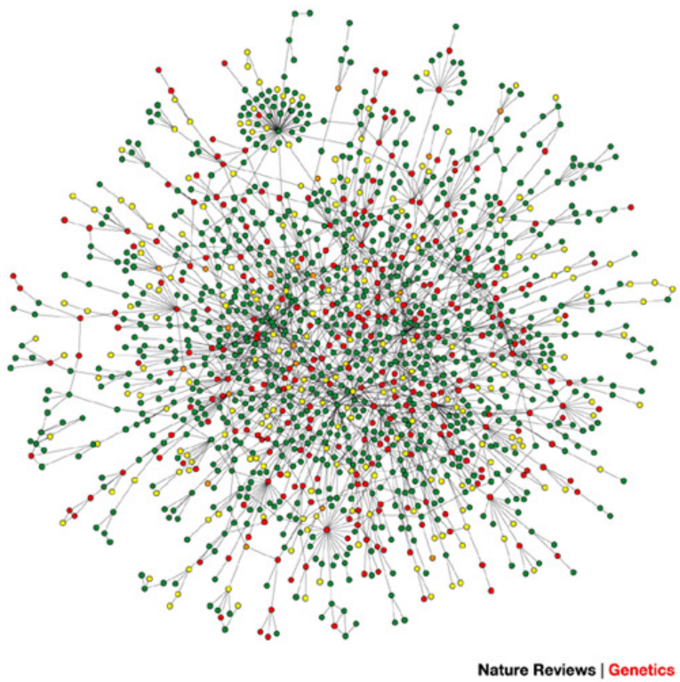
GRAPHES : REPRÉSENTATION ET PARCOURS

Graphes non-orientés

Déf. Un **graphe non-orienté** est representé par un couple (V, E) où $E \subseteq \binom{V}{2}$ (paires non-ordonnées).
V est l'ensemble des sommets et E est l'ensemble des arêtes.

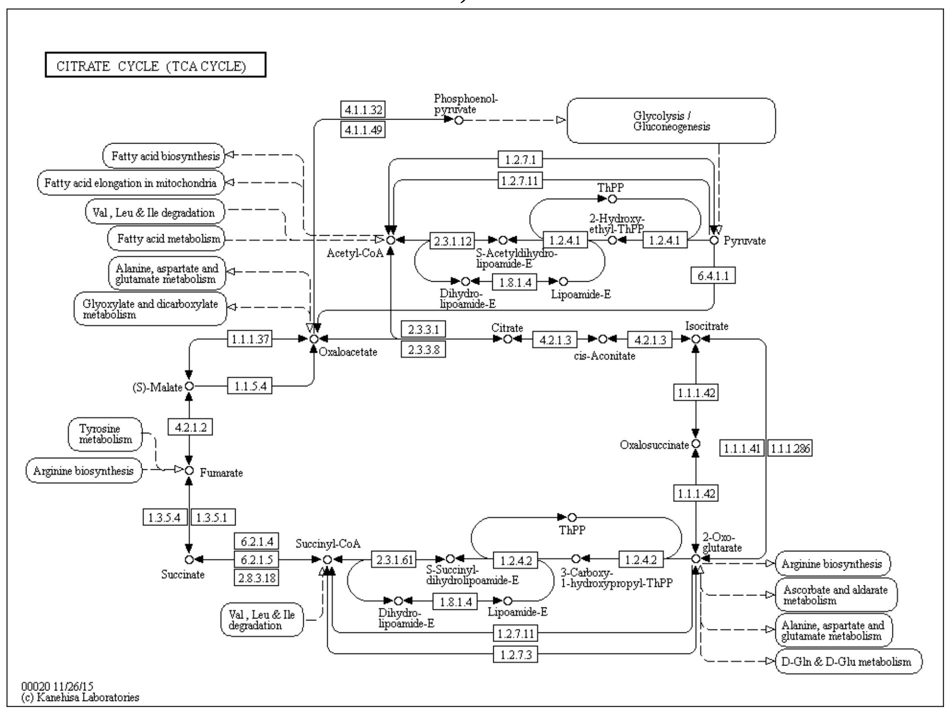


Barabási & Oltvai *Nature Rev Genet* 5 :101, 2004

Graphes orientés

Déf. Un **graphe orienté** est representé par un couple (V, E) où $E \subseteq V \times V$ (paires ordonnées).

