Chapitre 1 Complexité algorithmique

La théorie de la complexité se propose de déterminer les ressources nécessaires à l'exécution d'un algorithme. On distingue la complexité en temps et la complexité en espace mémoire. Nous commencerons ce chapitre en définissant des notations utilisées en théorie de la complexité.

5. Notations asymptotiques Θ(), O() et Ω()

5.1. Notation Θ()

Pour une fonction g(n) donnée, on note Θ(g(n)) l'ensemble des fonctions f définies par :

$$\Theta(g(n)) = \{f(n) : \exists\, c_1>0 \text{ et } c_2>0 \text{ et } n_0>0 \text{ tels que } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ pour tout } n \ge n_0\}$$

Une fonction appartient à l'ensemble Θ(g(n)) s'il existe des constantes strictement positives telles que f(n) puisse être prise en sandwich entre c_1g(n) et c_2g(n) pour n assez grand. Autrement dit, a fonction g(n) est égale à la fonction g(n) à une constante multiplicative près. On dit que g(n) est une borne approchée asymptotique pour f(n).

Exercice :
Montrer que

1. *si $f_1(n) = \Theta(g(n))$ et $f_2(n) = \Theta(g(n))$ alors $f_1(n)+ f_2(n) = \Theta(g(n))$*
2. *si $f_1(n) = \Theta(g(n))$ alors $n.f_1(n) = \Theta(n.g(n))$*
3. *$n^2-3n = \Theta(n^2)$*

Proposition :
Soit p un polynôme de degré d.

i. $p(n) \in \Theta(n^d)$
ii. $p(n) + \ln n \in \Theta(n^d)$
iii. $p(n) + \exp(n) \in \Theta(\exp(n))$

L'assertion i) signifie que les termes de degrés inférieurs au degré maximum d peuvent être négligés dans une étude asymptotique. Les assertions ii) et iii) montrent respectivement que la fonction *ln* est asymptotiquement négligeable devant n^d et la fonction n^d est négligeable devant la fonction *exp*.

5.2. Notations O() et Ω()

Pour une fonction g(n) donnée, on note O(g(n)) l'ensemble des fonctions f définies par :

$$O(g(n)) = \{f(n) : \exists\, c>0 \text{ et } n_0>0 \text{ tels que } 0 \le f(n) \le c.g(n) \text{ pour tout } n \ge n_0\}$$

La notation O() sert à donner une borne supérieure à une fonction à une coefficient multiplicatif près. La notation Θ() est donc plus forte que la notation O(). On note que pour toute fonction f

$$\Theta(f) \subseteq O(f)$$

Aussi la proposition de la section précédente peut se transformer aisément en remplaçant Θ par O.

De la même manière on définit la notation Ω par :

$$\Omega(g(n)) = \{f(n) : \exists\, c>0 \text{ et } n_0>0 \text{ tels que } 0 \le c.g(n) \le f(n) \text{ pour tout } n \ge n_0\}$$

La notation Ω () sert à donner une borne inférieure à une fonction f à une coefficient multiplicatif près. La notation Θ() est donc plus forte que la notation Ω (). On note que pour toute fonction f

$$\Theta(f) \subseteq \Omega(f)$$

Aussi la proposition de la section précédente peut se transformer aisément en remplaçant Θ par Ω

Théorème : Soient deux fonctions f et g.

$$f(n) = \Theta(g(n)) \Leftrightarrow \begin{array}{l} f(n) = \Omega(g(n)) \\ f(n) = O(g(n)) \end{array}$$

Autrement dit, $\Theta(g(n)) = \Omega(g(n)) \cap O(g(n))$

Complexité algorithmique en temps

5.3. Introduction

Evaluer le temps nécessaire à l'exécution d'un algorithme est trop dépendant du matériel utilisé. Afin de pallier ceci, on se propose de calculer le nombre d'exécutions d'instructions élémentaires, noté t. Les instructions élémentaires considérées seront :

- les affectations
- les opérations arithmétiques
- les tests

En général, le temps pris par un algorithme croit avec la taille de l'entrée; on a donc pris l'habitude de décrire le temps d'exécution d'un algorithme en fonction de la taille de l'entrée. La taille de l'entrée sera dans le cas des graphes, le nombre de sommets et/ou le nombre d'arcs.

Exemple : fonction factorielle.

