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

Pierre-Edouard Portier et Christine Solnon

INSA de Lyon - 3IF

2018/2019

1. **Introduction**
   - Organisation et objectifs pédagogiques
   - Modélisation de problèmes avec des graphes
2. **Définitions**
3. **Structures de données pour représenter un graphe**
4. **Parcours de graphes**
5. **Plus courts chemins**
6. **Problèmes de planification**
7. **Quelques problèmes NP-difficiles sur les graphes**

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

Unités d'enseignement du département IF :

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

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
- **Algo pour l'IA et les graphes**

Référentiel des compétences

Approfondissement de compétences abordées au semestre 1 :

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

$\rightsquigarrow$ Implémenter de bons logiciels

Nouvelles compétences :

- Modéliser et résoudre des problèmes à l'aide de graphes 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

- 5 cours : C. Solnon (du 5 février au 5 mars)
  $\leadsto$ Algorithmique avancée pour les graphes
- 4 cours : P.-E. Portier
  $\leadsto$ Algorithmique avancée pour l'IA

6 TD et 3 TP

- du 11 février au 7 juin

Evaluation

- 1 DS + questionnaires Moodle (sur les cours et sur les TP)

Pour en savoir plus...

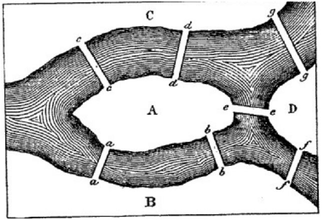
- **Sur l'algorithmique en général :**
  - *Algorithmique*
    T. Cormen, C. Leiserson, R. Rivest, C. Stein
    Editions Dunod - 2010
- **Sur les graphes :**
  - *La théorie des graphes*
    Aimé Sache
    Collection “Le sel et le fer”, n°22
    Editions Cassini - 2003

1. **Introduction**
   - Organisation et objectifs pédagogiques
   - Modélisation de problèmes avec des graphes
2. **Définitions**
3. **Structures de données pour représenter un graphe**
4. **Parcours de graphes**
5. **Plus courts chemins**
6. **Problèmes de planification**
7. **Quelques problèmes NP-difficiles sur les graphes**

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 :

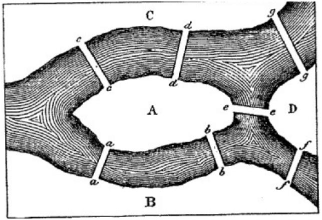
“ Peut-on arranger son parcours de telle sorte que l’on passe sur chaque pont, et que l’on ne puisse y passer qu’une seule fois ? Cela semble possible, disent les uns ; impossible, disent les autres ; cependant personne n’a la certitude de son sentiment. ”

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 :

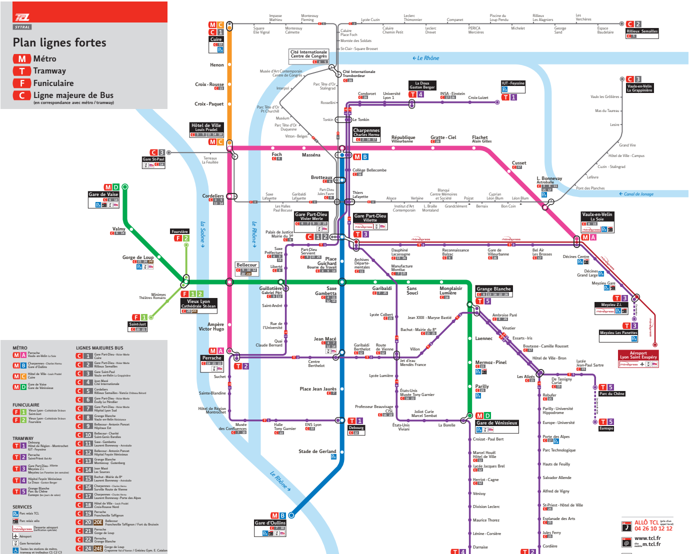
“ Peut-on arranger son parcours de telle sorte que l’on passe sur chaque pont, et que l’on ne puisse y passer qu’une seule fois ? Cela semble possible, disent les uns ; impossible, disent les autres ; cependant personne n’a la certitude de son sentiment. ”

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. ”

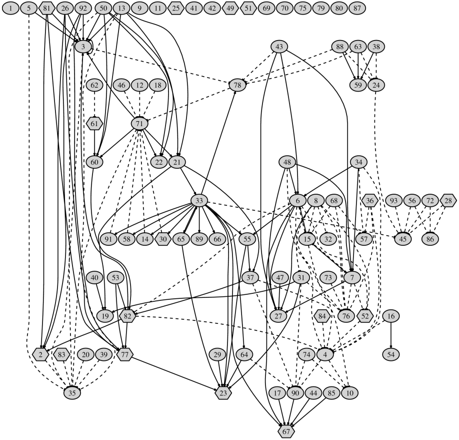
Exemples de modélisation par des graphes

Réseau de transport



Exemples de modélisation par des graphes

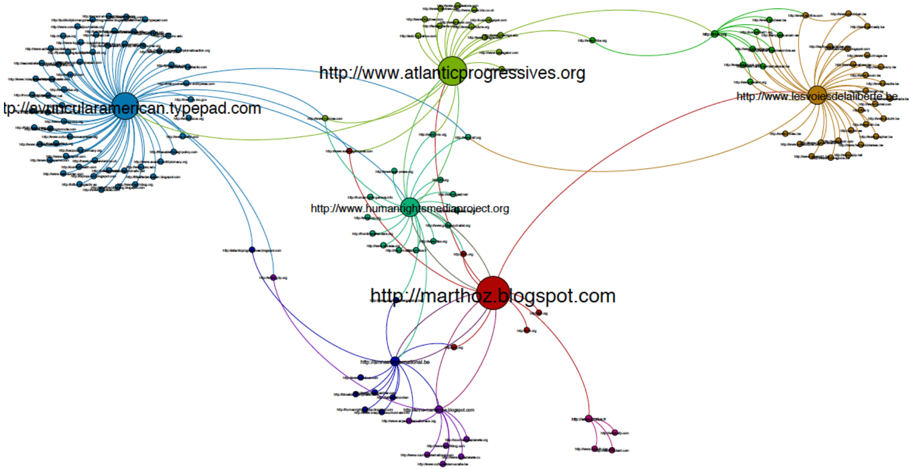
Réseaux de régulation génétique



- Sommets = gènes
- Arêtes = influence entre gènes

Exemples de modélisation par des graphes

Réseaux sociaux



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

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

1. Introduction
2. Définitions
3. Structures de données pour représenter un graphe
4. Parcours de graphes
5. Plus courts chemins
6. Problèmes de planification
7. Quelques problèmes NP-difficiles sur les graphes

Graphes non orientés

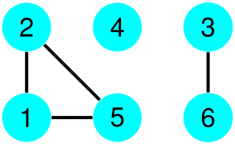
Définition

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**
  - La relation binaire définie par A est **symétrique**
    $\rightsquigarrow \forall (s_i, s_j) \in S \times S, (s_i, s_j) \in A \Leftrightarrow (s_j, s_i) \in A$ (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 : $d°(s_i) = \#adj(s_i)$
- Graphe **complet** si $\forall \{s_i, s_j\} \subseteq S, \{s_i, s_j\} \in A$

Graphes orientés

Définition

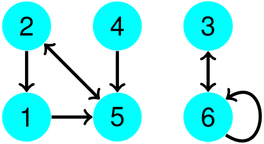
Un graphe orienté est défini par un couple (S, A) tel que

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

$\rightsquigarrow$ 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 :

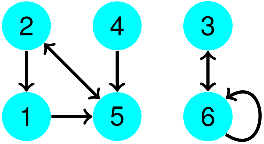
- s_j est **successeur** de s_i si $(s_i, s_j) \in A$: $succ(s_i) = \{s_j | (s_i, s_j) \in A\}$
- s_j est **prédécesseur** de s_i si $(s_j, s_i) \in A$: $pred(s_i) = \{s_j | (s_j, s_i) \in A\}$
- **demi-degré extérieur** = nb de successeurs : $d^{\circ +}(s_i) = \# succ(s_i)$
- **demi-degré intérieur**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

Graphes partiels

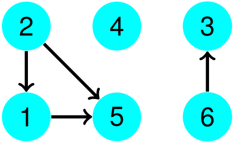
Définition

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

Exemple :



Graphe $G = (S, A)$



Graphe partiel de G

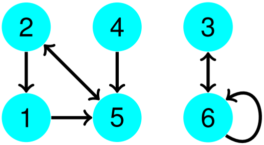
Sous-graphes

Définition

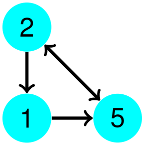
$G' = (S', A')$ est un sous-graphe de $G = (S, A)$ si $S' \subseteq S$ et $A' = A \cap S' \times S'$

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

Exemple :



Graphe $G = (S, A)$

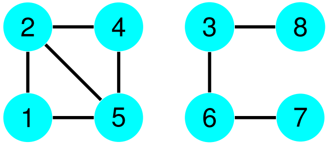


Sous-graphe de G induit par $\{1, 2, 5\}$

Cheminements et connexités

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

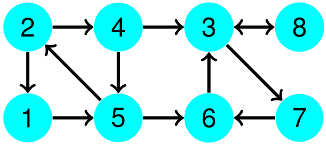
- **Chaîne** = Séquence de sommets $< s_0, s_1, s_2, ..., s_k >$ (notée $s_0 \sim s_k$) telle que $\forall i \in [1, k], \{s_{i-1}, s_i\} \in A$
- **Longueur** d'une chaîne = Nombre d'arêtes dans la chaîne
- **Chaîne élémentaire** = Chaîne dont tous les sommets sont distincts
- **Cycle** = Chaîne commençant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



Cheminements et connexités

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

- **Chemin** = Séquence de sommets $< s_0, s_1, s_2, ..., s_k >$ (notée $s_0 \leadsto s_k$) 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 $\forall (s_i, s_j) \in S^2, s_i \leadsto s_j$
- **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 chaine 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. **Définitions**
3. **Structures de données pour représenter un graphe**
   - Matrices d'adjacence
   - Listes d'adjacence
4. **Parcours de graphes**
5. **Plus courts chemins**
6. **Problèmes de planification**
7. **Quelques problèmes NP-difficiles sur les graphes**

Exemple d'algorithme :

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
  $d \leftarrow 0$
  pour *tout sommet* $s_j \in adj(s_i)$ **faire**
   $d \leftarrow d + 1$
  retourne d

Complexité de cet algorithme ?

Exemple d'algorithme :

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
    $d \leftarrow 0$
    pour *tout sommet* $s_j \in adj(s_i)$ **faire**
      $d \leftarrow d + 1$
    retourne d

Complexité de cet algorithme ?

Dépend des structures de données utilisées pour représenter le graphe !

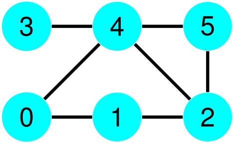
1. **Introduction**
2. **Définitions**
3. **Structures de données pour représenter un graphe**
   - Matrices d'adjacence
   - Listes d'adjacence
4. **Parcours de graphes**
5. **Plus courts chemins**
6. **Problèmes de planification**
7. **Quelques problèmes NP-difficiles sur les graphes**

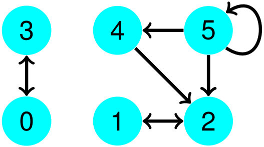
Matrice d'adjacence

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	1	0	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 :

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

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

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

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

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

Puissances de la matrice d'adjacence

Définition de M^k :

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

Théorème : $M^k[s_i][s_j]$ = nb de chemins de longueur k allant de s_i à s_j

$\leadsto$ 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 :

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

Théorème : $M^k[s_i][s_j]$ = nb de chemins de longueur k allant de s_i à s_j

$\leadsto$ Exercice : démonstration par récurrence sur k

Complexité pour un graphe ayant n sommets ?

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

Puissances de la matrice d'adjacence

Définition de M^k :

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

Théorème : $M^k[s_i][s_j]$ = nb de chemins de longueur k allant de s_i à s_j

$\rightsquigarrow$ Exercice : démonstration par récurrence sur k

Complexité pour un graphe ayant n sommets ?

- Complexité d'une multiplication de matrices $n \times n$: $\mathcal{O}(n^3)$
  $\rightsquigarrow$ Peut être optimisé dans le cas de matrices creuses
- Complexité du calcul de M^k ?
  - $\mathcal{O}(kn^3)$ si on fait k multiplications

Puissances de la matrice d'adjacence

Définition de M^k :

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

Théorème : $M^k[s_i][s_j]$ = nb de chemins de longueur k allant de s_i à s_j

$\rightsquigarrow$ Exercice : démonstration par récurrence sur k

Complexité pour un graphe ayant n sommets ?

- Complexité d'une multiplication de matrices $n \times n$: $\mathcal{O}(n^3)$
  $\rightsquigarrow$ Peut être optimisé dans le cas de matrices creuses
- 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 $M^{2x+1} = M(M^x)^2$

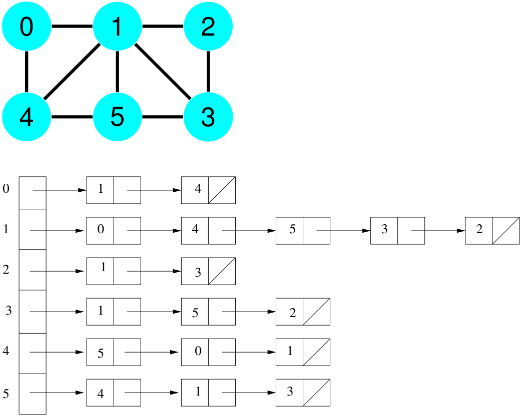
1. **Introduction**
2. **Définitions**
3. **Structures de données pour représenter un graphe**
   - Matrices d’adjacence
   - Listes d’adjacence
4. **Parcours de graphes**
5. **Plus courts chemins**
6. **Problèmes de planification**
7. **Quelques problèmes NP-difficiles sur les graphes**

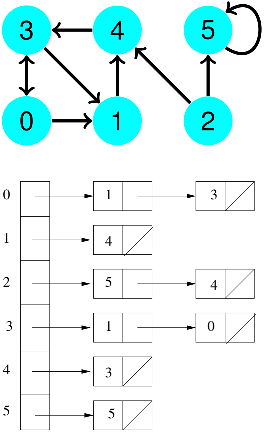
Listes d’adjacence

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

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

Exemples :





Complexité

Complexité en mémoire :

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

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

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

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

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

1. **Introduction**

2. **Définitions**

3. **Structures de données pour représenter un graphe**

4. **Parcours de graphes**
   - Généralités sur les parcours
   - Parcours en largeur (BFS)
   - Parcours en profondeur (DFS)

5. **Plus courts chemins**

6. **Problèmes de planification**

7. **Quelques problèmes NP-difficiles sur les graphes**

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

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

Mise en œuvre : Stockage des sommets gris dans une structure

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

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

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

Comment parcourir un graphe ?

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

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

Mise en œuvre : Stockage des sommets gris dans une structure

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

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

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

Comment parcourir un graphe ?

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

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

Mise en œuvre : Stockage des sommets gris dans une structure

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

Spécification d'un algorithme de parcours

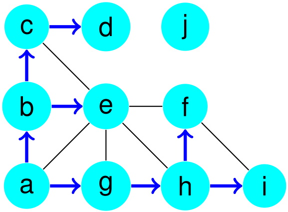
1 **Fonction** *Parcours(g, s_0)*
  Entrée : Un graphe g et un sommet s_0 de g
  Sortie : Arborescence π du parcours de g à partir de s_0

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

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

Arborescence associée à un parcours :

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

Quelle structure de données pour représenter l'arborescence ?

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

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

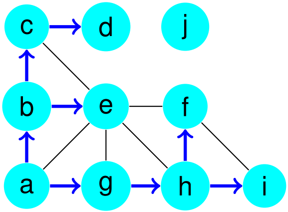


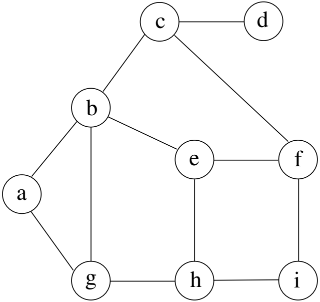
Tableau π correspondant :

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

1. **Introduction**
2. **Définitions**
3. **Structures de données pour représenter un graphe**
4. **Parcours de graphes**
   - Généralités sur les parcours
   - Parcours en largeur (BFS)
   - Parcours en profondeur (DFS)
5. **Plus courts chemins**
6. **Problèmes de planification**
7. **Quelques problèmes NP-difficiles sur les graphes**

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
    Soit f une file (FIFO) initialisée à vide
    pour *chaque sommet* s_i *de* g **faire**
        $\pi[s_i] \leftarrow$ *null*
        Colorier s_i en blanc
    Ajouter s_0 dans f et colorier s_0 en gris
    tant que f *n'est pas vide* **faire**
        Soit s_k le sommet le plus ancien dans f
        pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
            Ajouter s_i dans f et colorier s_i en gris
            $\pi[s_i] \leftarrow s_k$
        Enlever s_k de f et colorier s_k en noir
    retourne π

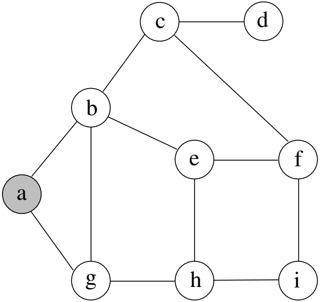


$f = <>$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
    Soit f une file (FIFO) initialisée à vide
    pour *chaque sommet* s_i *de* g **faire**
        $\pi[s_i] \leftarrow$ *null*
        Colorier s_i en blanc
    Ajouter s_0 dans f et colorier s_0 en gris
    tant que f *n'est pas vide* **faire**
        Soit s_k le sommet le plus ancien dans f
        pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
            Ajouter s_i dans f et colorier s_i en gris
            $\pi[s_i] \leftarrow s_k$
        Enlever s_k de f et colorier s_k en noir
    retourne π

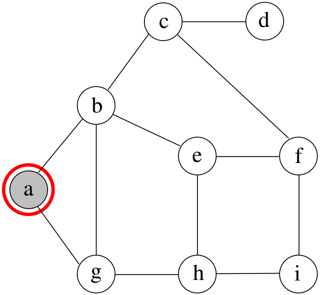


$f = < a >$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
    Soit f une file (FIFO) initialisée à vide
    pour *chaque sommet* s_i *de* g **faire**
        $\pi[s_i] \leftarrow null$
        Colorier s_i en blanc
    Ajouter s_0 dans f et colorier s_0 en gris
    tant que f *n'est pas vide* **faire**
        Soit s_k le sommet le plus ancien dans f
        pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
            Ajouter s_i dans f et colorier s_i en gris
            $\pi[s_i] \leftarrow s_k$
        Enlever s_k de f et colorier s_k en noir
    retourne π

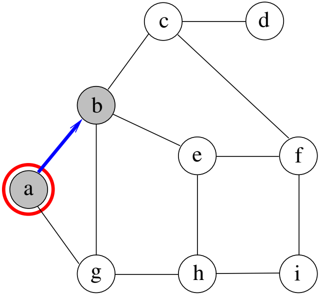


$f = < a >$
$s_k = a$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
    Soit f une file (FIFO) initialisée à vide
    pour *chaque sommet* s_i *de* g **faire**
        $\pi[s_i] \leftarrow null$
        Colorier s_i en blanc
    Ajouter s_0 dans f et colorier s_0 en gris
    tant que f *n'est pas vide* **faire**
        Soit s_k le sommet le plus ancien dans f
        pour chaque $s_i \in succ(s_k)$ **tq** s_i **blanc faire**
            Ajouter s_i **dans** f **et colorier** s_i **en gris**
            $\pi[s_i] \leftarrow s_k$
        Enlever s_k de f et colorier s_k en noir
    retourne π

