la solution trouvée est souvent moins bonne
- On peut itérer WA* avec des valeurs de w décroissantes jusqu'à $w = 1$

Exécution de WA* sur le monde des blocs

Exemple avec 4 piles et 23 blocs :

- $w=3$: Plan de coût 47 trouvé en 621 itérations et 0.01s
- $w=2$: Plan de coût 43 trouvé en 1 295 itérations et 0.02s
- $w=1.5$: Plan de coût 41 trouvé en 7 865 itérations et 0.09s
- $w=1.2$: Plan de coût 39 trouvé en 129 127 itérations et 1.47s
- $w=1$ (A*) : Plan de coût 38 trouvé en 4 330 580 itérations et 119s

Exemple avec 4 piles et 30 blocs :

- $w=3$: Plan de coût 76 trouvé en 30 953 itérations et 0.9s
- $w=2$: Plan de coût 62 trouvé en 16 851 itérations et 0.5s
- $w=1.5$: Plan de coût 57 trouvé en 3 868 099 itérations et 128s
- $w=1.2$: Memory out !

Inconvénient d'itérer WA* avec des valeurs de w décroissantes ?

Anytime Weighted A* (AWA*) (Hansen, Zhou 2007)

```

1 Fonction AWA*(E, F, A, einit, actions, t, h, w)
2   fermés ← ∅; ouverts ← {einit}; g[einit] ← 0; borne ← ∞
3   tant que ouverts ≠ ∅ faire
4     Soit ei l'état de ouverts minimisant g[ei] + w * h[ei]
5     pour tout a ∈ actions(ei) faire
6       ej ← t(ei, a)
7       si g[ei] + cout(a) < g[ej] alors
8         g[ej] ← g[ei] + cout(a)
9         π[ej] ← ei
10        si ej ∈ F alors borne ← g[ej]; Afficher borne ;
11        sinon si g[ej] + h[ej] < borne alors Ajouter ej dans ouverts;
12
13   Retirer ei de ouverts et l'ajouter dans fermés
14
15   retourne borne

```

Principales différences avec WA* :

- Test si l'état est final qd on atteint l'état (et non qd on l'enlève de *ouverts*)
 ↳ Pas d'arrêt, mais mise-à-jour et affichage de la borne
- Utilisation de la borne pour élaguer le graphe d'états

Exécution de AWA* sur le monde des blocs

4 piles et 23 blocs, et $w = 2$:

- Coût 42 : 1 904 itérations et 0.03s
- Coût 40 : 6 554 itérations et 0.08s
- Coût 39 : 9 199 itérations et 0.11s
- Coût 38 : 13 281 itérations et 0.14s
- Preuve d'opt. en 2 481 701 itér. et 31s

4 piles et 30 blocs, et $w = 2$:

- Coût 60 : 24 682 itérations et 0.29s
- Coût 59 : 30 053 itérations et 0.34s
- Coût 58 : 31 677 itérations et 0.36s
- Coût 57 : 93 181 itérations et 1.06s
- Coût 56 : 93 686 itérations et 1.07s
- Coût 55 : 766 526 itérations et 9.75s
- Coût 54 : 1 275 117 itérations et 16.03s
- Coût 53 : 5 581 916 itérations et 98.77s
- Coût 52 : 5 595 249 itérations et 99.49s
- Memory out

1 Introduction**2** Structures de données pour représenter un graphe**3** Arbres Couvrants Minimaux**4** Parcours de graphes**5** Plus courts chemins**6** Problèmes de planification**7** Problèmes NP-difficiles

- Classes de complexité
- Exploration exhaustive d'espaces de recherche
- Exploration incomplète d'espaces de recherche

8 Conclusion

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 $tab = \boxed{4 \ 4 \ 7 \ 8 \ 10 \ 11 \ 12} \rightsquigarrow$ Sortie : $i = 3$
- Entrées : $e = 9$ et $tab = \boxed{4 \ 4 \ 7 \ 8 \ 10 \ 11 \ 12} \rightsquigarrow$ Sortie : $i = 7$

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

Spécification du problème :

- Entrées : une formule booléenne F définie sur un ensemble X de n variables
- Sortie : un boolean B
- Précondition : F est sous forme normale conjonctive (CNF)
- Postrelation : F est satisfiable $\Leftrightarrow B=\text{vrai}$

Exemples d'instances :

- $X = \{a, b, c\}$ and $F = (a \vee \bar{b}) \wedge (b \vee c) \wedge (\bar{a} \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee c) \rightsquigarrow V = \text{true}$
- $X = \{a, b, c\}$ and $F = (a \vee \bar{b}) \wedge (b \vee c) \wedge (\bar{a} \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (a \vee b \vee \bar{c}) \rightsquigarrow V = \text{false}$

