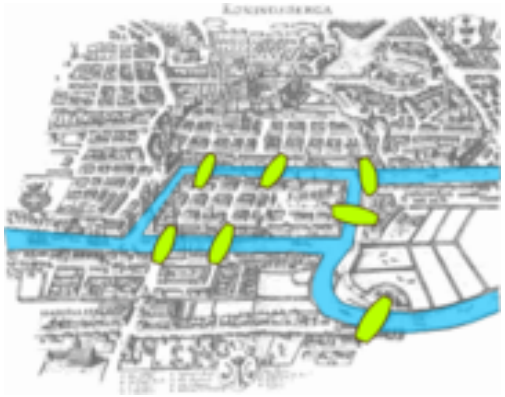
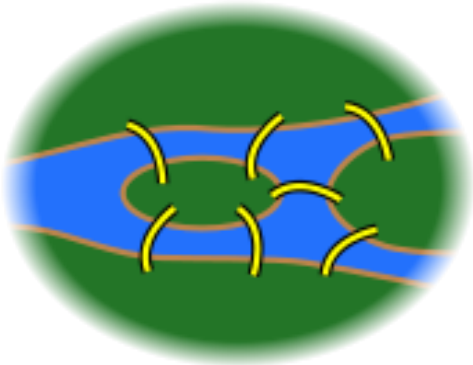
Algorithmique Avancée et Programmation en C
Graphes

