Leçon 925 : Graphes : représentations et algorithmes

Julie Parreaux

2018 - 2019

Références pour la leçon

[1] Beauquier, Berstel et Chretienne, *Éléments d'algorithmique.*
[2] Cormen, *Algorithmique.*
[3] Froidevaux, Gaudel et Soria, *Types de données et algorithmes.*

Développements de la leçon

Le tri topologique | Approximation ou non du problème TSP

Plan de la leçon

1 La structure d'un graphe **2**

2 Problèmes d'accessibilité **3**
2.1 Parcourir un graphe [2, p.549] . . . 3
2.2 Problème de la connexité [3, p.419] . . . 3
2.3 La robustesse d'un graphe [3, p.448] . . . 3

3 Problèmes de routage dans un graphe **3**
3.1 Le plus court chemin [2, p.495] . . . 3
3.2 Vers les problèmes de couverture . . . 4

4 Problèmes de couverture **4**
4.1 Arbre couvrant minimal [2, p.583] . . . 4
4.2 Problème de couverture par sommet . . . 4
4.3 Coloration d'un graphe . . . 5

5 Problème de flots **5**
5.1 Parcours de graphes et applications . . . 6

Motivation

Défense

Les graphes sont des objets qui permettent de modéliser de nombreux problèmes informatiques et de la vie de tous les jours : réseaux (transport, internet, information, eau, électri-

cité), relation entre des entité (réseaux sociaux, contraintes), circuit imprimé, des programmes ... Leurs études et les problèmes sont donc fondamentaux. Il existe également des problèmes calculatoires sur ces graphes. Lorsqu'on étudie les graphes, de nombreux paradigmes informatiques ressortent : on a différents paradigmes algorithmiques, du calcul de complexité, l'utilisation pertinente de structure de données ou encore l'apparition de quelques classes de problèmes.

Ce qu'en dit le jury

Cette leçon offre une grande liberté de choix au candidat, qui peut choisir de présenter des algorithmes sur des problèmes variés : connexité, diamètre, arbre couvrant, flot maximal, plus court chemin, cycle eulérien, etc. mais aussi des problèmes plus difficiles, comme la couverture de sommets ou la recherche d'un cycle hamiltonien, pour lesquels il pourra proposer des algorithmes d'approximation ou des heuristiques usuelles. Une preuve de correction des algorithmes proposés sera évidemment appréciée. Il est attendu que diverses représentations des graphes soient présentées et comparées, en particulier en termes de complexité.

1 La structure d'un graphe

Un graphe peut être vu comme un objet mathématiques servant à modéliser ou bien à une structure de données quelconque. Ici nous présentons le graphe vu comme un objet mathématiques sur lequel on peut rajouter des propriétés mais avec une représentation afin de pouvoir écrire des algorithmes.

- *Définition* [1, p.74] : graphe non-orienté et graphe orienté
- *Définition*[2, p.545] : un poids (rend l'arbre pondéré)
- *Définition* : l'arité + le degré d'un nœuds
- *Définition* [1, p.74] : graphe biparti
- Représentation des graphes : matrice d'adjacence, liste d'adjacence + *Remarque* : il en existe d'autre

Complexité	Liste d'adjacence	Matrice d'adjacence
Spatiale	$O(\|S\| + \|A\|)$	$O(\|S\|^2)$
Renvoyer la liste des voisins	$O(1)$	$O(\|S\|)$
Tester si deux sommets sont voisins	$O(\|S\|)$	$O(1)$
Parcours en largeur (Algorithme)	$O(\|S\| + \|A\|)$ (agrégat)	$O(\|S\|^2)$
Parcours en profondeur (Algorithme 1)	$O(\|S\| + \|A\|)$	$O(\|S\|^2)$
Composantes fortement connexes (Algorithmes)	$O(\|S\| + \|A\|)$	$O(\|S\|^2)$
2-connexité (Algorithme)	$O(\|S\| + \|A\|)$	$O(\|S\|^2)$
Bellman–Ford	$O(\|S\|\|A\|)$	$O(\|S\|^3)$
Dijkstra	$O((\|S\| + \|A\|) \log \|S\|)$	$O(\|S\|(\|S\| + \|A\|) \log \|S\|)$
Floyd–Warshall	X	$O(\|S\|^3)$
Prim	$O(\|A\| \log \|S\|)$	
Kruskal	$O(\|A\| \log \|A\|)$	