Reference :

Knuth: The Art of Computer Programming, Vol. 4B, 2022

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

~ Estimation dépendante de la taille n des paramètres en entrée

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

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

- $\mathcal{O}(1)$: constant
- $\mathcal{O}(\log_k(n))$: 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
~ Réponse $\in \{ \text{vrai}, \text{faux} \}$

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**

Ex. : Algorithme pour résoudre SAT sur une machine non déterministe

- 1 pour chaque variable $x_i \in X$ faire **Choisir** une valeur $v_i \in \{\text{vrai}, \text{faux}\}$;
- 2 si F est satisfaite quand $x_1 = v_1, \dots, x_n = v_n$ alors **retourne vrai**;

Réponse = vrai si au moins une branche a retourné vrai

Autrement dit :

$X \in \mathcal{NP}$ si pour toute instance I de X telle que $\text{réponse}(I) = \text{vrai}$, il existe un certificat $c(I)$ permettant de vérifier en temps polynomial que $\text{réponse}(I) = \text{vrai}$

Ex. : Certificat pour SAT = $V : X \rightarrow \{\text{vrai}, \text{faux}\}$ 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 ([Millenium Problems](#))

Problèmes \mathcal{NP} -complets :

- Les problèmes les plus difficiles de la classe \mathcal{NP} :
 $\sim X$ 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
 \sim cf ([Karp 1972](#)), ([Garey and Johnson 1979](#)), ([Wikipedia dynamic list](#)).

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

Problème de réduction entre problèmes

Définition du problème de réduction de P_1 vers P_2 :

- Entrée : une instance I_1 du problème de décision P_1
- Sortie : une instance I_2 du problème de décision P_2
- Postrelation : réponse de $P_1(I_1)$ = réponse de $P_2(I_2)$

Utilisation de réductions pour calculer une borne sur la complexité :

Etant donnés :

- un algorithme R de réduction de P_1 vers P_2
- un algorithme A résolvant le problème P_2 ,

Algorithme pour résoudre une instance I_1 de P_1 = $A(R(I_1))$

⇒ Complexité de $P_1 \leq$ Complexité de A et R

⇒ Si P_1 est \mathcal{NP} -complet, et R et A sont polynomiaux alors $\mathcal{P} = \mathcal{NP}$

Exercice

Description du problème Clique :

- Entrées : un graphe non orienté $G = (V, E)$ et un entier positif k
- Question : Existe-t-il $S \subseteq V$ tel que $\#S = k$ et $\forall \{i, j\} \subseteq S, \{i, j\} \in E$

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

~ Certificat ?

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

~ Réduction de SAT vers Clique :

- Donner un algorithme polynomial résolvant le problème de réduction :
 - Entrée : une instance de SAT = une formule CNF F
 - Sortie : une instance de Clique = un graphe G et un entier k
 - Postrelation : F est satisfiable $\Leftrightarrow G$ contient une clique d'ordre k

Solution

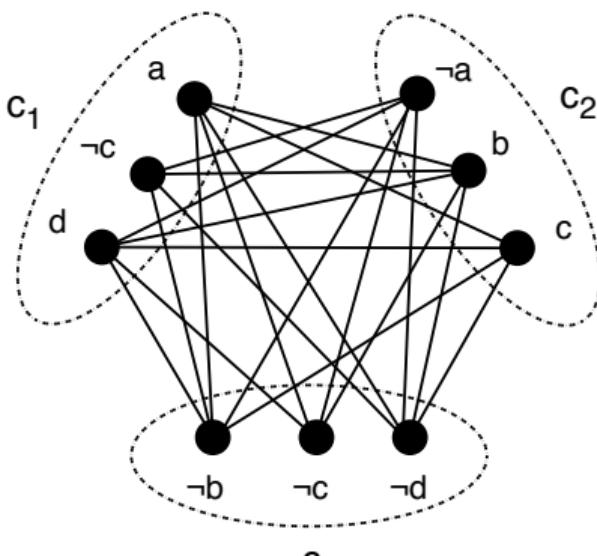
Graphe non orienté $G = (S, A)$ associé à une formule F :

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

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 NP -difficiles

Problèmes au moins aussi difficiles que ceux de NP :

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

Exemple : Problème de la clique exacte

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

Ce problème appartient-il à NP ?

Complexité des problèmes d'optimisation :

- Déterminée en fonction du problème de décision associé
- NP -difficile si le problème de décision est NP -complet

Comment résoudre un problème \mathcal{NP} -difficile ?

Commencer par vérifier que le problème est vraiment \mathcal{NP} -difficile !