Algorithme	**Opérations élémentaires**	**Nombre d'exécutions**
Fact (n)	-	
Début	-	
p entier	-	
Pour *i:=1 à n* ***faire***	1 test, 1 addition et 1 affectation	n
*p:=p*i*	1 multiplication et 1 affectation	n
Finpour	-	
Fact:=p	1 affectation	1
Fin	-	

Aussi le nombre :

- *de tests est n,*
- *d'additions est n,*
- *de multiplications est n,*
- *d'affectations est 2n+1*

Connaissant, pour un ordinateur donné, le temps requis pour ces instructions élémentaires, nous pouvons en déduire le temps. On appelle *inst* l'instruction élémentaire la plus rapide et *INST* l'instruction élémentaire la plus lente. En notant c le temps d'exécution de *inst* et C le temps d'exécution de *INST* on peut encadrer le temps d'éxécution T d'un programme par

$$\mathbf{ct \leq T \leq Ct \Leftrightarrow T = \Theta(t)}$$

<u>Exemple :</u> fonction factorielle
Le nombre total d'opérations effectuées est 5n+1 aussi

$$T=\Theta(n)$$

Dans l'exemple précédent, la taille de l'entrée permet de définir précisément le nombre d'opérations effectuées. D Dans certains cas, ceci est impossible. Imaginons que l'on souhaite trier, par ordre croissant, un tableau de nombres. Il semble intuitif de penser que plus le tableau sera grand, plus il sera long pour le trier. Aussi la taille du tableau est un paramètre important pour exprimer le temps d'exécution. Cependant il n'est pas suffisant pour obtenir une évaluation précise. En effet, il n'est pas difficile d'imaginer une procédure qui serait beaucoup plus efficace dans le cas ou le tableau est déjà trié ou s'il est trié aléatoirement. Aussi nous distinguons 3 types de complexité :

- la complexité au meilleur cas
- la complexité en moyenne
- la complexité au pire cas

5.4. Exemple 1: tri par Insertion

Algorithme	Coût	Nombre d'exécutions
Tri-Insertion (A)		
Début		
n:=taille de A	c_1	1
Pour *j:=2 à n*	c_2	n
c:=A[j]	c_3	n-1
i:=j-1	c_4	n-1
Tant que *i>0 et A[i]>c*	c_5	$\sum_{j=2 \text{ à } n} t_j$
A[i+1]:=A[i]	c_6	$\sum_{j=2 \text{ à } n} (t_j-1)$
i:=i-1	c_7	$\sum_{j=2 \text{ à } n} (t_j-1)$
Fin tantque		
A[i+1]:=c	c_8	n-1
Fin Pour		

où t_j représente le nombre de fois que le test de la boucle "tant que" est effectué pour un j donné. Aussi nous pouvons exprimer le coût total en additionnant les produits des colonnes "cout" et "nombre d'exécution". On obtient :

$$\mathbf{T(n)= c_1+c_2n+c_3(n-1)+c_4(n-1)+c_5\sum_{j=2 \text{ à } n} t_j + c_6\sum_{j=2 \text{ à } n} (t_j-1)+ c_7\sum_{j=2 \text{ à } n} (t_j-1)+c_8(n-1)}$$

On voit que le temps d'exécution ne dépend pas seulement de la taille du tableau mais aussi de la nature du tableau. En effet suivant que le tableau est trié ou qu'il est trié en ordre inverse les valeurs de t_j sont différentes. Le premier cas est le plus favorable alors que le second est le pire. Etudions ces deux cas et nous en déduirons les complexités respectivement au meilleur et au pire cas.

- **Etude de la complexité au meilleur cas**

Comme le tableau est déjà trié, à l'étape j, A[j-1]<c, aussi $t_j=1$. On appelle $t_{min}(n)$ le temps mis pour trier un tableau de taille n . Dans ce cas, $t_{min}(n)$ devient

$$\mathbf{T_{min}(n) = (c_1+c_2+c_3+c_4+c_5+c_8)n - (c_2+c_3+c_4+c_5+c_8)}$$

Aussi le temps d'exécution est une fonction linéaire de n. On dit que la complexité au meilleur cas est en $\Theta(n)$, i.e.