V est l'ensemble des nœuds ou sommets, et E est l'ensemble des arcs.



Terminologie

Graphe non-orienté

L'arête $\{u, v\}$ est dénotée par uv.
Si $uv \in E$, alors v est adjacent à u.
L'arête $uv \in E$ est incidente aux sommets u et v.
Le degré de $u \in V$ est le nombre d'arêtes qui y sont incidentes.

Graphe orienté

L'arc (u, v) est dénotée par uv : l'arc part de u et arrive à v.
Si $uv \in E$, alors v est adjacent à u.
Le degré sortant de $u \in V$ est le nombre d'arcs qui y partent ; le degré rentrant est le nombre d'arcs qui y arrivent.

Chemins

Un **chemin** de longueur ℓ est une séquence $v_0, v_1, \ldots, v_\ell$ où $v_{i-1}v_i \in E$ pour tout $i = 1, \ldots, \ell$.

($\ell = 0$ est OK : chemin sans arêtes/arcs.)

Si $v_0 = v_\ell$, alors le chemin forme un **cycle**.

(Plus précisement, les sommets initial et final ne sont pas distingués dans le cycle.)

Le chemin $v_0 \cdots v_\ell$ est **élémentaire** ssi $v_1, \ldots, v_\ell$ sont distincts. Si $v_0 = v_\ell$, alors le chemin forme un **cycle élémentaire**.

Un graphe sans cycle est dit **acyclique**.

Un graphe non-orienté est **connexe** si chaque paire de sommets est relié par un chemin.

Un graphe non-orienté connexe acyclique est un **arbre**.

Sous-graphes

Le graphe $G' = (V', E')$ est un **sous-graphe** de $G = (V, E)$ ssi $V' \subseteq V$ et $E' \subseteq E$.

Étant donné un sous-ensemble de sommets $V' \subseteq V$, le sous-graphe de G **engendré** par V' est le graphe $G' = (V', E')$ avec $E' = \Big\{uv \in E \colon u, v \in V'\Big\}$.

Pondération

Graphe pondéré : chaque arc (ou arête) possède un **poids** ou coût associé, défini par la fonction de pondération $c\colon E \mapsto \mathbb{R}$.

Poids d'un sous-graphe : somme de poids des arcs dans le sous-graphe

Poids d'un chemin : somme de poids des arcs dans le chemin

Questions intéressantes

Stocker les graphes dans le mémoire d'un ordinateur

Parcours d'un graphe

Vérifier si le graphe est connexe

Plus court chemins

Arbres couvrants

Comment stocker le graphe ?

Matrice d'adjacence : matrice $V \times V$, $A[u, v]$ donne le poids de uv ($\pm\infty$ ou NaN pour arcs non-existants), ou valeurs booléennes pour noter juste présence

Listes d'adjacence : liste $\mathsf{Adj}[u]$ pour chaque sommet u qui stocke l'ensemble $\bigl\{v \colon uv \in E\bigr\}$ ou l'ensemble des couples $\bigl\{\langle v, c(u, v)\rangle \colon uv \in E\bigr\}$.

Usage de mémoire : dépend de la densité du graphe $\frac{|E|}{|V|^2}$.

Déterminer si $uv \in E$ ou $c(u, v)$: rapide avec la matrice mais plus lente avec les listes d'adjacence

Parcours d’un arbre

Un parcours visite tous les nœuds de l’arbre.

⋆ **parcours préfixe** (*preorder traversal*) : chaque nœud est visité avant que ses enfants soient visités.

⋆ **parcours postfixe** (*postorder traversal*) : chaque nœud est visité après que ses enfants sont visités.

⋆ avec un arbre binaire, on a aussi le **parcours infixe** (*inorder traversal*) : chaque nœud est visité après son enfant gauche mais avant son enfant droit.

