Chapitre 3

Parcours de graphes

Il y a deux façons de parcourir exhaustivement tous les descendants d'un sommets à partir d'un sommet donné : le parcours en largeur et le parcours en profondeur. Intuitivement, la différence entre les deux est le fait que pour le parcours en largeur nous explorons tous les descendants d'un sommet et ensuite tous les descendants des descendants et ainsi de suite, alors que pour un parcours en profondeur nous explorons récursivement les descendants d'un sommet en faisant du backtracking (retour arrière) quand nous arrivons à un sommet que nous avons déjà visité.

3.1 Parcours en largeur

Étant donné un graphe $G = (X, U)$ ainsi qu'un sommet *origine* s le parcours en largeur explore tous les sommets accessibles à partir de s parc des chemins de G. Il doit son nom au fait qu'il découvre d'abord tous les sommets situés à une distance k de s avant de découvrir tout sommet situé à la distance $k + 1$. Pour garder une trace de sa progression l'algorithme colorie les sommets. Au début tous les sommets sont *blancs* et deviennent *gris* au moment de la première visite. Les sommets gris sont à la frontière de l'exploration. Un sommet est *noir* si l'exploration est finie pour lui.

La procédure de parcours en largeur suppose que le graphe d'entrée $G = (X, U)$ est représenté par une liste d'adjacences. Pour chaque sommet $x \in X$ nous avons une structure des données avec les informations suivantes :

— x . `couleur` représente la couleur du sommet x
— x . π représente le parent de x ; si x n'a pas de parent (c'est par exemple le sommet s) ou si x n'a pas encore était découvert, alors x . π `= null`
— x . `d` représente la distance entre le sommet s et le sommet x.

L'algorithme a également recours à une file pour gérer l'ensemble des sommets gris. L'algorithme 1 présente le pseudocode pour le parcours en largeur. La figure 3.1 contient un exemple.

Algorithm 1 Parcours en largeur

```
 1: procedure PARCOURS-LARGEUR(G = (X, U), s)
 2:     for x ∈ X − {s} do
 3:         x.couleur = BLANC
 4:         x.d = ∞
 5:         x.π = null
 6:     s.d = 0
 7:     s.couleur = GRIS
 8:     s.π = null
 9:     F = {s}
10:     while F ≠ ∅ do
11:         x = DÉFILE(F)
12:         for all y ∈ Adj[x] do
13:             if y.couleur == BLANC then
14:                 y.couleur = GRIS
15:                 y.d = x.d + 1
16:                 y.π = x
17:                 ENFILE(F, y)
18:         x.couleur = NOIR
```

Analyse de l'algorithme

Les lignes 2–9 initialisent les variables de l'algorithme et prennent un temps d'exécution $O(|X|)$. Après initialisation plus aucun sommet n'est colorié BLANC et le test de la ligne 13 garantit que chaque sommet est enfilé et défilé au plus une fois. Comme les operations d'enfilement et défilement sont $O(1)$ le temps total des operations dans la file est $O(|X|)$. Comme la liste d'adjacences pour chaque sommet n'est parcouru qu'au moment de défilement, elle est parcouru au plus une fois pour chaque sommet. La longueur totale des les listes d'adjacences étant $|U|$, le temps maximum pour parcourir les listes d'adjacences est $O(|U|)$. Par consequence, le coup total est $O(|X|) + O(|X| + |U|)$ qui est équivalent à $O(|X| + |U|)$.

Arborescence de parcours en largeur

Pendant l'exploration du graphe par l'algorithme de parcours en largeur, une arborescence est créé. Cette arborescence est représentée par le champ π de chaque sommet. Plus formellement, pour un graphe $G = (X, U)$ avec origine s nous définissons le sous-graphe prédécesseur de G par $G_\pi = (X_\pi, U_\pi)$ où :

