ALGORITHMES SUR LES GRAPHES

I – DÉFINITONS, TERMINOLOGIE

(1) Définitions et Notations

(2) Adjacences, Voisinages, Degrés

(3) Cheminements, Connexités

(4) Graphes Particuliers

(5) Structures de Données

II – PARCOURS DE GRAPHES

(1) Parcours en Largeur

(2) Parcours en Profondeur

(3) Applications Directes

(4) Prolongements

I – DÉFINITIONS, TERMINOLOGIE

La notion de Graphe permet de modéliser de très nombreuses situations concrètes, voir par exemple le cours de Recherche Opérationnelle. Tous les cas étudiés mettent en évidence des « objets » et des « liaisons » entre ces objets donc une **relation binaire sur cet ensemble d'objets**, une telle structure s'appelle un graphe.

(1) Définitions et Notations Élémentaires

Graphe simple non orienté G = (X, E)

La relation binaire est symétrique.

X = {1, 2, … , n} est l'ensemble des **sommets** (on nœuds ou points), **| X| = n**.

E est un ensemble de **paires** sur X, ce sont les **arêtes**. L'arête {x, y}, où x ≠ y, est notée xy ou yx. Par convention **| E | = m**. Le mot « simple » signifie qu'il y a au plus une arête entre deux sommets.

Graphe simple orienté G = (X, U)

X = {1, 2, … , n} est l'ensemble des **sommets**.

U est un ensemble de **couples** (x, y) où x ≠ y, on les appelle **arcs**, là aussi on note **| U | = m**.

Entre deux sommets x et y on peut avoir l'arc xy et l'arc yx, mais au plus un arc dans chaque sens.

Les graphes que nous avons définis sont sans boucle, c'est-à-dire sans arête ou arc du type xx. Ce sera le cas pour nous dans la suite. Mais ce n'est pas forcément toujours le cas dans certaines applications. On pense par exemple aux **Automates Finis**, en Théorie des Langages.

Multigraphe

Un multigraphe (orienté ou non) est un graphe où l'on s'autorise autant d'arêtes ou d'arcs que l'on veut, mais en nombre fini, entre deux sommets. S'il y a au plus k arêtes ou arcs entre deux sommets, k étant un entier positif, on parle de **k – graphe**. Dans un graphe simple k = 1.

Graphes et Multigraphes Valués

Si les sommets et/ou les arêtes (ou arcs) sont munies de **valuations**, on a à faire à des graphes ou multigraphes valués : G = (X, E, v, d) où v et d sont respectivement des applications de X vers R^k et de E vers R^k.

Ces valuations peuvent être des capacités de stockage, des coordonnées, des durées, … pour les sommets et des capacités de transport, des distances, des coûts … pour les arêtes ou les arcs.

Sous Structures, Graphes Complets, Graphes Stables

Soit G = (X, E), dans la mesure où G est défini par un couple, il existe plusieurs façons naturelles de définir une sous structure d'un graphe.

G' = (X', E') est un **graphe partiel de G** si X' = X et E' ⊆ E, autrement dit les sommets sont les mêmes mais des arêtes peuvent manquer.

G'' = (X'', E'') est un **sous graphe de G** (plus précisément le **sous graphe engendré** par X'') si X'' ⊆ X et E'' = E ∩ P_2(X''), c'est à dire qu'on ne garde que les arêtes ayant des extrémités dans X''.

G''' = (X''', E''') est un **sous graphe partiel** de G si X''' ⊆ X et E''' ⊆ E, c'est-à-dire le graphe partiel d'un sous graphe.

Ces définitions s'adaptent immédiatement aux cas des graphes orientés et des multigraphes.

On dit qu'un graphe G = (X, E) est **complet** (ou est une **clique**) si tous ses sommets sont reliés deux à deux, autrement dit E = P_2(E), on note **G = K_n**, il y a alors m = 1/2 . n(n – 1) arêtes.

On dit que G = (X, E) est un **stable** si E = Ø, autrement dit s'il n'a aucune arête, on note **G = S_n**.

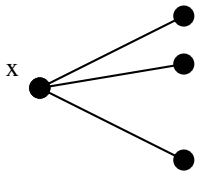
(2) Adjacences, Voisinages, Degrés

Soit **G = (X, E)** et x ∈ X, on note **V(x)** = {y ∈ X | xy ∈ E} les **Voisins de x** et **d(x)** le **degré de x** qui est le nombre d'arêtes adjacentes en x. Si le graphe est simple, au plus une arête entre deux sommets, on a d(x) = | V(x) |.

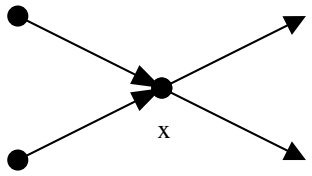
Soit **G = (X, U)** et x ∈ X, on note $\mathbf{V^-(x)}$ = {z ∈ X | zx ∈ u} l'ensemble des **prédécesseurs** de x et $\mathbf{V^+(x)}$ = {xy ∈ X | xy ∈ U} l'ensemble des **successeurs** de x, note également $\mathbf{d^-(x)}$, le **degré intérieur**, c'est le nombre d'arcs arrivant sur x et $\mathbf{d^+(x)}$, le **degré extérieur**, c'est le nombre d'arcs sortant de x. Et $\mathbf{d(x) = d^-(x) + d^+(x)}$.

Si le graphe est simple, au plus un arc entre tout couple de sommets, on a $d^-(x) = |V^-(x)|$ et $d^+(x) = |V^+(x)|$

.



V(x)



$V^-(x)$ $V^+(x)$

Propriété

$$\sum_{x \in X} d(x) = 2m \qquad et \qquad \sum_{x \in X} d^-(x) = \sum_{x \in X} d^+(x) = m$$

(3) Cheminements, Connexités

Soit **G = (X, E)**, on appelle **chaîne** toute suite $x_1, x_2, \ldots, x_k$ de sommets telle que $x_i x_{i+1} \in E$, sa longueur est k – 1 (le nombre d'arêtes). Une chaîne est dite **simple** (resp. **élémentaire**) si elle n'utilise pas deux fois la même arête (resp. le même sommet). Un **cycle** est une chaîne simple où l'origine et l'extrémité coïncident.

Soit **G = (X, U)**, on appelle **chemin** toute suite $x_1, x_2, \ldots, x_k$ de sommets telle que $x_i x_{i+1} \in U$, sa longueur est k – 1 (le nombre d'arcs). On définit de même les notions de simplicité et d'élémentarité en remplaçant arête par arc.