2 Problèmes d'accessibilité

Puis-je aller jusqu'à ce point ? Dans un graphe répondre à cette question est un problème en temps polynomiale. On va même aller plus loin en cherchant si tous les sommets peuvent aller jusqu'à n'import quelle autre sommets : c'est la connexité. On finira par parler un peu de robustesse du réseau avec la 2-connexité. Application : réseau électrique.

2.1 Parcourir un graphe [2, p.549]

- Parcours en largeur : principe + algorithme + complexité
- *Application* : Test le caractère biparti d'un graphe
- Parcours en largeur : principe + algorithme 1 + complexité
- *Application* : Tri topologique DEV

2.2 Problème de la connexité [3, p.419]

La vérification de la connexité d'un graphe : peut-on aller de tous sommets vers tous sommets est un problème pratique important (exemple : réseau électrique).

- *Définitions* [3, p.140] : composantes connexes (graphe non-orienté) et fortement connexes (graphe orienté)
- *Remarque* : Composante connexes : algorithme de parcours
- Algorithme de Kosaraju (application du parcours en profondeur)
- Algorithme de Tarjan (application du parcours en profondeur)

2.3 La robustesse d'un graphe [3, p.448]

Dans des réseaux de télécommunication, par exemple, il est intéressant que si un des noeuds tombe en panne, le reste du réseau reste fonctionnel (connexe). On se place dans le cadre d'un graphe non-orienté connexe et on souhaite savoir si il est robuste devant une panne.

- *Hypothèses* : Graphe non-orienté et connexe
- *Définition* : 2-connexité + point d'articulation + composante
- Algorithme + complexité

3 Problèmes de routage dans un graphe

Comment puis-je aller jusqu'à ce point ? Le problème de routage vient généraliser le problème d'accessibilité : en plus de savoir si deux noeuds sont accessibles on veut connaître un chemin entre ces deux noeuds. Souvent ce chemin vérifie une propriété. Application : internet

3.1 Le plus court chemin [2, p.495]

Le plus court chemin est un problème minimisant une certaine quantité. Souvent, on traite se problème dans un graphe pondéré. Il existe plusieurs variantes de ce problème : à destination unique, pour un couple de sommet ou entre tout couples de sommets. On va étudier des algorithmes résolvant ces variantes sous certaines hypothèses.

- *Définition* : problème du plus cours chemin dans un graphe

— *Remarque* : dans un graphe non pondéré un parcours en largeur suffit
— Algorithme de Bellman–Ford (graphe pondéré à valeurs réelles + origine unique) + complexité
— Algorithme de Dijkstra (graphe pondéré à valeurs positives + origine unique) + complexité DEV
— Algorithme de Floyd–Warshall (graphe pondéré à valeurs réelles sans cycle négatifs + tout couple de chemin) + complexité + application à la fermeture transitive (application aux composantes connexes)

Remarque : le problème du chemin le plus long est Np-complet.

3.2 Vers les problèmes de couverture

Les problèmes que l'on présente maintenant ne sont pas tout à fait des problèmes de routage : on ne cherche pas à relier deux points, ni tout à fait des problèmes de couverture car on cherche un chemin (ou un cycle) dans le graphe.

— *Définition* : problème du chemin eulérien
— *Proposition* : critère de la présence d'un chemin eulérien
— *Définition* : problème du chemin hamiltonien
— *Proposition* : chemin hamiltonien est NP-complet
— *Définition* : problème du TSP + version géométrique
— *Proposition* : TSP (géométrique) est NP-complet
— Approximation gloutonne + non ϵ-approximation DEV

4 Problèmes de couverture

Un problème de couverture est un problème qui cherche à couvrir notre graphe à l'aide d'un objet (soit tous les sommets, soit toutes les arêtes sont dans cet objets sans en violer les propriétés) : coloration, arbre couvrant, ... Les problèmes de clique permettent de modéliser des problèmes de serveurs web et les problèmes d'indépendant set modélisent les activités compatibles ou non.