Certains cas particuliers de problèmes \mathcal{NP} -difficiles sont dans \mathcal{P}

~ A-t-on bien pris en compte toutes les spécificités du problème à résoudre ?

Calculer une solution approchée avec des algo de ρ -approximation

(Valable uniquement pour les problèmes d'optimisation)

- Calcul en temps polynomial d'une solution de qualité bornée par $\rho * f$
où f = qualité de la solution optimale
- Exemple : approximation du TSP euclidien

Explorer l'espace des combinaisons candidates (espace de recherche)

- Exploration exhaustive par des approches complètes/exactes
~ Garantie d'optimalité, mais complexité exponentielle
- Exploration incomplète par mét-heuristiques
~ Aucune garantie d'optimalité, mais complexité polynomiale

1 Introduction**2** Structures de données pour représenter un graphe**3** Arbres Couvrants Minimaux**4** Parcours de graphes**5** Plus courts chemins**6** Problèmes de planification**7** Problèmes NP-difficiles

- Classes de complexité
- Exploration exhaustive d'espaces de recherche
- Exploration incomplète d'espaces de recherche

8 Conclusion

Comment explorer un espace de recherche de façon exhaustive ?

Etape 1 : Structurer l'espace sous la forme d'un graphe

- Exemple 1 : Graphe états-transition (cf partie 6 de ce cours)
- Exemple 2 : Arbre de recherche
 - Chaque nœud = Ensemble de combinaisons candidates
~~> Racine = ensemble initial
 - Fils d'un nœud s = partition de l'ens. des combi. associées à s

Etape 2 : Parcourir le graphe

- Meilleurs d'abord
 - ~~> Nécessite de mémoriser tous les nœuds "ouverts"
- En profondeur d'abord (récursevement)
 - ~~> Seuls les nœuds entre la racine et le nœud courant sont mémorisés
 - ~~> Ajout de techniques d'élagage/filtrage, d'heuristiques d'ordre, ...

Illustration : Recherche de cliques

Rappel du problème Clique :

- Entrées : un graphe non orienté $G = (V, E)$ et un entier positif k
- Question : Existe-t-il $S \subseteq V$ tel que $\#S = k$ et $\forall \{i, j\} \subseteq S, \{i, j\} \in E$

Quel espace de recherche ?

Illustration : Recherche de cliques

Rappel du problème Clique :

- Entrées : un graphe non orienté $G = (V, E)$ et un entier positif k
- Question : Existe-t-il $S \subseteq V$ tel que $\#S = k$ et $\forall \{i, j\} \subseteq S, \{i, j\} \in E$

Quel espace de recherche ?

Combinaisons candidates = Sous-ensembles de k sommets

~ Comment énumérer tous les sous-ensembles de k sommets ?

Enumération de sous-ensembles : V1

1 Proc *enum1*(S, c, k)

Entrée : Un ensemble S , un ensemble c , un entier k

Postcond. : Affiche tout ensemble c' tq $c \subseteq c' \subseteq c \cup S$ et $\#c' = k$

2 **si** $\#c = k$ **alors** afficher c ;

3 **sinon si** $\#c < k$ **alors**

4 **pour** chaque sommet $i \in S$ **faire** *enum1*($S \setminus \{i\}, c \cup \{i\}, k$) ;

Chaque appel récursif correspond à un nœud de l'arbre de recherche

Si $\#c < k$, fils de *enum1*(S, c, k) = $\{enum1(S \setminus \{i\}, c \cup \{i\}, k) | i \in S\}$

Exercice

Dessiner l'arbre de recherche dont la racine est *enum1*($\{a, b, c\}, \emptyset, 2$)

Quel est l'inconvénient de cette procédure ?

Enumération de sous-ensembles : V1

1 Proc *enum1*(S, c, k)

Entrée : Un ensemble S , un ensemble c , un entier k

Postcond. : Affiche tout ensemble c' tq $c \subseteq c' \subseteq c \cup S$ et $\#c' = k$

2 si $\#c = k$ alors afficher c ;

3 sinon si $\#c < k$ alors

4 pour chaque sommet $i \in S$ faire *enum1*($S \setminus \{i\}, c \cup \{i\}, k$);

Chaque appel récursif correspond à un nœud de l'arbre de recherche

Si $\#c < k$, fils de *enum1*(S, c, k) = $\{\text{iota}(S \setminus \{i\}, c \cup \{i\}, k) | i \in S\}$

Exercice

Dessiner l'arbre de recherche dont la racine est *enum1*($\{a, b, c\}, \emptyset, 2$)

Quel est l'inconvénient de cette procédure ?