Parcours préfixe et postfixe

Algo PARCOURS-PRÉFIXE(x)
if $x \neq \mathsf{null}$ **then**
    visiter x
    for $i \leftarrow 1, \ldots, k$ **do** PARCOURS-PRÉFIXE$\bigl(\mathsf{enfant}(x, i)\bigr)$

Algo PARCOURS-POSTFIXE(x)
if $x \neq \mathsf{null}$ **then**
    for $i \leftarrow 1, \ldots, k$ **do** PARCOURS-POSTFIXE$\bigl(\mathsf{enfant}(x, i)\bigr)$
    visiter x

($\mathsf{enfant}(x, i)$ donne l'enfant de x étiqueté par i — s'il n'y en a pas, alors null)

Maintenant PARCOURS-PRÉFIXE(racine) va visiter tous les nœuds dans l'arbre dans l'ordre préfixe.

Parcours infixe

Algo PARCOURS-INFIXE(x)

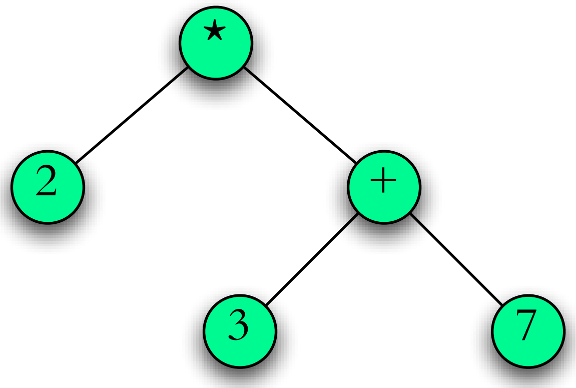
if $x \neq \mathsf{null}$ **then**

    PARCOURS-INFIXE$\big(\mathsf{gauche}(x)\big)$

    visiter x

    PARCOURS-INFIXE$\big(\mathsf{droit}(x)\big)$

Arbre syntaxique



notation infixe: 2⋆(3+7)
notation préfixe: ⋆ 2 + 3 7
notation postfixe: 2 3 7 + ⋆

Parcours avec pile

Algo PARCOURS-PILE

initialiser la pile P

P.push(root)

while $P \neq \emptyset$

  $x \leftarrow P$.pop()

  if $x \neq$ null **then**

    «visiter» x

    for $y \in x$.children **do** P.push(y)

Parcours avec queue

Algo PARCOURS-NIVEAU

initialiser la queue Q
Q.enqueue(root)
while $Q \neq \emptyset$
  $x \leftarrow Q$.dequeue()
  if $x \neq$ null **then**
    «visiter» x
    for $y \in x$.children **do** Q.enqueue(y)

PostScript

PostScript est un langage de programmation qui utilise une pile
Opérations en Postscript : `add`, `sub` `mul` `div`

P.e., la suite d'instructions `5 2 sub` place 5 et 2 sur la pile (dans cet ordre), et l'opérateur `sub` prend les deux éléments en haut de la pile pour les remplacer par le résultat de la soustraction. Dans ce cas-ci, la pile contiendra le seul élément 3 à la fin.