4.1 Arbre couvrant minimal [2, p.583]

— *Définition* : arbre couvrant minimal
— Algorithme de Prim (principe + complexité)
— Algorithme de Kruskal (principe + complexité)

4.2 Problème de couverture par sommet

— *Définition* : problème couverture par sommet (éclairage)
— *Proposition* : problème NP-complet
— Algorithme d'approximation

4.3 Coloration d'un graphe

— *Définition* : problème de coloration
— *Proposition* : problème NP-complet
— Restriction du problème pour le rendre P

5 Problème de flots

Quelle est la capacité de mon graphe ?

— *Définition* : problèmes Max flot / Min Cut + dualité
— Algorithme de Ford–Fulkerson (principe + complexité) + appli biparti

Remarque : le problème Max Cut est NP-complet.

Quelques notions importantes

Représentation d'un graphe

Il existe plusieurs représentations de graphes $G = (S, A)$. Nous allons allons étudier les trois plus classiques : matrice d'adjacence, la matrice d'incidence et la liste d'adjacence. Nous allons étudier leur points fort et leur points faible.

Matrice d'adjacence La matrice d'adjacence est une matrice carrée indicé par S et définie telle que

$$m_{ij} = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{sinon} \end{cases}$$

Le 0 et le 1 peuvent être des booléens ou bien des entiers. Dans le cas d'un graphe pondéré, le 1 peut être remplacé par la valeur du poids de la dite arête et 0 peut être remplacé par ∞.

Lorsque G est non-orienté, il peut être pratique de ne conservé que la partie triangulaire supérieure de la matrice (car elle est symétrique). Cette représentation est usuellement utilisé dans le cas de graphe dense ($|A|$ est proche de $|S|^2$) où lorsque les graphes sont raisonnablement petit.
Complexité spatiale : $\Theta(|S|^2)$
Complexité temporelle pour trouver si deux sommets sont adjacents : $O(1)$
Complexité temporelle pour trouver la liste des voisins d'un sommet : $O(|S|)$

Matrice d'incidence [1, p.76] On suppose que G est sans boucle. On définit la matrice d'incidence comme une matrice de taille $|S| \times |A|$ et définie telle que

$$\delta_{sa} = \begin{cases} 1 & \text{si } x \text{ est l'origine de } a \\ -1 & \text{si } x \text{ est la destination de } a \\ 0 & \text{sinon} \end{cases}$$

Cette matrice possède des propriétés intéressantes pour des problèmes combinatoires dans les problèmes de flots par exemple. En effet, le déterminant de toutes sous-matrice carrée extraite vaut $0, 1$ ou -1.

Liste d'adjacence On définit la liste d'adjacence comme un tableau de $|S|$ listes (indicé sur S) où chacune des cases (correspondant à un sommet s) continent l'ensemble des sommets adjacents à s. Pour stocker le poids d'une arête, si G est un graphe pondéré, on rajoute un champs à notre liste qui permet à côté du voisin d'indiquer son poids.

On utilise usuellement listes d'adjacences pour des graphe peut denses ($|A|$ est très inférieur à $|S|^2$).
Complexité spatiale : $\Theta(|S| + |A|)$
Complexité temporelle pour trouver si deux sommets sont adjacents : $O(|S|)$
Complexité temporelle pour trouver la liste des voisins d'un sommet : $O(1)$

5.1 Parcours de graphes et applications

Parcours en largeur

Parcours en profondeur *Stratégie du parcours* : Parcourir l'ensemble des nœuds du graphe en privilégiant la profondeur : on va le plus loin possible dans le dit graphe.