Elle énumère plusieurs fois un même sous-ensemble

Enumération de sous-ensembles : V2

1 Proc *enum2*(S, c, k)

Entrée : Un ensemble S , un ensemble c , un entier k

Postcond. : Affiche tout ensemble c' tq $c \subseteq c' \subseteq c \cup S$ et $\#c' = k$

2 si $\#c = k$ alors afficher c ;

3 sinon si $\#c < k$ alors

4 pour chaque sommet $i \in S$ tel que $\forall j \in c, i > j$ faire *enum2*($S \setminus \{i\}, c \cup \{i\}, k$) ;

Exercice

Dessiner l'arbre de recherche dont la racine est *enum2*($\{a, b, c, d\}, \emptyset, 2$)

Comment adapter cette procédure à la recherche de cliques ?

Enumération de sous-ensembles : V2

1 Proc *enum2*(S, c, k)

Entrée : Un ensemble S , un ensemble c , un entier k

Postcond. : Affiche tout ensemble c' tq $c \subseteq c' \subseteq c \cup S$ et $\#c' = k$

2 si $\#c = k$ alors afficher c ;

3 sinon si $\#c < k$ alors

4 pour chaque sommet $i \in S$ tel que $\forall j \in c, i > j$ faire *enum2*($S \setminus \{i\}, c \cup \{i\}, k$);

Exercice

Dessiner l'arbre de recherche dont la racine est *enum2*($\{a, b, c, d\}, \emptyset, 2$)

Comment adapter cette procédure à la recherche de cliques ?

~ Se limiter aux nœuds correspondant à des cliques

Recherche d'une clique d'ordre k

1 Fonction $\text{clique}(g, cand, c, k)$

Entrée : graphe $g = (S, A)$, $cand \subseteq S$, $c \subseteq S$ et entier k

Précond. : c est une clique de taille inférieure ou égale à k , $cand \cap c = \emptyset$, et $\forall i \in cand, \forall j \in c, \{i, j\} \in A$

Postcond. : Retourne vrai si \exists une clique c' de g tq $\#c' = k$ et $c \subseteq c' \subseteq c \cup cand$

2 **si** $\#c = k$ **alors retourne** **vrai**;

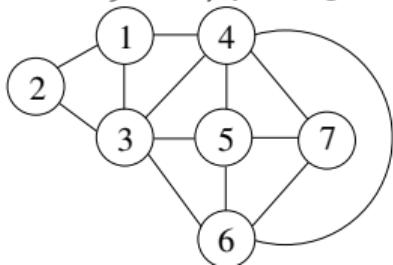
4 **pour** chaque sommet $i \in cand$ **faire**

5 **si** $\text{clique}(g, \{j \in cand | \{i, j\} \in A\}, c \cup \{i\}, k)$ **alors retourne** **vrai**;

6 **retourne** **faux**

Exercice :

Arbre de recherche de
 $\text{clique}(g, \{1, \dots, 7\}, \emptyset, 4)$ pour $g =$



Inconvénient de cette procédure ?

Recherche d'une clique d'ordre k

1 Fonction $clique(g, cand, c, k)$

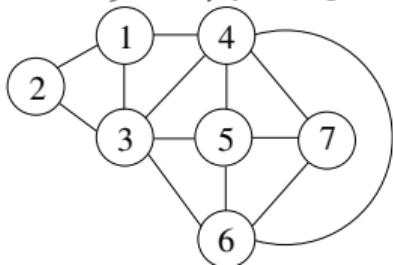
```

Entrée      : graphe  $g = (S, A)$ ,  $cand \subseteq S$ ,  $c \subseteq S$  et entier  $k$ 
Précond.    :  $c$  est une clique de taille inférieure ou égale à  $k$ ,  $cand \cap c = \emptyset$ , et  $\forall i \in cand, \forall j \in c, \{i, j\} \in A$ 
Postcond.   : Retourne vrai si  $\exists$  une clique  $c'$  de  $g$  tq  $\#c' = k$  et  $c \subseteq c' \subseteq c \cup cand$ 
si  $\#c = k$  alors retourne vrai;
pour chaque sommet  $i \in cand$  faire
  si  $clique(g, \{j \in cand | \{i, j\} \in A\}, c \cup \{i\}, k)$  alors retourne vrai;
retourne faux

```

Exercice :

Arbre de recherche de
 $clique(g, \{1, \dots, 7\}, \emptyset, 4)$ pour $g =$



Inconvénient de cette procédure ?

Elle continue la recherche même s'il ne peut pas y avoir de clique d'ordre k
 ~ Couper la branche dans ce cas !