Opérateurs en PS

pile avant	opérateur	pile après	description
Arithmétique			
$x_1\, x_2$	**add**	x	addition ($x = x_1 + x_2$)
$x_1\, x_2$	**sub**	x	soustraction ($x = x_1 - x_2$)
$x_1\, x_2$	**mul**	x	multiplication ($x = x_1 \cdot x_2$)
$x_1\, x_2$	**div**	x	division ($x = x_1 / x_2$)
$n\, m$	**mod**	r	reste ($r = n \,\mathsf{mod}\, m$)
n	**neg**	$-n$	négative
Manipulation de la pile			
x	**pop**	—	dépile le haut
$x_1\, x_2$	**exch**	$x_2\, x_1$	échange les deux éléments en haut
x	**dup**	$x\, x$	duplique l'élément en haut
$x_1\, x_2\, \ldots\, x_n\, n$	**copy**	$x_1\, x_2\, \ldots\, x_n\, x_1\, x_2\, \ldots\, x_n$	duplique n éléments en haut
$x_n\, x_{n-1}\, \ldots x_0\, n$	**index**	$x_n\, x_{n-1}\, \ldots x_0 x_n$	copiage d'un élément arbitraire
$x_{n-1}\, x_{n-2}\, \ldots\, x_0\, n\, j$	**roll**	$x_{(j-1)\,\mathsf{mod}\, n} \ldots x_0, x_{n-1} \ldots x_{j\,\mathsf{mod}\, n}$	décalage circulaire
Opérateurs de contrôle			
$b\, p$	**if**	—	exécute p si b est vraie
$b\, p_1\, p_2$	**ifelse**	—	exécute p_1 si b est vraie ; p_2 sinon
$n_0\, d\, n_{\mathsf{max}}\, p$	**for**	—	boucle exécutée avec $n_0, n_0 + d, \ldots n_{\mathsf{max}}$
$n\, p$	**repeat**	—	boucle exécutée n fois (sans variables d'itération)
Tableaux			
—	**[**	*marque*	début de construction de tableau
marque $x_0\, \ldots x_{n-1}$	**]**	*tableau*	finit la construction du tableau
tableau i	**get**	x_i	accès à élément avec indice i (=0 pour le premier)
tableau	**length**	n	longueur du tableau
tableau p	**forall**	—	boucle exécutée pour chaque élément du tableau
Affichage sur sortie standard			
x	**=**	—	affiche le haut de la pile sur le flux standard
—	**stack**	—	affiche le contenu de la pile sans la détruire

PostScript

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 612 792
%%EndComments
    `%%' dénote un commentaire spécial. Ici on montre le préambule minimal
    selon la spécification EPSF (encapsulated PostScript) qui inclut l'indication de la
    boîte englobant tout le dessin (BoundingBox) en points (=1/72 pouce)

% a. manipulation de la pile   ←`%' dénote un commentaire jusqu'à la fin de la ligne (comme // en Java).
1 2 3.25 4e10 -5 6 7 8 % valeurs empilees   ←on écrit les nombres comme en Java
4 1 roll      ←rotation circulaire de quatre éléments vers le bas de la pile
pop      ← dépile l'élément en haut
exch     ← échange les deux éléments en haut de la pile
7 -4 roll     ← rotation circulaire de sept éléments vers le haut de la pile

% b. nouvel operateur        on définit un nouvel opérateur op par /op { ... } def
/mystique    ← `/' sert comme caractère d'échappement; ceci évite l'évaluation immédiate
{              ← accolades enferment un bloc d'instructions; ceci aussi évite l'évaluation immédiate
    2 copy   ← duplique les deux éléments en haut (avant: x y, après: x y x y)
    gt         ← test logique de «plus grand que»; remplace les deux éléments en haut par true ou false
    {
        exch
    } if     ← énoncé conditionnel; s'écrit par { ... } if. Cela consume la valeur booléenne en haut de la pile,
    pop      et exécute le code bloc si elle est vraie. Pour avoir une branche «sinon», utiliser {code pour si}
} def        {code pour sinon} ifelse

10 25 add /dfrnc exch def   ← on définit une variable var par /var valeur def. Souvent, on combine
                            avec exch quand la valeur est le résultat d'un calcul (pour lisibilité).
/sigle (IFT2015\tA13:) def  ← une chaîne de caractères s'écrit entre parenthèses (et non pas guillemets);
                            '\t' est le caractère de tabulation, échappée comme en Java.
/montableau [1 2 3 -4] def  ← on définit un tableau par crochets
   [0 2 1000 {} for] ← définition d'un tableau contenant 0,2,4,6,...,1000; for définit une boucle (init,
/autretbl exch def   incrément, valeur finale). (Logique de PostSript: il exécute le bloc {...} après avoir placé la
                     valeur de la variable d'itération sur la pile à chaque itération. Comme on ne dépile pas,
                     toutes les valeurs restent sur la pile et ']' les enlève jusqu'au crochet ouvert. )
% c. parcours d'un tableau
/tbl.? % [x0 x1 ...] tbl.? x
    Par convention, on décrit l'usage d'un opérateur en spécifiant les
    arguments (de gauche à droit, dans l'ordre d'empiler), et le résultat
    de l'opération remplaçant les arguments. Il n'y a pas de restriction
    sur le nom d'opérateurs — on peut utiliser tout caractère sauf %.
{
    0 exch
    {
        0 lt {1 add } if  ← lt est le test logique de «plus petit que»
    } forall  ← boucle sur les éléments du tableau: {...} forall. En chaque itération,
} def         le prochain élément du tableau est automatiquement empilé et le code bloc
              est exécuté; le tableau même est dépilé avant la première itération.