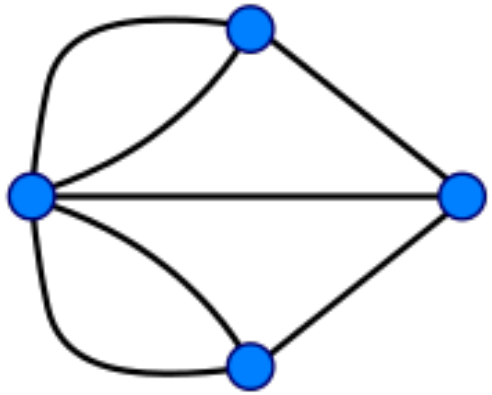
Rym Guibadj

LISIC, EILCO

Problème des sept ponts de Königsberg

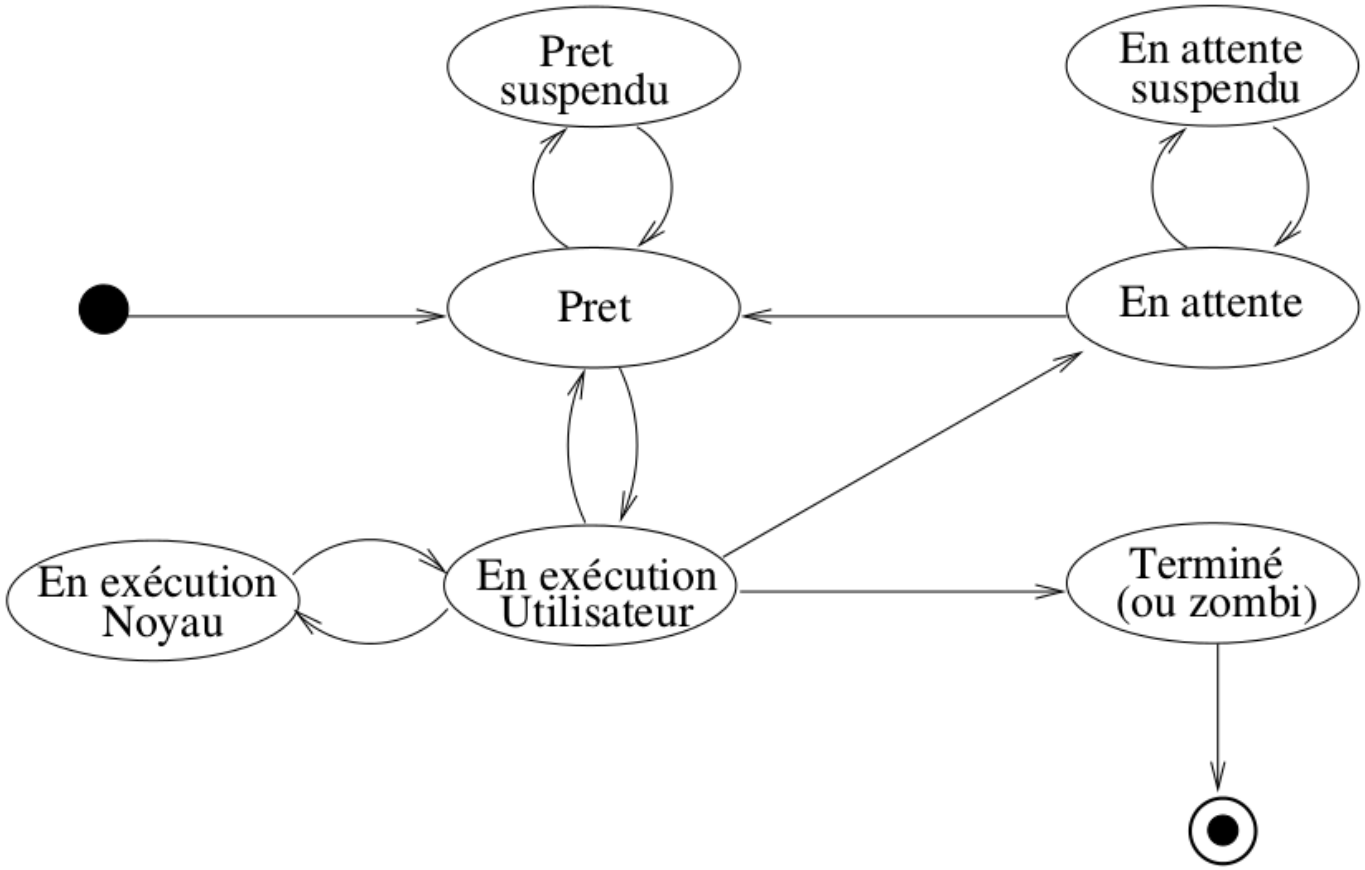






Existe il une promenade dans les rues de Königsberg permettant, à partir d'un point de départ au choix, qui permet de passer une et une seule fois par chaque pont, et de revenir à son point de départ ?

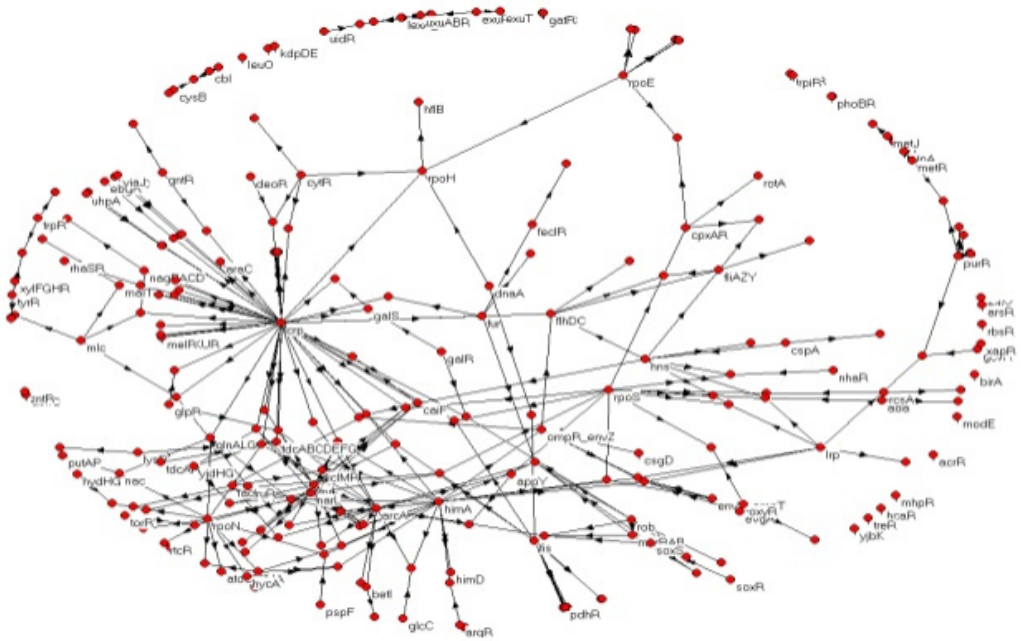
Exemples de modélisation par les graphes



Digramme UML

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

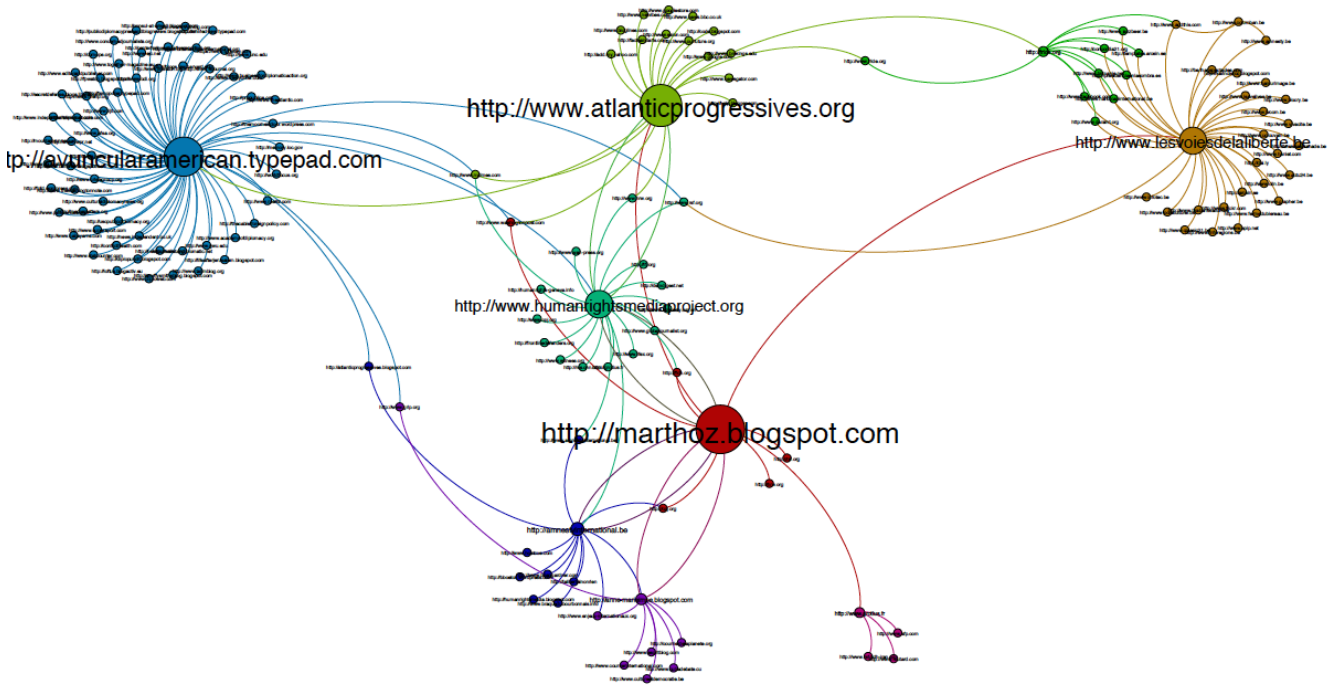
Exemples de modélisation par les graphes