$$\mathbf{T_{min}(n) \in \Theta(n)}$$

- **Etude de la complexité au pire cas**

Si le tableau est déjà trié en sens inverse, i.e en ordre décroissant, on se trouve dans le pire cas (facile à montrer). On appelle $t_{max}(n)$ le temps nécessaire pour trier un tableau de taille n. On doit alors

comparer chaque élément A[j] avec chaque élément de sous tableau déjà trié A[1,…,j-1] et donc t_j=j. On en déduit aisément que la complexité au pire cas de la procédure Tri-Insertion est en $\Theta(n^2)$, i.e.

$$\mathbf{T_{max}(n) \in \Theta(n^2)}$$

5.5. Exemple 2 : tri par Fusion

Le tri fusion est un tri récursif qui consiste à diviser le tableau de n éléments en deux sous-tableaux de n/2 éléments, de trier les deux sous-tableaux puis de fusionner ces deux sous tableaux. On nomme *Fusionner(A,p,q,r)*, la procédure qui fusionne deux sous-tableaux A[p,…,q] et A[q+1,…,n] supposés triés ; le résultat est donc un sous-tableau A[p,…,r] triés.

Exercice : Ecrire la procédure fusionner(A,p,q,r)

Bien programmé, la complexité de cette procédure est (aussi bien au meilleur cas qu'au pire cas) en $\Theta(n)$. On peut maintenant écrire la procédure récursive Tri-Fusion(A,p,q).

Tri-Fusion(A,p,q)

Debut
si *p<r,* ***alors***
q:=⌊(p+r)/2⌋
Tri-Fusion(A,p,q)
Tri-Fusion(A,q+1,r)
Fusionner(A,p,q,r)
Fin si
Fin

Supposons, pour simplifier, que n est une puissance de 2. On peut exprimer le temps d'exécution de cette procédure par la formule de récurrence suivante :

$T(n) = \Theta(1)$ si n=1
$2T(n/2) + \Theta(n)$ si n>1

On montre par récurrence que

$$\mathbf{T(n) = \Theta(n.\ln n)}$$

Exercice : Démontrer par récurrence la complexité précédente.

5.6. Analyse de l'étude de la complexité

La complexité au meilleur cas, n'a pas beaucoup d'intérêt en pratique, la complexité en moyenne est souvent assez difficile à calculer. Nous nous limiterons dans ce cours, à la complexité au pire cas.

Exercice :

Supposons que l'on dispose de deux algorithmes A et B pour résoudre un problème. La complexité au pire cas de A en en $\Theta(n^2)$ alors que celle de B est en $\Theta(n.\ln n)$. Peut on assurer que B s'exécutera plus vite que A sur une donnée? On justifiera la réponse en utilisant deux arguments.

Réponse :

L'étude en complexité est une étude asymptotique au pire cas. Aussi, la seule chose que l'on puisse assurer est que l'algorithme B s'exécutera plus vite que A, lorsque l'entrée correspondra au cas le plus défavorable et ce lorsque n sera grand.

La taille des constantes est d'une part inhérente à l'algorithme lui-même, à la qualité du programmeur ainsi qu'aux caractéristiques de la machine.
Aussi l'analyse de la complexité ne permet pas toujours de dire qu'un algorithme est plus rapide qu'un autre dans une application donnée, mais représente un bon indicateur pour une étude théorique.
Le tableau suivant donne quelques exemples de complexité avec le temps d'exécution correspondant

Complexité\taille	**20**	**50**	**100**	**200**	**500**	**1000**
$\mathbf{10^3.n}$	0,02s	0,05s	0,1s	0,2s	0,5s	1s
$\mathbf{10^3.n.\ln n}$	0,09s	0,3s	0,6s	1,5s	4,5s	10s
$\mathbf{10^2.n^2}$	0,04s	0,25s	1s	4s	25s	2min
$\mathbf{10.n^3}$	0,02s	1s	10s	1min	21min	27h
$\mathbf{n^{\ln n}}$	0,4s	1,1h	220j	12500ans	5.10^{10} ans	--
$\mathbf{2^n}$	1s	36ans	--	--	--	--
$\mathbf{3^n}$	58min	2.10^{11} ans	--	--	--	--
n!	77100ans	--	--	--	--	--

- **Complexité polynomiale - classe P**

Un algorithme est de complexité polynomiale s'il existe un entier p tel que sa complexité soit en $O(n^p)$. La classe P (comme polynomiale) regroupe tous les problèmes tel qu'il existe un algorithme resolvant ce problème. En pratique, on dira qu'un problème est "effectivement" soluble s'il appartient à la classe P.