Un **circuit** est un chemin simple dont l'extrémité coïncide avec l'origine.

Propriété

Tout graphe orienté fini sans circuit admet une source, c'est-à-dire un sommet sans prédécesseur.

On dit qu'un graphe G = (X, E) est **Hamiltonien** (resp. **Pseudo Hamiltonien**) s'il admet un cycle (resp. une chaîne) passant une fois et une seule par chacun de ses sommets.

On dit qu'un graphe G = (X, E) est **Eulérien** (resp. **Pseudo Eulérien**) s'il admet un cycle (resp. une chaîne) passant une fois et une seule par chacune de ses arêtes.

On définit de même ces notions sur les graphes orientés (avec des circuits et des chemins).

Alors qu'il existe une bonne caractérisation des Graphes Eulériens, c'est-à-dire conduisant à des algorithmes très efficaces de vérification et de construction d'un cycle eulérien, il n'existe, à ce jour aucune méthode rapide (c'est à dire polynomiale en n et m) pour vérifier si un graphe est Hamiltonien : ce problème est **NP - Complet**.

Soit **G = (X, E)** et x, y deux sommets quelconques, on dit que x et y sont **connectés** si il existe une chaîne de x à y. Le graphe G est dit **connexe** si ∀x, y ∈ X x et y sont connectés.

Propriété

Tout graphe connexe vérifie :

$$n - 1 \leq m \leq 1/2 \ . \ n(n-1)$$

Si G n'est pas connexe, on peut écrire $\mathbf{G = G_1 \oplus G_2 \oplus \dots \oplus G_k}$ (somme disjointe) avec $k \geq 2$ où les G_i sont les **Composantes Connexes** c'est-à-dire les sous graphes connexes maximaux de G.

Ces composantes peuvent aussi se définir de la façon suivante : soit R la relation sur X définie par x R y si et seulement si x = y ou x est connecté à y. C'est une relation d'équivalence (réflexive, symétrique et transitive).

Les classes d'équivalence $X_1, X_2, \dots, X_k$ (partition de X) correspondent aux sous ensembles de sommets des composantes connexes $G_1, G_2, \dots$ et G_k de G : G_i est le sous graphe engendré par X_i.

La notion de connexité, dans un graphe simple non orienté, peut s'étendre à la **k-connexié** et à la **k- arête-connexité** ($k \geq 2$) : un graphe est **k-connexe** (resp. **k-arête-connexe**) si entre 2 sommets quelconques x et y il existe au moins k chaînes élémentaires disjointes (sauf en x et y) au sens des sommets (resp. au sens des arêtes).

On dit que **G = (X, U)** est **Fortement Connexe** si entre tout couple de sommets x et y il existe à la fois un chemin de x à y et un chemin de y à x.

Si G n'est pas fortement connexe, il est décomposable en sous graphes maximaux fortement connexes, ce sont ses **Composantes Fortement Connexes**, qu'on peut définir comme les sous graphes engendrés par les classes d'équivalence de la relation binaire R définie sur X par : x R y si et seulement si x = y ou il existe un chemin de x à y et un chemin de y à x.

Si $G_1, G_2, \dots, G_k$ ($k \geq 2$) sont les composantes fortement connexes de G on peut définir le **Graphe Réduit** de G, $\mathbf{G_R = (X_R, U_R)}$, comme le graphe quotient de G par ses composantes fortement connexes :

$$\mathbf{X_R = \{X_1, X_2, \dots, X_K\}}$$

$$\mathbf{U_R = \{ X_iX_j \mid \exists\ x \in X_i \text{ et } \exists\ y \in X_j \text{ tels que } xy \in U \}}$$

Autrement dit il y a un sommet par composante fortement connexe et deux composantes sont reliées s'il existe un arc les reliant dans G.

Propriété

G_R est un graphe simple orienté sans circuit.

(4) Graphes Particuliers : Arbres, Forêts, Arborescences, Graphes Bipartis, Graphes Planaires

Un graphe simple non orienté est un **Arbre** s'il est **connexe** et **sans cycle** (à ne pas confondre avec une arborescence, voir plus bas). Une **Forêt** est un graphe dont chacune des composantes connexes est un arbre.

Propriété

Pour G = (X, E) un graphe simple non orienté, les 6 propriétés suivantes sont équivalentes :

- ***G est un arbre (par définition : connexe et sans cycle)***
- ***G est sans cycle et m = n –1***
- ***G est connexe et m = n – 1***
- ***G est sans cycle et l'ajout d'une arête quelconque entre deux sommets créé exactement un cycle***
- ***G est connexe et la suppression d'une arête quelconque le déconnecte***
- ***Toute paire de sommets est reliée par une chaîne unique***

Un graphe simple orienté, **G = (X, U)**, est une **Arborescence** s'il admet une **Racine** (c'est à dire un sommet **r** tel qu'il existe un chemin de r à tout autre sommet) et si le graphe non orienté correspondant (on dit aussi le graphe non orienté sous-jacent) est un arbre.

Un graphe **G = (X, E)** est **Biparti** si X peut être partitionné en deux sous ensembles **Stables** A et B, autrement dit toute arête n'existe qu'entre A et B.

On note $K_{p,q}$ le **Graphe Biparti Complet**, c'est-à-dire le graphe où $|A| = p$ et $|B| = q$ ayant p . q arêtes : tout sommet de A est relié à tout sommet de B.

Propriété

Pour tout graphe simple non orienté G = (X, E) les trois propriétés suivantes sont équivalentes :

- ***G est biparti***
- ***G est 2-colorable***
- ***G n'a pas de cycle impair***

On dit qu'un graphe **G = (X, E)** est **Planaire** s'il peut être représenté dans le plan en utilisant des points pour les sommets et des segments de droite pour les arêtes sans que deux arêtes ne se coupent. La représentation du graphe partitionne le plan en régions connexes (au sens topologique du terme) qu'on appelle les **Faces**. Notons **f** le **nombre de faces** (y compris la face extérieure).