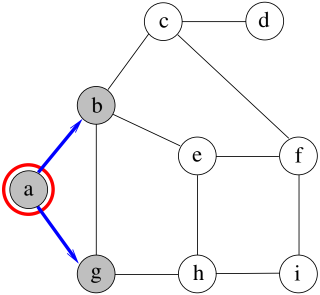


$f =< b, a >$
$s_k = a$, $s_i = b$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow$ *null*
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour chaque $s_i \in succ(s_k)$ **tq** s_i **blanc faire**
      Ajouter s_i **dans** f **et colorier** s_i **en gris**
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

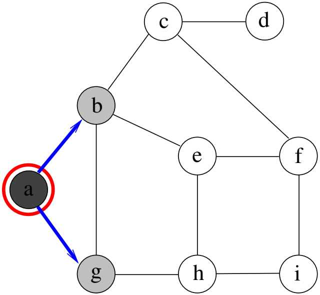


$f =< g, b, a >$
$s_k = a$, $s_i = g$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS(g, s_0)*
- Soit f une file (FIFO) initialisée à vide
- **pour** *chaque sommet s_i de g* **faire**
  - $\pi[s_i] \leftarrow$ *null*
  - Colorier s_i en blanc
- Ajouter s_0 dans f et colorier s_0 en gris
- **tant que** *f n'est pas vide* **faire**
  - Soit s_k le sommet le plus ancien dans f
  - **pour** *chaque $s_i \in succ(s_k)$ tq s_i est blanc* **faire**
    - Ajouter s_i dans f et colorier s_i en gris
    - $\pi[s_i] \leftarrow s_k$
  - **Enlever s_k de f et colorier s_k en noir**
- **retourne** π



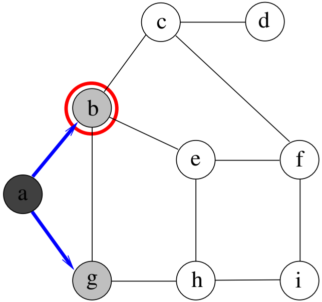
$f = < g, b >$

$s_k = a$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS(g, s_0)*
    Soit f une file (FIFO) initialisée à vide
    pour *chaque sommet s_i de g* **faire**
        $\pi[s_i] \leftarrow$ *null*
        Colorier s_i en blanc
    Ajouter s_0 dans f et colorier s_0 en gris
    tant que *f n'est pas vide* **faire**
        Soit s_k le sommet le plus ancien dans f
        pour *chaque $s_i \in succ(s_k)$ tq s_i est blanc* **faire**
            Ajouter s_i dans f et colorier s_i en gris
            $\pi[s_i] \leftarrow s_k$
        Enlever s_k de f et colorier s_k en noir
    retourne π

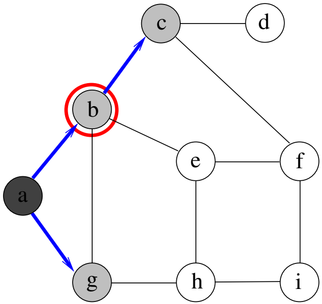


$f = < g, b >$
$s_k = b$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour chaque $s_i \in succ(s_k)$ **tq** s_i **blanc faire**
      Ajouter s_i **dans** f **et colorier** s_i **en gris**
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

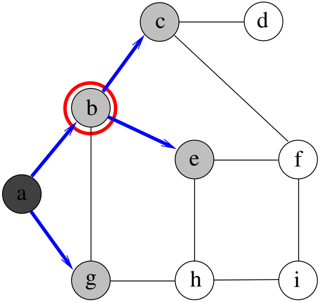


$f = < c, g, b >$
$s_k = b$, $s_i = c$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow$ *null*
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour chaque $s_i \in succ(s_k)$ **tq** s_i **blanc faire**
      Ajouter s_i **dans** f **et colorier** s_i **en gris**
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

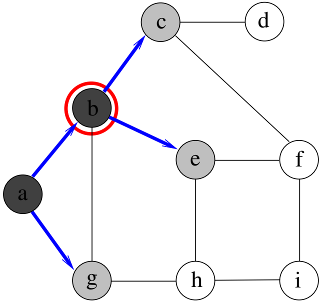


$f = < e, c, g, b >$
$s_k = b$, $s_i = e$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
- Soit f une file (FIFO) initialisée à vide
- **pour** *chaque sommet* s_i *de* g **faire**
  - $\pi[s_i] \leftarrow$ *null*
  - Colorier s_i en blanc
- Ajouter s_0 dans f et colorier s_0 en gris
- **tant que** f *n'est pas vide* **faire**
  - Soit s_k le sommet le plus ancien dans f
  - **pour** *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
    - Ajouter s_i dans f et colorier s_i en gris
    - $\pi[s_i] \leftarrow s_k$
  - **Enlever s_k de f et colorier s_k en noir**
- **retourne** π

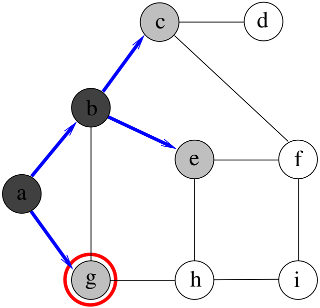


$f = < e, c, g >$
$s_k = b$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
      Ajouter s_i dans f et colorier s_i en gris
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

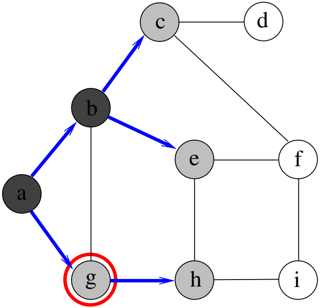


$f = < e, c, g >$
$s_k = g$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
- Soit f une file (FIFO) initialisée à vide
- **pour** *chaque sommet* s_i *de* g **faire**
  - $\pi[s_i] \leftarrow null$
  - Colorier s_i en blanc
- Ajouter s_0 dans f et colorier s_0 en gris
- **tant que** f *n'est pas vide* **faire**
  - Soit s_k le sommet le plus ancien dans f
  - **pour chaque** $s_i \in succ(s_k)$ **tq** s_i **blanc faire**
    - **Ajouter** s_i **dans** f **et colorier** s_i **en gris**
    - $\pi[s_i] \leftarrow s_k$
  - Enlever s_k de f et colorier s_k en noir
- **retourne** π



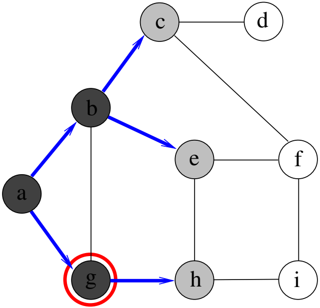
$f = < h, e, c, g >$

$s_k = g$, $s_i = h$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
      Ajouter s_i dans f et colorier s_i en gris
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

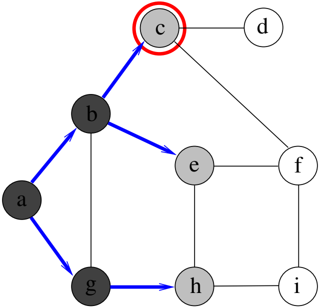


$f = < h, e, c >$
$s_k = g$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
      Ajouter s_i dans f et colorier s_i en gris
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

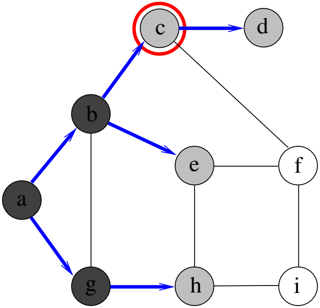


$f = < h, e, c >$
$s_k = c$

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

Parcours en largeur (Breadth First Search / BFS)

```
Fonction BFS(g, s0)
    Soit f une file (FIFO) initialisée à vide
    pour chaque sommet si de g faire
        π[si] ← null
        Colorier si en blanc
    Ajouter s0 dans f et colorier s0 en gris
    tant que f n'est pas vide faire
        Soit sk le sommet le plus ancien dans f
        pour chaque si ∈ succ(sk) tq si blanc faire
            Ajouter si dans f et colorier si en gris
            π[si] ← sk
        Enlever sk de f et colorier sk en noir
    retourne π
```



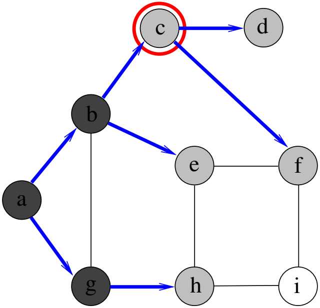
$f = < d, h, e, c >$

$s_k = c$, $s_i = d$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS(g, s_0)*
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet s_i de g* **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que *f n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour chaque $s_i \in succ(s_k)$ tq s_i blanc faire
      Ajouter s_i dans f et colorier s_i en gris
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

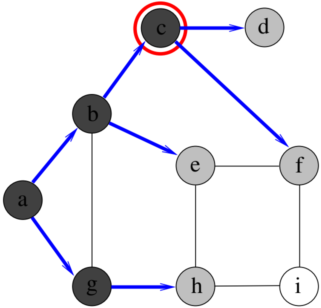


$f = < f, d, h, e, c >$
$s_k = c$, $s_i = f$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow$ *null*
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
      Ajouter s_i dans f et colorier s_i en gris
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

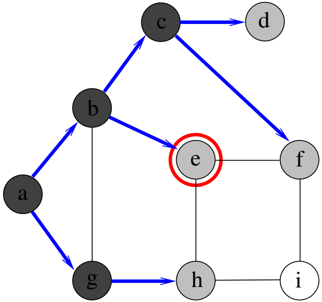


$f = < f, d, h, e >$
$s_k = c$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
      Ajouter s_i dans f et colorier s_i en gris
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

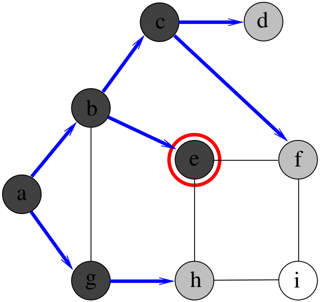


$f = < f, d, h, e >$
$s_k = e$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow$ *null*
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
      Ajouter s_i dans f et colorier s_i en gris
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

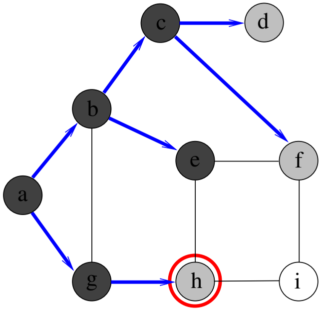


$f = < f, d, h >$
$s_k = e$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
- Soit f une file (FIFO) initialisée à vide
- **pour** *chaque sommet* s_i *de* g **faire**
  - $\pi[s_i] \leftarrow null$
  - Colorier s_i en blanc
- Ajouter s_0 dans f et colorier s_0 en gris
- **tant que** f *n'est pas vide* **faire**
  - **Soit s_k le sommet le plus ancien dans f**
  - **pour** *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
    - Ajouter s_i dans f et colorier s_i en gris
    - $\pi[s_i] \leftarrow s_k$
  - Enlever s_k de f et colorier s_k en noir
- **retourne** π



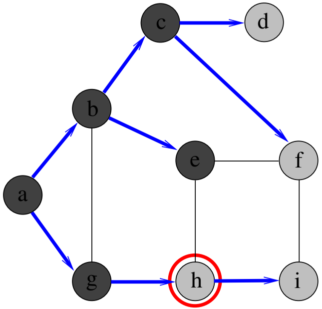
$f = < f, d, h >$

$s_k = h$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
- Soit f une file (FIFO) initialisée à vide
- **pour** *chaque sommet* s_i *de* g **faire**
  - $\pi[s_i] \leftarrow$ *null*
  - Colorier s_i en blanc
- Ajouter s_0 dans f et colorier s_0 en gris
- **tant que** f *n'est pas vide* **faire**
  - Soit s_k le sommet le plus ancien dans f
  - **pour chaque** $s_i \in succ(s_k)$ **tq** s_i **blanc faire**
    - **Ajouter** s_i **dans** f **et colorier** s_i **en gris**
    - $\pi[s_i] \leftarrow s_k$
  - Enlever s_k de f et colorier s_k en noir
- **retourne** π

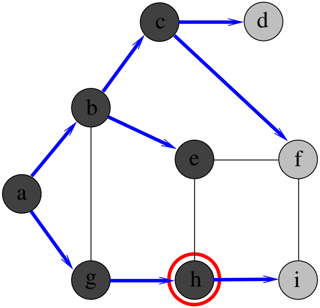


$f =< i, f, d, h >$
$s_k = h$, $s_i = i$

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

Parcours en largeur (Breadth First Search / BFS)

```
Fonction BFS(g, s0)
    Soit f une file (FIFO) initialisée à vide
    pour chaque sommet si de g faire
        π[si] ← null
        Colorier si en blanc
    Ajouter s0 dans f et colorier s0 en gris
    tant que f n'est pas vide faire
        Soit sk le sommet le plus ancien dans f
        pour chaque si ∈ succ(sk) tq si est blanc faire
            Ajouter si dans f et colorier si en gris
            π[si] ← sk
        Enlever sk de f et colorier sk en noir
    retourne π
```



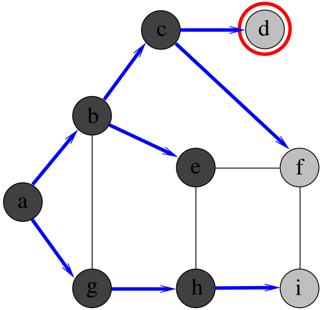
$f =< i, f, d >$

$s_k = h$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
      Ajouter s_i dans f et colorier s_i en gris
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

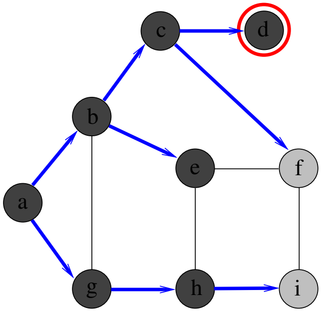


$f = < i, f, d >$
$s_k = d$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
      Ajouter s_i dans f et colorier s_i en gris
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

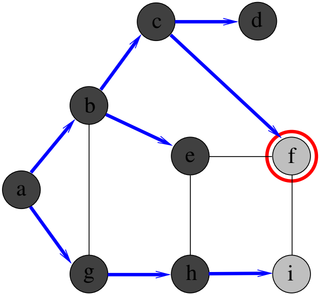


$f = < i, f >$
$s_k = d$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
      Ajouter s_i dans f et colorier s_i en gris
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

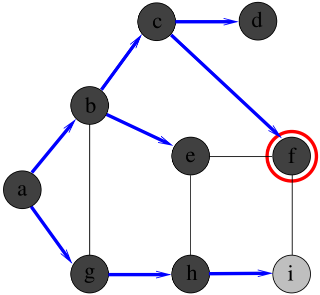


$f = < i, f >$
$s_k = f$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
- Soit f une file (FIFO) initialisée à vide
- **pour** *chaque sommet* s_i *de* g **faire**
  - $\pi[s_i] \leftarrow null$
  - Colorier s_i en blanc
- Ajouter s_0 dans f et colorier s_0 en gris
- **tant que** f *n'est pas vide* **faire**
  - Soit s_k le sommet le plus ancien dans f
  - **pour** *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
    - Ajouter s_i dans f et colorier s_i en gris
    - $\pi[s_i] \leftarrow s_k$
  - **Enlever s_k de f et colorier s_k en noir**
- **retourne** π



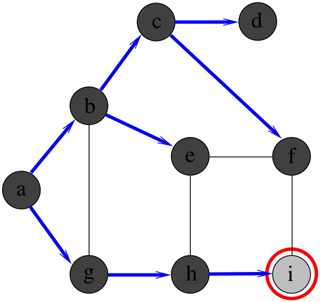
$f = < i >$

$s_k = f$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow$ *null*
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
      Ajouter s_i dans f et colorier s_i en gris
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

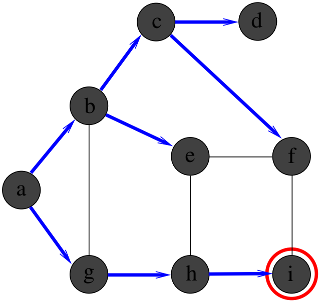


$f = < i >$
$s_k = i$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
  Soit f une file (FIFO) initialisée à vide
  pour *chaque sommet* s_i *de* g **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Ajouter s_0 dans f et colorier s_0 en gris
  tant que f *n'est pas vide* **faire**
    Soit s_k le sommet le plus ancien dans f
    pour *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
      Ajouter s_i dans f et colorier s_i en gris
      $\pi[s_i] \leftarrow s_k$
    Enlever s_k de f et colorier s_k en noir
  retourne π

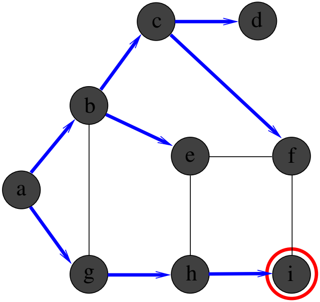


$f = <>$
$s_k = i$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS(g, s_0)*
- Soit f une file (FIFO) initialisée à vide
- **pour** *chaque sommet s_i de g* **faire**
  - $\pi[s_i] \leftarrow$ *null*
  - Colorier s_i en blanc
- Ajouter s_0 dans f et colorier s_0 en gris
- **tant que** *f n'est pas vide* **faire**
  - Soit s_k le sommet le plus ancien dans f
  - **pour** *chaque $s_i \in succ(s_k)$ tq s_i est blanc* **faire**
    - Ajouter s_i dans f et colorier s_i en gris
    - $\pi[s_i] \leftarrow s_k$
  - Enlever s_k de f et colorier s_k en noir
- **retourne** π



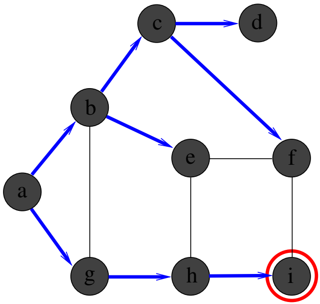
$f = <>$

$s_k = i$

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

Parcours en largeur (Breadth First Search / BFS)

Fonction *BFS*(g, s_0)
- Soit f une file (FIFO) initialisée à vide
- **pour** *chaque sommet* s_i *de* g **faire**
  - $\pi[s_i] \leftarrow$ *null*
  - Colorier s_i en blanc
- Ajouter s_0 dans f et colorier s_0 en gris
- **tant que** f *n'est pas vide* **faire**
  - Soit s_k le sommet le plus ancien dans f
  - **pour** *chaque* $s_i \in succ(s_k)$ *tq* s_i *est blanc* **faire**
    - Ajouter s_i dans f et colorier s_i en gris
    - $\pi[s_i] \leftarrow s_k$
  - Enlever s_k de f et colorier s_k en noir
- **retourne** π



$f = <>$

$s_k = i$

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

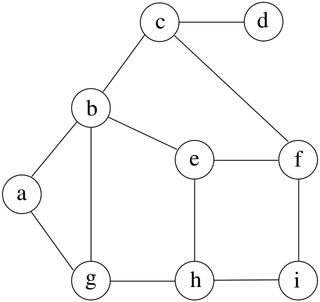
$\rightsquigarrow \mathcal{O}(n+p)$ (sous réserve d'une implémentation par listes d'adjacence)

Utilisation de BFS : Recherche de plus courts chemins

Définition : Soient s_0 et s_i deux sommets tels que $s_0 \leadsto s_i$

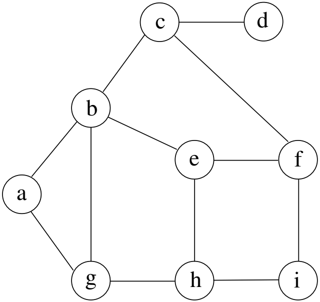
- Plus court chemin entre s_0 et s_i = chemin de longueur minimale
- Distance entre s_0 et s_i = $\delta(s_0, s_i)$ = longueur du plus court chemin

Exemple :



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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