Astuces : On utilise une coloration des nœuds du graphe afin de savoir quels sont les nœuds que nous avons déjà pris : pas étudier, en cours, fini. Cette coloration nous permet de nous assurer d'avoir parcouru tous les nœuds et de l'avoir fait une unique fois. On utilise aussi des dates de début et de fin de traitement pour chacun des nœuds. Ces dates jouent le même rôle que les couleurs et peuvent être exploité dans certaines applications comme le calcul de composantes fortement connexes. Ces deux astuces ne sont pas obligatoirement utiliser en même temps même si l'une ou l'autre peut ainsi simplifier les preuves de correction, de terminaison ou de complexité.
Applications du parcours en profondeur :

- Tri topologique : à la sortie de VISITER, on ajoute le sommet que l'on vient de visiter dans une liste.
- Composantes fortement connexes : on effectue deux passages de l'algorithme (un sur le graphe et le deuxième sur le graphe complémentaire dont l'ordre de l'étude des sommets est donné par les dates calculées lors du premier parcours.)

Présentation de l'algorithmique autours du parcours en profondeur de graphe Nous présentons ici un algorithme récursif (Algorithme 1) du parcours de graphe en profondeur [2, p.558]. Cet algorithme fait appelle à une fonction auxiliaire (qui porte la récursivité) VISITE (Algorithme 2) qui traite chacun des sommets du graphe et décide de l'ordre dans lequel on les traites. Nous prouverons également la correction et la terminaison de cet algorithme. Nous finirons par étudier sa complexité. La figure 1 nous donne un exemple d'application de l'algorithme de parcours en profondeur.

Algorithm 1 Le parcours en profondeur d'un graphe.

```
1: function PARCOURS-PROFONDEUR(G)
2:     for tout u ∈ G.V do
3:         u.couleur ← blanc
4:         u.π ← NIL                ▷ Prédécesseurs
5:     end for
6:     date ← 0
7:     for tout u ∈ G.V do
8:         if u.couleur = blanc then
9:             VISITE(G, u)
10:        end if
11:    end for
12: end function
```

Algorithm 2 Visiter un sommet : traiter ce sommet lors du parcours

```
1: function VISITE(G, u)
2:     date ← date + 1   ▷ u juste découvert
3:     u.d ← date
4:     u.couleur = gris
5:     for tout v ∈ Adj(u) do
6:         if v.couleur = blanc then
7:             v.π ← u
8:             VISITE(G, u)
9:         end if
10:    end for
11:    u ← noir
12:    date ← date + 1
13:    u.f ← date
14: end function
```

Théorème (Terminaison)**.** *Soit $G = (V, E)$ un graphe. Le parcours en profondeur sur ce graphe termine.*

Démonstration. La terminaison provient du fait qu'on appel la fonction Visite au plus $|V|$. □

Remarque. L'ordre dans lequel les sommets sont examinés influe sur la sortie de l'algorithme. Mais en pratique, cet ordre nous importe peu car les sorties possibles sont équivalentes.

Théorème (Complexité)**.** *Soit $G = (V, E)$ un graphe. Le parcours en profondeur sur ce graphe se fait en $O(|V| + |E|)$.*

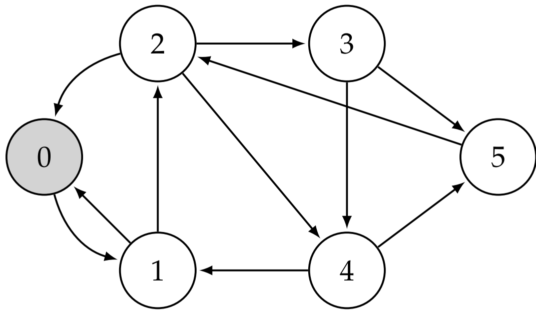
Démonstration. On applique la méthode de l'agrégat. On applique exactement $|V|$ fois la procédure Visite. De plus, pendant tout le parcours, la boucle dans Visite s'exécute $\sum_{v \in S} |Adj(v)| = \Theta(|E|)$. Le temps d'exécution du parcours en profondeur est $\Theta(|S| + |E|)$. □

Quelques propriétés que l'on peut déduire du parcours Le parcours en profondeur d'un graphe révèle quelques unes de ces structures.