- **Problèmes difficiles**

Les algorithmes non-polynomiaux, i.e les algorithmes en $\Omega(2^n)$ seront considérés comme non-utilisables en pratique.

Exemple : *Problème du voyageur de commerce*

Supposons un ensemble de villes toutes reliées entre elles par une route. Un représentant de commerce se pose le problème suivant:

"Comment, en partant de chez lui, le représentant peut-il passer une et une seule fois par toutes les villes en minimisant la distance parcourue?"

Ce problème ne possède pas de solution polynomiale.

Tous les problèmes de graphes présentés dans ce cours peuvent être résolus en énumérant l'ensemble des solutions. Malheureusement la complexité de tels algorithmes est souvent exponentielle. Aussi tout l'enjeu de la théorie des graphes est de trouver des algorithmes polynomiaux, de degré aussi petit que possible, pour résoudre les différents problèmes.

Exemple :

Trouver le plus court chemin entre deux villes.

Complexité en espace

La complexité en espace est chargée d'évaluer les besoins en mémoire d'un algorithme. Souvent, la complexité en espace et la complexité en temps sont antinomiques ; parfois conserver des calculs intermédiaires en mémoire permet d'accélérer un algorithme. Lors de ce cours, nous ne parlerons pas de complexité en espace. Pour la plupart des algorithmes présentés dans ce cours, les complexités en espace sont faibles. Nous nous concentrerons donc sur les ressources en temps.

Chapitre 2 : Graphes orientés

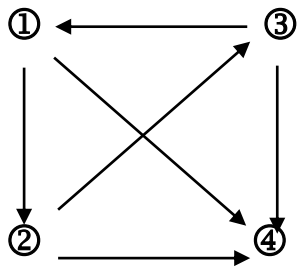
1 Quelques définitions sur les graphes orientés

5.7. Définition des graphes orientés

Définition 1 On appelle *graphe orienté* la donnée d'un couple **G**=(**X,U**) telle que :
X = $\{x_1,\ldots,x_n\}$ est un ensemble fini d'éléments appelés sommets.
U = $\{u_1,\ldots,u_m\}$ est un ensemble fini de couples de sommets appelés *arcs*
Dans toute la suite, on notera n le nombre de sommets et m le nombre d'arcs.

Remarque **:** dans un souci de simplification et lorsqu'il n'y aura pas d'ambiguité, un arc (u_i,u_j) sera noté simplement par (i,j)

Exemple 1 : Résultats d'un tournoi d'échec ayant 4 concurrents x_1, x_2, x_3, x_4
L'écriture d'un arc (i,j) indique que x_i a remporté la partie unique l'opposant à x_j
Un tournoi se formalise par un graphe orienté comme suit :
X={ x_1, x_2, x_3, x_4 }
U= { (1,2), (1,4), (2,3), (2,4), (3,1), (3,4) }



5.8. Terminologies

i. Soit u=(x,y) un arc. On dit que x est l'extrémité initiale de u et y est l'extrémité terminale de u.
ii. Une *boucle* est un arc (x,x) dont les extrémités coïncident.
iii. Des *arcs* sont dits *adjacents* s'ils possèdent au moins une extrémité commune.
iv. On dit que y est un *successeur* de x s'il existe un arc ayant x comme extrémité initiale et y comme extrémité terminale. $\Gamma^+(x)$ désigne l'ensemble des successeurs de x.
v. On dit que y est un *prédécesseur* de x s'il existe un arc ayant y comme extrémité initiale et x comme extrémité terminale. $\Gamma^-(x)$ désigne l'ensemble des prédécesseurs de x.
vi. x est un sommet *adjacent* au sommet y s'il est prédécesseur ou successeur de y. On dit aussi que x et y sont *voisins*. $\Gamma(x) = \Gamma^-(x) \cup \Gamma^+(x)$ note l'ensemble des voisins de x.
vii. Un sommet est dit *isolé* s'il ne possède pas de voisin.