Réseaux de régulation génétique

- Sommets = gènes
- Arcs = influence entre gènes

Exemples de modélisation par les graphes



Web et Réseaux sociaux

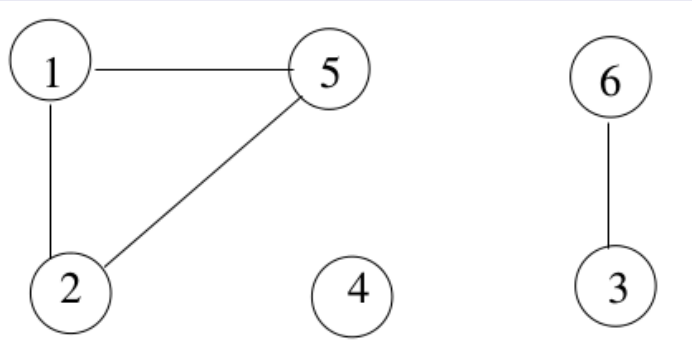
- Sommets = URL de blogs
- Arcs = Hyper-liens

Graphes non orientés

Définition

$G = (S, A)$ est non orienté si $\forall (s_i, s_j) \in S \times S, (s_i, s_j) \in A \Leftrightarrow (s_j, s_i) \in A$. La relation binaire définie par A est **symétrique**.

Exemple



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

Terminologie :

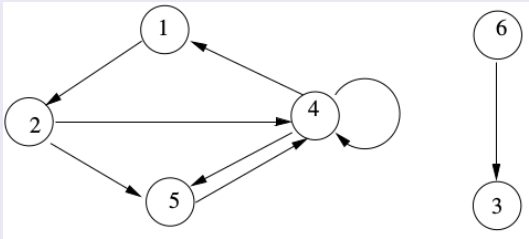
- Les éléments de A sont appelés **arêtes**
- s_i est **adjacent** à s_j si $(s_i, s_j) \in A : adj(s_i) = s_j | (s_i, s_j) \in A$
- **degré** d'un sommet = nombre de sommets adjacents : $d(s_i) = |adj(s_i)|$
- Graphe **complet** si $A = (s_i, s_j) \in S \times S | s_i \neq s_j$

Graphes orientés

Définition

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

Exemple



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

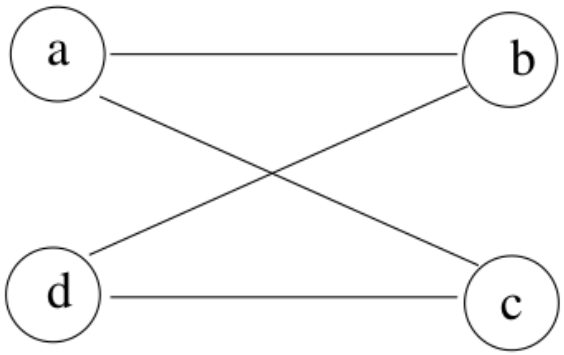
Terminologie :

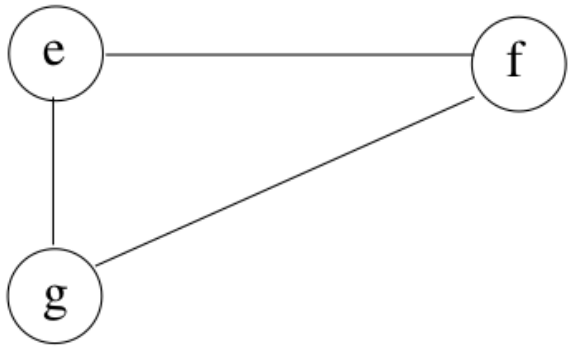
- Les éléments de A sont appelés **arcs**
- s_j est **successeur** de s_i si $(s_i, s_j) \in A : succ(s_i) = s_j | (s_i, s_j) \in A$
- s_j est **prédécesseur** de s_i si $(s_j, s_i) \in A : pred(s_i) = s_j | (s_j, s_i) \in A$
- **demi-degré extérieur** = nombre de successeurs :$d^+(s_i) = |succ(s_i)|$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