$$X_\pi = \{v \in X : v.\pi \neq \texttt{null}\} \cup \{s\}$$

et

$$U_\pi = \{(v.\pi, v) : v \in X_\pi - \{s\}\}$$

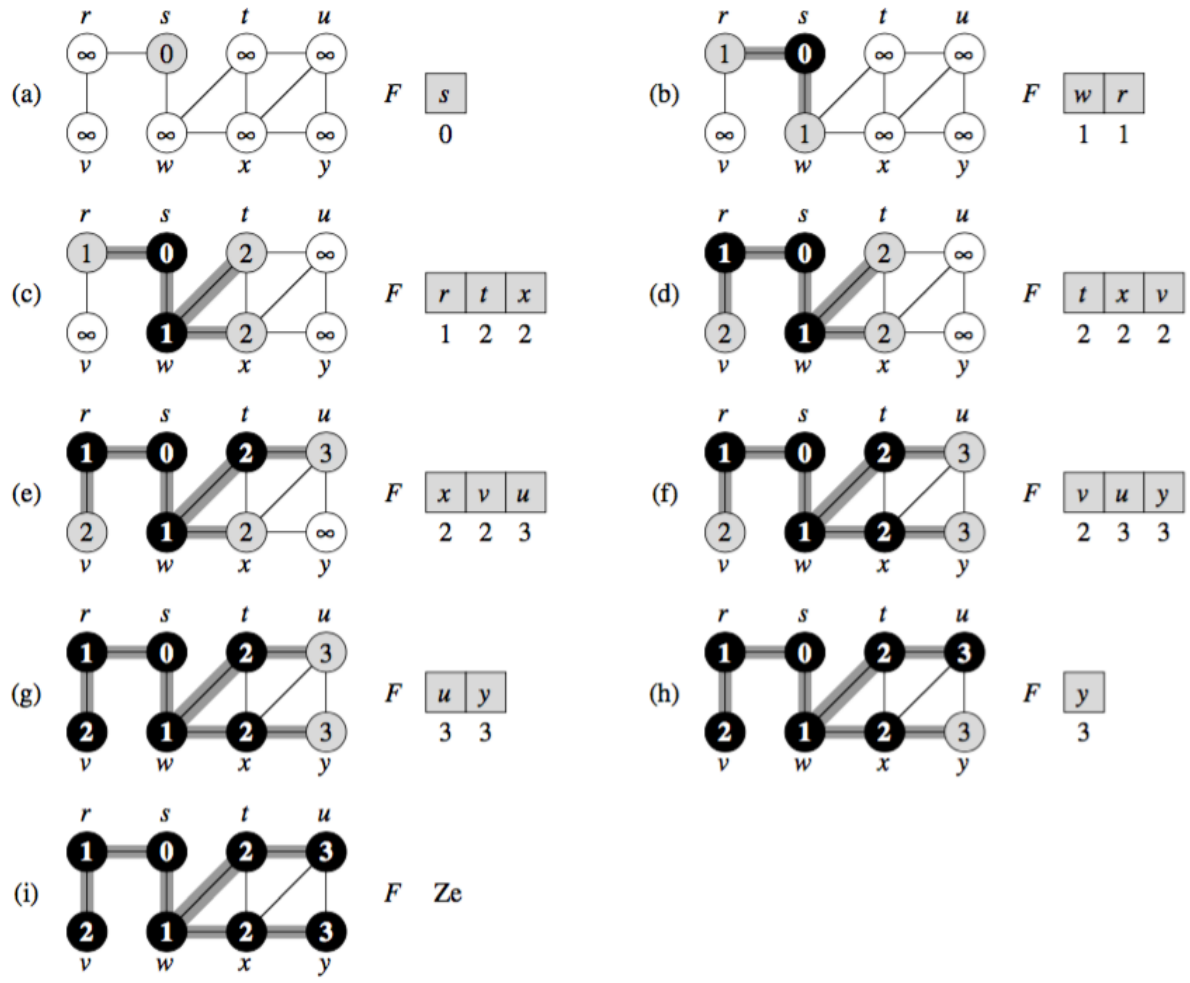


FIGURE 3.1 – Execution du parcours en largeur étape par étape (image issue de [1])

On appelle le sous-graphe des prédécesseurs G_π une *arborescence de parcours en largeur*. L'algorithme suivant imprime les sommets du chemin reliant s à y. Notons que cette arborescence contient seulement les sommets accessibles à partir de s.

```
1: procedure IMPRIMER-CHEMIN(G, s, y)
2:     if y == s then
3:         imprimer s
4:     else if y.π == null then
5:         imprimer " il n'existe aucun chemin de " s " à " y
6:     else
7:         IMPRIMER-CHEMIN(G, s, v.π)
8:         imprimer y
```

3.2 Parcours en profondeur

En contrast avec le parcours en largeur, le but du parcours en profondeur est d'explorer recursivement, tant que possible, les descendant d'un sommet—au lieu d'explorer tous ses enfants et ensuite les enfants des enfants, etc. Quand l'algorithme se trouve à un sommet qu'il a déjà explorer, il fait pas en arrière (backtrack) pour continuer l'exploration et ce jusqu'au avoir exploré le graphe entier.