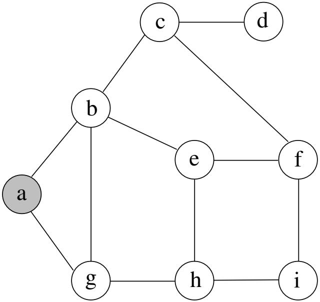


$f = <>$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

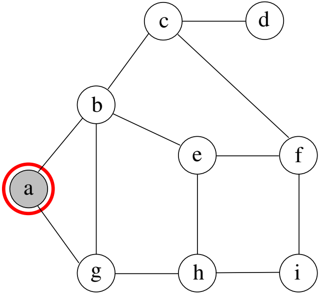


$f =< a >$
$d[a] = 0$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

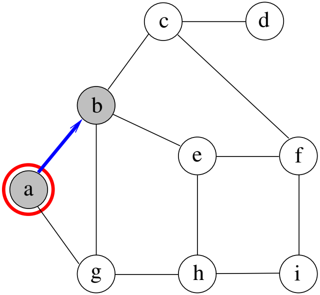


$f = < a >$
$d[a] = 0$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

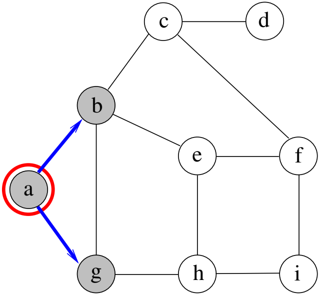


$f =< b, a >$
$d[a] = 0$, $d[b] = 1$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

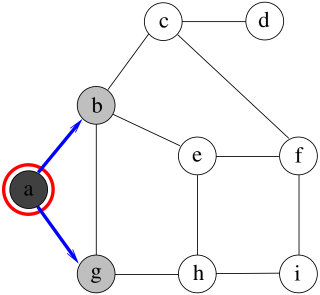


$f =< g, b, a >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2   **pour** *chaque sommet* s_i *de* g **faire**
3         $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

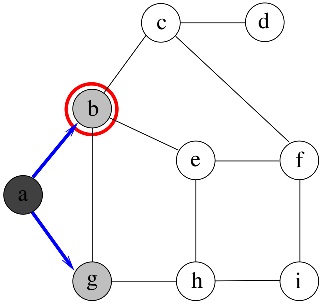


$f = < g, b >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2   **pour** *chaque sommet* s_i *de* g **faire**
3         $\pi[s_i] \leftarrow$ *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* $s_i \in succ(s_k)$ *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

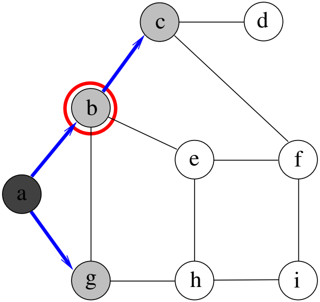


$f =< g, b >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2   **pour** *chaque sommet* s_i *de* g **faire**
3         $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

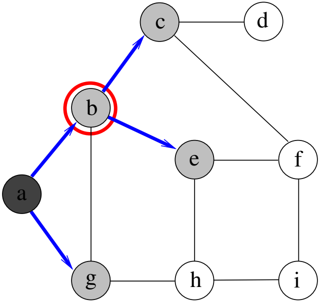


$f = < c, g, b >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

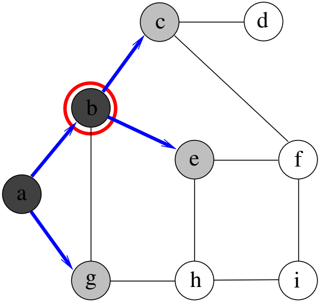


$f = < e, c, g, b >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow$ *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* $s_i \in succ(s_k)$ *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

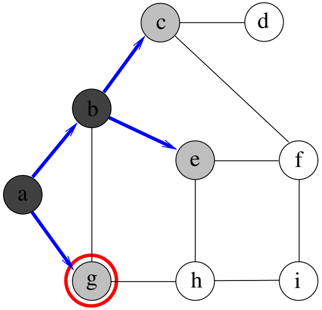


$f = < e, c, g >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2   **pour** *chaque sommet* s_i *de* g **faire**
3         $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

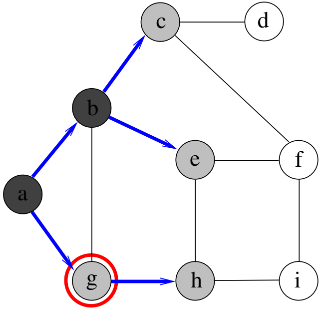


$f = < e, c, g >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

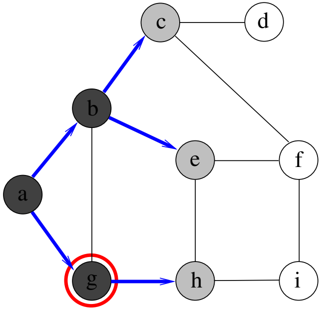


$f =< h, e, c, g >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2   **pour** *chaque sommet* s_i *de* g **faire**
3         $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

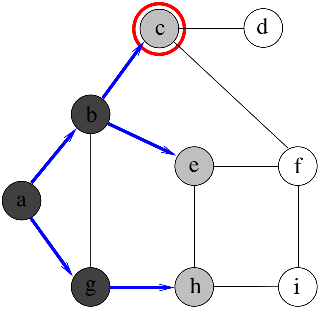


$f = < h, e, c >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

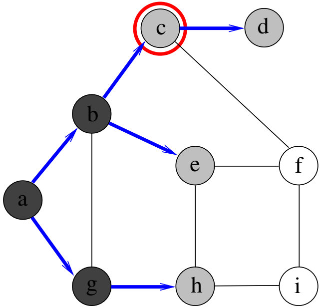


$f = < h, e, c >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2   **pour** *chaque sommet* s_i *de* g **faire**
3     $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

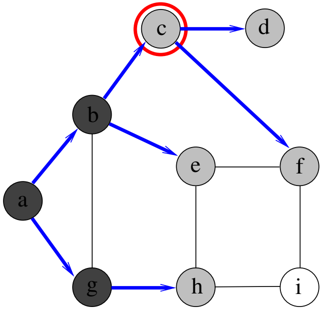


$f = < d, h, e, c >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

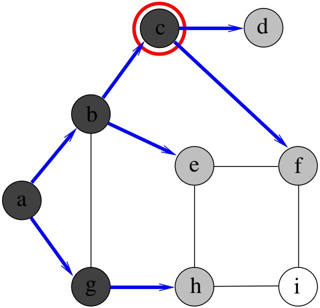


$f =< f, d, h, e, c >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$, $d[c] = 2$, $d[e] = 2$, $d[h] = 2$, $d[d] = 3$, $d[f] = 3$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2   **pour** *chaque sommet* s_i *de* g **faire**
3         $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

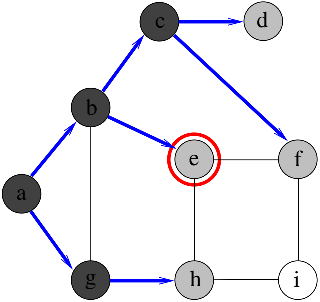


$f =< f, d, h, e >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$, $d[f] = 3$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

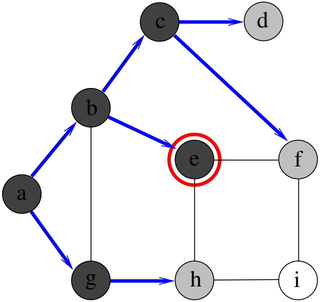


$f = < f, d, h, e >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$, $d[f] = 3$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow$ *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* $s_i \in succ(s_k)$ *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

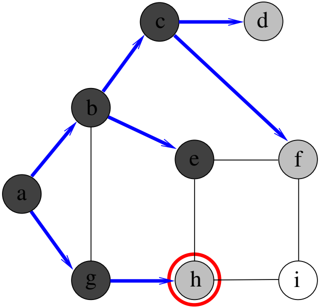


$f = < f, d, h >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$, $d[f] = 3$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2   **pour** *chaque sommet* s_i *de* g **faire**
3     $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

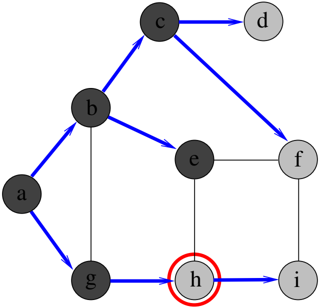


$f = < f, d, h >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$, $d[f] = 3$

Preuve : propriétés invariantes à la ligne 9

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

```
1  Fonction calculeDistances(g, s0)
2      pour chaque sommet si de g faire
3          π[si] ← null
4          Colorier si en blanc
5          d[si] ← ∞
6      Ajouter s0 dans f et colorier s0 en gris
7      d[s0] ← 0
8      tant que f n'est pas vide faire
9          Soit sk le sommet le plus ancien dans f
10         pour chaque si ∈ succ(sk) tq si est blanc faire
11             Ajouter si dans f et colorier si en gris
12             d[si] ← d[sk] + 1
13             π[si] ← sk
14         Enlever sk de f et colorier sk en noir
15     retourne d
```

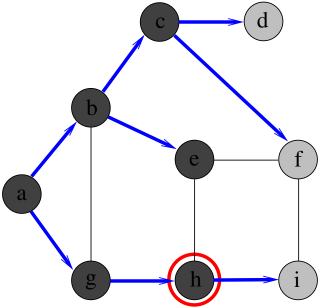


$f = < i, f, d, h >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$, $d[f] = 3$, $d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de g* **faire**
3 $\pi[s_i] \leftarrow$ *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* $s_i \in succ(s_k)$ *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

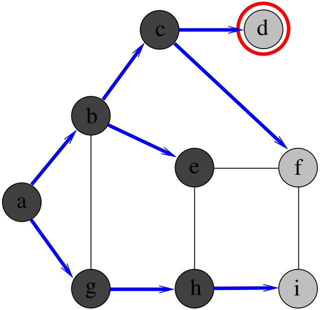


$f =< i, f, d >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$, $d[f] = 3$, $d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow$ *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* $s_i \in succ(s_k)$ *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

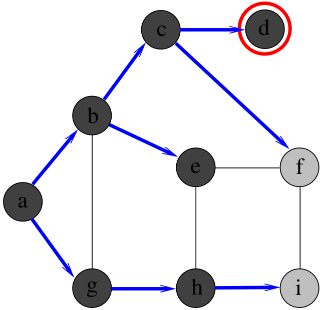


$f =< i, f, d >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$, $d[f] = 3$, $d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

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

```
1  Fonction calculeDistances(g, s0)
2      pour chaque sommet si de g faire
3          π[si] ← null
4          Colorier si en blanc
5          d[si] ← ∞
6      Ajouter s0 dans f et colorier s0 en gris
7      d[s0] ← 0
8      tant que f n'est pas vide faire
9          Soit sk le sommet le plus ancien dans f
10         pour chaque si ∈ succ(sk) tq si est blanc faire
11             Ajouter si dans f et colorier si en gris
12             d[si] ← d[sk] + 1
13             π[si] ← sk
14         Enlever sk de f et colorier sk en noir
15     retourne d
```



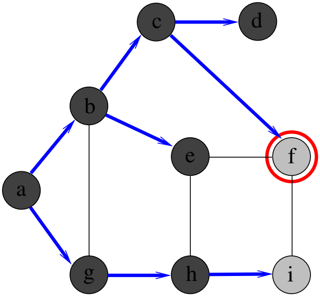
$f =< i, f >$

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

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

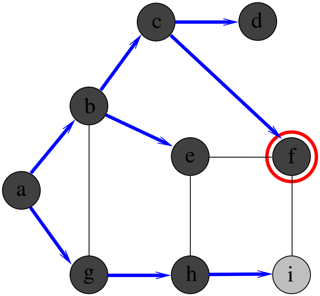


$f =< i, f >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$, $d[f] = 3$, $d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2   **pour** *chaque sommet* s_i *de* g **faire**
3         $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

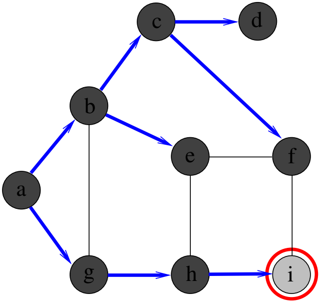


$f =< i >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$, $d[f] = 3$, $d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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

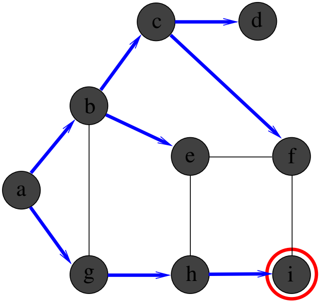


$f = < i >$
$d[a] = 0$, $d[b] = 1$, $d[g] = 1$,
$d[c] = 2$, $d[e] = 2$, $d[h] = 2$,
$d[d] = 3$, $d[f] = 3$, $d[i] = 3$

Preuve : propriétés invariantes à la ligne 9

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

1 **Fonction** *calculeDistances*(g, s_0)
2 **pour** *chaque sommet* s_i *de* g **faire**
3 $\pi[s_i] \leftarrow null$
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* $s_i \in succ(s_k)$ *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



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

Preuve : propriétés invariantes à la ligne 9

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

Affichage du plus court chemin

Proc *plusCourtChemin*(s_0, s_j, π)
- **Entrée** : 2 sommets s_0 et s_j, et une arborescence π
- **Précond.** : π = arborescence retournée par *calculeDistance*(g, s_0)
- **Postcond.** : Affiche un plus court chemin pour aller de s_0 jusque s_j

si $s_0 = s_j$ **alors** afficher(s_0);
sinon si $\pi[s_j] =$ *null* **alors** afficher("Pas de chemin !");
sinon
    plusCourtChemin($s_0, \pi[s_j], \pi$)
    afficher(" suivi de ",s_j)

Exemple :

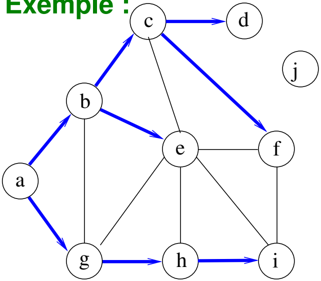


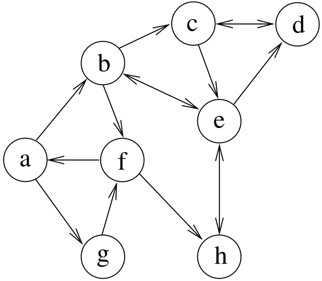
Tableau π correspondant :

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

1. **Introduction**
2. **Définitions**
3. **Structures de données pour représenter un graphe**
4. **Parcours de graphes**
   - Généralités sur les parcours
   - Parcours en largeur (BFS)
   - Parcours en profondeur (DFS)
5. **Plus courts chemins**
6. **Problèmes de planification**
7. **Quelques problèmes NP-difficiles sur les graphes**

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
- Soit p une pile (LIFO) initialisée à vide
- **pour** *tout sommet* $s_i \in S$ **faire**
  - $\pi[s_i] \leftarrow$ *null*
  - Colorier s_i en blanc
- Empiler s_0 dans p et colorier s_0 en gris
- **tant que** *p n'est pas vide* **faire**
  - Soit s_i le dernier sommet entré dans p
  - **si** $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
    - Empiler s_j dans p et colorier s_j en gris
    - $\pi[s_j] \leftarrow s_i$
  - **sinon**
    - Dépiler s_i de p et colorier s_i en noir
- **retourne** π

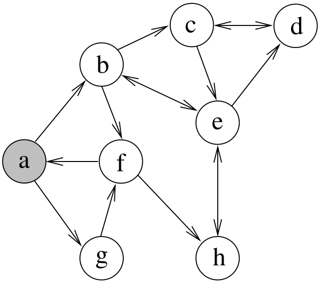


$p = <>$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
- Soit p une pile (LIFO) initialisée à vide
- **pour** *tout sommet* $s_i \in S$ **faire**
  - $\pi[s_i] \leftarrow null$
  - Colorier s_i en blanc
- **Empiler s_0 dans p et colorier s_0 en gris**
- **tant que** *p n'est pas vide* **faire**
  - Soit s_i le dernier sommet entré dans p
  - **si** $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
    - Empiler s_j dans p et colorier s_j en gris
    - $\pi[s_j] \leftarrow s_i$
  - **sinon**
    - Dépiler s_i de p et colorier s_i en noir
- **retourne** π

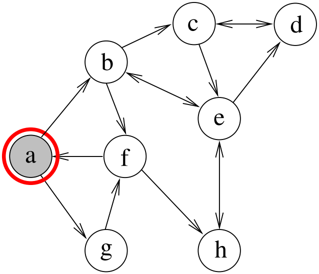


$p = < a >$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
  Soit p une pile (LIFO) initialisée à vide
  pour *tout sommet* $s_i \in S$ **faire**
    $\pi[s_i] \leftarrow$ *null*
    Colorier s_i en blanc
  Empiler s_0 dans p et colorier s_0 en gris
  tant que *p n'est pas vide* **faire**
    Soit s_i le dernier sommet entré dans p
    si $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
      Empiler s_j dans p et colorier s_j en gris
      $\pi[s_j] \leftarrow s_i$
    sinon
      Dépiler s_i de p et colorier s_i en noir
  retourne π

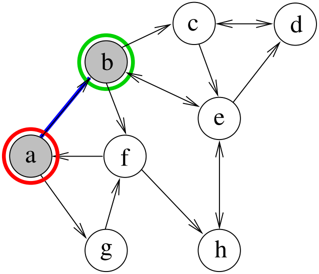


$p = < a >$
$s_i = a$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
  Soit p une pile (LIFO) initialisée à vide
  pour *tout sommet* $s_i \in S$ **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Empiler s_0 dans p et colorier s_0 en gris
  tant que *p n'est pas vide* **faire**
    Soit s_i le dernier sommet entré dans p
    si $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
      Empiler s_j dans p et colorier s_j en gris
      $\pi[s_j] \leftarrow s_i$
    sinon
      Dépiler s_i de p et colorier s_i en noir
  retourne π

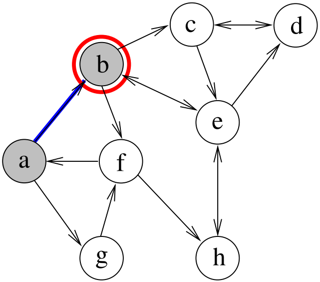


$p =< a, b >$
$s_i = a, s_j = b$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
- Soit p une pile (LIFO) initialisée à vide
- **pour** *tout sommet* $s_i \in S$ **faire**
  - $\pi[s_i] \leftarrow null$
  - Colorier s_i en blanc
- Empiler s_0 dans p et colorier s_0 en gris
- **tant que** *p n'est pas vide* **faire**
  - **Soit s_i le dernier sommet entré dans p**
  - **si** $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
    - Empiler s_j dans p et colorier s_j en gris
    - $\pi[s_j] \leftarrow s_i$
  - **sinon**
    - Dépiler s_i de p et colorier s_i en noir
- **retourne** π

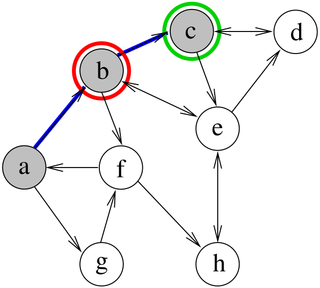


$p = < a, b >$
$s_i = b$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS(g, s_0)*
    Soit p une pile (LIFO) initialisée à vide
    pour *tout sommet $s_i \in S$* **faire**
        $\pi[s_i] \leftarrow$ *null*
        Colorier s_i en blanc
    Empiler s_0 dans p et colorier s_0 en gris
    tant que *p n'est pas vide* **faire**
        Soit s_i le dernier sommet entré dans p
        si $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
            Empiler s_j dans p et colorier s_j en gris
            $\pi[s_j] \leftarrow s_i$
        sinon
            Dépiler s_i de p et colorier s_i en noir
    retourne π

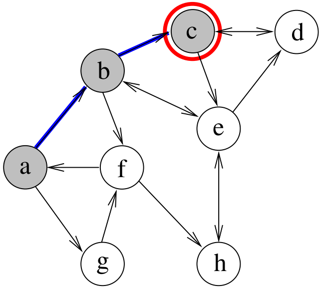


$p = < a, b, c >$
$s_i = b$, $s_j = c$

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

Parcours en profondeur (Depth First search / DFS)