Le célèbre Théorème suivant relie les nombres de sommets, d'arêtes et de faces :

Théorème (Relation d'EULER)

Tout graphe planaire connexe vérifie n – m + f = 2

Corollaire

Tout graphe planaire connexe ayant au moins 3 sommets vérifie l'inégalité $m \leq 3n - 6$, de plus si tous ces cycles ont au moins quatre arêtes alors $m \leq 2n - 4$

On en déduit que **K_5 et $K_{3,3}$ ne sont pas planaires**. (*)

Parmi les caractérisations des graphes planaires la plus célèbre est sans doute la suivante :

Théorème (KURATOWSKI, 1930)

Un graphe est planaire si et seulement si il ne contient pas de subdivision (graphe obtenu en rajoutant autant de sommets que l'on veut sur des arêtes) de K_5 ni de $K_{3,3}$

Exercice

Prouver les **Propriétés** énoncées p. 2, 3 et 4 (il y en a 6) ainsi que le **Corollaire** ci dessus et le corollaire du corollaire : l'affirmation (*).

Remarques

Attention aux notations : G = (X, E) signifie non orienté, tandis que G = (X, U) signifie orienté.

Certaines notions sont non orientées (ex. Chaîne, Cycle, Arbre, Connexité, …), d'autres sont orientées (ex. Chemin, Circuit, Arborescence, Forte Connexité, …).

Dans les définitions données a priori tous les graphes sont simples (au plus une arête ou un arc dans chaque sens entre deux sommets) sauf mention du contraire, même ci certaines définitions peuvent s'étendre à des multigraphes orientés ou non.

(5) Structures de Données

Pour pouvoir manipuler un graphe, c'est-à-dire le stocker, lui appliquer un algorithme, … il faut en avoir une représentation en machine, les deux principales structures de données sont soit la **Matrice d'Adjacence** soit le **Tableau des Listes d'Adjacence**, si le graphe est non orienté, ou le **Tableau des Listes de Successeurs** ou de **Prédécesseurs**, s'il est orienté.

Quitte à renuméroter les sommets nous supposerons que $\mathbf{X = \{1, 2, \dots, n\}}$.
Nous noterons comme d'habitude $\mathbf{|E| = |U| = m}$.

Matrice d'Adjacence

La **Matrice d'Adjacence** $\mathbf{M_G}$ d'un graphe G est une matrice booléenne de taille n.n définie par :

$$M_G(i,j) = \begin{cases} 1 & si\ ij \in G \\ 0 & si\ ij \notin G \end{cases}$$

Si **G est non orienté** la matrice est symétrique, la ligne ou la colonne du sommet i représente ses voisins.
Si **G est orienté**, la ligne du sommet i permet de repérer ses successeurs, la colonne du sommet i permet d'obtenir ses prédécesseurs.

Dans les deux cas, la taille de cette structure est $\mathbf{n^2}$.
Remarquons que cette structure n'est pas très adaptée au stockage d'un **Arbre** ou d'un **Graphe Planaire** par exemple, pourquoi ?

Listes d'Adjacence, Listes de Successeurs ou de Prédécesseurs

Le **Tableau des Listes d'Adjacence** est un tableau **G(1), G(2), … , G(n)**, indicé sur la numérotation des sommets du graphe, tel que G(i) contient un pointeur vers la liste, **V(i)**, des **Voisins du sommet i** lorsque le graphe est non orienté ou la **Liste des Successeurs**, $\mathbf{V^+(i)}$ (ou la **Liste des Prédécesseurs**, $\mathbf{V^-(i)}$) lorsque le graphe est orienté.

La taille de cette structure est en $\mathbf{\Theta(n + m)}$, plus exactement n + 4m cases mémoires si G est non orienté et n + 2m cases mémoires si G est orienté. Pourquoi ?

Structures pour les Multigraphes et les Graphes Valués

Si G est un **multigraphe** (qu'il soit orienté ou non) alors $\mathbf{M_G(i, j) = m_{i,j}}$ où $m_{i,j}$ est la **multiplicité** de l'arête ou de l'arc ij. Dans les listes on rajoutera dans chaque structure un champ contenant cette multiplicité.

Lorsque le graphe est **valué** alors $\mathbf{M_G(i, j) = d(i, j)}$, qui est la **valuation de l'arête ou de l'arc** et dans les listes on rajoute un champ où figure cette valuation.

Une autre structure peut tout simplement consister à faire la liste des arêtes ou des arcs, avec leurs multiplicités ou leurs valuations éventuelles.

Mais retenons que la structure choisie dépendra du problème à résoudre, du type de graphe à traiter ainsi que des opérations nécessaires à sa résolution et des performances globales attendues de l'algorithme utilisé.

II – PARCOURS DE GRAPHES

Les parcours de graphes qui sont des méthodes systématiques d'exploration de tous les sommets et de toutes les arêtes ou des arcs d'un graphe sont à la base de presque tous les algorithmes de résolution de problèmes se modélisant sous la forme de graphe.

Il existe essentiellement deux techniques d'exploration d'un graphe :

- le **Parcours en Largeur** ou concentrique (Breadth-First Search, BFS)
- le **Parcours en Profondeur** (Depth-First Search, DFS)

(1) Parcours en Largeur

Principe

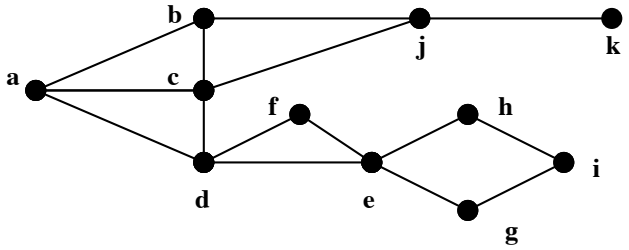
On choisit un sommet initial x qui devient visité, puis à partir de x on visite tous ses voisins (ou ses successeurs, si le graphe est orienté), puis tous les voisins (ou successeurs) non visités des voisins (ou successeurs) et ainsi de suite jusqu'à visiter tous les sommets du graphe, auquel cas le parcours est terminé ou bien le parcours reprend en un sommet non encore visité, s'il en existe.

Par cette méthode tous les sommets et toutes les arêtes (ou arcs) sont examinés et il faudra une procédure de marquage des sommets afin de reconnaître les sommets visités.

Examinons ce parcours sur le cas d'un graphe non orienté puis d'un graphe orienté.