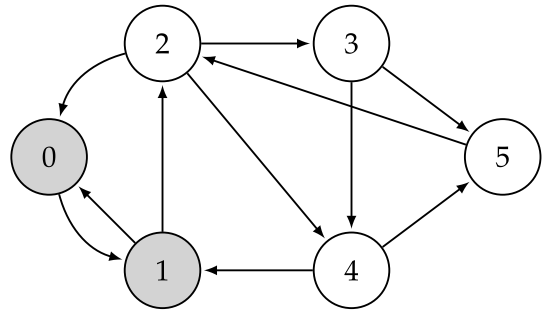
Théorème (Théorème des parenthèses)**.** *Dans un parcours en profondeur d'un graphe $G = (V, E)$, pour deux sommets quelconques u et v, une et une seule des trois conditions suivantes est vérifiée :*

- *les intervalles $[u.d, u.f]$ et $[v.d, v.f]$ sont disjoints, et ni u ni v n'est un descendant de l'autre dans la forêt de parcours en profondeur ;*
- *l'intervalle $[u.d, u.f]$ est entièrement inclus dans l'intervalle $[v.d, v.f]$, et ni u est un descendant de v dans un arbre de parcours en profondeur ;*
- *l'intervalle $[v.d, v.f]$ est entièrement inclus dans l'intervalle $[u.d, u.f]$, et ni v est un descendant de u dans un arbre de parcours en profondeur.*

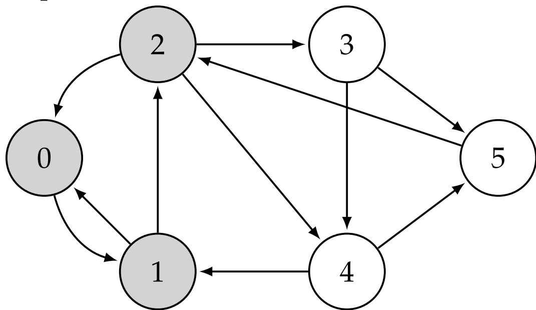
Démonstration. Supposons que $u.d < u.f$. On distingue alors deux sous-cas. Si $v.d < u.f$, alors v a été découvert pendant que u était encore gris (u est cours de traitement). Donc v est un descendant de u. Puisque la découverte de v est plus récente que u, le traitement de v se termine avant le traitement de u. Donc, l'intervalle $[v.d, v.f]$ est entièrement inclus dans l'intervalle $[u.d, u.f]$.



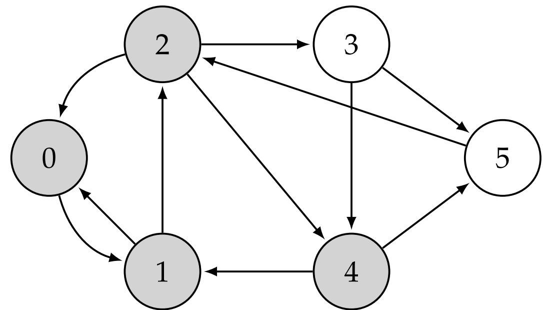
(a) Graphe sur lequel on applique le parcours en profondeur



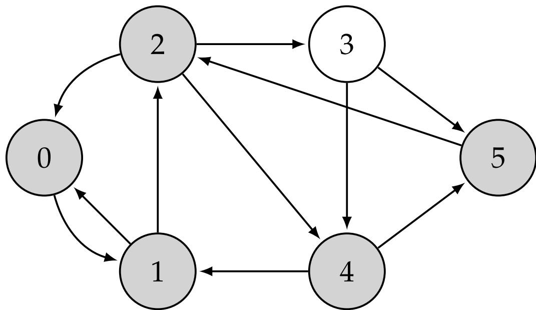
(b) Visite du sommet 1.