Exemple : en reprenant l'exemple du tournoi d'échecs, on obtient

	x_1	x_2	x_3	x_4
Γ^+	$\{x_2,x_4\}$	$\{x_3,x_4\}$	$\{x_1,x_4\}$	$\varnothing$
Γ^-	$\{x_3\}$	$\{x_1\}$	$\{x_2\}$	$\{x_1,x_2,x_3\}$
Γ	$\{x_2, x_3, x_4\}$	$\{x_1, x_3, x_4\}$	$\{x_1, x_2, x_4\}$	$\{x_1, x_2, x_3\}$

Exemple : fonctions Γ^+, Γ^- et Γ du tournoi d'échec

viii. On dit que u est incident à x vers l'extérieur et incident à y vers l'intérieur. $\mathbf{d^+_G(x)}$ désigne le **nombre d'arcs incidents à** x **vers l'extérieur** et s'appelle le demi-degré extérieur de x. $\mathbf{d^-_G(x)}$ désigne le **nombre d'arcs incidents à** x vers l'**intérieur** et s'appelle le demi-degré intérieur de x. Le degré d'un sommet x vaut : $\mathbf{d_G(x)= d^+_G(x) + d^-_G(x)}$ **.**

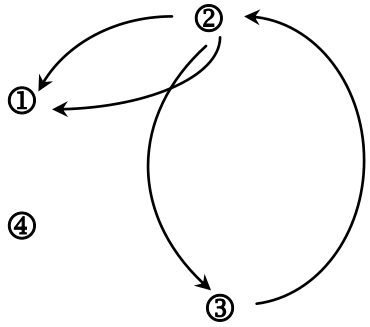
Exemple :

	$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	$\mathbf{x_4}$
$\mathbf{d^+_G(x)}$	2	2	2	0
$\mathbf{d^-_G(x)}$	1	1	1	3
$\mathbf{d_G(x)}$	3	3	3	3

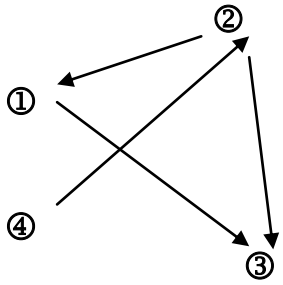
Exemple : fonctions d^+_G , d^-_G, d_G du tournoi d'échec

5.9. p-graphes

Définition 2. On appelle ***p-graphe*** tout graphe tel qu'il existe *au plus* p arcs joignants deux sommets quelconques de ce graphe.



Exemple de 2-graphe



Exemple de 1-graphe

Sauf mention express du contraire, dans la suite tous les graphes seront supposés être des 1-graphes.

Remarquons qu'un **G** graphe est défini ou bien par un couple **(X,U)**, ou bien par l'application $\mathbf{\Gamma^+}$, ou bien par l'application $\mathbf{\Gamma^-}$.

6. Sous structures d'un graphe.

6.1. Définitions

Soit **G**=(**X**, **U**) un graphe.