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

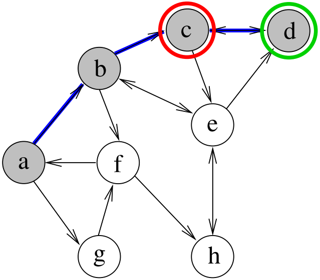


$p = < a, b, c >$
$s_i = c$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
- Soit p une pile (LIFO) initialisée à vide
- **pour** *tout sommet* $s_i \in S$ **faire**
  - $\pi[s_i] \leftarrow null$
  - Colorier s_i en blanc
- Empiler s_0 dans p et colorier s_0 en gris
- **tant que** *p n'est pas vide* **faire**
  - Soit s_i le dernier sommet entré dans p
  - **si** $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
    - **Empiler s_j dans p et colorier s_j en gris**
    - $\pi[s_j] \leftarrow s_i$
  - **sinon**
    - Dépiler s_i de p et colorier s_i en noir
- **retourne** π

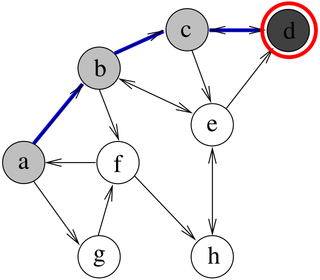


$p = < a, b, c, d >$
$s_i = c, s_j = d$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS(g, s_0)*
- Soit p une pile (LIFO) initialisée à vide
- **pour** *tout sommet* $s_i \in S$ **faire**
  - $\pi[s_i] \leftarrow$ *null*
  - Colorier s_i en blanc
- Empiler s_0 dans p et colorier s_0 en gris
- **tant que** *p n'est pas vide* **faire**
  - **Soit s_i le dernier sommet entré dans p**
  - **si** $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
    - Empiler s_j dans p et colorier s_j en gris
    - $\pi[s_j] \leftarrow s_i$
  - **sinon**
    - **Dépiler s_i de p et colorier s_i en noir**
- **retourne** π

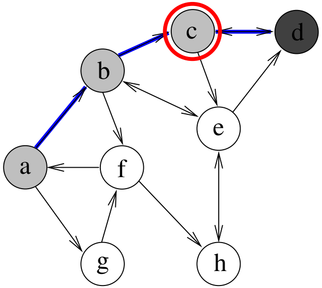


$p = < a, b, c >$
$s_i = d$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
  Soit p une pile (LIFO) initialisée à vide
  pour *tout sommet* $s_i \in S$ **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Empiler s_0 dans p et colorier s_0 en gris
  tant que *p n'est pas vide* **faire**
    Soit s_i le dernier sommet entré dans p
    si $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
      Empiler s_j dans p et colorier s_j en gris
      $\pi[s_j] \leftarrow s_i$
    sinon
      Dépiler s_i de p et colorier s_i en noir
  retourne π

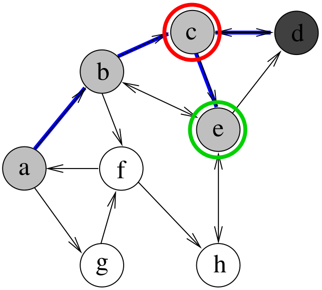


$p =< a, b, c >$
$s_i = c$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
- Soit p une pile (LIFO) initialisée à vide
- **pour** *tout sommet* $s_i \in S$ **faire**
  - $\pi[s_i] \leftarrow null$
  - Colorier s_i en blanc
- Empiler s_0 dans p et colorier s_0 en gris
- **tant que** *p n'est pas vide* **faire**
  - Soit s_i le dernier sommet entré dans p
  - **si** $\exists s_j \in succ(s_i)$ *tel que* s_j *soit blanc* **alors**
    - **Empiler s_j dans p et colorier s_j en gris**
    - $\pi[s_j] \leftarrow s_i$
  - **sinon**
    - Dépiler s_i de p et colorier s_i en noir
- **retourne** π

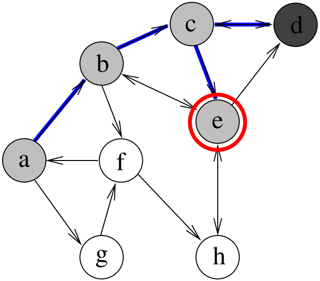


$p = < a, b, c, e >$
$s_i = c, s_j = e$

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

Parcours en profondeur (Depth First search / DFS)

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

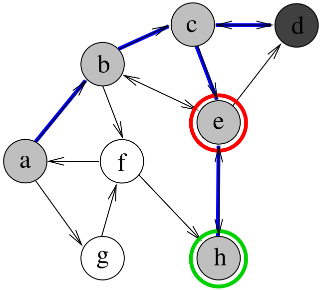


$p = < a, b, c, e >$
$s_i = e$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS(g, s_0)*
- Soit p une pile (LIFO) initialisée à vide
- **pour** *tout sommet* $s_i \in S$ **faire**
  - $\pi[s_i] \leftarrow null$
  - Colorier s_i en blanc
- Empiler s_0 dans p et colorier s_0 en gris
- **tant que** *p n'est pas vide* **faire**
  - Soit s_i le dernier sommet entré dans p
  - **si** $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
    - **Empiler s_j dans p et colorier s_j en gris**
    - $\pi[s_j] \leftarrow s_i$
  - **sinon**
    - Dépiler s_i de p et colorier s_i en noir
- **retourne** π

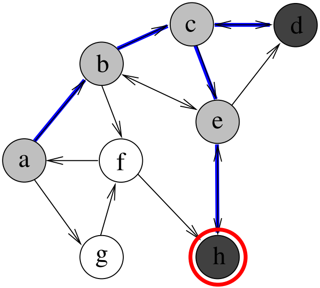


$p = < a, b, c, e, h >$
$s_i = e$, $s_j = h$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
  Soit p une pile (LIFO) initialisée à vide
  pour *tout sommet* $s_i \in S$ **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Empiler s_0 dans p et colorier s_0 en gris
  tant que *p n'est pas vide* **faire**
    Soit s_i le dernier sommet entré dans p
    si $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
      Empiler s_j dans p et colorier s_j en gris
      $\pi[s_j] \leftarrow s_i$
    sinon
      Dépiler s_i de p et colorier s_i en noir
  retourne π

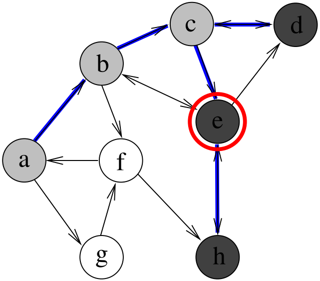


$p = < a, b, c, e >$
$s_i = h$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
- Soit p une pile (LIFO) initialisée à vide
- **pour** *tout sommet* $s_i \in S$ **faire**
  - $\pi[s_i] \leftarrow null$
  - Colorier s_i en blanc
- Empiler s_0 dans p et colorier s_0 en gris
- **tant que** *p n'est pas vide* **faire**
  - **Soit s_i le dernier sommet entré dans p**
  - **si** $\exists s_j \in succ(s_i)$ *tel que* s_j *soit blanc* **alors**
    - Empiler s_j dans p et colorier s_j en gris
    - $\pi[s_j] \leftarrow s_i$
  - **sinon**
    - **Dépiler s_i de p et colorier s_i en noir**
- **retourne** π

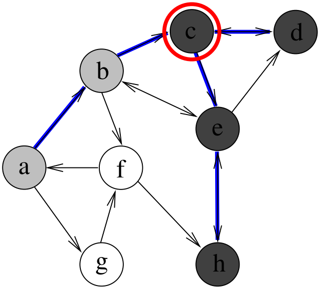


$p = < a, b, c >$
$s_i = e$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
  Soit p une pile (LIFO) initialisée à vide
  pour *tout sommet* $s_i \in S$ **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Empiler s_0 dans p et colorier s_0 en gris
  tant que *p n'est pas vide* **faire**
    Soit s_i le dernier sommet entré dans p
    si $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
      Empiler s_j dans p et colorier s_j en gris
      $\pi[s_j] \leftarrow s_i$
    sinon
      Dépiler s_i de p et colorier s_i en noir
  retourne π

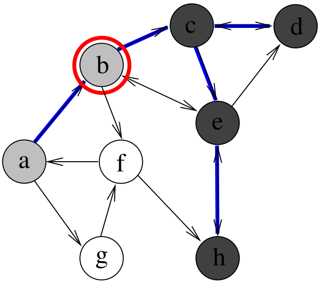


$p = < a, b >$
$s_i = c$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
    Soit p une pile (LIFO) initialisée à vide
    pour *tout sommet* $s_i \in S$ **faire**
        $\pi[s_i] \leftarrow$ *null*
        Colorier s_i en blanc
    Empiler s_0 dans p et colorier s_0 en gris
    tant que *p n'est pas vide* **faire**
        Soit s_i le dernier sommet entré dans p
        si $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
            Empiler s_j dans p et colorier s_j en gris
            $\pi[s_j] \leftarrow s_i$
        sinon
            Dépiler s_i de p et colorier s_i en noir
    retourne π

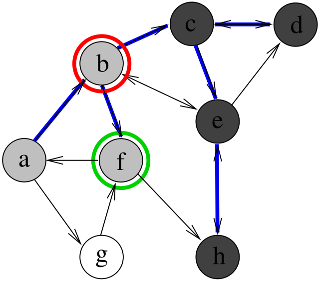


$p = < a, b >$
$s_i = b$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS(g, s_0)*
    Soit p une pile (LIFO) initialisée à vide
    pour *tout sommet $s_i \in S$* **faire**
        $\pi[s_i] \leftarrow$ *null*
        Colorier s_i en blanc
    Empiler s_0 dans p et colorier s_0 en gris
    tant que *p n'est pas vide* **faire**
        Soit s_i le dernier sommet entré dans p
        si $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
            Empiler s_j dans p et colorier s_j en gris
            $\pi[s_j] \leftarrow s_i$
        sinon
            Dépiler s_i de p et colorier s_i en noir
    retourne π

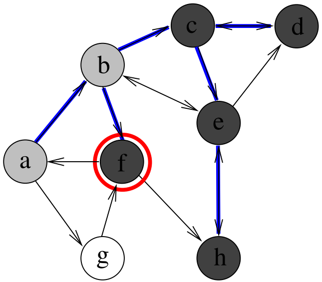


$p = < a, b, f >$
$s_i = b, s_j = f$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS(g, s_0)*
- Soit p une pile (LIFO) initialisée à vide
- **pour** *tout sommet* $s_i \in S$ **faire**
  - $\pi[s_i] \leftarrow$ *null*
  - Colorier s_i en blanc
- Empiler s_0 dans p et colorier s_0 en gris
- **tant que** *p n'est pas vide* **faire**
  - **Soit s_i le dernier sommet entré dans p**
  - **si** $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
    - Empiler s_j dans p et colorier s_j en gris
    - $\pi[s_j] \leftarrow s_i$
  - **sinon**
    - **Dépiler s_i de p et colorier s_i en noir**
- **retourne** π

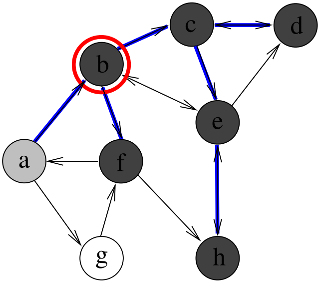


$p = < a, b >$
$s_i = f$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
- Soit p une pile (LIFO) initialisée à vide
- **pour** *tout sommet* $s_i \in S$ **faire**
  - $\pi[s_i] \leftarrow$ *null*
  - Colorier s_i en blanc
- Empiler s_0 dans p et colorier s_0 en gris
- **tant que** *p n'est pas vide* **faire**
  - **Soit s_i le dernier sommet entré dans p**
  - **si** $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
    - Empiler s_j dans p et colorier s_j en gris
    - $\pi[s_j] \leftarrow s_i$
  - **sinon**
    - **Dépiler s_i de p et colorier s_i en noir**
- **retourne** π

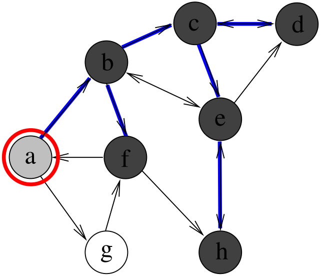


$p = < a >$
$s_i = b$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
    Soit p une pile (LIFO) initialisée à vide
    pour *tout sommet* $s_i \in S$ **faire**
        $\pi[s_i] \leftarrow$ *null*
        Colorier s_i en blanc
    Empiler s_0 dans p et colorier s_0 en gris
    tant que *p n'est pas vide* **faire**
        Soit s_i le dernier sommet entré dans p
        si $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
            Empiler s_j dans p et colorier s_j en gris
            $\pi[s_j] \leftarrow s_i$
        sinon
            Dépiler s_i de p et colorier s_i en noir
    retourne π

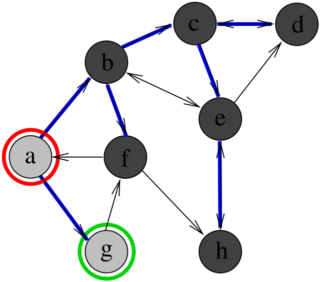


$p = < a >$
$s_i = a$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS(g, s_0)*
  Soit p une pile (LIFO) initialisée à vide
  pour *tout sommet* $s_i \in S$ **faire**
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  Empiler s_0 dans p et colorier s_0 en gris
  tant que *p n'est pas vide* **faire**
    Soit s_i le dernier sommet entré dans p
    si $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
      Empiler s_j dans p et colorier s_j en gris
      $\pi[s_j] \leftarrow s_i$
    sinon
      Dépiler s_i de p et colorier s_i en noir
  retourne π

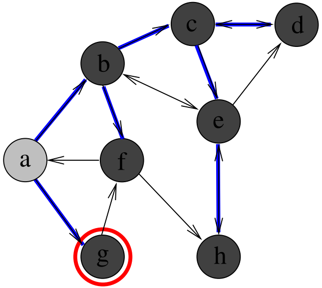


$p = < a, g >$
$s_i = a$, $s_j = g$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
  Soit p une pile (LIFO) initialisée à vide
  pour *tout sommet* $s_i \in S$ **faire**
    $\pi[s_i] \leftarrow$ *null*
    Colorier s_i en blanc
  Empiler s_0 dans p et colorier s_0 en gris
  tant que *p n'est pas vide* **faire**
    Soit s_i le dernier sommet entré dans p
    si $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
      Empiler s_j dans p et colorier s_j en gris
      $\pi[s_j] \leftarrow s_i$
    sinon
      Dépiler s_i de p et colorier s_i en noir
  retourne π

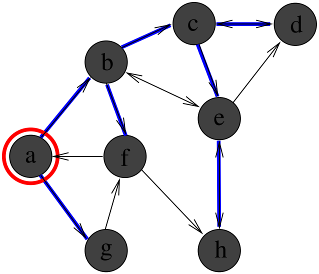


$p = < a >$
$s_i = g$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS(g, s_0)*
- Soit p une pile (LIFO) initialisée à vide
- **pour** *tout sommet $s_i \in S$* **faire**
  - $\pi[s_i] \leftarrow null$
  - Colorier s_i en blanc
- Empiler s_0 dans p et colorier s_0 en gris
- **tant que** *p n'est pas vide* **faire**
  - **Soit s_i le dernier sommet entré dans p**
  - **si** $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
    - Empiler s_j dans p et colorier s_j en gris
    - $\pi[s_j] \leftarrow s_i$
  - **sinon**
    - **Dépiler s_i de p et colorier s_i en noir**
- **retourne** π

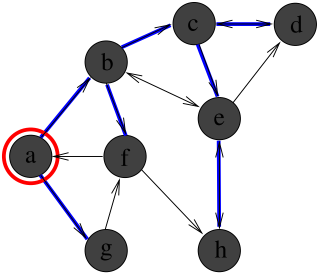


$p = <>$
$s_i = a$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS(g, s_0)*
- Soit p une pile (LIFO) initialisée à vide
- **pour** *tout sommet $s_i \in S$* **faire**
  - $\pi[s_i] \leftarrow null$
  - Colorier s_i en blanc
- Empiler s_0 dans p et colorier s_0 en gris
- **tant que** *p n'est pas vide* **faire**
  - Soit s_i le dernier sommet entré dans p
  - **si** $\exists s_j \in succ(s_i)$ *tel que s_j soit blanc* **alors**
    - Empiler s_j dans p et colorier s_j en gris
    - $\pi[s_j] \leftarrow s_i$
  - **sinon**
    - Dépiler s_i de p et colorier s_i en noir
- **retourne** π

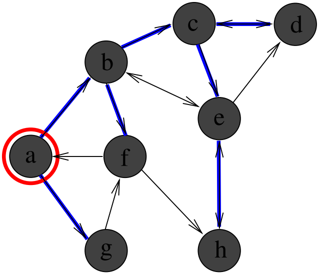


$p = <>$
$s_i = a$

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

Parcours en profondeur (Depth First search / DFS)

Fonction *DFS*(g, s_0)
    Soit p une pile (LIFO) initialisée à vide
    pour *tout sommet* $s_i \in S$ **faire**
        $\pi[s_i] \leftarrow$ *null*
        Colorier s_i en blanc
    Empiler s_0 dans p et colorier s_0 en gris
    tant que p *n'est pas vide* **faire**
        Soit s_i le dernier sommet entré dans p
        si $\exists s_j \in succ(s_i)$ *tel que* s_j *soit blanc* **alors**
            Empiler s_j dans p et colorier s_j en gris
            $\pi[s_j] \leftarrow s_i$
        sinon
            Dépiler s_i de p et colorier s_i en noir
    retourne π



$p = <>$
$s_i = a$

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

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

Version récursive de DFS

Proc *DFSrec*(g, s_0)

  Entrée : Un graphe g et un sommet s_0

  Précond. : s_0 est blanc

  début

    Colorier s_0 en gris

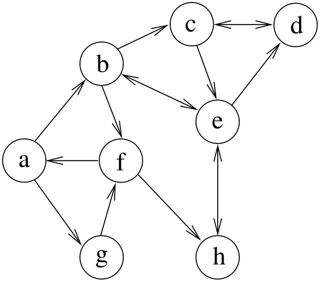
    pour *tout* $s_j \in succ(s_0)$ **faire**

      si s_j *est blanc* **alors**

        $\pi[s_j] \leftarrow s_0$

        DFSrec(g, s_j)

    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

Fonction *booléen possèdeCircuit(g, s_0)*

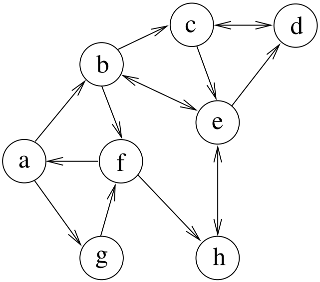
Entrée : Un graphe g et un sommet s_0

Sortie : Vrai si g possède un circuit ; faux sinon

Précond. : s_0 est blanc

début

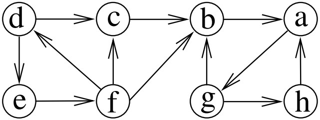
- Colorier s_0 en gris
- **pour** *tout* $s_j \in succ(s_0)$ **faire**
  - **si** s_j *est gris* **alors retourne** *vrai* **;**
  - **sinon si** s_j *est blanc* **alors**
    - **si** *possèdeCircuit*(g, s_j) **alors**
      - **retourne** *vrai*
- Colorier s_0 en noir
- **retourne** *faux*



Numérotation des sommets

Proc *DFSnum(g)*
  $cpt \leftarrow 1$
  Colorier tous les sommets en blanc
  pour *chaque sommet s_i de g* **faire**
    si *s_i est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
  début
    Colorier s_0 en gris
    pour *tout* $s_j \in succ(s_0)$ **faire**
      si *s_j est blanc* **alors**
        $\pi[s_j] \leftarrow s_0$
        DFSrec(g, s_j)
    Colorier s_0 en noir
    $num[s_0] \leftarrow cpt$
    $cpt \leftarrow cpt + 1$