Cas d'un graphe non orienté

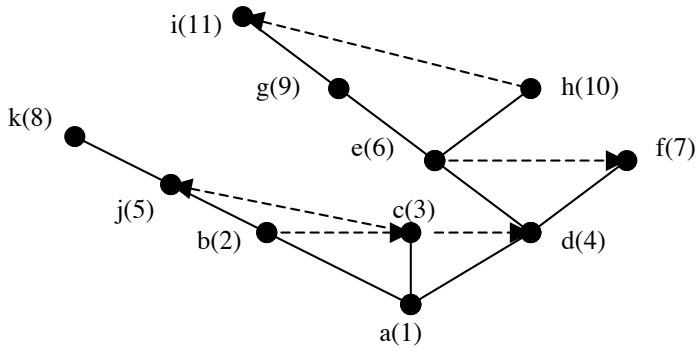
Soit G = (X, E) le graphe suivant :



On suppose G donné par ses listes d'adjacence (le graphe a n = 11 sommets et m = 15 arêtes, la matrice d'adjacence serait de taille 121):

a : b, c, d	**d** : a, c, e, f	**g** : e, i	**j** : b, c, k
b : a, c, j	**e** : d, f, g, h	**h** : e, i	**k** : j
c : a, b, d, j	**f** : d, e	**i** : g, h	

Partant du sommet a on obtient l'arborescence suivante :



Les sommets sont numérotés dans l'ordre de visite. Le parcours oriente naturellement chaque arête en arc dans le sens de sa première exploration (ce sont les segments dessinés en traits pleins, orientés du bas vers le haut) et lui associe donc une **Arborescence de racine a**.

Les arêtes restantes, elles aussi devenues arcs, ce sont les **co-arcs**, sont représentées en pointillé.

Algorithme

On suppose que $\mathbf{X = \{1, 2, \ldots, n\}}$. L'algorithme nécessite un tableau **n°visite(1)**, **n°visite(2)**, … , **n°visite(n)**, indicé par les sommets, initialisé à 0, stockant l'ordre de visite et permettant de savoir si un sommet x a déjà été visité, c'est-à-dire quand n°visite(x) ≠ 0.

La gestion des sommets se fait en utilisant un **File de Priorité FIFO** (first in, first out) : à chaque étape on explore les voisins du sommet stocké en début de file qu'on enlève de la file et chaque nouveau sommet est mis en fin de file. Cette structure peut être représentée soit par une liste soit par un tableau.

L'arborescence (ou la forêt d'arborescences) du parcours, **F**, peut être construite au cours de l'algorithme et peut être représentée soit par un tableau de listes de successeurs soit par un tableau stockant pour chaque sommet son père dans l'arborescence.

L'algorithme peut alors s'écrire :

```
N° ← 1
FILE ← Ø
pour i ← 1 à n faire n°visite(i) ← 0
pour i ← 1 à n faire
        si n°visite(i) = 0 alors
                n°visite(i) ← N°
                N° ← N° + 1
                FILE ← FILE + {i}
                Largeur(i)

Largeur(i)
        tant que FILE ≠ Ø faire
                x ← premier (FILE)
                FILE ← FILE – {x}
                pour « chaque voisin y de x » faire
                        si n°visite(y) = 0 alors  n°visite(y) ← N°
                                                  N° ← N° + 1
                                                  FILE ← FILE + {y}
                                                  F(x) ← F(x) + {y}
```

Propriétés

- *L'algorithme visite tous les sommets du graphe et chacun d'eux apparaît exactement une fois dans la file (la taille de la file est donc O(n)).*

- *Chaque arête du graphe est considérée deux fois (une fois dans chaque sens) par l'algorithme.*

- ***Largeur(i)** visite tous les sommets connectés à i et fournit donc la composante connexe correspondante C. **Largeur(i)** construit une arborescence recouvrante de C de racine i. Toute autre arête de C (co-arc) relie deux sommets de même niveau ou de niveaux consécutifs dans l'arborescence.*

- *L'algorithme fournit une arborescence recouvrante (en fait un **arbre recouvrant**, c'est-à-dire un graphe partiel qui est un arbre) si le graphe est connexe ou une forêt recouvrante, à raison d'une arborescence (ou arbre) par composante connexe.*

Complexités

D'après les propriétés précédentes le parcours en largeur est en $\Omega(n+m)$ en espace et en temps.

Propriété

Si G est représenté par un tableau de listes d'adjacence alors le parcours en largeur peut être implémenté en $\Theta(n + m)$ en espace et en temps.

Preuve

Les opérations de base sont :

- soit l'accès à un élément de tableau soit l'ajout ou la suppression d'un élément en début ou en fin de liste ou de tableau, elles sont toutes en O(1)
- soit le parcours d'une liste de p éléments (pour l'accès aux voisins d'un sommet), ce qui se fait en O(p)
- soit des opérations d'incrémentation, d'affectation ou de gestion de boucle, d'appel de fonction, … toutes faisables en O(1).

Soient $G_1 = (X_1, E_1)$, $G_2 = (X_2, E_2)$, … , $G_k = (X_k, E_k)$ les composantes connexes de G. Si on a $|X_i| = n_i$ et $|E_i| = m_i$ alors $\mathbf{n_1 + \ldots + n_k = n}$ et $\mathbf{m = m_1 + \ldots + m_k = m}$.

Largeur(1) visite tous les sommets de sa composante connexe, soit G_1. Pour chaque sommet x de G_1, x est mis une fois dans la **FILE**, l'algorithme passe en revue tous ses voisins et effectue sur chacun un travail constant.

Donc :

$$coût(L\arg eur(1)) = \sum_{x \in X_1} (cte + cte'.d(x)) = am_1 + bn_1 \in \Theta(n_1 + m_1)$$

où cte, cte', a et b sont des constantes strictement positives.

La deuxième boucle du programme principal passe en revue tous les sommets du graphe, mais n'appelle la fonction **Largeur** qu'une seule fois sur chaque composante connexe de G. Donc globalement (avec les initialisations) la **Complexité en Temps** est :

$$T(n,m) = cte + cte'.n + \sum_{i=1}^{k} (am_i + bn_i) = \alpha m + \beta n + \gamma \in \Theta(n+m)$$

où α, β et γ sont des constantes strictement positives.

La **Complexité en Espace** est aussi en Θ(n + m) car :

- la structure de donnée de G est en Θ(n + m)
- la FILE et le tableau n°visite sont en Θ(n)
- la forêt d'arborescence est représentable en Θ(n)
- il y a enfin un nombre constant de variables auxiliaires.