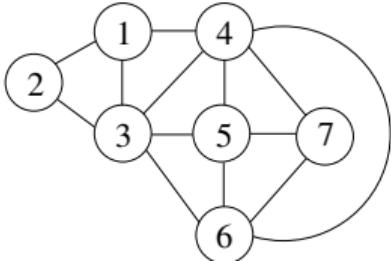
Recherche d'une clique d'ordre k

1 Fonction $\text{clique}(g, cand, c, k)$

Entrée : graphe $g = (S, A)$, $cand \subseteq S$, $c \subseteq S$ et entier k
Précond. : c est une clique de taille inférieure ou égale à k , $cand \cap c = \emptyset$, et $\forall i \in cand, \forall j \in c, \{i, j\} \in A$
Postcond. : Retourne vrai si \exists une clique c' de g tq $\#c' = k$ et $c \subseteq c' \subseteq c \cup cand$
si $\#c = k$ **alors retourne** vrai;
si $\#c + \#cand < k$ **alors retourne** faux;
pour chaque sommet $i \in cand$ **faire**
 ↳ **si** $\text{clique}(g, \{j \in cand | \{i, j\} \in A\}, c \cup \{i\}, k)$ **alors retourne** vrai;
retourne faux

Exercice :

Arbre de recherche de
 $\text{clique}(g, \{1, \dots, 7\}, \emptyset, 4)$ pour $g =$



Techniques d'élagage :

- Procédure pour décider si l'appel courant peut retourner vrai
- Exemple : test sur la cardinalité de $cand$
- D'autres idées ?

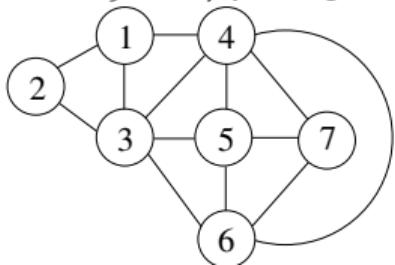
Recherche d'une clique d'ordre k

1 Fonction $\text{clique}(g, cand, c, k)$

Entrée : graphe $g = (S, A)$, $cand \subseteq S$, $c \subseteq S$ et entier k
Précond. : c est une clique de taille inférieure ou égale à k , $cand \cap c = \emptyset$, et $\forall i \in cand, \forall j \in c, \{i, j\} \in A$
Postcond. : Retourne vrai si \exists une clique c' de g tq $\#c' = k$ et $c \subseteq c' \subseteq c \cup cand$
si $\#c = k$ **alors retourne** vrai;
si $\#c + \#cand < k$ **alors retourne** faux;
pour chaque sommet $i \in cand$ **faire**
 └ **si** $\text{clique}(g, \{j \in cand | \{i, j\} \in A\}, c \cup \{i\}, k)$ **alors retourne** vrai;
retourne faux

Exercice :

Arbre de recherche de
 $\text{clique}(g, \{1, \dots, 7\}, \emptyset, 4)$ pour $g =$



Heuristiques d'ordre :

- Que change l'ordre de parcours des sommets de $cand$ (ligne 4) sur l'arbre de recherche ?
- Quel ordre choisir ?

Autre illustration : Voyageur de commerce

Définition du problème du voyageur de commerce :

- Entrées : un graphe $G = (S, A)$ et une fonction $c : A \rightarrow \mathbb{R}$
- Sortie : un circuit hamiltonien (passant par chaque sommet de S exactement une fois) de coût total minimal

Résolution en TP !

- TP3 : Programmation dynamique, A*, WA* et AWA*
- TP4 : Branch and Bound

1 Introduction**2** Structures de données pour représenter un graphe**3** Arbres Couvrants Minimaux**4** Parcours de graphes**5** Plus courts chemins**6** Problèmes de planification**7** Problèmes NP-difficiles

- Classes de complexité
- Exploration exhaustive d'espaces de recherche
- Exploration incomplète d'espaces de recherche

8 Conclusion

Méta-heuristiques

Qu'est-ce qu'une méta-heuristique ?

Approche générique pour explorer un espace de recherche

- Exploration incomplète, sans garantie d'optimalité...
...mais complexité polynomiale
- Approche anytime \leadsto qualité améliorée au fil du temps

Deux grandes familles de méta-heuristiques :

- Approches perturbatives, modifiant itérativement des combinaisons
 - modifications élémentaires \leadsto recherche locale
 - croisements et mutations \leadsto algorithmes génétiques
 - ...
- Approches constructives, basées sur des modèles
 - modèles gloutons (aléatoires)
 - modèles appris (EDA, ACO, RL)

Approches perturbatives : Recherche locale (LS)