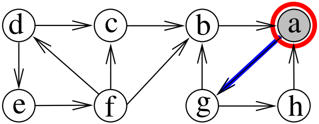


$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

Proc *DFSnum(g)*
  $cpt \leftarrow 1$
  Colorier tous les sommets en blanc
  pour *chaque sommet s_i de g* **faire**
    si *s_i est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
  début
    Colorier s_0 en gris
    pour *tout $s_j \in succ(s_0)$* **faire**
      si *s_j est blanc* **alors**
        $\pi[s_j] \leftarrow s_0$
        DFSrec(g, s_j)
    Colorier s_0 en noir
    $num[s_0] \leftarrow cpt$
    $cpt \leftarrow cpt + 1$



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

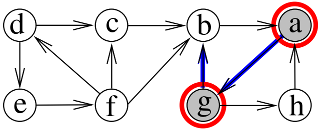
$\rightsquigarrow$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

Proc *DFSnum(g)*
  $cpt \leftarrow 1$
  Colorier tous les sommets en blanc
  pour *chaque sommet s_i de g* **faire**
    si *s_i est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
  début
    Colorier s_0 en gris
    pour *tout $s_j \in succ(s_0)$* **faire**
      si *s_j est blanc* **alors**
        $\pi[s_j] \leftarrow s_0$
        DFSrec(g, s_j)
    Colorier s_0 en noir
    $num[s_0] \leftarrow cpt$
    $cpt \leftarrow cpt + 1$

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

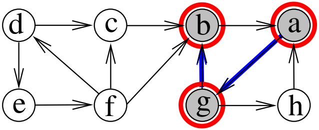


Pile = $< a, g >$
cpt = 1
$s_0 = g$; $s_j = b$

Numérotation des sommets

Proc *DFSnum(g)*
- $cpt \leftarrow 1$
- Colorier tous les sommets en blanc
- **pour** *chaque sommet s_i de g* **faire**
  - **si** s_i *est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
- **début**
  - **Colorier s_0 en gris**
  - **pour** *tout* $s_j \in succ(s_0)$ **faire**
    - **si** s_j *est blanc* **alors**
      - $\pi[s_j] \leftarrow s_0$
      - DFSrec(g, s_j)
  - Colorier s_0 en noir
  - $num[s_0] \leftarrow cpt$
  - $cpt \leftarrow cpt + 1$



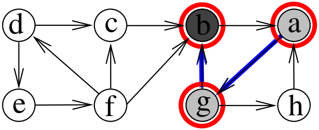
Pile = $< a, g, b >$
cpt = 1
$s_0 = b$

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

Numérotation des sommets

Proc *DFSnum(g)*
- $cpt \leftarrow 1$
- Colorier tous les sommets en blanc
- **pour** *chaque sommet* s_i *de* g **faire**
  - **si** s_i *est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
- **début**
  - Colorier s_0 en gris
  - **pour** *tout* $s_j \in succ(s_0)$ **faire**
    - **si** s_j *est blanc* **alors**
      - $\pi[s_j] \leftarrow s_0$
      - DFSrec(g, s_j)
  - **Colorier s_0 en noir**
  - $num[s_0] \leftarrow cpt$
  - $cpt \leftarrow cpt + 1$



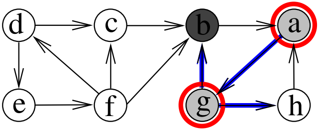
Pile = $< a, g, b >$
cpt = 2
$s_0 = b$
num[b]=1

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

Numérotation des sommets

Proc *DFSnum(g)*
- $cpt \leftarrow 1$
- Colorier tous les sommets en blanc
- **pour** *chaque sommet s_i de g* **faire**
  - **si** *s_i est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
- **début**
  - Colorier s_0 en gris
  - **pour** *tout $s_j \in succ(s_0)$* **faire**
    - **si** *s_j est blanc* **alors**
      - $\pi[s_j] \leftarrow s_0$
      - **DFSrec**(g, s_j)
  - Colorier s_0 en noir
  - $num[s_0] \leftarrow cpt$
  - $cpt \leftarrow cpt + 1$



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

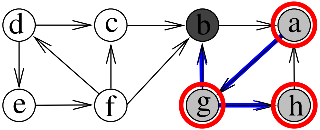
$\leadsto$ *num* = Numéro d'ordre de coloriage en noir

Numérotation des sommets

Proc *DFSnum(g)*
    $cpt \leftarrow 1$
    Colorier tous les sommets en blanc
    pour *chaque sommet* s_i *de g* **faire**
        si s_i *est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec*(g, s_0)
    début
        Colorier s_0 en gris
        pour *tout* $s_j \in succ(s_0)$ **faire**
            si s_j *est blanc* **alors**
                $\pi[s_j] \leftarrow s_0$
                DFSrec(g, s_j)
        Colorier s_0 en noir
        $num[s_0] \leftarrow cpt$
        $cpt \leftarrow cpt + 1$

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

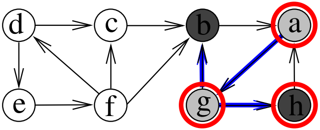


Pile = $< a, g, h >$
cpt = 2
$s_0 = h$
num[b]=1

Numérotation des sommets

Proc *DFSnum(g)*
  $cpt \leftarrow 1$
  Colorier tous les sommets en blanc
  pour *chaque sommet* s_i *de* g **faire**
    si s_i *est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec*(g, s_0)
  début
    Colorier s_0 en gris
    pour *tout* $s_j \in succ(s_0)$ **faire**
      si s_j *est blanc* **alors**
        $\pi[s_j] \leftarrow s_0$
        DFSrec(g, s_j)
    Colorier s_0 en noir
    $num[s_0] \leftarrow cpt$
    $cpt \leftarrow cpt + 1$



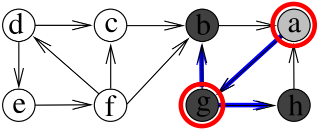
Pile = $< a, g, h >$
cpt = 3
$s_0 = h$
num[b]=1, num[h]=2

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

Numérotation des sommets

Proc *DFSnum(g)*
  $cpt \leftarrow 1$
  Colorier tous les sommets en blanc
  pour *chaque sommet s_i de g* **faire**
    si s_i *est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
  début
    Colorier s_0 en gris
    pour *tout* $s_j \in succ(s_0)$ **faire**
      si s_j *est blanc* **alors**
        $\pi[s_j] \leftarrow s_0$
        DFSrec(g, s_j)
    Colorier s_0 en noir
    $num[s_0] \leftarrow cpt$
    $cpt \leftarrow cpt + 1$



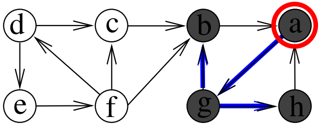
Pile = $< a, g >$
cpt = 4
$s_0 = g$
num[b]=1, num[h]=2, num[g]=3

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

Numérotation des sommets

Proc *DFSnum(g)*
- $cpt \leftarrow 1$
- Colorier tous les sommets en blanc
- **pour** *chaque sommet* s_i *de g* **faire**
  - **si** s_i *est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec*(g, s_0)
- **début**
  - Colorier s_0 en gris
  - **pour** *tout* $s_j \in succ(s_0)$ **faire**
    - **si** s_j *est blanc* **alors**
      - $\pi[s_j] \leftarrow s_0$
      - DFSrec(g, s_j)
  - **Colorier s_0 en noir**
  - $num[s_0] \leftarrow cpt$
  - $cpt \leftarrow cpt + 1$



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

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

Numérotation des sommets

Proc *DFSnum(g)*

  $cpt \leftarrow 1$

  Colorier tous les sommets en blanc

  pour *chaque sommet s_i de g* **faire**

    si *s_i est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*

  début

    Colorier s_0 en gris

    pour *tout $s_j \in succ(s_0)$* **faire**

      si *s_j est blanc* **alors**

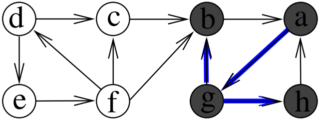
        $\pi[s_j] \leftarrow s_0$

        DFSrec(g, s_j)

    Colorier s_0 en noir

    $num[s_0] \leftarrow cpt$

    $cpt \leftarrow cpt + 1$



Pile = $<>$
cpt = 5

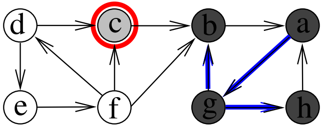
num[b]=1, num[h]=2, num[g]=3, num[a]=4

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

Numérotation des sommets

Proc *DFSnum(g)*
- $cpt \leftarrow 1$
- Colorier tous les sommets en blanc
- **pour** *chaque sommet s_i de g* **faire**
  - **si** *s_i est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
- **début**
  - **Colorier s_0 en gris**
  - **pour** *tout $s_j \in succ(s_0)$* **faire**
    - **si** *s_j est blanc* **alors**
      - $\pi[s_j] \leftarrow s_0$
      - DFSrec(g, s_j)
  - Colorier s_0 en noir
  - $num[s_0] \leftarrow cpt$
  - $cpt \leftarrow cpt + 1$



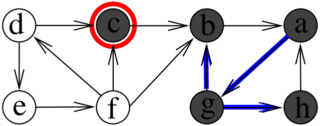
Pile = $< c >$
cpt = 5
$s_0 = c$
num[b]=1, num[h]=2, num[g]=3, num[a]=4

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

Numérotation des sommets

Proc *DFSnum(g)*
- $cpt \leftarrow 1$
- Colorier tous les sommets en blanc
- **pour** *chaque sommet s_i de g* **faire**
  - **si** s_i *est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
- **début**
  - Colorier s_0 en gris
  - **pour** *tout* $s_j \in succ(s_0)$ **faire**
    - **si** s_j *est blanc* **alors**
      - $\pi[s_j] \leftarrow s_0$
      - DFSrec(g, s_j)
  - **Colorier s_0 en noir**
  - $num[s_0] \leftarrow cpt$
  - $cpt \leftarrow cpt + 1$



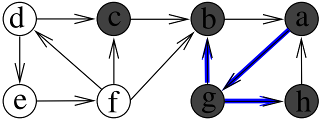
Pile = $< c >$
cpt = 6
$s_0 = c$
num[b]=1, num[h]=2, num[g]=3, num[a]=4, num[c]=5

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

Numérotation des sommets

Proc *DFSnum(g)*
- $cpt \leftarrow 1$
- Colorier tous les sommets en blanc
- **pour** *chaque sommet s_i de g* **faire**
  - **si** s_i *est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
- **début**
  - Colorier s_0 en gris
  - **pour** *tout* $s_j \in succ(s_0)$ **faire**
    - **si** s_j *est blanc* **alors**
      - $\pi[s_j] \leftarrow s_0$
      - DFSrec(g, s_j)
  - Colorier s_0 en noir
  - $num[s_0] \leftarrow cpt$
  - $cpt \leftarrow cpt + 1$



Pile = $<>$
cpt = 6

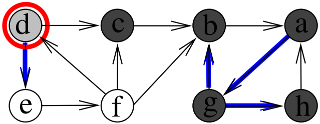
num[b]=1, num[h]=2, num[g]=3, num[a]=4, num[c]=5

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

Numérotation des sommets

```
Proc DFSnum(g)
    cpt ← 1
    Colorier tous les sommets en blanc
    pour chaque sommet s_i de g faire
        si s_i est blanc alors DFSrec(g, s_i);

Proc DFSrec(g, s_0)
    début
        Colorier s_0 en gris
        pour tout s_j ∈ succ(s_0) faire
            si s_j est blanc alors
                π[s_j] ← s_0
                DFSrec(g, s_j)
        Colorier s_0 en noir
        num[s_0] ← cpt
        cpt ← cpt + 1
```



Pile = $< d >$

cpt = 6

$s_0 = d$; $s_j = e$

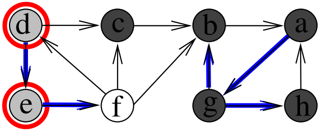
num[b]=1, num[h]=2, num[g]=3, num[a]=4, num[c]=5

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

Numérotation des sommets

Proc *DFSnum(g)*
  $cpt \leftarrow 1$
  Colorier tous les sommets en blanc
  pour *chaque sommet* s_i *de g* **faire**
    si s_i *est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec*(g, s_0)
  début
    Colorier s_0 en gris
    pour *tout* $s_j \in succ(s_0)$ **faire**
      si s_j *est blanc* **alors**
        $\pi[s_j] \leftarrow s_0$
        DFSrec(g, s_j)
    Colorier s_0 en noir
    $num[s_0] \leftarrow cpt$
    $cpt \leftarrow cpt + 1$



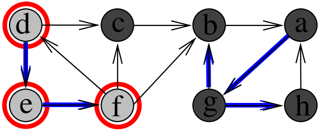
Pile = $< d, e >$
cpt = 6
$s_0 = e$; $s_j = f$
num[b]=1, num[h]=2, num[g]=3, num[a]=4, num[c]=5

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

Numérotation des sommets

Proc *DFSnum(g)*
  $cpt \leftarrow 1$
  Colorier tous les sommets en blanc
  pour *chaque sommet s_i de g* **faire**
    si s_i *est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
  début
    Colorier s_0 en gris
    pour *tout* $s_j \in succ(s_0)$ **faire**
      si s_j *est blanc* **alors**
        $\pi[s_j] \leftarrow s_0$
        DFSrec(g, s_j)
    Colorier s_0 en noir
    $num[s_0] \leftarrow cpt$
    $cpt \leftarrow cpt + 1$



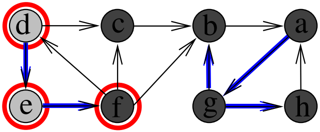
Pile = $< d, e, f >$
cpt = 6
$s_0 = f$
num[b]=1, num[h]=2, num[g]=3, num[a]=4, num[c]=5

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

Numérotation des sommets

Proc *DFSnum(g)*
- $cpt \leftarrow 1$
- Colorier tous les sommets en blanc
- **pour** *chaque sommet* s_i *de* g **faire**
  - **si** s_i *est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
- **début**
  - Colorier s_0 en gris
  - **pour** *tout* $s_j \in succ(s_0)$ **faire**
    - **si** s_j *est blanc* **alors**
      - $\pi[s_j] \leftarrow s_0$
      - DFSrec(g, s_j)
  - **Colorier s_0 en noir**
  - $num[s_0] \leftarrow cpt$
  - $cpt \leftarrow cpt + 1$



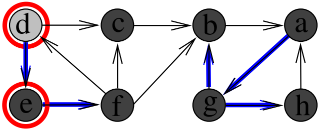
Pile = $< d, e, f >$
cpt = 7
$s_0 = f$
num[b]=1, num[h]=2, num[g]=3, num[a]=4, num[c]=5, num[f]=6

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

Numérotation des sommets

Proc *DFSnum(g)*
  $cpt \leftarrow 1$
  Colorier tous les sommets en blanc
  pour *chaque sommet s_i de g* **faire**
    si *s_i est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
  début
    Colorier s_0 en gris
    pour *tout $s_j \in succ(s_0)$* **faire**
      si *s_j est blanc* **alors**
        $\pi[s_j] \leftarrow s_0$
        DFSrec(g, s_j)
    Colorier s_0 en noir
    $num[s_0] \leftarrow cpt$
    $cpt \leftarrow cpt + 1$



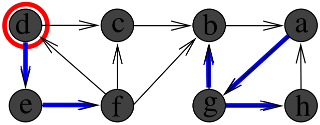
Pile = $< d, e >$
cpt = 8
$s_0 = e$
num[b]=1, num[h]=2, num[g]=3, num[a]=4, num[c]=5, num[f]=6, num[e]=7

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

Numérotation des sommets

Proc *DFSnum(g)*
  $cpt \leftarrow 1$
  Colorier tous les sommets en blanc
  pour *chaque sommet s_i de g* **faire**
    si *s_i est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
  début
    Colorier s_0 en gris
    pour *tout $s_j \in succ(s_0)$* **faire**
      si *s_j est blanc* **alors**
        $\pi[s_j] \leftarrow s_0$
        DFSrec(g, s_j)
    Colorier s_0 en noir
    $num[s_0] \leftarrow cpt$
    $cpt \leftarrow cpt + 1$



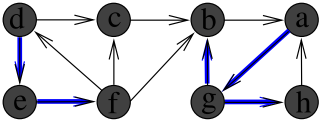
Pile = $< d >$
cpt = 9
$s_0 = d$
num[b]=1, num[h]=2, num[g]=3, num[a]=4, num[c]=5, num[f]=6, num[e]=7, num[d]=8

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

Numérotation des sommets

Proc *DFSnum(g)*
- $cpt \leftarrow 1$
- Colorier tous les sommets en blanc
- **pour** *chaque sommet s_i de g* **faire**
  - **si** s_i *est blanc* **alors** DFSrec(g, s_i);

Proc *DFSrec(g, s_0)*
- **début**
  - Colorier s_0 en gris
  - **pour** *tout* $s_j \in succ(s_0)$ **faire**
    - **si** s_j *est blanc* **alors**
      - $\pi[s_j] \leftarrow s_0$
      - DFSrec(g, s_j)
  - Colorier s_0 en noir
  - $num[s_0] \leftarrow cpt$
  - $cpt \leftarrow cpt + 1$



Pile = $<>$
cpt = 9

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

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

Tri topologique d'un DAG

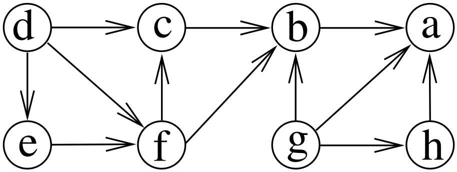
Définitions :

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

Théorème :

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

Exercice 1 : Tri topologique du graphe suivant



Tri topologique d'un DAG

Définitions :

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

Théorème :

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

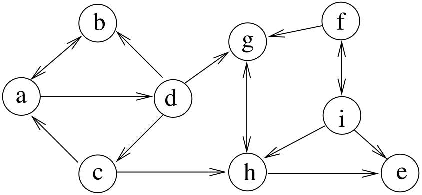
Exercice 2 : Comparer la complexité avec un algo *decrease and conquer*

Soit L une liste vide
tant que *G possède un sommet v tq* $d^-(v) = 0$ **faire**
    Ajouter v dans L
    Supprimer v (et les arcs incidents à v) de G
si *G n'a plus de sommets* **alors retourne** L;
sinon afficher "Le graphe n'est pas un DAG !";

Recherche des composantes fortement connexes

Fonction *SCC(g)*
- $SCC \leftarrow \emptyset$
- $DFSnum(g)$
- Construire le graphe $g^t = (S, A^t)$ tel que $A^t = \{(s_i, s_j) \mid (s_j, s_i) \in A\}$
- Colorier tous les sommets de g^t en blanc
- **pour** *chaque sommet s_i pris par ordre de num décroissant* **faire**
  - **si** *s_i est blanc* **alors**
    - $Blanc \leftarrow \{s_j \in S | s_j$ est blanc$\}$
    - DFSrec(g^t, s_i)
    - Ajouter à SCC l'ensemble $\{s_j \in Blanc \mid s_j$ est noir$\}$
- **retourne** *SCC*

Exercice :



Recherche des composantes fortement connexes

Fonction *SCC(g)*
    $SCC \leftarrow \emptyset$
    $DFSnum(g)$
    Construire le graphe $g^t = (S, A^t)$ tel que $A^t = \{(s_i, s_j) \mid (s_j, s_i) \in A\}$
    Colorier tous les sommets de g^t en blanc
    pour *chaque sommet s_i pris par ordre de num décroissant* **faire**
        si s_i *est blanc* **alors**
            $Blanc \leftarrow \{s_j \in S | s_j \text{ est blanc}\}$
            DFSrec(g^t, s_i)
            Ajouter à SCC l'ensemble $\{s_j \in Blanc \mid s_j \text{ est noir}\}$
    retourne *SCC*

Preuve de correction :

$\leadsto$ Faite en TD

1. Introduction
2. Définitions
3. Structures de données pour représenter un graphe
4. Parcours de graphes
5. Plus courts chemins
   - Définitions et propriétés
   - 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. Quelques problèmes NP-difficiles sur les graphes

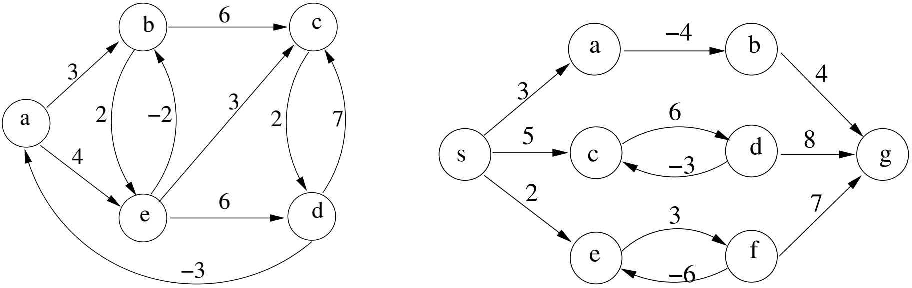
Définitions :

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

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

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

Exemples :



Principe d'optimalité d'une sous-structure

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

Soit $p =< s_0, s_1, \ldots, s_k >$ un plus court chemin
$\forall i, j$ tel que $0 \leq i \leq j \leq k$: $p_{ij} =< s_i, s_{i+1}, \ldots, s_j >$ est un plus court chemin

Exercice :

Démonstration...