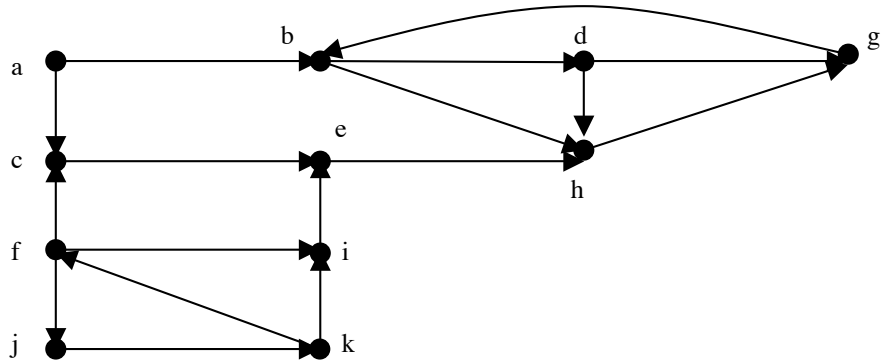
Remarque

Si G est représenté par sa matrice d'adjacence alors les complexités en temps et en espace sont en …

Cas d'un graphe orienté

Soit G = (X, U) le graphe suivant :

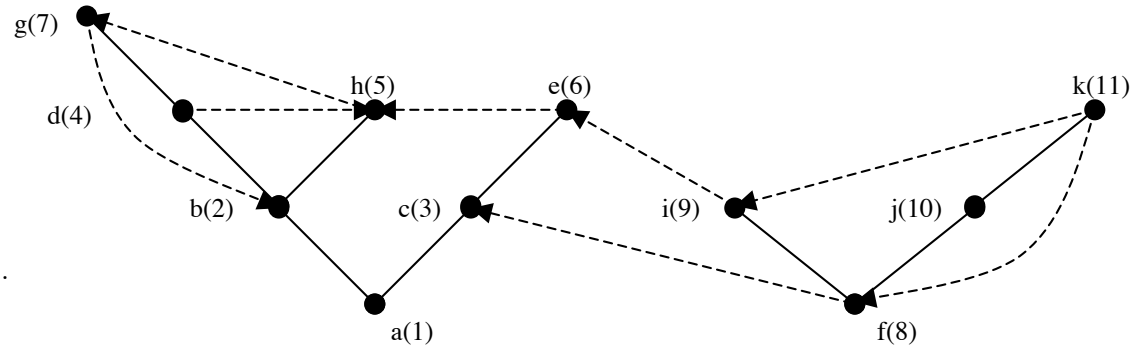
.



Les listes de successeurs (le graphe a n = 11 sommets et m = 17 arcs, la matrice d'adjacence serait de taille 121) sont les suivantes :

a : b, c	**d** : g, h	**g** : b	**j** : k
b : d, h	**e** : h	**h** : g	**k** : f, i
c : e	**f** : c, i, j	**i** : e	

Le parcours en largeur, appliqué en respectant l'orientation, avec les listes de successeurs, à partir de a, donne la forêt d'exploration :



.

Les sommets sont numérotés dans l'ordre de visite. La Forêt du Parcours est obtenue en retenant les arcs explorés pour la première fois (arcs représentés en traits pleins, orientés implicitement de bas en haut), les arcs restants, les co-arcs, sont représentés en pointillé.

Remarquons que bien que le graphe soit connexe, au sens non orienté, le parcours donne une forêt de deux arborescences.

Algorithme

Il est identique à l'algorithme donné pour le cas non orienté en remplaçant voisins par successeurs (ou éventuellement prédécesseurs).

Propriétés

- *L'algorithme visite tous les sommets du graphe, chacun d'eux apparaissant une et une seule fois dans la FILE, et examine chaque arc exactement une fois.*

- *L'arborescence de racine 1 (le premier sommet visité) obtenu par* ***Largeur(1)*** *à partir des listes de successeurs (resp. des prédécesseurs) contient tous les* ***Descendants*** *(resp.* ***Ascendants****) de 1 dans G. Mais ce n'est pas vrai pour les autres sommets dans la suite du parcours. Si x est un sommet du graphe on dit qu'un sommet y (resp. z) est* ***un descendant*** *de x (resp.* ***un ascendant*** *de x) si il existe dans G un chemin de x à y (resp. de z à x).*

- *Le parcours construit une Forêt d'Arborescences qui recouvre le graphe.*

Complexités

Le raisonnement est identique à celui fait pour un graphe non orienté, mais utilise le fait que le parcours construit une forêt recouvrante en utilisant les listes de successeurs. On obtient de même :

Propriété

Si G est représenté par un tableau de listes de successeurs (ou de prédécesseurs) alors le parcours en largeur peut être implémenté en $\Theta(n + m)$ en espace et en temps.

Remarque

Si G est représenté par sa matrice d'adjacence alors les complexités en temps et en espace sont en …

(2) Parcours en Profondeur

Principe

On choisit un sommet initial x_0 qui devient visité. En chaque nouveau sommet visité x on choisit l'un de ses voisins (ou successeurs) non encore visité y, s'il en existe, qui est à son tour visité et qu'on explore de la même façon, s'il n'en existe pas on revient au sommet ayant précédé x dans le parcours s'il existe sinon la recherche reprend en un sommet non encore exploré ou bien se termine si tous les sommets sont visités.

Remarques

C'est une technique « vieille comme le monde » … En effet, c'est celle utilisée par Thésée dans le fameux Labyrinthe du roi Minos, construit par Dédale, pour y tuer le monstrueux Minotaure fruit des amours de Pasiphaé, l'épouse de Minos, et d'un taureau envoyé de la mer par Poséidon. Thésée en sortit vainqueur grâce au fil d'Ariane, fille de Minos, tombée amoureuse du héros …

La première utilisation explicite a été faite par **TREMAUX** (cité par **LUCAS**, 1882 et 1894) et **TARRY** (1886 et 1895), puis **SANTE-LAGUE** (1926) pour résoudre des problèmes de labyrinthe. Mais la première définition et utilisation algorithmique rigoureuse (« moderne ») a été faite par **R. E. TARJAN** en 1972.