1 Fonction LS

- 2 **Entrées** : un espace de recherche E , une fct de voisinage $v : E \rightarrow \mathcal{P}(E)$
- 3 Générer (aléatoirement) une combinaison initiale $e \in E$
- 4 **tant que** critères d'arrêt non atteints **faire**
- 5 Choisir $e' \in v(e)$
- 6 $e \leftarrow e'$
- 7 **retourner** la meilleure combinaison construite

Choix de la fonction de voisinage v :

- Les voisins doivent être générés et évalués très rapidement
- Exemple pour le problème de la clique maximum :
 $v(e) = \text{ens. des cliques obtenues en enlevant/supprimant } k \text{ sommets}$
- Exemple pour le problème du voyageur de commerce :
 $v(e) = \text{ensemble des tours obtenus en échangeant } k \text{ arêtes (k-opt)}$

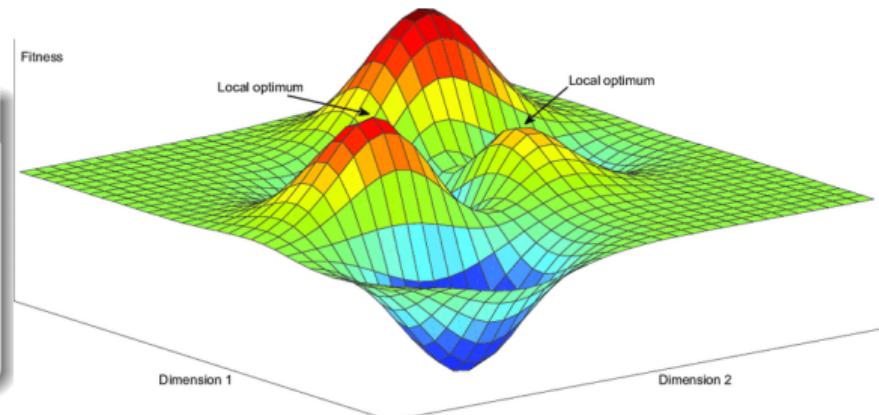
Approches perturbatives : Recherche locale (LS)

1 Fonction LS

- 2 **Entrées** : un espace de recherche E , une fct de voisinage $v : E \rightarrow \mathcal{P}(E)$
- 3 Générer (aléatoirement) une combinaison initiale $e \in E$
- 4 **tant que** critères d'arrêt non atteints faire
- 5 Choisir $e' \in v(e)$
- 6 $e \leftarrow e'$
- 7 **retourner** la meilleure combinaison construite

Fitness landscape de (E, v) :

- Graphe dont les sommets sont les combinaisons de E , et les arêtes les relations de voisinage définies par v
- Recherche locale = chemin dans ce graphe



Approches perturbatives : Recherche locale (LS)

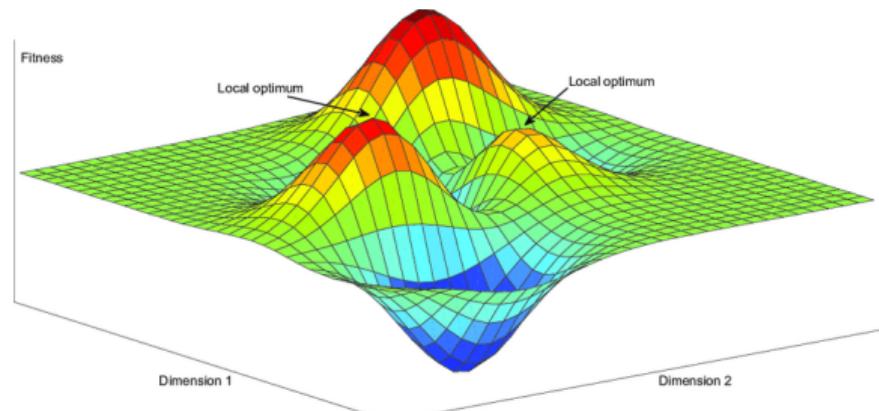
1 Fonction LS

- 2 **Entrées** : un espace de recherche E , une fct de voisinage $v : E \rightarrow \mathcal{P}(E)$
- 3 Générer (aléatoirement) une combinaison initiale $e \in E$
- 4 **tant que** critères d'arrêt non atteints faire
 - 5 Choisir $e' \in v(e)$
 - 6 $e \leftarrow e'$
- 7 **retourner** la meilleure combinaison construite

Stratégies pour choisir $e' \in v(e)$:

- Glouton (montée de gradient)
- Random Walk
- Recuit simulé
- Tabou
- Variable Neighbourhood Search
- Iterated Local Search (cf TD)

...et plein d'autres encore, cf (Hao, Solnon 2020) pour plus de détails



Approches perturbatives : Algorithmes Génétiques (GA)