Exploitation par des algorithmes :

- Dijkstra, Bellman-Ford :

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

- 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. Définitions
3. Structures de données pour représenter un graphe
4. Parcours de graphes
5. Plus courts chemins
   - Définitions et propriétés
   - 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. Quelques problèmes NP-difficiles sur les graphes

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

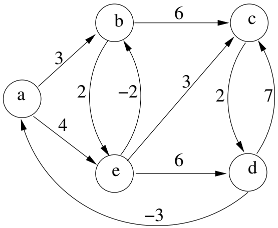
Fonction *PlusCourtsChemins*(g, c, s_0)

- **Entrée** : Un graphe (orienté ou non) $g = (S, A)$
  Une fonction de coût $c : A \to \mathbb{R}$
  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 $s_i \to s_j$ 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 :



Principe des algorithmes de recherche de plus courts chemins :

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

Proc *relacher*($(s_i, s_j), \pi, d$)

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

```
début
    si d[s_j] > d[s_i] + c(s_i, s_j) alors
        d[s_j] ← d[s_i] + c(s_i, s_j)
        π[s_j] ← s_i
```

Algorithmes de recherche de plus courts chemins

Principe commun à tous les algorithmes :

- Grignotage progressif de *d* en relâchant des arcs

Question : Dans quel ordre relâcher les arcs ?

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

Principe de l'algorithme de Dijkstra

Généralisation d'un BFS à des graphes valués :

- Procède par coloriage des sommets :
  - s_i est blanc s'il n'a pas encore été découvert
    $\leadsto d[s_i] = +\infty$
  - 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
- A chaque itération, un sommet gris est colorié en noir et ses arcs sont relâchés
  - Stratégie **gloutonne** pour choisir ce sommet gris
    $\leadsto$ Sommet gris minimisant d
  - **Ne marche que si tous les coûts sont positifs**
    $\leadsto$ Précondition à Dijkstra : Pour tout arc $(s_i, s_j) \in A, cout(s_i, s_j) \geq 0$

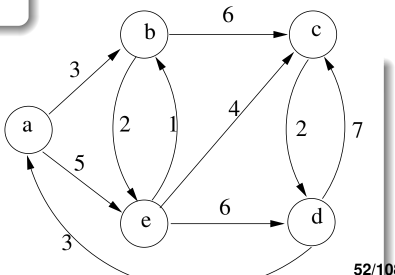
1 **Fonction** *Dijkstra*(g, c, s_0)
2 **pour** *chaque sommet* $s_i \in S$ **faire**
3 $d[s_i] \leftarrow +\infty$; $\pi[s_i] \leftarrow null$; Colorier s_i en blanc
4 $d[s_0] \leftarrow 0$; Colorier s_0 en gris
5 **tant que** *il existe un sommet gris* **faire**
6 Soit s_i le sommet gris tq $d[s_i]$ 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**
11 Colorier s_j en gris
12 Colorier s_i en noir
13 **retourne** π *et* d

Propriété invariante ligne 5 :

Pour tout sommet s_i,

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

Exemple :



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

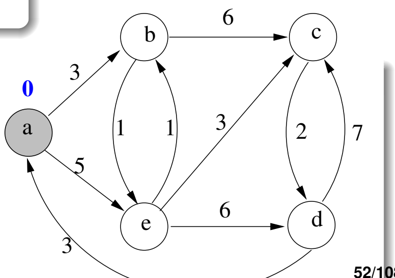
Propriété invariante ligne 5 :

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

Exemple :

$s_0 = a$



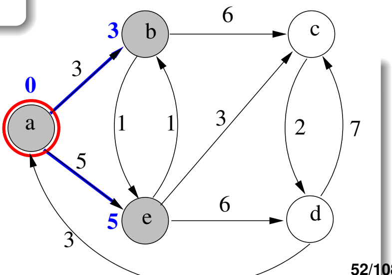
1 **Fonction** *Dijkstra*(g, c, s_0)
2 **pour** *chaque sommet* $s_i \in S$ **faire**
3 $d[s_i] \leftarrow +\infty$; $\pi[s_i] \leftarrow null$; Colorier s_i en blanc
4 $d[s_0] \leftarrow 0$; Colorier s_0 en gris
5 **tant que** *il existe un sommet gris* **faire**
6 **Soit s_i le sommet gris tq $d[s_i]$ 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**
11 **Colorier s_j en gris**
12 Colorier s_i en noir
13 **retourne** π *et* d

Propriété invariante ligne 5 :

Pour tout sommet s_i,

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

Exemple :



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

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

Propriété invariante ligne 5 :

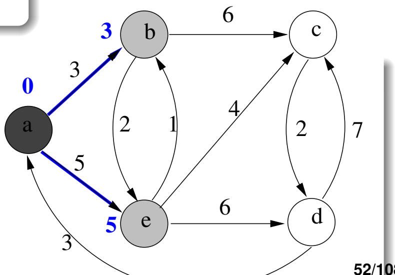
Pour tout sommet s_i,

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

Exemple :

$s_i = a$

Arcs relâchés : $(a, b), (a, e)$



1 **Fonction** *Dijkstra*(g, 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**
11   **Colorier s_j en gris**
12   Colorier s_i en noir
13   **retourne** π *et* d

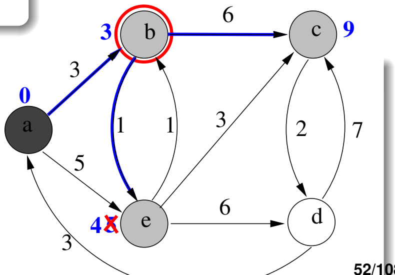
Propriété invariante ligne 5 :

Pour tout sommet s_i,

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

Exemple :

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



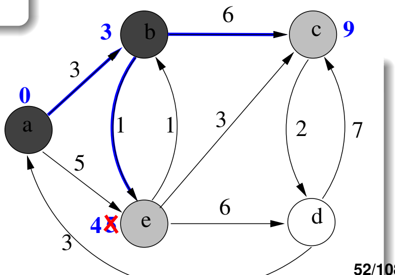
1 **Fonction** *Dijkstra*(g, c, s_0)
2 **pour** *chaque sommet* $s_i \in S$ **faire**
3 $d[s_i] \leftarrow +\infty$; $\pi[s_i] \leftarrow null$; Colorier s_i en blanc
4 $d[s_0] \leftarrow 0$; Colorier s_0 en gris
5 **tant que** *il existe un sommet gris* **faire**
6 Soit s_i le sommet gris tq $d[s_i]$ 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**
11 Colorier s_j en gris
12 **Colorier s_i en noir**
13 **retourne** π *et* d

Propriété invariante ligne 5 :

Pour tout sommet s_i,

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

Exemple :



$s_i = b$

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

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

Propriété invariante ligne 5 :

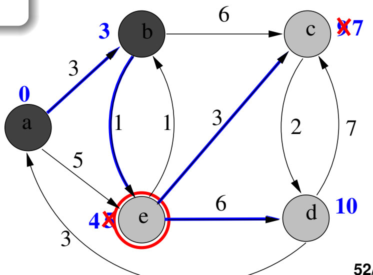
Pour tout sommet s_i,

- 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

Exemple :

$s_i = e$

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



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

Propriété invariante ligne 5 :

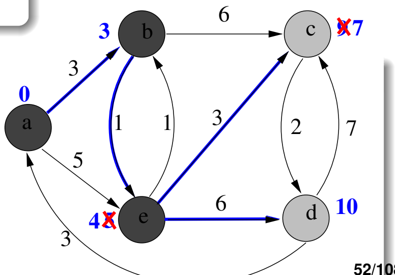
Pour tout sommet s_i,

- 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

Exemple :

$s_i = e$

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



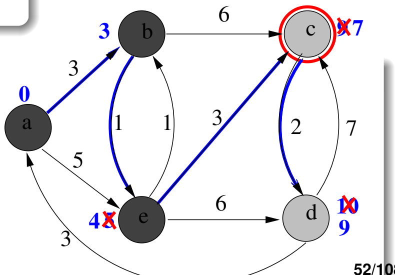
```
1  Fonction Dijkstra(g, c, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet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

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

Exemple :



$s_i = c$

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

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

Propriété invariante ligne 5 :

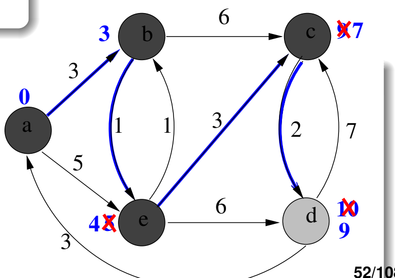
Pour tout sommet s_i,

- 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

Exemple :

$s_i = c$

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



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

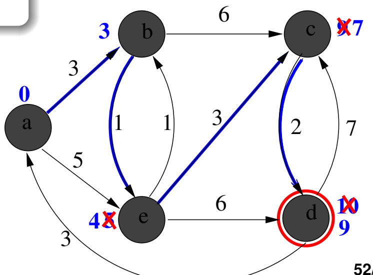
Propriété invariante ligne 5 :

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

Exemple :

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



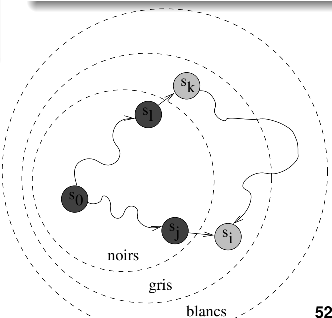
1 **Fonction** *Dijkstra*(g, c, s_0)
2 **pour** *chaque sommet* $s_i \in S$ **faire**
3 $d[s_i] \leftarrow +\infty$; $\pi[s_i] \leftarrow$ *null* ; Colorier s_i en blanc
4 $d[s_0] \leftarrow 0$; Colorier s_0 en gris
5 **tant que** *il existe un sommet gris* **faire**
6 Soit s_i le sommet gris tq $d[s_i]$ 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**
11 Colorier s_j en gris
12 Colorier s_i en noir
13 **retourne** π *et* d

Propriété invariante ligne 5 :

Pour tout sommet s_i,

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

Exemple :



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

Propriété invariante ligne 5 :

Pour tout sommet s_i,

- Si s_i est gris alors $d[s_i]$ = longueur du plus court chemin de s_0 à s_i ne passant que par des sommets noirs
- Si s_i est noir alors $d[s_i] = \delta(s_0, s_i)$
- 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, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet sj ∈ succ(si) faire
8              si sj est blanc ou gris alors
9                  relacher((si, sj), π, d)
10                 si sj est blanc alors
11                     Colorier sj en gris
12         Colorier si en noir
13     retourne π et d
```

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**
11   Colorier s_j en gris
12   Colorier s_i en noir
13   **retourne** π *et d*

Propriété invariante ligne 5 :

Pour tout sommet s_i,

- Si s_i est gris alors $d[s_i]$ = longueur du plus court chemin de s_0 à s_i ne passant que par des sommets noirs
- Si s_i est noir alors $d[s_i] = \delta(s_0, s_i)$
- 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
  $\leadsto$ 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

Algorithme TopoDAG

Fonction *TopoDAG*(g, c, s_0)

  Précond. : g est un DAG

  pour *chaque sommet* $s_i \in S$ **faire**

    $d[s_i] \leftarrow +\infty$

    $\pi[s_i] \leftarrow null$

  $d[s_0] \leftarrow 0$

  Trier topologiquement les sommets de g à l'aide d'un parcours en profondeur

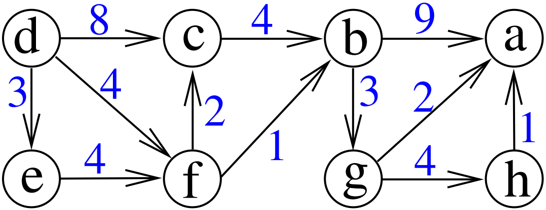
  pour *chaque sommet* s_i *pris selon l'ordre topologique* **faire**

    pour *chaque sommet* $s_j \in succ(s_i)$ **faire**

      relacher$((s_i, s_j), \pi, d)$

  retourne π *et* d

Exercice :



Algorithme TopoDAG

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

Complexité pour un graphe ayant n sommets et p arcs ?

Algorithme TopoDAG

Fonction *TopoDAG*(g, c, s_0)

  Précond. : g est un DAG

  pour *chaque sommet* $s_i \in S$ **faire**

    $d[s_i] \leftarrow +\infty$

    $\pi[s_i] \leftarrow null$

  $d[s_0] \leftarrow 0$

  Trier topologiquement les sommets de g à l'aide d'un parcours en profondeur

  pour *chaque sommet* s_i *pris selon l'ordre topologique* **faire**

    pour *chaque sommet* $s_j \in succ(s_i)$ **faire**

      relacher$((s_i, s_j), \pi, d)$

  retourne π *et* d

Complexité pour un graphe ayant n sommets et p arcs ?

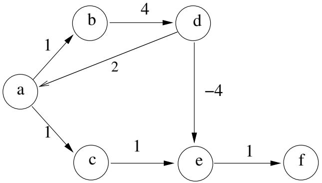
$\leadsto \mathcal{O}(n + p)$

Points communs et limitations de Dijkstra et TopoDAG

Disjkstra et TopoDAG sont des algorithmes gloutons

- Si $d[s_i] = \delta(s_0, s_i)$, alors on peut relâcher chaque arc (s_i, s_j)
  $\leadsto$ Principe d'optimalité des sous-chemins :
  $$\delta(s_0, s_j) = \min_{s_i \in pred(s_j)} \delta(s_0, s_i) + c(s_i, s_j)$$
- Principe glouton pour choisir s_i :
  - Dijkstra : Choisir s_i gris qui minimise d
    $\leadsto$ Si tous les coûts sont positifs, $d[s_i]$ ne pourra plus diminuer
  - TopoDAG : Choisir s_i selon un ordre topologique
    $\leadsto$ 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, ..

$\leadsto$ 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 récursivement la solution d'un sous-problème

- Identifier les cas de base, où la solution est connue sans calcul
- Définir la solution aux sous-problèmes plus compliqués en fonction des solutions de sous-problèmes plus simples

Eq. de Bellman basées sur le principe **d'optimalité des sous-structures**

Etape 3 : Analyse des performances

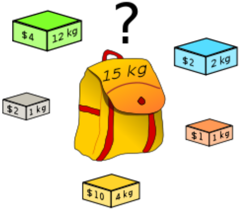
Combien de sous-problèmes différents doivent être résolus ?

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

- Top-Down : Implémentation récursive + mémoïsation
  $\rightsquigarrow$ Facile à programmer (et à prouver correct !)
- Bottom-Up : Implémentation itérative, en remplissant un tableau
  $\rightsquigarrow$ Parfois plus économe en mémoire

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

Description du problème :



- Entrée : un poids max p_{max} et un ensemble d'objets $O = \{1, \ldots, n\}$
  $\rightsquigarrow$ 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

$\forall i \in [1, n], \forall p \in [1, p_{max}]$: $m(i, p) =$ sol. opt. qd $O = \{1, \ldots, i\}$ 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 s_i : sous-problème $d(k, s_i)$
$\leadsto$ longueur du + court chemin de s_0 jusque s_i **utilisant au + k arcs**

Etape 2 : Equations de Bellman

- $k = 0$: $d(0, s_i) = 0$ si $s_i = s_0$ et $d(0, s_i) = +\infty$ sinon
- $k > 1$:
  $d(k, s_i) = \min(\{d(k-1, s_i)\} \cup \{d(k-1, s_j) + c(s_j, s_i) | s_j \in pred(s_i)\}$

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

Etape 3 : Analyse des performances :

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

Etape 4 : Calcul récursif Top-down

Rappel des équations de Bellman :

- $k = 0$: $d(0, s_i) = 0$ si $s_i = s_0$ et $d(0, s_i) = +\infty$ sinon
- $k > 1$:
  $d(k, s_i) = \min(\{d(k-1, s_i)\} \cup \{d(k-1, s_j) + c(s_j, s_i) | s_j \in pred(s_i)\}$

Fonction *Calcul-récursif*($g, c, s_0, \mathbf{k}, \mathbf{s_i}$)
    si $k = 0$ **alors**
        si $s_i = s_0$ **alors retourne** *0*;
        retourne ∞
    $d \leftarrow$ Calcul-récursif($g, c, s_0, \mathbf{k-1}, \mathbf{s_i}$)
    pour *chaque sommet* $s_j \in pred(s_i)$ **faire**
        $d \leftarrow \min(d,$ Calcul-récursif$(g, c, s_0, \mathbf{k-1}, \mathbf{s_j}) + c(s_j, s_i))$
    retourne d

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

Complexité de cet algorithme ?

Etape 4 : Calcul récursif Top-down

Rappel des équations de Bellman :

- $k = 0$: $d(0, s_i) = 0$ si $s_i = s_0$ et $d(0, s_i) = +\infty$ sinon
- $k > 1$:
  $d(k, s_i) = \min(\{d(k-1, s_i)\} \cup \{d(k-1, s_j) + c(s_j, s_i) | s_j \in pred(s_i)\}$

Fonction *Calcul-récursif*$(g, c, s_0, \mathbf{k}, \mathbf{s_i})$
    si $k = 0$ **alors**
        si $s_i = s_0$ **alors retourne** *0*;
        retourne ∞
    $d \leftarrow$ Calcul-récursif$(g, c, s_0, \mathbf{k-1}, \mathbf{s_i})$
    pour *chaque sommet* $s_j \in pred(s_i)$ **faire**
        $d \leftarrow \min(d,$ Calcul-récursif$(g, c, s_0, \mathbf{k-1}, \mathbf{s_j}) + c(s_j, s_i))$
    retourne d

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

Complexité de cet algorithme ?

Il faut mémoïser pour ne pas calculer plusieurs fois $d(k, s_i)$!