montableau tbl.?  ← exécuter tbl.? avec montableau
20 string cvs     ← conversion en string: le syntaxe est x s cvs s', où x est la valeur à convertir, s est un string
                  de longueur suffisante, allouée par 20 string. Le résultat s' remplace le contenu de s
/Times-Roman 12 selectfont ← sélection de police de 12 pt. Times est une des polices prédéfinies.
10 20 moveto ← déplacement aux coordonnées (10, 20). Axes X et Y vont de gauche à droit et de bas vers haut
sigle show show   ← dessin de 2 strings à partir de la position courante (10,20).
5 3.5 rmoveto 612 792 lineto stroke ← déplacement+dessin de ligne; stroke raie le chemin construit
%%EOF   ←commentaire spécial exigé par la spécification EPSF à la fin du fichier.
```

Parcours de graphe

Idée générale : suivre toujours une arête qui mène d'un sommet visité à un sommet non-visité

Parcours par profondeur : choix d'arête/arc uv où u est visité le plus récemment (pile)

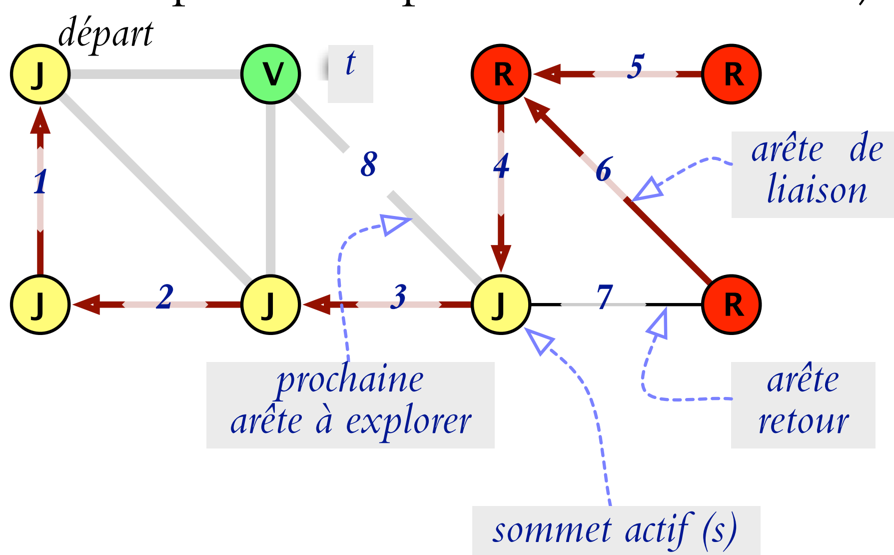
Parcours par largeur : choix d'arête/arc uv où u est visité le moins récemment (queue)

Parcours en profondeur — DFS

Technique essentielle : marquage/coloriage «jamais vu», «déjà vu»

(init)	$\textsf{parent}[s] \leftarrow s$; **for** $u \leftarrow 0, 1, \ldots, n-1$ **do** $\textsf{couleur}[u] \leftarrow \textsf{verte}$	
	DFS(s)	*// parcours en profondeur à partir de sommet s*
D1	$\textsf{couleur}[s] \leftarrow \textsf{jaune}$	*// prévisite de s*
D2	**for** $st \in \textsf{Adj}[s]$ **do**	*// pour tout sommet t adjacent à s*
D3	**if** $\textsf{couleur}[t] = \textsf{verte}$ **then** DFS(t) ; $\textsf{parent}[t] \leftarrow s$	*// visite du voisin t*
D4	$\textsf{couleur}[s] \leftarrow \textsf{rouge}$	*// post-visite de s*