Pour le parcours en profondeur nous utiliserons également une liste d'adjacences et un champ π pour chaque sommet : si, pour un sommet x déjà découverte nous arrivons à un nouveau sommet y, alors $y.\pi = x$.

Au contraire de la version du parcours en largeur que nous avons montré au section précédant, le parcours en profondeur peux continuer l'exploration du graphe même pour les sommets non-accessibles à partir de s, en désignant arbitrairement un sommet non-exploré comme sommet origine. Le résultat de cela est le fait que le parcours en profondeur ne produit plus une arborescence mais un forêt. Le sous-graphe des prédécesseurs est donc définit un peu différemment : on pose $G_\pi = (X, U_\pi)$ où :

$$U_\pi = \{(x.\pi, x) : x \in X \text{ et } x.\pi \neq \texttt{null}\}$$

Comme dans le parcours en largeur, les sommets sont coloriés pendant le parcours pour indiquer leur état. Chaque sommet est initialement blanc, puis gris quand il est découvert pendant le parcours, et enfin noirci en fin de traitement, c'est-à-dire quand sa liste d'adjacence a été complètement examinée. Cette technique assure que chaque sommet appartient à une arborescence de parcours en profondeur et une seul, de sorte que ces arborescences sont disjointes.

En plus de créer une forêt de parcours en profondeur, le parcours en profondeur date chaque sommet. Chaque sommet x porte deux dates : la première, $x.d$, marque le moment où x a été découvert pour la première fois (et colorié en gris) *date de pré-visite*, et la seconde, $y.f$, enregistre le moment ou le parcours a fini d'examiner la liste d'adjacence de y (et le colorie en noir), appelée *date de post-visite*. Ces dates sont utilisées dans de nombreux algorithmes de graphes et sont utiles pour analyser le comportement du parcours en profondeur. $y.d$ et $y.f$ sont des entiers compris entre 1 et $2|X|$ et pour tout sommet

$$x : u.d < u.f$$

Le pseudocode suivant donne le parcours en profondeur.

```
1: procedure PARCOURS-PROFONDEUR(G = (X, U))
2:     for all x ∈ X do
3:         x.couleur = Blanc
4:         x.π = null
5:     date = 0
6:     for all x ∈ X do
7:         if x.couleur == Blanc then
8:             VISITE-PARCOURS-PROFONDEUR(G, x)
```

1: **procedure** VISITE-PARCOURS-PROFONDEUR(G, x)
2: $date = date + 1$ ▷ le sommet blanc x vient d'être découvert
3: x.`couleur` $=$ `Gris`
4: $x.d = date$
5: **for all** $y \in G$.`Adj[x]` **do** ▷ Exploration de l'arc (x, y)
6: **if** y.`couleur` == BLANC **then**
7: $y.\pi = x$
8: VISITE-PARCOURS-PROFONDEUR(G, v)
9: x.`couleur` $=$ `Noir` ▷ noircir x, car on en a fini avec lui
10: $date = date + 1$
11: $x.f = date$

La figure 3.2 montre la progression de l'execution d'un parcours un profondeur sur un graphe donné.