Etape 4 : Calcul récursif Top-down avec mémoïsation

Rappel des équations de Bellman :

- $k = 0$: $d(0, s_i) = 0$ si $s_i = s_0$ et $d(0, s_i) = +\infty$ sinon
- $k > 1$:
  $d(k, s_i) = \min(\{d(k-1, s_i)\} \cup \{d(k-1, s_j) + c(s_j, s_i) | s_j \in pred(s_i)\}$

Utilisation d'une var. globale d pour mémoïser les valeurs calculées :

```
Fonction Calcul-réc-mémo(g, c, s0, k, si)
    si k = 0 alors
        si si = s0 alors retourne 0 sinon retourne ∞;
    si d[k][si] = null alors
        d[k][si] ← Calcul-réc-mémo(g, c, s0, k − 1, si)
        pour chaque sommet sj ∈ pred(si) faire
            d[k][si] ← min(d[k][si], Calcul-réc-mémo(g, c, s0, k − 1, sj) + c(sj, si))
    retourne d[k][si]
```

Complexité :

Etape 4 : Calcul récursif Top-down avec mémoïsation

Rappel des équations de Bellman :

- $k = 0$: $d(0, s_i) = 0$ si $s_i = s_0$ et $d(0, s_i) = +\infty$ sinon
- $k > 1$:
  $d(k, s_i) = \min(\{d(k-1, s_i)\} \cup \{d(k-1, s_j) + c(s_j, s_i) | s_j \in pred(s_i)\}$

Utilisation d'une var. globale d pour mémoïser les valeurs calculées :

```
1 Fonction Calcul-réc-mémo(g, c, s0, k, si)
2     si k = 0 alors
3         si si = s0 alors retourne 0 sinon retourne ∞;
4     si d[k][si] = null alors
5         d[k][si] ← Calcul-réc-mémo(g, c, s0, k − 1, si)
6         pour chaque sommet sj ∈ pred(si) faire
7             d[k][si] ← min(d[k][si], Calcul-réc-mémo(g, c, s0, k − 1, sj) + c(sj, si))
8     retourne d[k][si]
```

Complexité : $\mathcal{O}(np)$

- Lignes 5-7 exécutées pour chaque $k \in [0, n-1]$ et $s_i \in S \Rightarrow \mathcal{O}(n^2)$
- Chaque exécution $\Rightarrow$ Parcours de l'ensemble $pred(s_i)$

Etape 4 : Calcul itératif, de bas en haut

Rappel des équations de Bellman :

- $k = 0$: $d(0, s_i) = 0$ si $s_i = s_0$ et $d(0, s_i) = +\infty$ sinon
- $k > 1$:
  $d(k, s_i) = \min(\{d(k-1, s_i)\} \cup \{d(k-1, s_j) + c(s_j, s_i) | s_j \in pred(s_i)\})$

Fonction *Calcul-itératif(g, c, s_0)*
    pour *chaque sommet* $s_i \in S$ **faire** $d[0][s_i] \leftarrow +\infty$;
    $d[0][s_0] \leftarrow 0$
    pour k *variant de* 1 *à* $\#S - 1$ **faire**
        pour *chaque sommet* $s_i \in S$ **faire**
            $d[k][s_i] \leftarrow d[k-1][s_i]$
            pour *chaque sommet* $s_j \in pred(s_i)$ **faire**
                $d[k][s_i] \leftarrow \min(d[k][s_i], d[k-1][s_j] + c(s_j, s_i))$
    retourne $d[\#S - 1]$

Complexité de cet algorithme ?

Etape 4 : Calcul itératif, de bas en haut

Rappel des équations de Bellman :

- $k = 0$: $d(0, s_i) = 0$ si $s_i = s_0$ et $d(0, s_i) = +\infty$ sinon
- $k > 1$:
  $d(k, s_i) = \min(\{d(k-1, s_i)\} \cup \{d(k-1, s_j) + c(s_j, s_i) | s_j \in pred(s_i)\}$

Fonction *Calcul-itératif*(g, c, s_0)
    pour *chaque sommet* $s_i \in S$ **faire** $d[0][s_i] \leftarrow +\infty$;
    $d[0][s_0] \leftarrow 0$
    pour k *variant de* 1 *à* $\#S - 1$ **faire**
        pour *chaque sommet* $s_i \in S$ **faire**
            $d[k][s_i] \leftarrow d[k-1][s_i]$
            pour *chaque sommet* $s_j \in pred(s_i)$ **faire**
                $d[k][s_i] \leftarrow \min(d[k][s_i], d[k-1][s_j] + c(s_j, s_i))$
    retourne $d[\#S - 1]$

Complexité de cet algorithme ?

$\mathcal{O}(np)$

Fonction *Calcul-itératif*(g, c, s_0)

  pour *chaque sommet* $s_i \in S$ **faire** $d[0][s_i] \leftarrow +\infty$;

  $d[0][s_0] \leftarrow 0$

  pour k *variant de* 1 *à* $\#S - 1$ **faire**

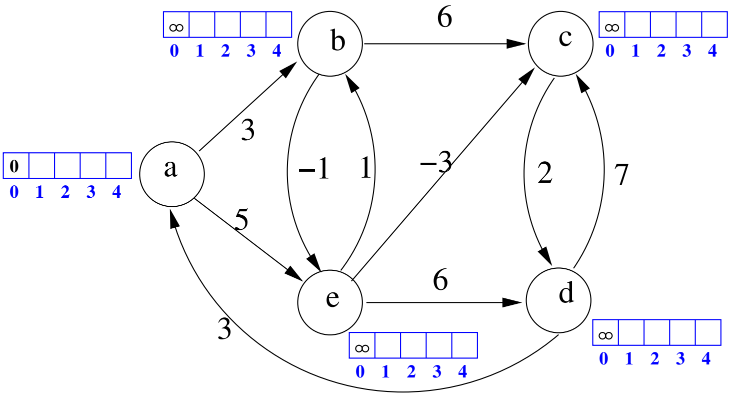
    pour *chaque sommet* $s_i \in S$ **faire**

      $d[k][s_i] \leftarrow d[k-1][s_i]$

      pour *chaque sommet* $s_j \in pred(s_i)$ **faire**

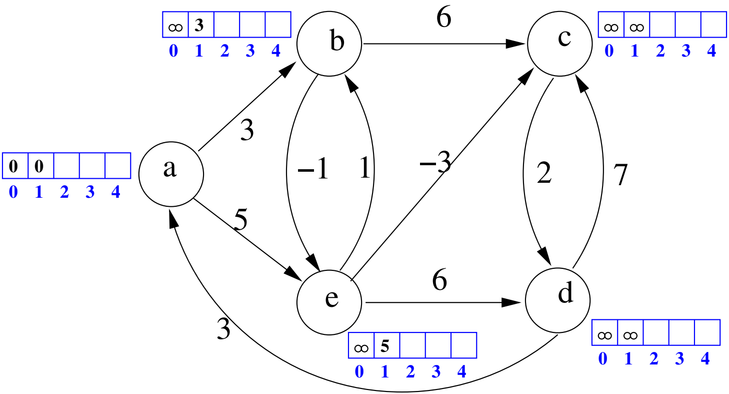
        $d[k][s_i] \leftarrow \min(d[k][s_i], d[k-1][s_j] + c(s_j, s_i))$

  retourne $d[\#S - 1]$



Fonction *Calcul-itératif*(g, c, s_0)

- **pour** *chaque sommet* $s_i \in S$ **faire** $d[0][s_i] \leftarrow +\infty$;
- $d[0][s_0] \leftarrow 0$
- **pour** k *variant de* 1 *à* $\#S - 1$ **faire**
  - **pour** *chaque sommet* $s_i \in S$ **faire**
    - $d[k][s_i] \leftarrow d[k-1][s_i]$
    - **pour** *chaque sommet* $s_j \in pred(s_i)$ **faire**
      - $d[k][s_i] \leftarrow \min(d[k][s_i], d[k-1][s_j] + c(s_j, s_i))$
- **retourne** $d[\#S - 1]$



Fonction *Calcul-itératif(g, c, s_0)*

  pour *chaque sommet* $s_i \in S$ **faire** $d[0][s_i] \leftarrow +\infty$;

  $d[0][s_0] \leftarrow 0$

  pour *k variant de* 1 *à* $\#S - 1$ **faire**

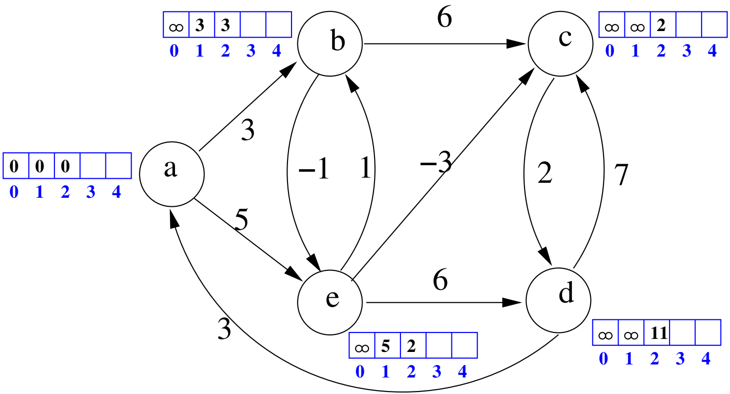
    pour *chaque sommet* $s_i \in S$ **faire**

      $d[k][s_i] \leftarrow d[k-1][s_i]$

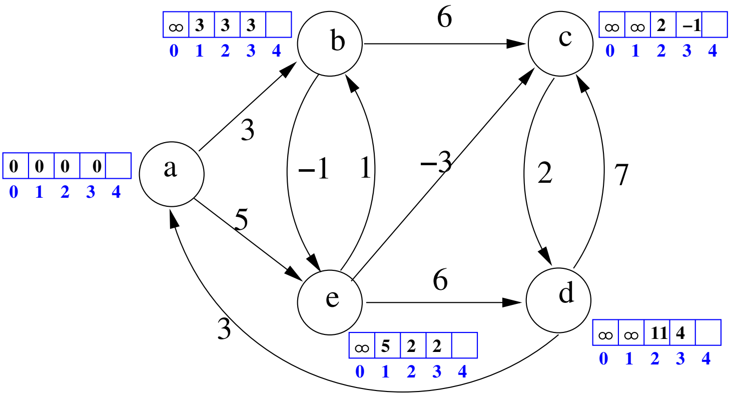
      pour *chaque sommet* $s_j \in pred(s_i)$ **faire**

        $d[k][s_i] \leftarrow \min(d[k][s_i], d[k-1][s_j] + c(s_j, s_i))$

  retourne $d[\#S - 1]$

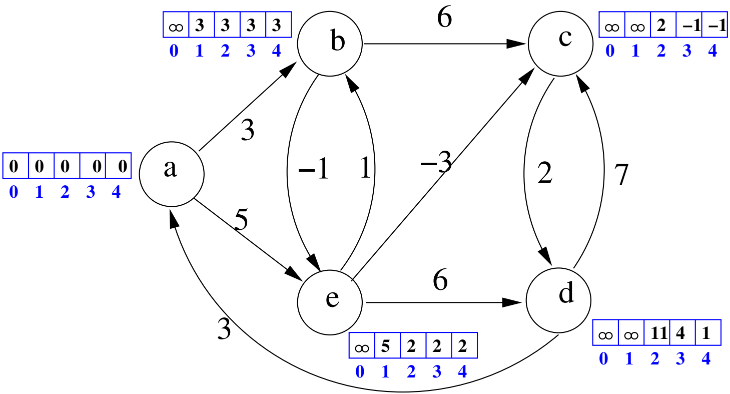


```
Fonction Calcul-itératif(g, c, s_0)
    pour chaque sommet s_i ∈ S faire d[0][s_i] ← +∞;
    d[0][s_0] ← 0
    pour k variant de 1 à #S − 1 faire
        pour chaque sommet s_i ∈ S faire
            d[k][s_i] ← d[k − 1][s_i]
            pour chaque sommet s_j ∈ pred(s_i) faire
                d[k][s_i] ← min(d[k][s_i], d[k − 1][s_j] + c(s_j, s_i))
    retourne d[#S − 1]
```



Fonction *Calcul-itératif(g, c, s_0)*

- **pour** *chaque sommet* $s_i \in S$ **faire** $d[0][s_i] \leftarrow +\infty$;
- $d[0][s_0] \leftarrow 0$
- **pour** *k variant de* 1 *à* $\#S - 1$ **faire**
  - **pour** *chaque sommet* $s_i \in S$ **faire**
    - $d[k][s_i] \leftarrow d[k-1][s_i]$
    - **pour** *chaque sommet* $s_j \in pred(s_i)$ **faire**
      - $d[k][s_i] \leftarrow \min(d[k][s_i], d[k-1][s_j] + c(s_j, s_i))$
- **retourne** $d[\#S - 1]$



```
1 Fonction Calcul-itératif(g, c, s_0)
2     pour chaque sommet s_i ∈ S faire d[0][s_i] ← +∞;
3     d[0][s_0] ← 0
4     pour k variant de 1 à #S − 1 faire
5         pour chaque sommet s_i ∈ S faire
6             d[k][s_i] ← d[k − 1][s_i]
7             pour chaque sommet s_j ∈ pred(s_i) faire
8                 d[k][s_i] ← min(d[k][s_i], d[k − 1][s_j] + c(s_j, s_i))
9     retourne d[#S − 1]
```

Détecter les circuits absorbants :

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

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* $s_i \in S$ **faire** $d[0][s_i] \leftarrow +\infty$;
3     $d[0][s_0] \leftarrow 0$
4   **pour** k *variant de* 1 *à* $\#S - 1$ **faire**
5   **pour** *chaque sommet* $s_i \in S$ **faire**
6             $d[k][s_i] \leftarrow d[k-1][s_i]$
7   **pour** *chaque sommet* $s_j \in pred(s_i)$ **faire**
8                 $d[k][s_i] \leftarrow \min(d[k][s_i], d[k-1][s_j] + c(s_j, s_i))$
9   **retourne** $d[\#S - 1]$

Détecter les circuits absorbants :

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

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* $s_i \in S$ **faire**
3         $d[s_i] \leftarrow +\infty$
4         $\pi[s_i] \leftarrow null$
5     $d[s_0] \leftarrow 0$
6   **pour** k *variant de* 1 *à* $\#S - 1$ **faire**
7   **pour** *chaque sommet* $s_i \in S$ **faire**
8   **pour** *chaque sommet* $s_j \in pred(s_i)$ **faire**
9   relacher$((s_i, s_j), \pi, d)$
10   **retourne** π *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 (s_i, s_j) pour lesquels $d[s_j]$ a été modifié

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

1 **Fonction** *Bellman-Ford*(g, c, s_0)
2   **pour** *chaque sommet* $s_i \in S$ **faire**
3         $d[s_i] \leftarrow +\infty$
4         $\pi[s_i] \leftarrow null$
5     $d[s_0] \leftarrow 0$
6   **pour** k *variant de* 1 *à* $\#S - 1$ **faire**
7   **pour** *chaque sommet* $s_i \in S$ **faire**
8   **pour** *chaque sommet* $s_j \in pred(s_i)$ **faire**
9   relacher$((s_i, s_j), \pi, d)$
10   **retourne** π *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 (s_i, s_j) pour lesquels $d[s_i]$ a été modifié

1 Introduction

2 Définitions

3 Structures de données pour représenter un graphe

4 Parcours de graphes

5 Plus courts chemins
- Définitions et propriétés
- 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 Quelques problèmes NP-difficiles sur les graphes

Spécification du pb de + court chemin pour tout couple de sommets

1 **Fonction** *PlusCourtsChemins(g, c)*

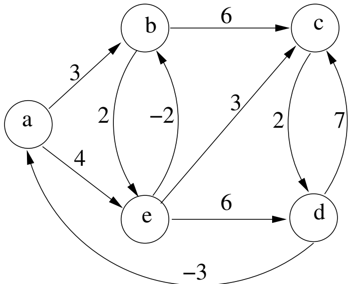
Entrée : Un graphe $g = (S, A)$ et une fct coût $c : A \rightarrow \mathbb{R}$

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

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

- Si $i = j$ ou $s_i \not\rightsquigarrow s_j$, alors $\pi[s_i][s_j] = null$
- Sinon : $\pi[s_i][s_j] =$ préd. de s_j dans un + court chemin de s_i à s_j

Exemple :



Résolution par programmation dynamique

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

Pour tout $k \in [0, |S|]$ et pour tout sommet s_i : sous-problème $d(k, s_i, s_j)$
$\rightsquigarrow$ longueur du + court chemin de s_i jusque s_j **en n'utilisant que les sommets** $\{s_1, \ldots, s_k\}$

Etape 2 : Equations de Bellman

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

Etape 3 : Analyse des performances :

- Combien de sous-problèmes différents pour un graphe ayant 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* $(s_i, s_j) \in S \times S$ **faire**
3   **si** $(s_i, s_j) \in A$ **alors** $d[0][s_i][s_j] \leftarrow c[s_i][s_j]$;
4   **sinon** $d[0][s_i][s_j] \leftarrow +\infty$;
5   **pour** *k variant de 1 à n* **faire**
6   **pour** *chaque couple de sommets* $(s_i, s_j) \in S \times S$ **faire**
7             $d[k][s_i][s_j] \leftarrow \min(d[k-1][s_i][s_j], d[k-1][s_i][s_k] + d[k-1][s_k][s_j])$
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
  - $\pi[k][s_i][s_j] = \pi[k-1][s_i][s_j]$ ou $\pi[k-1][s_k][s_j]$ selon le min (ligne 7)
- $\pi[n]$ est la matrice de liaisons finale

Algorithme de Floyd-Warshall

1 **Fonction** *Floyd-Warshall(g, c)*
2   **pour** *chaque couple de sommets* $(s_i, s_j) \in S \times S$ **faire**
3   **si** $(s_i, s_j) \in A$ **alors** $d[0][s_i][s_j] \leftarrow c[s_i][s_j]$;
4   **sinon** $d[0][s_i][s_j] \leftarrow +\infty$;
5   **pour** *k variant de 1 à n* **faire**
6   **pour** *chaque couple de sommets* $(s_i, s_j) \in S \times S$ **faire**
7             $d[k][s_i][s_j] \leftarrow \min(d[k-1][s_i][s_j], d[k-1][s_i][s_k] + d[k-1][s_k][s_j])$
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
  - $\pi[k][s_i][s_j] = \pi[k-1][s_i][s_j]$ ou $\pi[k-1][s_k][s_j]$ selon le min (ligne 7)
- $\pi[n]$ est la matrice de liaisons finale

Algorithme de Floyd-Warshall

1 **Fonction** *Floyd-Warshall*(g, c)
2   **pour** *chaque couple de sommets* $(s_i, s_j) \in S \times S$ **faire**
3   **si** $(s_i, s_j) \in A$ **alors** $d[0][s_i][s_j] \leftarrow c[s_i][s_j]$;
4   **sinon** $d[0][s_i][s_j] \leftarrow +\infty$;
5   **pour** *k variant de 1 à n* **faire**
6   **pour** *chaque couple de sommets* $(s_i, s_j) \in S \times S$ **faire**
7             $d[k][s_i][s_j] \leftarrow \min(d[k-1][s_i][s_j], d[k-1][s_i][s_k] + d[k-1][s_k][s_j])$
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
  - $\pi[k][s_i][s_j] = \pi[k-1][s_i][s_j]$ ou $\pi[k-1][s_k][s_j]$ selon le min (ligne 7)
- $\pi[n]$ est la matrice de liaisons finale

1. Introduction
2. Définitions
3. Structures de données pour représenter un graphe
4. Parcours de graphes
5. Plus courts chemins
   - Définitions et propriétés
   - 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. Quelques problèmes NP-difficiles sur les graphes

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 proba des arcs de $\pi < b$ 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 ?

Condition commune à tous les algorithmes vus :

Optimalité des sous-structures
$\rightsquigarrow$ Tout sous-chemin d'un chemin optimal est optimal

Cette propriété 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 la cond. d'optimalité des ss-structures n'est pas vérifiée ?

Le problème devient $\mathcal{NP}$-difficile
$\rightsquigarrow$ 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 $cout(s_0 \rightsquigarrow s_i \rightarrow s_j) \geq cout(s_0 \rightsquigarrow s_i)$
- Si on cherche un max, alors $cout(s_0 \rightsquigarrow s_i \rightarrow s_j) \leq cout(s_0 \rightsquigarrow s_i)$

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

Exemple : Chemin le plus long

- OK si $cout(s_0 \leadsto s_i \rightarrow s_j) = cout(s_0 \leadsto s_i) + c(s_i, s_j) \leq cout(s_0 \leadsto s_i)$
  $\leadsto$ Il faut que $c(s_i, s_j)$ soit **négatif** pour tout arc (s_i, s_j)
- Initialiser d à $-\infty$, sauf pour $d[s_0]$ qui est initialisé à 0
- Procédure de relâchement :

```
Proc relacher((s_i, s_j), π, d)
    si d[s_j] > d[s_i] + c(s_i, s_j) alors
        d[s_j] ← d[s_i] + c(s_i, s_j)
        π[s_j] ← 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 $s_0 \rightsquigarrow s_i$ ne peut que dégrader son coût :

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

Exemple : Chemin le plus probable

- OK si $cout(s_0 \rightsquigarrow s_i \rightarrow s_j) = cout(s_0 \rightsquigarrow s_i) * p(s_i, s_j) \leq cout(s_0 \rightsquigarrow s_i)$
  $\rightsquigarrow$ Toujours vrai car $0 \leq p(s_i, s_j) \leq 1$, pour tout arc (s_i, s_j)
- Initialiser d à 0, sauf pour $d[s_0]$ qui est initialisé à 1
- Procédure de relâchement :

```
Proc relacher((s_i, s_j), π, d)
    si d[s_j] < d[s_i] * p(s_i, s_j) alors
        d[s_j] ← d[s_i] * p(s_i, s_j)
        π[s_j] ← 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 quand le coût d’un chemin est égal au coût de son plus petit arc

- Initialiser d à $-\infty$, sauf pour $d[s_0]$ qui est initialisé à $+\infty$
- Procédure de relâchement :

Proc *relacher*$((s_i, s_j), \pi, d)$
    si $d[s_j] < min(d[s_i], cout(s_i, s_j))$ **alors**
        $d[s_j] \leftarrow min(d[s_i], cout(s_i, s_j))$
        $\pi[s_j] \leftarrow s_i$

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 quand le coût d’un chemin est égal au coût de son plus petit arc

- Initialiser d à $-\infty$, sauf pour $d[s_0]$ qui est initialisé à $+\infty$
- Procédure de relâchement :

```
Proc relacher((s_i, s_j), π, d)
    si d[s_j] < min(d[s_i], cout(s_i, s_j)) alors
        d[s_j] ← min(d[s_i], cout(s_i, s_j))
        π[s_j] ← s_i
```

Aurait-on pu utiliser Dijkstra dans ce cas ?