Cette technique s'est depuis révélée très performante pour mettre au point de bons algorithmes, c'est-à-dire rapides et concis, pour résoudre de nombreux problèmes :

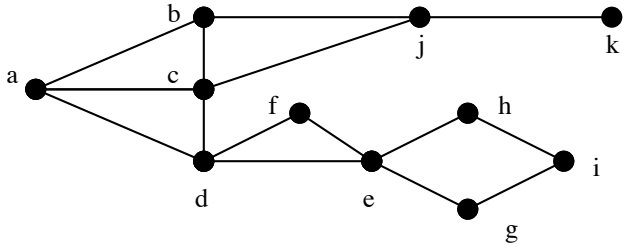
- Composantes Fortement Connexes et 2-Connexes d'un graphe en $\Theta(n + m)$ (**TARJAN**, 1972)
- Planarité d'un graphe en $\Theta(n + m)$ (**HOPCROFT, TARJAN**, 1974), donc en $\Theta(n)$
- Composantes 3-Connexes d'un graphe en $\Theta(n + m)$ (**HOPCROFT, TARJAN**, 1974)
- …

Comme le parcours en largeur, cette méthode atteint tous les sommets et toutes les arêtes (ou arcs), il faudra donc des instructions de marquage des sommets afin de reconnaître les sommets visités.

Examinons ce parcours sur le cas d'un graphe non orienté puis d'un graphe orienté.

Cas d'un graphe non orienté

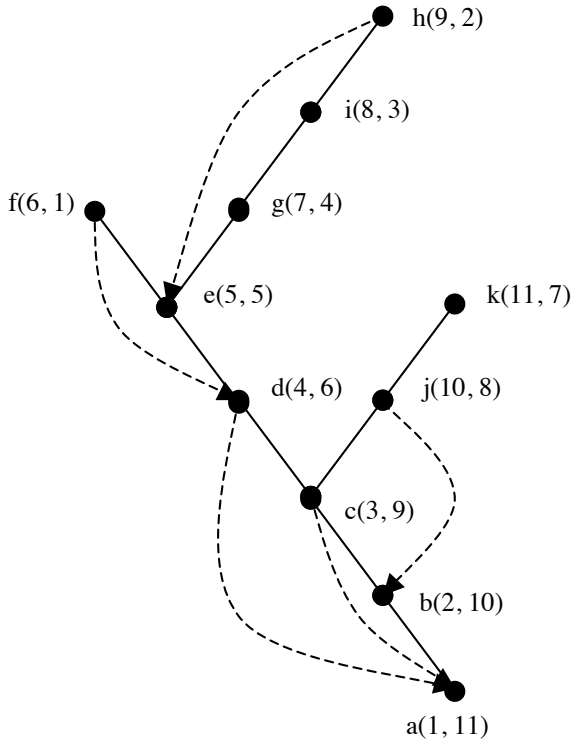
Soit G = (X, E) le graphe suivant (celui déjà utilisé) :



On suppose G donné par ses listes d'adjacence :

a : b, c, d	**d** : a, c, e, f	**g** : e, i	**j** : b, c, k
b : a, c, j	**e** : d, f, g, h	**h** : e, i	**k** : j
c : a, b, d, j	**f** : d, e	**i** : g, h	

Partant du **sommet a** on obtient l'arborescence suivante :



Les couples d'entiers associés à chaque sommet x représentent respectivement l'ordre de visite et l'ordre de fin de visite, (**n°visite(x)**, **finvisite(x)**).

Comme le parcours en largeur, ce parcours oriente naturellement chaque arête en arc dans les sens de sa première exploration (ce sont les segments dessinés en traits pleins, orientés du bas vers le haut) et lui associe donc une **Arborescence de racine a**. Les arêtes restantes, elles aussi devenues arcs, ce sont les **co-arcs**, sont représentées en pointillé.

Algorithme

On suppose que **X = {1, 2, … , n}**. L'algorithme nécessite un tableau **n°visite(1)**, **n°visite(2)**, … , **n°visite(n)**, indicé par les sommets, initialisé à 0, stockant l'ordre de visite et permettant de savoir si un sommet x a déjà été déjà visité, c'est-à-dire quand n°visite(x) ≠ 0.

On peut aussi utiliser un tableau **finvisite(1)**, **finvisite(2)**, … , **finvisite(n)** lui aussi initialisé à 0, ce tableau n'est pas nécessaire mais peut être utile, nous le verrons dans la suite. Ce tableau indique la fin de l'exploration d'un sommet, c'est-à-dire lorsque tous les voisins de ce sommet ont été visités.

L'**Arborescence** (ou la **Forêt d'arborescences**) **du Parcours**, **F**, peut être construite au cours de l'algorithme et peut être représentée soit par un tableau de listes de successeurs soit par un tableau stockant pour chaque sommet son père dans l'arborescence.

L'algorithme s'écrit très naturellement de façon récursive sous la forme :

```
N° ← 0
FIN ← 0
pour i ← 1 à n faire n°visite(i) ← 0 ; finvisite(i) ← 0
pour i ← 1 à n faire
        si n°visite(i) = 0 alors Profondeur(i)

Profondeur(x)
        N° ← N° + 1
        n°visite(x) ← N°
        pour « chaque voisin y de x » faire
                si n°visite(y) = 0 alors F(x) ← F(x) + {y}
                                        Profondeur(y)
        FIN ← FIN + 1
        finvisite(x) ← FIN
```

Remarque

L'algorithme est écrit sous forme récursive, il peut s'écrire sous forme itérative avec l'utilisation d'une **PILE** (priorité LIFO : last in, first out), stockant les sommets dans l'ordre de visite, **n°visite** est alors le numéro d'entrée dans la pile.

Un sommet est dépilé lorsque tous ses voisins ont été visités. Dans cette version de l'algorithme, **finvisite** est l'ordre de dépilement des sommets. La suite de l'exploration se poursuit avec le sommet situé en haut de pile, cela correspond au fameux « fil d'Ariane », pourquoi ?

La fonction **Profondeur** s'écrit alors :

```
Profondeur(x)

        EMPILER(x)
        N° ← N° + 1
        n°visite(x) ← N°

        tant que PILE ≠ Ø faire
                y ← haut de PILE

                si « y a un voisin z non visité »

                        alors EMPILER(z)
                                N° ← N° + 1
                                n°visite(z) ← N°

                        sinon DÉPILER(y)
                                FIN ← FIN + 1
                                finvisite(y) ← FIN
```

Propriétés

- *L'algorithme visite tous les sommets du graphe et examine deux fois chaque arête (une fois dans chaque sens).*
- *Profondeur(1) visite tous les sommets connectés à 1 et permet de générer la composante connexe correspondante.*

- *Le parcours permet de construire une Forêt d'Arborescences Recouvrante (à raison d'une arborescence par composante connexe) F = (X, A) sur le graphe et l'oriente en **PALMIER** (palm-tree) P = (X, U), toute arête de G est orientée dans le sens de sa première exploration et toute autre, tout co-arc, est un arc « retour » (backward edge) c'est-à-dire un arc allant d'un sommet à l'un de ses ancêtres ou ascendant dans la forêt.*

  Remarquons que tout orientation de G en palmier peut être obtenu par un parcours en profondeur (non nécessairement unique).