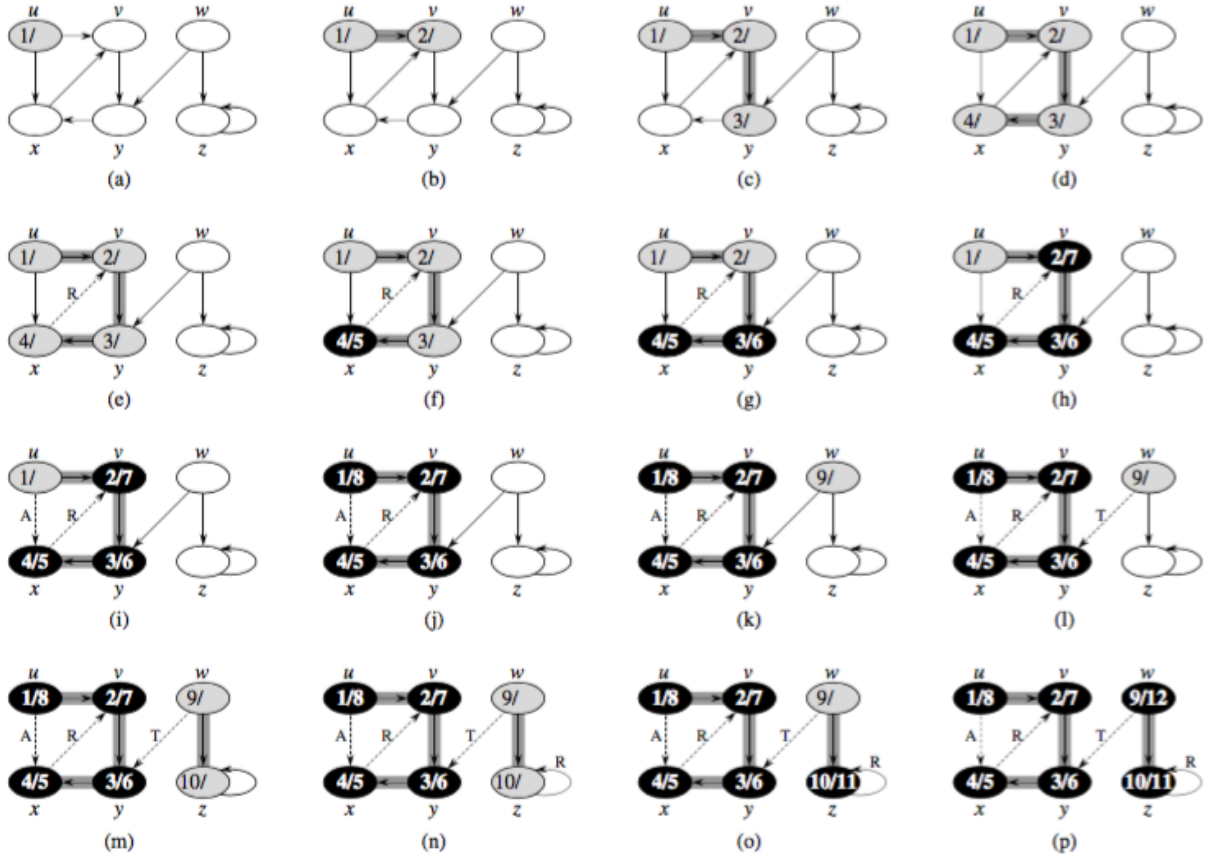


FIGURE 3.2 – Execution du parcours en profondeur étape par étape (image issue de [1])

Analyse de l'algorithme

Le boucle des lignes 2–4 et 6–8 du PARCOURS-PROFONDEUR requièrent un temps d'execution $\Theta(X)$, sans compter le temps d'execution des appels VISITE-PARCOURS-PROFONDEUR. Cette dernière procedure est appelée exactement une fois pour chaque sommet $y \in X$ puisqu'elle n'est invoquée que sur les sommets blancs, et qu'elle commence par les peindre en gris. Pendant l'exécution de VISITE-PARCOURS-PROFONDEUR la boucle des lignes 5–8 est exécuté $|\texttt{Adj}\,[x]|$ fois. Or,

$$\sum_{x \in X} |\texttt{Adj}\,[x]| = |U|$$

le coût total de la procedure VISITE-PARCOURS-PROFONDEUR est $\Theta(U)$. Donc, le temps d'exécution de PARCOURS-PROFONDEUR est $\Theta(X + U)$.

3.3 Tri Topologique

Le tri topologique d'un graphe orienté sans circuit $G = (X, U)$ consiste à ordonner tous ses sommets de sorte que, si G contient un arc (u, v), x apparaisse avant y dans le tri.