Extensions de Bellman-Ford et Floyd-Warshall

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

Pas de circuit absorbant (et optimalité des sous-structures, évidemment)

Exemples :

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

Et si on recherche un plus court chemin pour aller de s_0 vers un unique sommet destination s_i ?

En théorie :

Pas d'algorithme plus efficace que TopoDAG (si pas de circuits) ou Dijkstra (si tous les coûts sont positifs)

En pratique :

Possibilité d'améliorer les performances en utilisant des bornes
(par exemple : distance euclidienne)
$\leadsto$ Algorithme A* étudié dans la deuxième partie du cours !

1. Introduction
2. Définitions
3. Structures de données pour représenter un graphe
4. Parcours de graphes
5. Plus courts chemins
6. Problèmes de planification
7. Quelques problèmes NP-difficiles sur les graphes

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* : $E \rightarrow \mathcal{P}(A)$
  $\rightsquigarrow$ $actions(e)$ = ensemble des actions pouvant être appliquées à l'état e
- Une fonction de transition $t : E \times A \rightarrow E$
  $\rightsquigarrow$ si $a \in actions(e)$ alors $t(e, a)$ = état obtenu quand on applique a sur e

Données en sortie :

- Un plan d'action = une séquence $< e_1, a_1, \ldots, e_n, a_n, e_{n+1} >$ telle que
  - $e_1 = e_{init}$
  - $e_{n+1} \in F$
  - $\forall i \in [1, n], a_i \in actions(e_i)$ et $e_{i+1} = t(e_i, a_i)$
- Variante : Trouver le meilleur plan (plus court, moins coûteux, etc)

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
- $actions(e) = \{x \;\; op \;\; y | \{x, y\} \subseteq e, op \in \{+, -, *, /\}\}$
- $t(e, x \;\; op \;\; y) = e \setminus \{x, y\} \cup \{x \;\; op \;\; y\}$

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

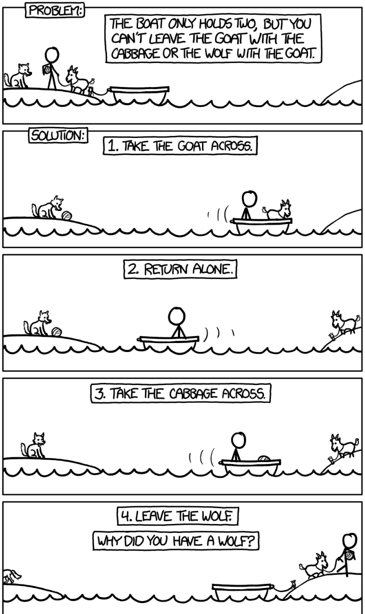
- $e_1 = \{1, 3, 4, 5, 7, 10, 25\}$ et $a_1 = 25 * 3$
- $e_2 = \{1, 4, 5, 7, 10, 75\}$ et $a_2 = 75 + 5$
- $e_3 = \{1, 4, 7, 10, 80\}$ et $a_3 = 80 * 4$
- $e_4 = \{1, 7, 10, 320\}$ et $a_4 = 320 + 1$
- $e_5 = \{7, 10, 321\}$

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

Objectif du problème :

Faire traverser une rivière à un groupe P en respectant des contraintes

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



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

Formalisation du problème :

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

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

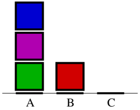
Exemple 3 : Le monde des blocs

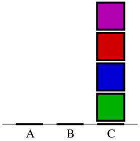
Objectif du problème :

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
- $actions(c) = \{(p_1, p_2) | p_1$ est une pile contenant au - 1 bloc$\}$
- $t((p_1, p_2), c) =$ faire passer le bloc au sommet de p_1 sur p_2

Exemple de plan pour E_{init} = 

et $F = \{$ 

$\}$

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

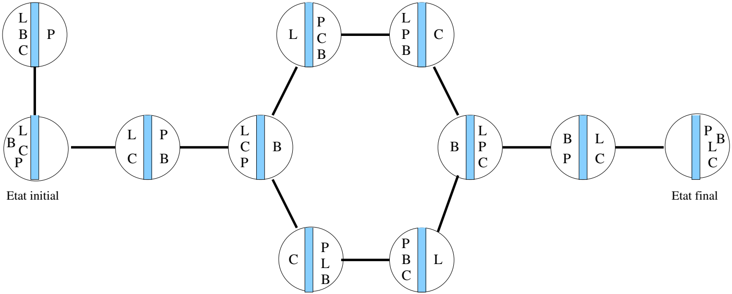
Modéliser un problème de planification à l'aide d'un graphe

Définition du graphe 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 pour la traversée de rivière



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

Quels algorithmes utiliser pour...

- Déterminer s'il existe un plan d'actions ?
- Chercher le plan d'action le plus court ?
- Chercher le plan d'action le moins coûteux (si chaque action a un coût) ?
- Compter le nombre de plans d'actions différents possibles ?

Quelles sont les complexités de ces algorithmes ?

- Par rapport au graphe Etats-Transitions ?
- Par rapport au problème de planification ?

On peut faire mieux !

$\leadsto$ cf deuxième partie du cours...

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

Quels algorithmes utiliser pour...

- Déterminer s'il existe un plan d'actions ?
- Chercher le plan d'action le plus court ?
- Chercher le plan d'action le moins coûteux (si chaque action a un coût) ?
- Compter le nombre de plans d'actions différents possibles ?

Quelles sont les complexités de ces algorithmes ?

- Par rapport au graphe Etats-Transitions ?
- Par rapport au problème de planification ?

On peut faire mieux !

$\rightsquigarrow$ cf deuxième partie du cours...

1. Introduction
2. Définitions
3. Structures de données pour représenter un graphe
4. Parcours de graphes
5. Plus courts chemins
6. Problèmes de planification
7. Quelques problèmes NP-difficiles sur les graphes
   - Classes de complexité
   - Recherche de cliques
   - Coloriage de graphes

Problèmes, instances et algorithmes (rappels)

Spécification d'un problème :

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

Instance d'un problème :

Valuation des paramètres en entrée satisfaisant les préconditions

Algorithme pour un problème P :

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

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

Spécification du problème :

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

Exemples d'instances :

- Entrées : $e = 8$ et $tab =$

4	4	7	8	10	11	12

  $\leadsto$ Sortie : $i = 3$

- Entrées : $e = 9$ et $tab =$

4	4	7	8	10	11	12

  $\leadsto$ Sortie : $i = 7$

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

Spécification du problème :

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

Exemple d'instance

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

Complexité d'un algorithme (rappel)

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

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

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

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

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

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

Complexité des problèmes de décision

Problèmes de décision :

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

Exemple : Description du problème de décision *Recherche*

- Entrées = un tableau *tab* contenant *n* entiers et un entier *e*
- Question = Existe-t-il un élément de *tab* qui soit égal à *e* ?

Complexité d'un problème X :

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

La classe $\mathcal{P}$

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

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

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

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

La classe $\mathcal{NP}$

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

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

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

- Description du problème SAT (rappel) :
  - Entrées = une formule bool. F portant sur un ens. X de n variables
  - Question = Existe-t-il une valuation des var. de X qui satisfait F ?
- Algorithme pour une machine non déterministe :

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

Réponse = vrai si au moins une branche a retourné vrai
Certificat = une valuation $V : X \rightarrow \{vrai, faux\}$ qui satisfait F

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

- $\mathcal{P} \subseteq \mathcal{NP}$
- Conjecture : $\mathcal{P} \neq \mathcal{NP}$
  1 million de dollars à gagner (cf www.claymath.org/millennium-problems)

Problèmes $\mathcal{NP}$-complets :

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

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

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

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 A_r de réduction de P_1 vers P_2
- un algorithme A_2 résolvant le problème P_2,

Algorithme pour résoudre une instance I_1 de P_1 = $A_2(A_r(I_1))$
$\Rightarrow$ Complexité de $P_1 \leq$ Complexité de A_2 et A_r
$\Rightarrow$ Si P_1 est $\mathcal{NP}$-complet, et A_r et A_2 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 $\in \mathcal{NP}$:

$\rightsquigarrow$ Certificat ?

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

$\rightsquigarrow$ 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

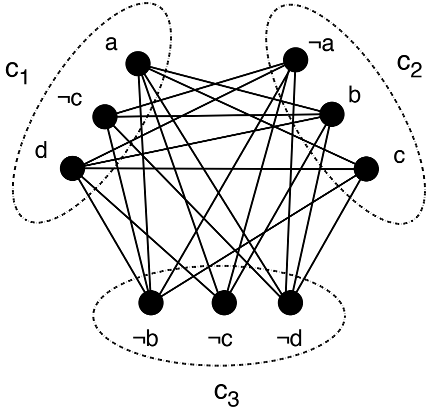
Solution

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

- S associe un sommet à chaque littéral de chaque clause de F
  $\rightsquigarrow$ $c(u)$ 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 $\mathcal{NP}$-difficiles

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

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

Exemple : Problème de la clique exacte

- Entrées : un graphe 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 $\mathcal{NP}$?

Complexité des problèmes d'optimisation :

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

Problèmes indécidables

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

Exemple 1 : Problème de l'arrêt

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

Exemple 2 : Problème de Post

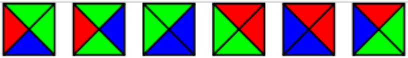
- Entrée : 2 listes finies $\alpha_1, \alpha_2, \ldots, \alpha_n$ et $\beta_1, \beta_2, \ldots, \beta_n$ de mots
- Question : $\exists k$ indices $i_1, i_2, \ldots, i_k$ tq $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_n} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_n}$?
- Exemple d'instance : Entrée =

α_1	α_2	α_3
a	ab	bba

β_1	β_2	β_3
baa	aa	bb

Exemple 3 : Problème de pavage

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



1. Introduction
2. Définitions
3. Structures de données pour représenter un graphe
4. Parcours de graphes
5. Plus courts chemins
6. Problèmes de planification
7. Quelques problèmes NP-difficiles sur les graphes
   - Classes de complexité
   - Recherche de cliques
   - Coloriage de graphes

Enumération de toutes les cliques d'un graphe

Fonction *enumClique*(g, c)
- **Entrée** : Un graphe $g = (S, A)$ et un ens. de sommets $c \subseteq S$
- **Précond.** : c est une clique de g
- **Postcond.** : Affiche toute clique c' de c telle que $c \subseteq c'$

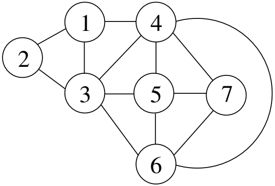
```
Afficher c
cand ← {s_i ∈ S | ∀s_j ∈ c, {s_i, s_j} ∈ A et s_i > s_j}
pour chaque sommet s_i ∈ cand faire
    enumClique(g, c ∪ {s_i})
```

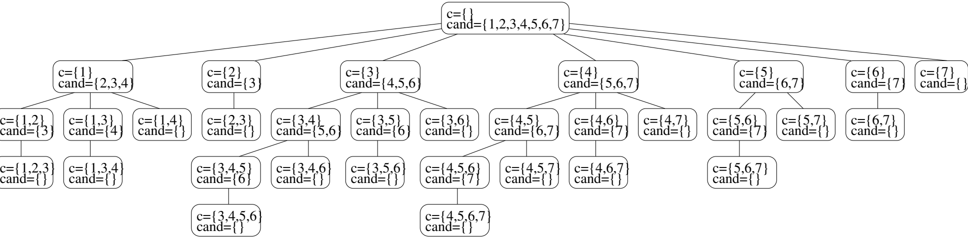
Arbre de recherche associé à une exécution de *enumClique* :

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

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

Exemple d’arbre de recherche :





Fonction *enumClique*(g, c)

Entrée : Un graphe $g = (S, A)$ et un ens. de sommets $c \subseteq S$

Précond. : c est une clique de g

Postcond. : Affiche toute clique c' de g telle que $c \subseteq c'$

Affiche c

$cand \leftarrow \{s_i \in S \mid \forall s_j \in c, \{s_i, s_j\} \in A \text{ et } s_i > s_j\}$

pour *chaque sommet* $s_i \in cand$ **faire**

    enumClique$(g, c \cup \{s_i\})$

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

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

$\leadsto$ Complexité = $\mathcal{O}(nx)$ (*incremental polynomial time*)

Recherche d'une clique d'ordre k

Fonction *chercheClique*(g, c, k)

  Entrée : graphe $g = (S, A)$, $c \subseteq S$ et entier k

  Précond. : c est une clique de taille inférieure ou égale à k

  Postcond. : Retourne vrai si $\exists$ une clique c' de g tq $\#c' = k$ et $c \subseteq c'$; faux sinon

  si $\#c = k$ **alors retourne** *vrai*;

  $cand \leftarrow \{s_i \in S \mid \forall s_j \in c, \{s_i, s_j\} \in A \wedge s_i > s_j\}$

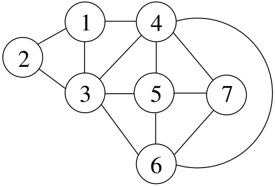
  si $\#c + \#cand < k$ **alors retourne** *faux*;

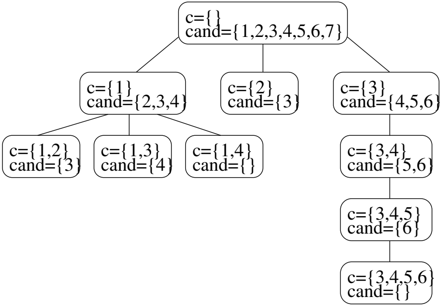
  pour *chaque sommet* $s \in cand$ **faire**

    si *chercheClique*$(g, c \cup \{s\}, k)$ **alors retourne** *vrai*;

  retourne *faux*

Arbre de recherche pour $k = 4$:

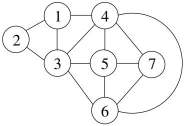


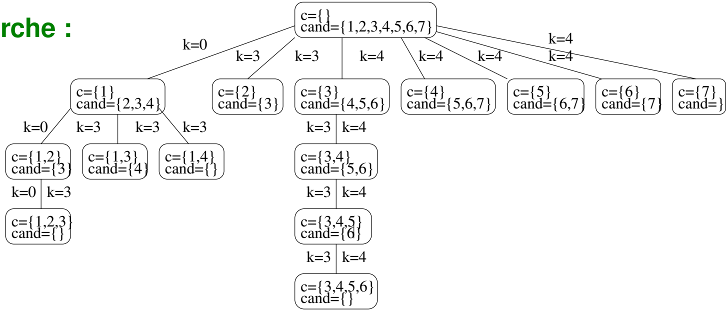


Recherche d'une clique maximum

```
Fonction chercheCliqueMax(g, c, k)
    Entrée    : graphe g = (S, A), c ⊆ S et entier k
    Précond.  : c est une clique
    Postcond. : ret. max(k, k′) où k′ = taille de la + grande clique de g contenant c
    cand ← {s_i ∈ S | ∀s_j ∈ c, {s_i, s_j} ∈ A ∧ s_i > s_j}
    si cand = ∅ alors retourne max(#c, k);
    pour chaque sommet s ∈ cand et tant que #c + #cand > k faire
        k ←chercheCliqueMax(g, c ∪ {s}, k)
    retourne k
```

Arbre de recherche :

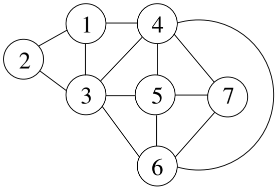




Construction gloutonne d'une clique

Fonction *chercheCliqueGlouton*(g)
    Entrée : Un graphe $g = (S, A)$
    Postcond. : retourne une clique de g
    $cand \leftarrow S$
    $c \leftarrow \emptyset$
    tant que $cand \neq \emptyset$ **faire**
        Soit s_i le sommet de *cand* maximisant $\#(cand \cap adj(s_i))$
        $c \leftarrow c \cup \{s_i\}$
        $cand \leftarrow cand \cap adj(s_i)$
    retourne c

Exercice :



Construction gloutonne d'une clique

Fonction *chercheCliqueGlouton*(g)

Entrée : Un graphe $g = (S, A)$

Postcond. : retourne une clique de g

$cand \leftarrow S$

$c \leftarrow \emptyset$

tant que $cand \neq \emptyset$ **faire**

  Soit s_i le sommet de *cand* maximisant $\#(cand \cap adj(s_i))$

  $c \leftarrow c \cup \{s_i\}$

  $cand \leftarrow cand \cap adj(s_i)$

retourne c

Complexité ?

1. **Introduction**

2. **Définitions**

3. **Structures de données pour représenter un graphe**

4. **Parcours de graphes**

5. **Plus courts chemins**

6. **Problèmes de planification**

7. **Quelques problèmes NP-difficiles sur les graphes**
   - Classes de complexité
   - Recherche de cliques
   - Coloriage de graphes

Coloriage de graphes

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

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

Complexité :

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

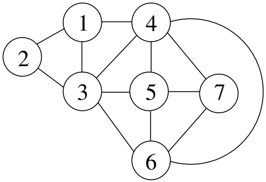
Relation entre coloriage et cliques :

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

Algorithme glouton de Brélaz (DSATUR)

Fonction *brélaz*(g)
  Postcond. **:** retourne une borne supérieure de $\chi(g)$
  $borne_\chi \leftarrow 0$
  tant que *tous les sommets ne sont pas coloriés* **faire**
    Choisir un sommet s_i non colorié tel que :
    - s_i = sommet ayant le plus de voisins coloriés avec des valeurs différentes
    - en cas d'ex æquo, s_i = sommet ayant le plus de voisins non coloriés
    si $\forall k \in [1, borne_\chi], \exists s_j \in adj(s_i)$ *tel que* s_j *est colorié avec* k **alors**
      $borne_\chi \leftarrow borne_\chi + 1$
    $k \leftarrow$ plus petite couleur $\in [1, borne_\chi]$ non prise par un voisin de s_i
    Colorier s_i avec k
  retourne $borne_\chi$

Exercice :



Algorithme glouton de Brélaz (DSATUR)

Fonction *brélaz*(g)

  Postcond. : retourne une borne supérieure de $\chi(g)$

  $borne_\chi \leftarrow 0$

  tant que *tous les sommets ne sont pas coloriés* **faire**

    Choisir un sommet s_i non colorié tel que :

    - $s_i =$ sommet ayant le plus de voisins coloriés avec des valeurs différentes

    - en cas d'ex æquo, $s_i =$ sommet ayant le plus de voisins non coloriés

    si $\forall k \in [1, borne_\chi], \exists s_j \in adj(s_i)$ *tel que s_j est colorié avec k* **alors**

      $borne_\chi \leftarrow borne_\chi + 1$

    $k \leftarrow$ plus petite couleur $\in [1, borne_\chi]$ non prise par un voisin de s_i

    Colorier s_i avec k

  retourne $borne_\chi$

Complexité ?

Au delà des cliques et du coloriage

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

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

Ces problèmes sont rencontrés dans de très nombreuses applications

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

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

C'est ce que vous verrez dans la suite du cours (entre autres...)