- *Les numérotations n°visite et finvisite définissent une relation d'ordre partiel (celle de l'arborescence) sur les sommets :*

  $x \leq y \Leftrightarrow n°visite(x) \leq n°visite(y)$ *et* $finvisite(x) \geq finvisite(y)$
  $\Leftrightarrow$ *x est ancêtre de y dans la forêt*

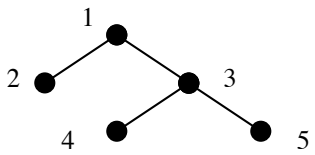
Remarques

- Les numérotations **n°visite** et **finvisite** correspondent respectivement à un **ordre d'empilement** et à un **ordre de dépilement** lié à cet ordre d'empilement (dans le sens où si on empile a puis b on ne peut pas dépiler a avant b).

  Elles permettent aussi de coder et de reconstituer de façon unique toute arborescence ayant 1, 2, … , n pour sommets.

  Exemples :

  - (1, 2, 3, 4, 5) et (2, 4, 5, 3, 1) correspondent à l'unique arborescence :



  - (1, 2, 3) et (2, 1, 3) correspondent aussi à une unique forêt d'arborescences, laquelle ?

  - Mais il n'existe aucune arborescence associée aux permutations (1, 2, 3) et (3, 1, 2).

  Dans ces exemples, (1, 2, … , n) et l'autre permutation sur {1, 2, … , n} doivent correspondre respectivement à un n°visite, ordre d'empilement, et à un finvisite, ordre de dépilement.

  Questions :

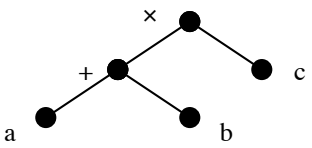
  - Comment caractériser les ordres de dépilement ?
  - Comment reconnaître un ordre de dépilement en $\Theta(n)$?
  - Comment construire l'unique arborescence ou forêt d'arborescences correspondant à un ordre de dépilement en $\Theta(n)$?
  - Démontrer qu'étant donnée une arborescence ou une forêt d'arborescences A, les numérotations n°visite et finvisite définissent A de façon unique.

- Ces deux numérotations sont fondamentales en Informatique sur les arborescences : n°visite correspond au **PRÉORDRE** (c'est-à-dire à la **Notation Polonaise Préfixée**) et finvisite au **POSTORDRE** (c'est-à-dire à la **Notation Polonaise Postfixée**) (notations ou Langages de **LUCASIEWICZ**).

  Ce sont des notations que l'on retrouve dans les **Parcours d'Arbre** et en **COMPILATION**.

Exemple :

On peut représenter l'expression arithmétique $(a+b)\times c$ par l'arborescence suivante :



Elle peut aussi se noter $\times + a\,b\,c$ en **Notation Polonaise Préfixée** (correspond à la numérotation n°visite de l'arbre) ou $a\,b + c\,\times$ en **Notation Polonaise Postfixée** (correspond à la numérotation finvisite de l'arbre).

Complexités

D'après les propriétés précédentes, le parcours en profondeur est en Ω(n+m) en espace et en temps.

Propriété

Si G est représenté par un tableau de listes d'adjacence le parcours en profondeur peut être implémenté en Θ(n + m) en espace et en temps.

Preuve

Les opérations de base sont les mêmes que celles qui sont explicitées dans l'analyse du parcours en largeur.

Soient $G_1 = (X_1, E_1), G_2 = (X_2, E_2), \ldots, G_k = (X_k, E_k)$ les **composantes connexes** de G. Si $|X_i| = n_i$ et $|E_i| = m_i$ alors $n_1 + \ldots + n_k = n$ et $m = m_1 + \ldots + m_k = m$.

Profondeur(x) n'est appelé, dans le programme principal, que pour un sommet x non encore visité et explore la composante connexe correspondante. Sur chaque sommet le travail est en O(1) et l'on parcourt la liste de ses voisins. Donc pour la $i^{ième}$ composante connexe G_i le coût en temps est :

$$\sum_{x \in X_i} (cte + cte'.d(x)) = am_i + bn_i \in \Theta(n_i + m_i)$$

où cte, cte', a et b sont des constantes strictement positives.

La deuxième boucle du programme principal passe en revue tous les sommets du graphe, mais n'appelle la fonction **Profondeur** qu'une seule fois sur chaque composante connexe de G. Donc globalement (avec les initialisations) la **Complexité en Temps** est :

$$T(n,m) = cte + cte'.n + \sum_{i=1}^{k} (am_i + bn_i) = \alpha m + \beta n + \gamma \in \Theta(n+m)$$

où α, β et γ sont des constantes strictement positives.

La **Complexité en Espace** est aussi en Θ(n + m) car :

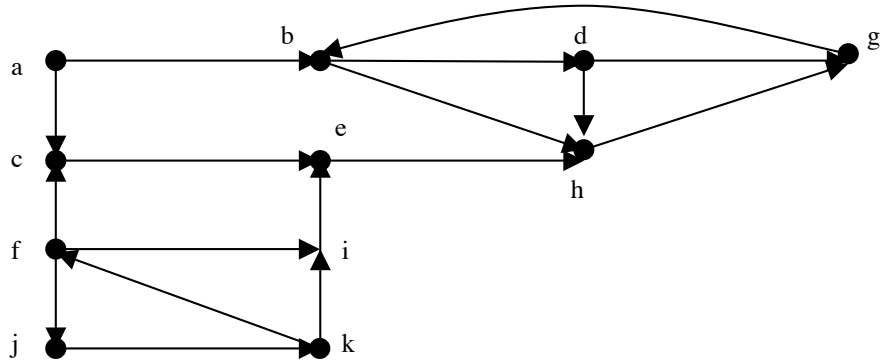
- la structure de donnée de G est en Θ(n + m)
- les tableaux n°visite et finvisite sont en Θ(n)
- la forêt d'arborescence est représentable en Θ(n)
- il y a enfin un nombre constant de variables auxiliaires
- dans la version itérative la PILE est de taille O(n).