Cheminements et connexités

Définition 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

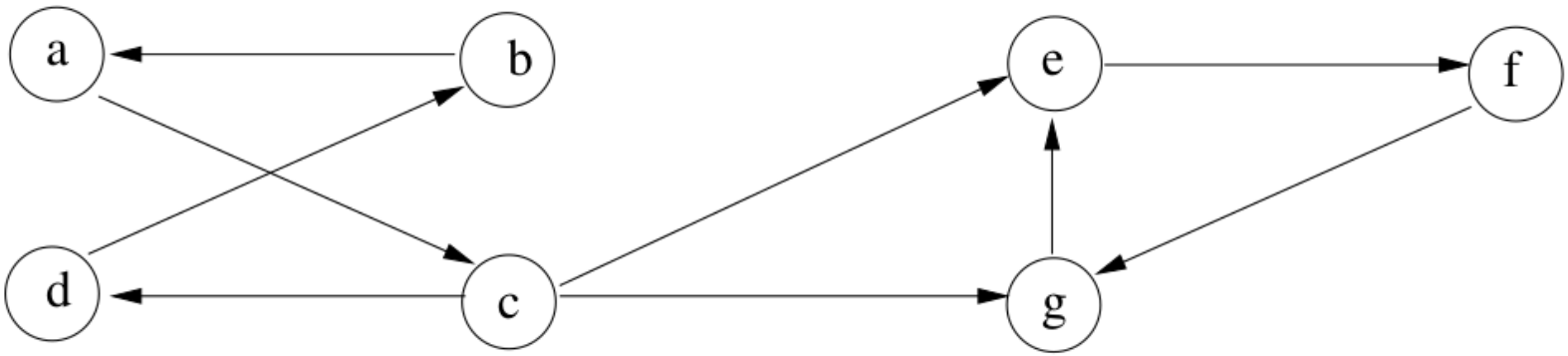




Cheminements et connexités

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

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

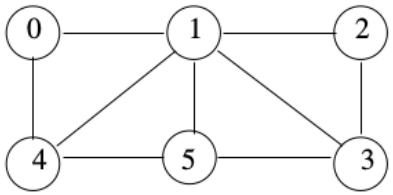


Matrice d'adjacence

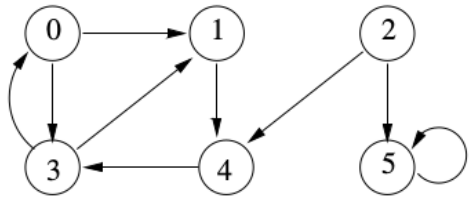
Définition : matrice d'adjacence d'un graphe $G = (S, A)$

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

Exemples :



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



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

Matrice d'adjacence

Complexité en mémoire

$O(n^2)$ avec n = nombre de sommets de g

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

$O(1)$

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

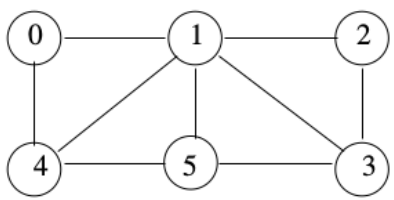
$O(n)$ avec n le nombre de sommets de g

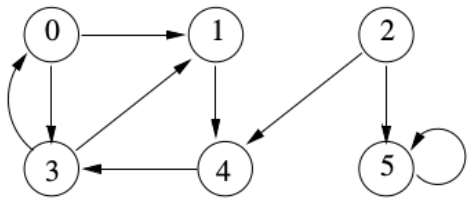
Listes d'adjacence

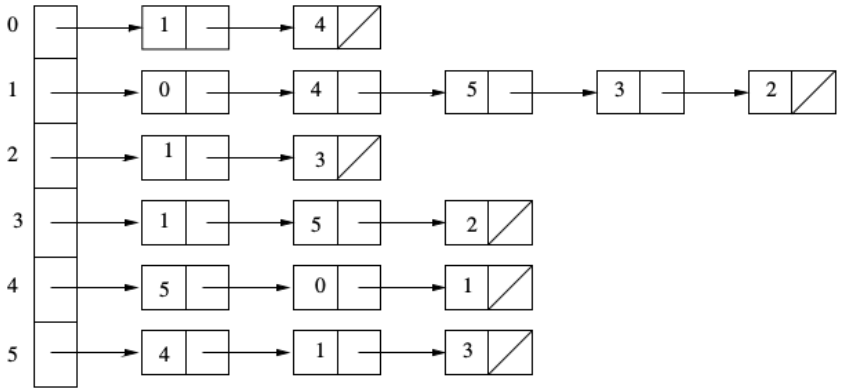
Définition : listes d'adjacence d'un graphe $G = (S, A)$

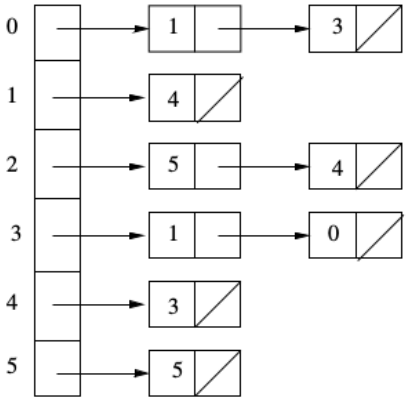
Tableau *succ* tel que $succ[s_i]$ = liste des successeurs de s_i

Exemples :









Listes d'adjacence

Complexité en mémoire

$O(n+p)$ 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 :

$O(d(s_i))$

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

$O(d(s_i))$

Généralités

Parcourir un graphe :

Visiter tous les sommets accessibles depuis un sommet de départ donné

Comment parcourir un graphe ?