(Si le graphe n'est pas sans circuit, aucun ordre linéaire n'est possible.) Le tri topologique d'un graphe peut être vu comme un alignement de ses sommets le long d'une ligne horizontale de manière que tous les arcs soient orientés de gauche à droite.

Le tri topologique peut être utile dans de nombreuses applications, comme par exemple l'ordonnancement de tâches (trouver quel tâches exécuter avant quelle autre tâche selon leur dépendances), plus court chemin (Bellman sans circuit).

3.3.1 Détection des circuits : mise en niveau

Algorithme de mise en niveau :

1. faire le dictionnaire des prédécesseurs
2. le niveau N_0 est l'ensemble des sommets sans prédécesseurs
3. k=1
4. le niveau N_k est l'ensemble des sommets sans prédécesseurs dans G privés des sommets $N_0, \ldots N_{k-1}$
5. répéter l'étape précédente pour $k + 1$.
6. jusqu'à ce qu'il n'y ait plus de sommets sans prédécesseurs.

Si à la fin de l'algorithme tous les sommets ont pu être placés dans des niveaux le graphe est sans-circuit et on peut le dessiner par niveau sinon il a au moins un circuit.

Les sommets pris dans l'ordre des niveaux forment un tri topologique.

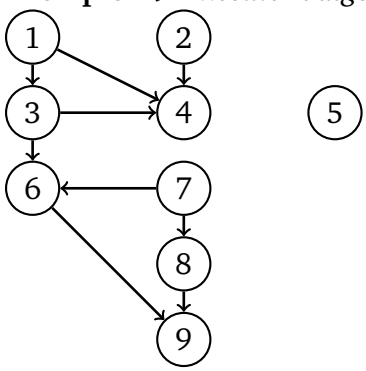
3.3.2 Algorithme de tri topologique

Voici un algorithme simple et efficace qui nous permet d'effectuer un tri topologique :

1: **procedure** TRI-TOPOLOGIQUE(G)
2: PARCOURS-PROFONDEUR(G) ▷ pour calculer les dates de fin de traitement $y.f$ pour chaque sommet y
3: chaque fois que le traitement d'un sommet se termine, insérer le sommet dans une pile
4: **renvoyer** la pile des sommets

On peut effectuer un tri topologique en $\Theta(X+U)$, car le parcours en profondeur prend $\Theta(X+U)$ et l'insertion de chacun des $|X|$ sommets au début de la pilee nécessite $O(1)$.

Exemple 19 *Executer l'algorithme sur le graphe suivant :*



3.4 Composantes fortement connexes

Nous avons vu au chapitre 1 qu'une composante fortement connexe d'un graphe orienté $G=(X,U)$ est un ensemble maximal de sommets $C \subseteq X$ tel que, pour chaque paire de sommets x et y de C, on ait à la fois $x \rightsquigarrow y$ et $y \rightsquigarrow x$. On peut aussi définir la composante connexe $\overline{x}$ de x comme l'intersection des descendants et des ascendants de x : $\overline{x} = D(x) \cap A(x)$.

Afin de trouver de façon efficace les composantes fortement connexes d'un graphe orienté nous introduisons la notion de *graphe transposé*. Étant donné un graphe orienté $G=(X,U)$ son graphe transposé G^T est défini par $G^T=(X,U^T)$ où

$$U^T = \{(u,v) : (v,u) \in U\}$$

Autrement dit, G^T est constitué des arcs de G dont le sens a été inversé. La complexité pour la création du G^T, étant donnée une représentation par liste d'adjacence, est $O(X+U)$.

L'algorithme suivant calcule les composantes fortement connexes.

1: **procedure** COMPOSANTES-FORTEMENT-CONNEXES(G)

2: PARCOURS-PROFONDEUR(G) ▷ pour calculer les dates de fin de traitement $y.f$ pour chaque sommet y
3: Calculer $G^T = (X, U^T)$
4: PARCOURS-PROFONDEUR(G^T) en considérant les sommets par ordre de $x.f$ décroissants
5: imprimer les sommets de chaque arborescence de la forêt obtenue en ligne 3 en tant que composante fortement connexe distincte

Exemple 20 *Executer l'algorithme sur le graphe suivant :*