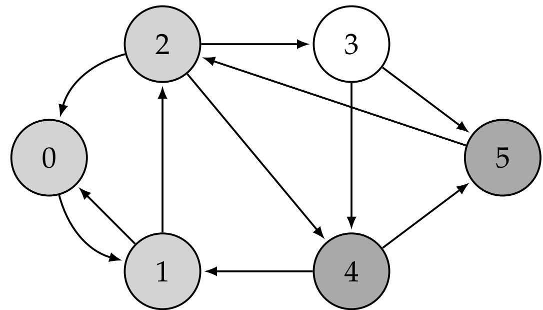
(c) Visite du sommet 2.



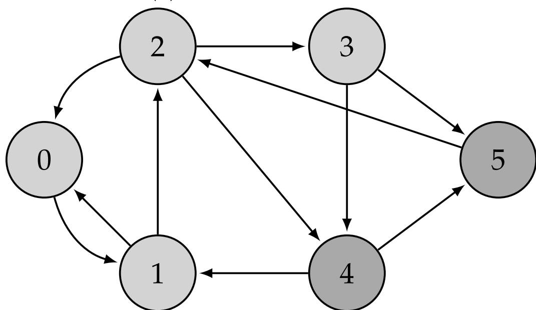
(d) Visite du sommet 4.



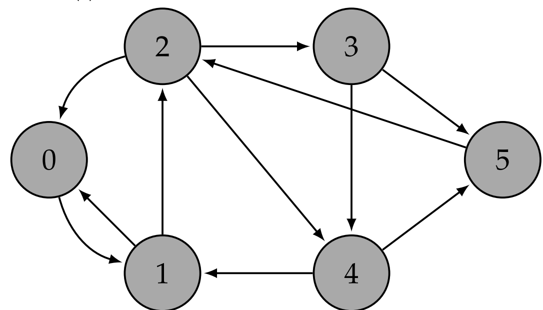
(e) Visite du sommet 5.



(f) Fin des visites des sommets 4 et 5.



(g) Visite du sommet 3.



(h) Fin des visites des sommets 3, 2, 1 et 0.

FIGURE 1 – Le parcours en profondeur d'un graphe. On obtient alors : 0, 1, 2, 4, 5 et 3.

Dans l'autre sous-cas, $u.f < v.d$ ce qui implique que $u.d < u.f < v.d < v.f$ et donc les intervalles $[u.d, u.f]$ et $[v.d, v.f]$ sont disjoints. Comme les intervalles sont disjoints, aucun des deux sommets n'a été découvert pendant que l'autre était gris. Donc, aucun sommet n'est un descendant de l'autre.

Dans le cas $v.d < u.d$, on raisonne de manière analogue en inversant les rôles de u et v. □

Corollaire (Imbrication des intervalles descendants). *Le sommet v est un descendant propre du sommet u dans la forêt de parcours en profondeur d'un graphe G si et seulement si $u.d < v.d < v.f < u.f$.*

Démonstration. Immédiat par le théorème précédent. □

Théorème (Théorème du chemin blanc). *Dans une forêt de parcours en profondeur d'un graphe $G = (V, E)$, un sommet v est un descendant d'un sommet u si et seulement si, au moment $u.d$ où le parcours découvre u, il existe un chemin de u à v composé uniquement de sommets blancs.*

Démonstration. $\Rightarrow$: Si $v = u$, alors le chemin de u à v contient uniquement le sommet u, qui est encore blanc quand on définit la valeur de $u.f$. Supposons que v soit un descendant propre de u dans la forêt de parcours en profondeur. D'après ca qui précède, $u.d < v.d$ et donc v est blanc à l'instant $u.d$. Comme v peut être un descendant quelconque de u, tous les sommets situés sur le chemin simple unique entre u et v dans la forêt de parcours sont blanc à l'instant $u.d$.