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

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

Mise en oeuvre : 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

Fonction parcours (g, s_0) :

- **Entrée** : Un graphe g et un sommet s_0 de g
- **Sortie** : Retourne l'arborescence π du parcours de g à partir de s_0

Arborescence associée à un parcours :

- s_i est le père de s_j si c'est s_i qui a colorié s_j en gris
- s_i est racine si $s_i = s_0$ ou si pas de chemin de s_0 jusque s_i
- Mémorisation dans un tableau π tel que $\pi[s_i] = \mathit{null}$ si s_i est racine, et $\pi[s_i] =$ père de s_i sinon

Exemples :

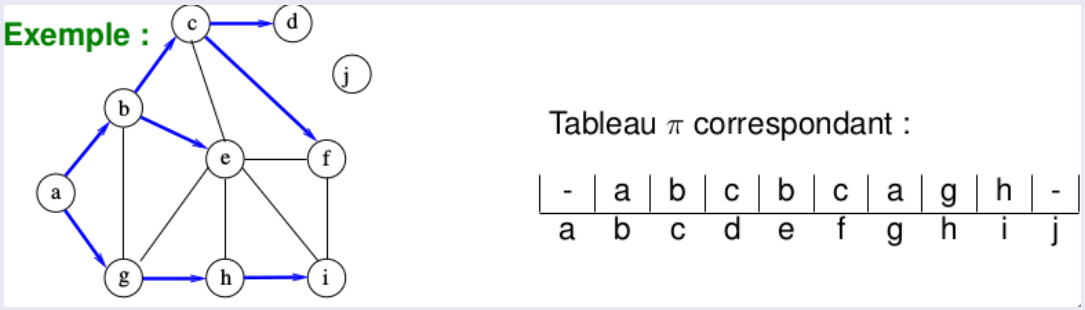


Tableau π correspondant :

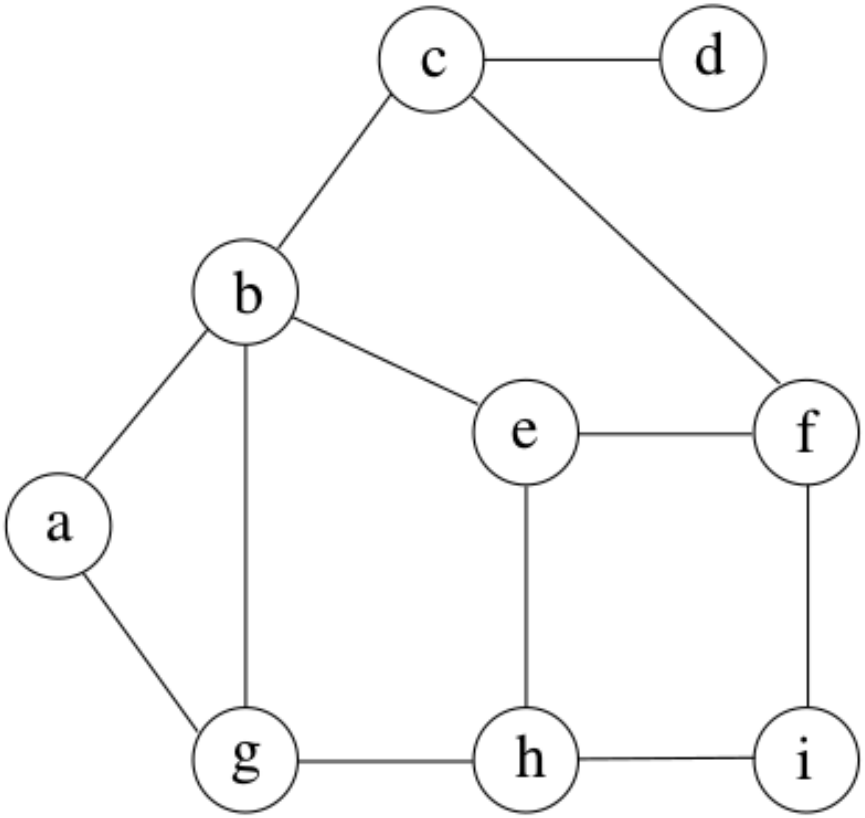
-	a	b	c	b	c	a	g	h	-
a	b	c	d	e	f	g	h	i	j

Parcours en largeur (Breadth First Search / BFS)

Algorithm 1: BFS(g, s_0)

foreach $s_i \in S$ **do**
    Colorier s_i en blanc ;
$f = enfiler(f, s_0)$;
Colorier s_0 en gris ;
while *fileNonVide(f)* **do**
    $f = defiler(f, s_k)$;
    while $\exists s_i \in succ(s_k)$ *tq* s_i *soit blanc* **do**
        $f = enfiler(f, s_i)$;
        Colorier s_i en gris ;
    Colorier s_k en noir ;
    Afficher s_k ;