Remarque

Si G est représenté par sa matrice d'adjacence alors les complexités en temps et en espace sont en …

Cas d'un graphe orienté

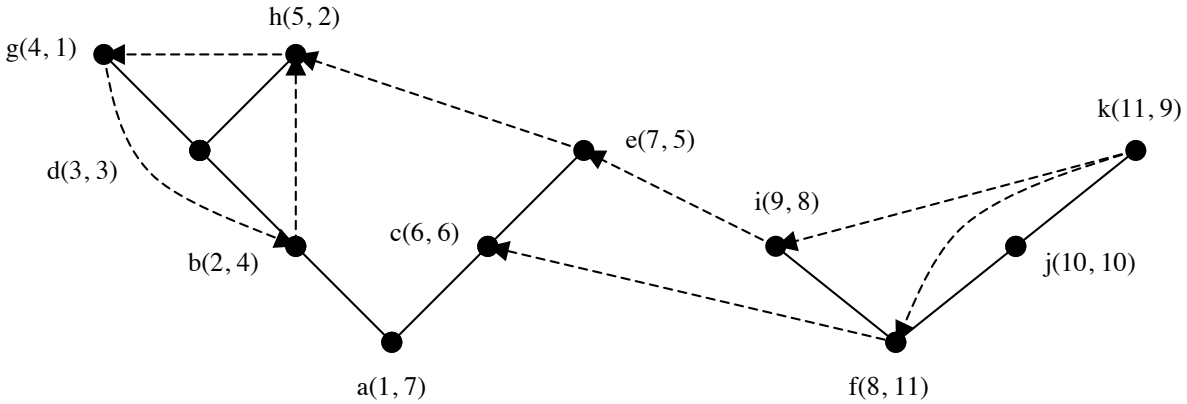
Soit G = (X, U) le graphe suivant (celui déjà utilisé):



Les listes de successeurs sont :

a : b, c	**d** : g, h	**g** : b	**j** : k
b : d, h	**e** : h	**h** : g	**k** : f, i
c : e	**f** : c, i, j	**i** : e	

Le parcours en profondeur, appliqué en respectant l'orientation, avec les listes de successeurs, à partir de a, donne la forêt d'exploration :



Les sommets sont numérotés dans l'ordre de visite. La **Forêt du Parcours** est obtenue en retenant les arcs explorés pour la première fois (arcs représentés en traits pleins, orientés implicitement de bas en haut), les arcs restants, les **co-arcs**, sont représentés en pointillé.

Remarquons que bien que le graphe soit connexe, au sens non orienté, le parcours donne une forêt de deux arborescences.

Il est instructif de comparer la forêt obtenue avec celle obtenue avec le parcours en largeur.

Algorithme

Il est identique à l'algorithme donné pour le cas non orienté en remplaçant voisins par successeurs (ou éventuellement prédécesseurs).

Propriétés

- *L'algorithme visite tous les sommets du graphe et examine chaque arc exactement une fois.*
- *L'arborescence de racine 1 (le premier sommet visité) obtenu par* ***Profondeur(1)*** *à partir des listes de successeurs (resp. des prédécesseurs) contient tous les* ***Descendants*** *(resp.* ***Ascendants****) de 1 dans G. Mais ce n'est pas vrai pour les autres sommets dans la suite du parcours.*
- *Le parcours permet de construire une Forêt d'Arborescences qui recouvre le graphe.*
- *Les deux numérotations, n°visite et finvisite, partitionnent les arcs de G en quatre classes :*
  - *les* ***arcs du parcours*** *(arcs de la forêt)*
  - *les* ***arcs transverses****, ce sont les arcs xy vérifiant :*

    $$n°visite(x) > n°visite(y) \text{ et } finvisite(x) > finvisite(y)$$

  - *les* ***arcs retour****, ce sont les arcs xy vérifiant :*

    $$n°visite(x) > n°visite(y) \text{ et } finvisite(x) < finvisite(y)$$

  - *les* ***arcs directs****, ce sont les arcs de transitivité de la forêt, ils vérifient :*

    $$n°visite(x) < n°visite(y) \text{ et } finvisite(x) > finvisite(y)$$

Remarques

Tous les arcs de transitivité du graphe ne sont pas nécessairement des arcs de transitivité de la forêt.

Il n'existe pas d'arcs xy vérifiant **n°visite(x) < n°visite(y)** et **finvisite(x) < finvisite(y)**. Pourquoi ?

Complexités

Le raisonnement est identique à celui fait pour un graphe non orienté, et utilise le fait que le parcours construit une forêt recouvrante en utilisant les listes de successeurs. On obtient de même :

Propriété

Si G est représenté par un tableau de listes de successeurs (ou de prédécesseurs) alors le parcours en profondeur peut être implémenté en $\Theta(n + m)$ en espace et en temps.

Remarque

Si G est représenté par sa matrice d'adjacence alors les complexités en temps et en espace sont en …

(3) Applications Directes des Parcours

Les parcours en Largeur et en Profondeur permettent facilement de tester la structure de certains graphes, de déterminer quelques paramètres et propriétés classiques.

Pour les **Graphes Non Orientés** :

- Connexité et Composantes Connexes
- Distances entre sommets, les Centres, le Rayon et le Diamètre du graphe
- Bipartisme
- Tester si le graphe est un Arbre, ou une Forêt
- …

Pour les **Graphes Orientés** :

- Calculs de la Fermeture Transitive et d'une Réduction Transitive
- Déterminer si le graphe est sans circuit
- Tester si le graphe est une Arborescence ou une Forêt d'Arborescences
- …

Nous laissons cela en Exercices, voir aussi TD 6.

(4) Prolongements

Pours des applications « plus concrètes », plus spécifiques voire plus approfondies utilisant les notions citées, voir le Cours de Recherche Opérationnelle, où sont mentionnés et traités de nombreux problèmes se modélisant naturellement sous forme de graphes, comme :

- des problèmes d'organisation, de gestion, de planification, …
- des problèmes d'Ordonnancement
- des problèmes de Plus Courts Chemins, d'Arbre Recouvrant
- des problèmes de Flots

Nous parlons aussi bien sûr du polycopié « Graphes, Quelques Notions de Base », du cours de RO, où sont reprises toutes les définitions nécessaires et où sont cités de nombreux exemples de modélisations et d'applications, ne figurant pas ici, ainsi que quelques notions supplémentaires, comme les couplages, les colorations de graphes…