1 Fonction GA

- 2 Initialiser une population avec un ensemble de combinaisons générées aléatoirement
- 3 **tant que** critères d'arrêt non atteints **faire**
- 4 Sélectionner des combinaisons de la population
- 5 Créer de nouvelles combinaisons par croisement et mutation
- 6 Mettre à jour la population
- 7 **retourner** la meilleure combinaison ayant appartenu à la population

Généralisation de la recherche locale :

LS = GA avec une population réduite à une combinaison

cf (Hao, Solnon 2020) pour plus de détails

Approches constructives gloutonnes

Algorithme glouton :

- Algorithme faisant des choix en utilisant des heuristiques
- En général, complexité polynomiale

Dans certains cas, un algorithme glouton peut être optimal

- Dijkstra quand tous les coûts sont positifs
- Prim et Kruskal

Bien souvent, un algorithme glouton ne garantit pas l'optimalité

- Calcul d'une solution approchée en temps polynomial
- Contrairement aux algorithmes de ρ -approximation, pas de garantie théorique sur l'écart entre solution calculée et solution optimale

Exemple 1 : Construction gloutonne d'une clique

1 Fonction *chercheCliqueGlouton*(g)

Entrée : Un graphe $g = (S, A)$

Postcond. : retourne une clique de g

2 $cand \leftarrow S$

3 $c \leftarrow \emptyset$

4 tant que $cand \neq \emptyset$ faire

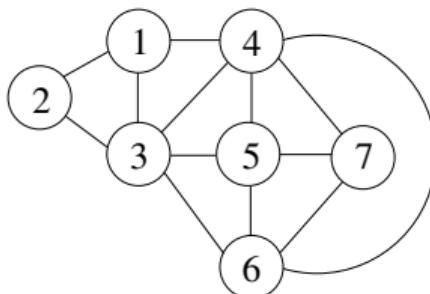
5 Soit s_i le sommet de $cand$ maximisant $\#(cand \cap adj(s_i))$

6 $c \leftarrow c \cup \{s_i\}$

7 $cand \leftarrow cand \cap adj(s_i)$

8 retourne c

Exercice :



Exemple 2 : Algorithme de (Brélaz 1979) pour colorier un graphe

1 Fonction $DSATUR(g)$

2 $borne \leftarrow 0$

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

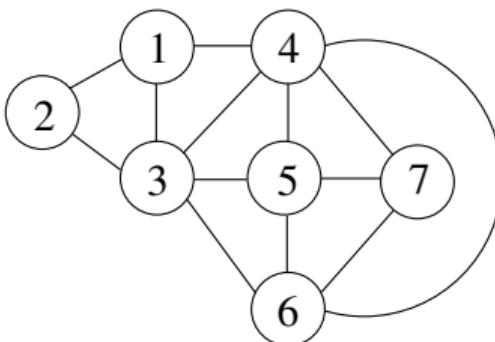
7 **si** $\forall k \in [1, borne], \exists s_j \in adj(s_i)$ tel que s_j est colorié avec k **alors** $borne \leftarrow borne + 1$;

8 $k \leftarrow$ plus petite couleur $\in [1, borne]$ non prise par un voisin de s_i ;

9 Colorier s_i avec k

10 **retourne** $borne$

Exercice :



Approches constructives gloutonnes aléatoires

Une approche gloutonne construit toujours la même solution

~ Introduire de l'aléatoire pour diversifier les solutions calculées

Comment introduire de l'aléatoire dans un algorithme glouton ?

- Solution 1 : Choisir selon une probabilité proportionnelle à h : $p(i) = \frac{h(i)^\beta}{\sum_{j \in \text{cand}} h(j)^\beta}$
- Solution 2 : Choisir aléatoirement un candidat parmi les k meilleurs

~ β et k = paramètres pour régler le degré d'aléatoire

Approches constructives basées sur des modèles appris :

~ Biaiser les probabilités en exploitant les constructions passées

- Greedy Randomised Adaptive Search Procedure (GRASP)
- Estimation of Distribution Algorithms (EDA)
- Ant Colony Optimisation (ACO)

Ex. : ACO pour la recherche de cliques max (Solnon, Fenet 2006)

- initialiser les traces de phéromone
- repeat
 - ① chaque fourmi construit une clique
 - ② mettre à jour les traces de phéromone
- until optimal clique found or stagnation

Ex. : ACO pour la recherche de cliques max (Solnon, Fenet 2006)

- initialiser les traces de phéromone
- repeat
 - ① chaque fourmi construit une clique
 - ② mettre à jour les traces de phéromone
- until optimal clique found or stagnation

La phéromone est déposée sur les sommets du graphe :