Parcours en largeur (Breadth First Search / BFS) : Exemple



Parcours en largeur (Breadth First Search / BFS) : Exemple

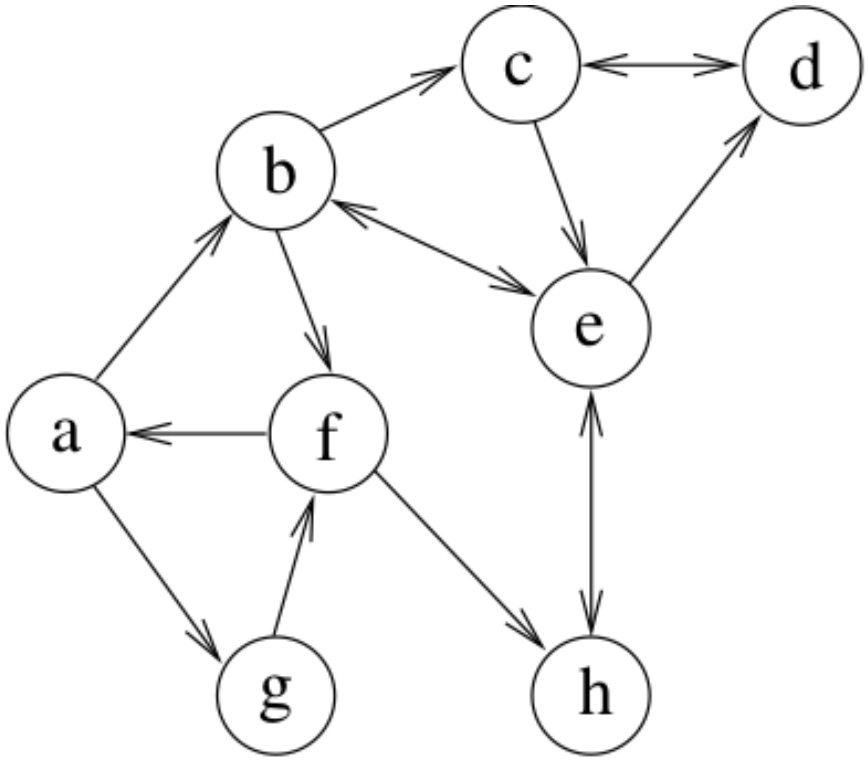
Contenu de la File	Sommet noir affiché
$\{a\}$	
$\{b, g\}$	a
$\{g, c, e\}$	a, b
$\{c, e, h\}$	a, b, g
$\{e, h, d\}$	a, b, g, c
$\{h, d, f\}$	a, b, g, c, e
$\{d, f, i\}$	a, b, g, c, e, h
$\{f, i\}$	a, b, g, c, e, h, d
$\{i\}$	a, b, g, c, e, h, d, f
$\{\}$	$a, b, g, c, e, h, d, f, i$

Parcours en profondeur (Depth First search / DFS)

Algorithm 2: DFS(g, s_0)

foreach $s_i \in S$ **do**
    $\pi[s_i] = null$;
    Colorier s_i en blanc ;
$pile = empiler(pile, s_0)$;
Colorier s_0 en gris ;
while *pileNonVide(pile)* **do**
    s_k = le dernier sommet empilé (en sommet de pile) ;
    if $\exists s_i \in succ(s_k)$ *tq* s_i *soit blanc* **then**
        $pile = empiler(pile, s_i)$;
        Colorier s_i en gris ;
        $\pi[s_i] = s_k$;
    else
        $pile = depiler(pile, s_k)$;
        Colorier s_k en noir ;

Parcours en profondeur (Depth First search / DFS)



Parcours en profondeur (Depth First search / DFS)

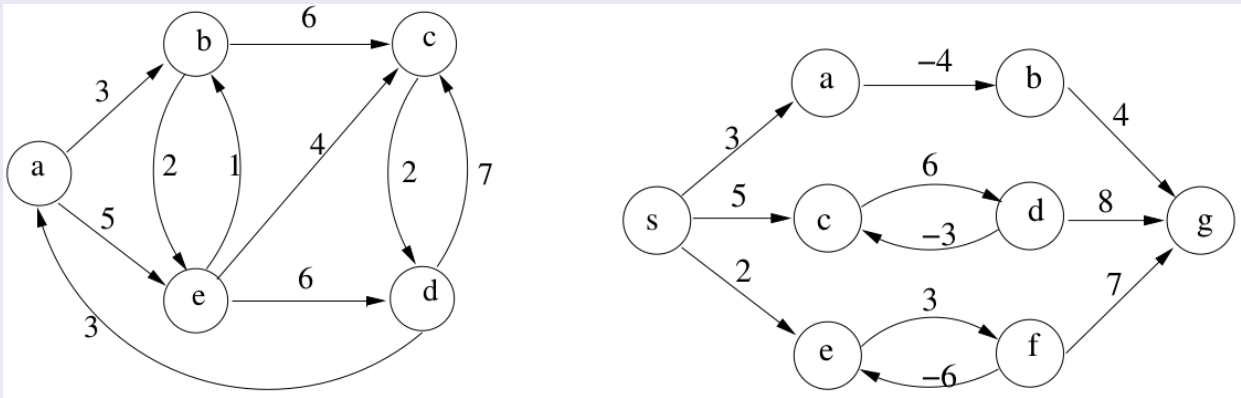
Pile	Sommet noir affiché
$\{a\}$	
$\{b, a\}$	
$\{c, b, a\}$	
$\{d, c, b, a\}$	
$\{c, b, a\}$	d
$\{e, c, b, a\}$	d
$\{h, e, c, b, a\}$	d
$\{e, c, b, a\}$	d, h
$\{c, b, a\}$	d, h, e
$\{b, a\}$	d, h, e, c
$\{f, b, a\}$	d, h, e, c
$\{b, a\}$	d, h, e, c, f
$\{a\}$	d, h, e, c, f, b
$\{g, a\}$	d, h, e, c, f, b
$\{a\}$	d, h, e, c, f, b, g
$\{\}$	d, h, e, c, f, b, g, a