(Généralisation du parcours préfixe ou postfixe sur les arbres.)



Graphe biparti (avec DFS)

(init) partition$[s] \leftarrow 1$; **for** $u \leftarrow 0, 1, \ldots, n-1$ **do** couleur$[u] \leftarrow$ verte

DFS-BIP(s) *// tester si la composante connexe des est biparti*

couleur$[s] \leftarrow$ jaune

for $st \in \mathsf{Adj}[u]$ **do**

    if couleur$[t] =$ verte **then** *// arête de liaison*

        partition$[t] = 3 -$ partition$[s]$ // $1 \leftrightarrow 2$

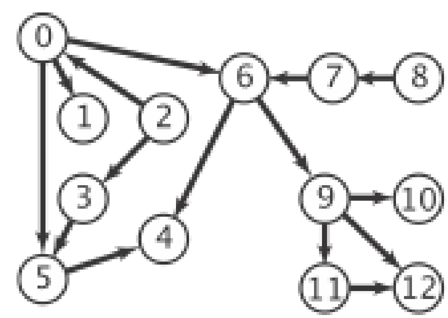
        DFS-BIP(t)

    else if couleur$[t] =$ jaune **then** *// parent ou arête retour*

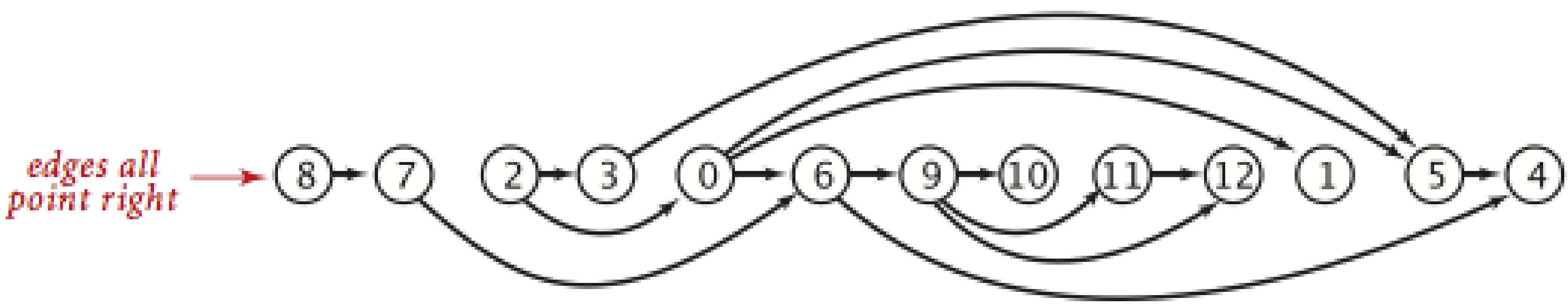
        if partition$[s] \neq$ partition$[t]$ **then** die(«Cycle de longueur impaire»)

Tri topologique (avec DFS)

(init)	parent$[s] \leftarrow s$; **for** $u \leftarrow 0, 1, \ldots, n-1$ **do** couleur$[u] \leftarrow$ verte
	DFS(s) *// parcours en profondeur à partir de sommet s*
D1	couleur$[s] \leftarrow$ jaune *// prévisite de s*
D2	**for** $st \in$ Adj$[s]$ **do** *// pour tout sommet t adjacent à s*
D3	**if** couleur$[t]$ = verte **then** DFS(t) ; parent$[t] \leftarrow s$ *// visite du voisin t*
D4	couleur$[s] \leftarrow$ rouge ; P.push(u)



À la fin, P.pop défile dans l'ordre topologique.



Parcours en largeur

Utiliser une file FIFO (*queue*) Q

```
BFS(s)
for u ← 0, 1, ..., n − 1 do couleur[u] ← jaune
d[s] ← 0 ; parent[s] ← s
Q.enqueue(s); couleur[s] ← jaune
while Q ≠ ∅ do
    u ← Q.dequeue()
    for uv ∈ Adj[u] do
        if couleur[v] = verte then
            d[v] ← d[u] + 1; parent[v] ← u
            Q.enqueue(v); couleur[v] ← jaune
        end
    end
    couleur[u] ← rouge
end
```

BFS trouve les plus courts chemins