$\sim \tau(i)$ = désirabilité apprise de sélectionner i dans une clique

Initialiser $\tau(i)$ à τ_{max} , pour chaque sommet i

$\sim \tau_{max}$ = paramètre

Ex. : ACO pour la recherche de cliques max (Solnon, Fenet 2006)

- initialiser les traces de phéromone
- repeat
 - ① **chaque fourmi construit une clique**
 - ② mettre à jour les traces de phéromone
- until optimal clique found or stagnation

Construction gloutonne aléatoire d'une clique \mathcal{C}

- Choisir aléatoirement $i \in V$ et initialiser \mathcal{C} à $\{i\}$
- Tant que $cand = \{j \in V \setminus \mathcal{C} : \forall i \in \mathcal{C}, \{i, j\} \in E\} \neq \emptyset$ faire :
 - Sélectionner aléatoirement un sommet $j \in cand$ selon la probabilité $p(j) = \frac{[\tau(j)]^\alpha}{\sum_{k \in cand} [\tau(k)]^\alpha}$
où α = paramètre réglant le poids de la phéromone
 - Ajouter j à \mathcal{C}
- Retourner \mathcal{C}

Ex. : ACO pour la recherche de cliques max (Solnon, Fenet 2006)

- initialiser les traces de phéromone
- repeat
 - ① chaque fourmi construit une clique
 - ② **mettre à jour les traces de phéromone**
- until optimal clique found or stagnation

Etape de mise à jour de la phéromone

- Evaporation : multiplier les traces de phéromone par $(1 - \rho)$
 $\leadsto \rho$ = taux d'évaporation ($0 \leq \rho \leq 1$)
- Renforcement : ajout de phéromone sur les sommets de la meilleure clique
- Borner les traces de phéromone pour éviter une convergence prématurée :
 - Si $\tau(i) < \tau_{min}$ alors $\tau(i) \leftarrow \tau_{min}$
 - Si $\tau(i) > \tau_{max}$ alors $\tau(i) \leftarrow \tau_{max}$

- 1 Introduction
- 2 Structures de données pour représenter un graphe
- 3 Arbres Couvrants Minimaux
- 4 Parcours de graphes
- 5 Plus courts chemins
- 6 Problèmes de planification
- 7 Problèmes NP-difficiles
- 8 Conclusion

Conclusion : Ce qu'on a vu dans ce cours

Les graphes permettent de modéliser de nombreux problèmes

- On a vu des algorithmes efficaces pour :
 - Calculer des arbres couvrants minimaux (Kruskal, Prim)
 - Parcourir un graphe (BFS, DFS)
 - Calculer des meilleurs chemins (BFS, TopoDAG, Dijkstra, Bellman-Ford, Floyd-Warshall)
- Les problèmes de planification se ramènent à des recherches de chemins dans des graphes états-transitions
 - Ces graphes sont souvent de très grande taille (voir infini)
 - A* (et ses variantes WA*, AWA*, ...) utilise des heuristiques pour guider la recherche dans ces graphes

Certains problèmes sont plus difficiles à résoudre que d'autres

- Les problèmes \mathcal{NP} -difficiles ne peuvent être résolus en temps polynomial si $\mathcal{P} \neq \mathcal{NP}$
- Pour les résoudre, il faut introduire un peu d'intelligence (artificielle ?) pour contenir ou contourner l'explosion combinatoire

Conclusion : Ce qu'on n'a pas vu dans ce cours

D'autres problèmes dans les graphes peuvent être résolus efficacement

- Problèmes de flot max / coupe min, de couplage, d'affectation, etc
- Voir (Cormen et al, 2009) ou (Roughgarden, 2023), par exemple

La résolution de pb \mathcal{NP} -difficiles est un sujet de recherche très actif

- De nouveaux algorithmes sont régulièrement publiés
 - ~ Voir The Art Of Computer Programming, Vol. 4B, Knuth, 2022
... et le futur volume 4C de TAOCP !
- Certains de ces algorithmes sont intégrés dans des bibliothèques :
Pour résoudre un problème \mathcal{NP} -difficile avec ces bibliothèques, il suffit de décrire le pb en termes de variables, contraintes, et fonction objectif
 - ~ Voir Choco, PyCSP³ ou OR-Tools, par exemple

Conclusion : Ce que vous verrez dans un TP de l'EC REVE

L'optimisation n'est pas neutre

- Elle permet de massifier la production, et donc la consommation
- Elle change en profondeur l'organisation du travail et de la société
- Elle peut se faire au détriment de la résilience

~ Illustrations sur des problèmes de sac-à-dos, de voyageur de commerce, de plus courts chemins, d'arbres couvrants minimaux, ... et d'autres problèmes non étudiés en AAIA

Défendez vos valeurs quand vous choisissez d'optimiser !