Généralités

Définitions

- Soit $G = (S, A)$ un graphe (orienté ou non) et une fonction coût $c : A \rightarrow R$
- Coût d'un chemin $p =< s_0, s_1, s_2, ..., s_k >$: $c(p) = \sum_{i=1}^{k} c(s_{i-1}, s_i)$
- Coût d'un plus court chemin de s_i vers $s_j = \delta(s_i, s_j)$
  - $\delta(s_i, s_j) = +\infty$ si $\nexists$ chemin de s_i vers s_j
  - $\delta(s_i, s_j) = -\infty$ si $\exists$ circuit absorbant
  - $\delta(s_i, s_j) = min\{c(p)|p =$ chemin de s_i a $s_j\}$ sinon

Exemples



Principe d'optimalité d'une sous-structure

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

Si $p =< s_0, s_1, ..., s_k >$ est un plus court chemin alors
$\forall i, j$ tq $0 \leq i \leq j \leq k, p_{ij} =< s_i, s_{i+1}, ..., s_j >$ est un plus court chemin

Exploitation par des algorithmes gloutons (Dijkstra)

Si $s_i \rightsquigarrow s_j \rightarrow s_k$ est un plus court chemin, alors $\delta(s_i, s_k) = \delta(s_i, s_j) + c(s_j, s_k)$
$\rightarrow$ Dès qu'on connait $\delta(s_i, s_j)$, on peut calculer $\delta(s_i, s_k)$

Exploitation par programmation dynamique (Bellman-Ford)

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

Algorithm 3: relacher$((s_i, s_j), \pi, d)$

if $d[s_j] > d[s_i] + c(s_i, s_j)$ **then**
  $d[s_j] = d[s_i] + c(s_i, s_j)$;
  $\pi[s_j] = s_i$;

relacher(s_i, s_j) = mettre à jour $d[s_j]$ en considérant l'arc (s_i, s_j)

Principe de l'algorithme de Dijkstra

- Procède par coloriage des sommets :
  - s_i est blanc s'il n'a pas encore été découvert : $d[s_i] = +\infty$
  - s_i est gris s'il a été découvert et sa borne peut encore diminuer : $\delta(s_0, s_i) \leq d[s_i] < +\infty$
  - s_i est noir si sa borne ne peut plus diminuer : $d[s_i] = \delta(s_0, s_i)$ et 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 : sommet gris minimisant d
  - **Ne marche que si tous les coûts sont positifs** : Précondition à Dijkstra : Pour tout arc $(s_i, s_j) \in A$, $cout(s_i, s_j) \geq 0$

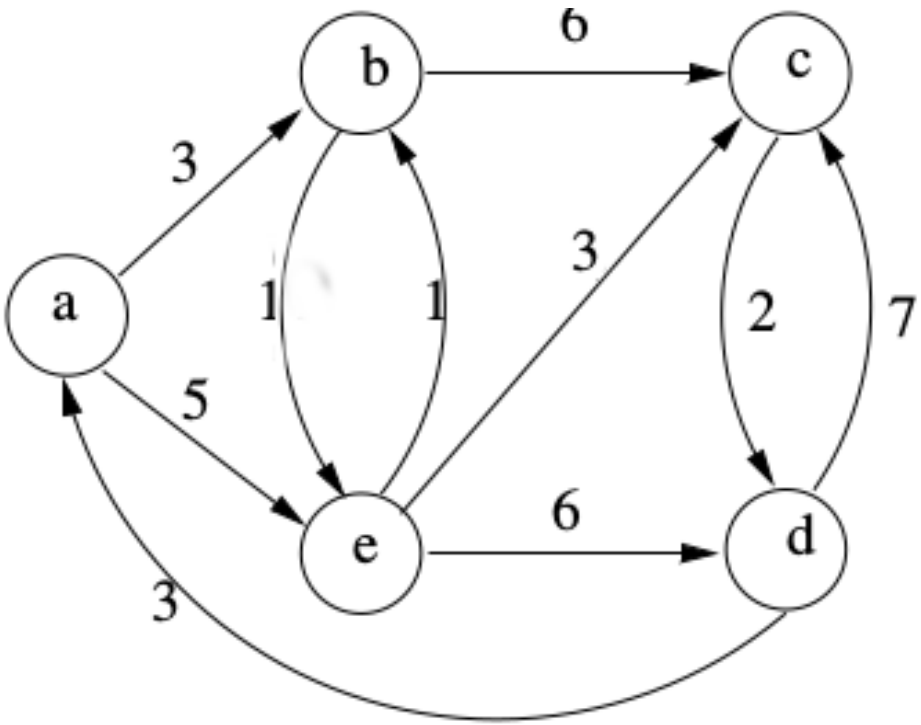
Algorithme de Dijkstra

Algorithm 4: Dijkstra(g, c, s_0)

foreach $s_i \in S$ **do**
    $d[s_i] = +\infty$;
    $\pi[s_i] = null$;
    Colorier s_i en blanc ;
$d[s_0] = 0$;
Colorier s_0 en gris ;
while *il existe un sommet gris* **do**
    Soit s_i le sommet gris tel que $d[s_i]$ soit minimal ;
    foreach $s_j \in Successeur(s_i)$ **do**
        if s_j *est blanc ou gris* **then**
            relacher($(s_i, s_j), \pi, d$) ;
            if s_j *est blanc* **then**
                Colorier s_j en gris ;
    Colorier s_i en noir ;

Algorithme de Dijkstra

Déroulez l'algorithme de Dijkstra sur le graphe suivant :



Algorithme de Dijkstra

s_i	Arcs Relachés	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$
a		0	∞	∞	∞	∞
a	$\{(a,b),(a,e)\}$	0	3	∞	∞	5
b	$\{(b,c),(b,e)\}$	0	3	9	∞	4
e	$\{(e,b),(e,c),(e,d)\}$	0	3	7	10	4
c	$\{(c,d)\}$	0	3	7	9	4

s_i	Arcs Relachés	$\pi[a]$	$\pi[b]$	$\pi[c]$	$\pi[d]$	$\pi[e]$
a		*null*	*null*	*null*	*null*	*null*
a	$\{(a,b),(a,e)\}$	*null*	a	*null*	*null*	a
b	$\{(b,c),(b,e)\}$	*null*	a	b	*null*	b
e	$\{(e,b),(e,c),(e,d)\}$	*null*	a	e	e	b
c	$\{(c,d)\}$	*null*	a	e	c	b

Algorithme de Dijkstra

Complexité pour un graphe ayant n sommets et p arcs ?