$\Leftarrow$: Supposons qu'à l'instant $u.d$ il existe un chemin constitué de sommets blanc entre u et v, mais que v ne devient pas un descendant de u dans l'arbre de parcours en profondeur. Sans perte de généralité, on suppose que tous les autres sommets de ce chemin (sauf v) deviennent un descendant de u dans l'arbre. Si ce n'est pas le cas, on choisi le plus proche de u qui vérifie cette propriété. Soit w le prédécesseur de v dans ce chemine (u et w peuvent être le même chemin). D'après ce qui précède, $w.f \leq u.f$. Comme v doit être découvert après u, mais avant que le traitement de w soit terminé, on a $u.d < v.d < w.f \leq u.f$. Le théorème des intervalles implique alors que l'intervalle $[v.d, v.f]$ est entièrement inclus dans l'intervalle $[u.d, u.f]$. Donc, par le corollaire, v est un descendant de u. □

Classification des arcs Le parcours en profondeur peut permettre de classer les arcs du graphe $G = (V, E)$. Ce type peut fournir des informations sur le graphe.

Définition (Classification des arcs). Soit $G = (V, E)$ un graphe. Le parcours en profondeur donne une classification des arcs du graphe.

1. Les arcs de liaison sont les arcs de la forêt de parcours en profondeur. L'arc (u, v) est un arc de liaison si v a été découvert la première fois pendant le parcours de l'arc (u, v).
2. Les arcs arrière sont les arcs (u, v) reliant un sommet u à un ancêtre v dans un arbre de parcours en profondeur. Les boucles (dans un graphes orienté) sont considérées comme des arcs arrière.
3. Les arcs avant sont les arcs (u, v) qui ne sont pas des arcs de liaison et qui relient un sommet u à un descendant v dans un arbre de parcours en profondeur.
4. Les arcs transverse sont tous les autres arcs. Ils peuvent relier deux sommets d'un même arbre de parcours en profondeur, du moment que l'un des sommets n'est pas un ancêtre de l'autre. Ils peuvent aussi deux sommets appartenant à des arbres de parcours en profondeur différents.

Lors du parcours en profondeur du graphe, on peut déterminer la nature de l'arc. Lorsque l'arc est exploré pour la première fois :

1. blanc indique un arc de liaison ;
2. gris indique un arc arrière ;
3. noir indique un arc avant ou de transverse.

Proposition. *Dans un parcours en profondeur d'un graphe non orienté G, chaque arête de G est soit un arc de liaison, soit un arc arrière.*

Démonstration. Soit (u, v) un arc de G tel que $u.d < v.d$. Alors, v doit être découvert et son traitement terminé avant le traitement de u (pendant que u est gris) (v se trouve dans la liste d'adjacence de u). Si l'arête (u, v) est d'abord exploré dans le sens u vers v, alors v n'a pas été découvert (blanc) jusqu'ici (sinon on aurait déjà exploré cet arête dans la direction de v vers u). Dans ce cas, (u, v) est un arc de liaison.

Si (u, v) est exploré dans l'autre sens, on a un arc arrière car u est encore gris lors de la première exploration. □

Proposition. *Un graphe G est acyclique si et seulement si un parcours en profondeur de G ne génère aucun arc arrière.*

Démonstration. $\Rightarrow$: Supposons qu'un parcours en profondeur produise un arc arrière. Alors, le sommet v est un ancêtre du sommet u dans la forêt de parcours en profondeur. Donc, il existe un chemin v à u dans G et l'arc arrière complète le cycle partant de v.

$\Leftarrow$: Supposons que G contienne un cycle c. On va montrer qu'un parcours en profondeur de G génère un arc arrière. Soit v le premier sommet découvert dans c et soit (u, v) l'arc précédent dans c. A l'instant $v.d$, les sommets de c forment un chemin entre v et u composé de sommets blanc. D'après le théorème du chemin blanc, le sommet u devient un descendant de v dans la forêt de parcours en profondeur. Donc (u, v) est donc un arc arrière. □

Structures de données et algorithmes affiliés Le parcours en profondeur peut être écrit de manière impérative à l'aide d'une pile. Dans le cas de l'algorithme récursif, la pile est caché dans le boucle pour et de la fonction Visite.

Références

[1] D. Beauquier, J. Berstel, and P. Chrétienne. *Eléments d'algorithmique*. Manuels informatiques Masson. Masson, 1992.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Algorithmique, 3ème édition*. Dunod, 2010.

[3] C. Froidevaux, M.C. Gaudel, and M. Soria. *Types de données et algorithmes*. McGraw-Hill, 1990.