- $O(n^2)$ si recherche linéaire du sommet gris minimisant d
- $O((n+p) * log(n))$ si les sommets gris sont stockés dans un tas binaire

Programmation dynamique pour le calcul de plus courts chemins

Programmation dynamique

- Idée : Décomposer le problème en sous-problèmes (diviser pour régner)
  - Définition de la solution optimale par des équations récursives
  - Calculer en partant des cas de base
  - Ne pas recalculer plusieurs fois la même chose ; "Mémoization"
- Proposé par Bellman (1952) pour résoudre des problèmes de planification

Equations récursives pour calculer des plus courts chemins :

- Utiliser l'optimalité des sous-structures pour décomposer le problème
- $\delta^k(s_i)$ = longueur du plus court chemin de s_0 jusque s_i passant par au plus k arcs
- Définition récursive (sur k) de $\delta^k(s_i)$:
  - Si $k = 0$: $\delta^0(s_0) = 0$ et $\delta^0(s_i) = +\infty, \forall s_i \in S \{s_0\}$
  - Si $k > 1$: $\delta^k(s_i) = min(\{\delta^{k-1}(s_i)\} \cup \{\delta^{k-1}(s_j) + c(s_j, s_i) | sj \in pred(s_i)\})$
- Pour quelle valeur de k a-t-on $\delta^k(s_i) = \delta(s_0, s_i)$?

Algorithme de Bellman-Ford

Algorithm 5: Bellman-Ford(g, c, s_0)

foreach $s_i \in S$ **do**
    $d[s_i] = +\infty$;
    $\pi[s_i] = null$;
    Colorier s_i en blanc ;
$d[s_0] = 0$;
for $k \leftarrow 1$ **to** $|S| - 1$ **do**
    foreach $(s_i, s_j) \in A$ **do**
        relacher($(s_i, s_j), \pi, d$) ;

Algorithme de Bellman-Ford

Algorithm 6: Bellman-Ford(g, c, s_0)

foreach $s_i \in S$ **do**
    $d[s_i] = +\infty$;
    $\pi[s_i] = null$;
    Colorier s_i en blanc ;
$d[s_0] = 0$;
for $k \leftarrow 1$ **to** $|S| - 1$ **do**
    foreach $(s_i, s_j) \in A$ **do**
        relacher($(s_i, s_j), \pi, d$) ;

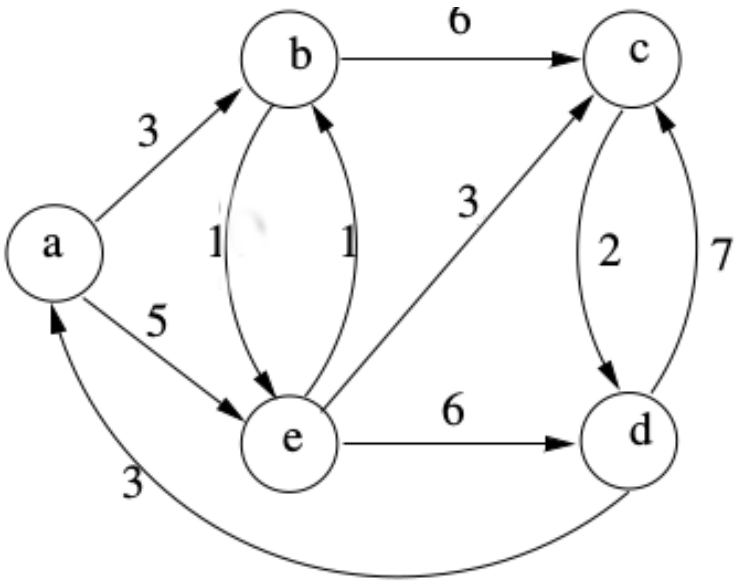
Détecter les circuits absorbants :

Tester si $\exists (s_i, s_j) \in A$ tel que $d[s_j] > d[s_i] + cout(s_i, s_j)$

Complexité pour un graphe ayant n sommets et p arcs :

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

Algorithme de Bellman-Ford



	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$
init	0	∞	∞	∞	∞
$k=1$	0	3	∞	∞	4
$k=2$	0	3	8	11	4
$k=3$	0	3	8	10	4
$k=4$	0	3	7	9	4

	$\pi[a]$	$\pi[b]$	$\pi[c]$	$\pi[d]$	$\pi[e]$
init	0	*null*	*null*	*null*	*null*
$k=1$	0	*a*	*null*	*null*	*a*
$k=2$	0	*a*	*b*	*e*	*b*
$k=3$	0	*a*	*e*	*c*	*b*
$k=4$	0	*a*	*e*	*c*	*b*

.