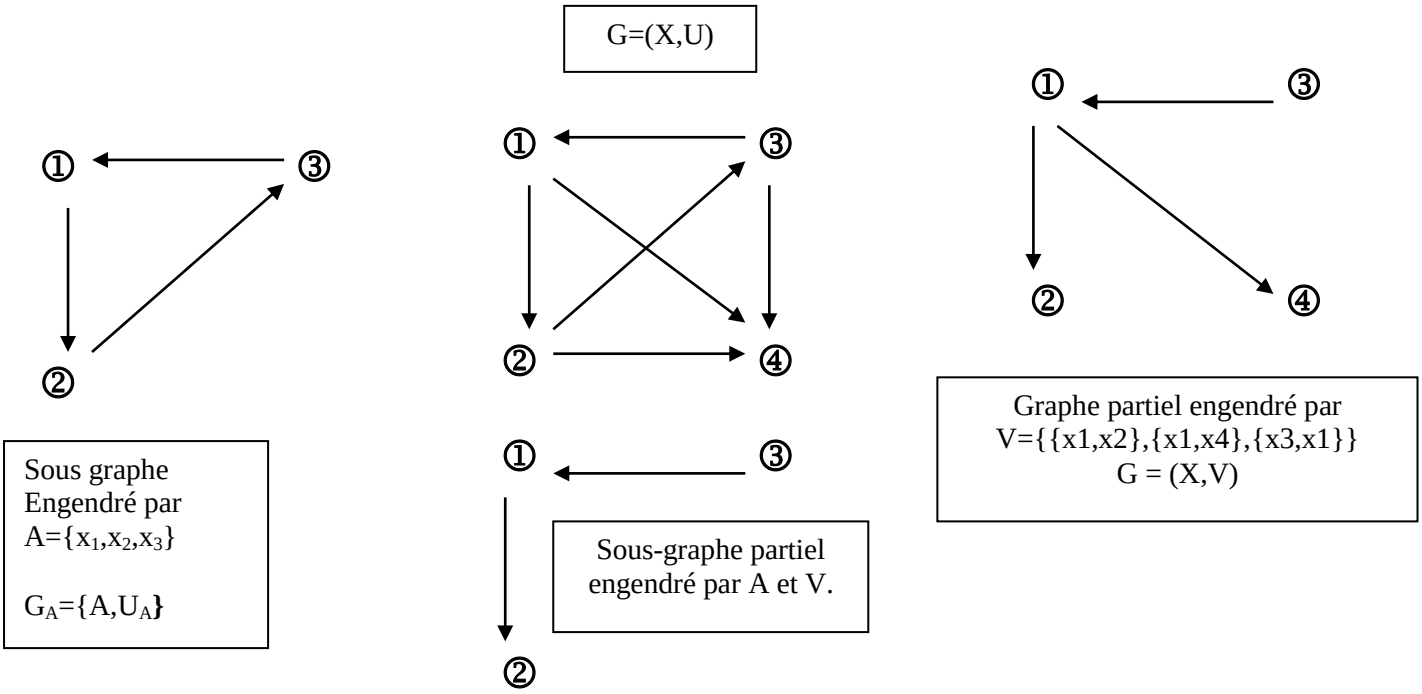
i. On appelle *sous graphe* de **G** tout graphe $(\mathbf{A},\mathbf{U_A})$ où :
- A est un sous ensemble de X
- U_A est *le* sous ensemble de U ayant leurs extrémités dans A.

$(\mathbf{A},\mathbf{U_A})$ est le *sous graphe de* **G** *engendré par* **A** et, ce graphe est noté par $\mathbf{G_A}$.

ii. On appelle *graphe partiel* de **G** tout graphe (**X**,**V**) où **V** est un sous ensemble d'arcs.

iii. Etant donné $\mathbf{A \subseteq X}$ et $\mathbf{V \subseteq U}$, le sous graphe partiel engendré par **A** et par **V** est le graphe partiel de $\mathbf{G_A}$=$(\mathbf{A},\mathbf{V_A})$ engendré par $\mathbf{V \cap V_A}$**.**

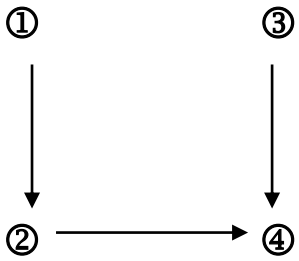
6.1.1.Exemple



6.2. Définition d'une chaîne

Soit G=(X,U) un graphe et C={u_1,…,u_q} une suite d'arcs de G.
On dit que C est une *chaîne* (de cardinalité q) si chaque arc u_k, $2 \leq k \leq q-1$, possède une extrémité en commun avec l'arc précédent u_{k-1} et l'autre extrémité en commun avec l'arc suivant u_{k+1} . x et y sont dites les extrémités de la chaîne C.
Une *chaîne* est *élémentaire* si en la parcourant on ne rencontre pas deux le même sommet

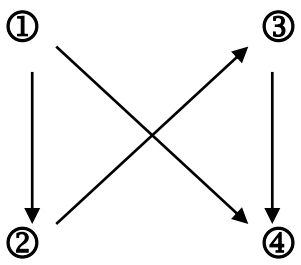
Exemple de chaîne élémentaire



6.3. Définition d'un chemin

On appelle *chemin* de cardinalité q, toute chaîne {u_1,…,u_q} de cardinalité q dont tous les arcs sont orientés dans le *même sens*, c'est à dire que l'extrémité terminale de u_k coïncide avec l'extrémité finale de u_{k+1}, $1 \leq k < q$.
Soient x l'extrémité initiale de u_1 et y l'extrémité finale u_q d'un tel chemin C={u_1,…,u_q}.
C est dit chemin (allant) de x vers y, x et y étant les extrémités du chemin c.
C est un chemin élémentaire si en le parcourant, on ne rencontre pas deux fois le même sommet.

Exemple de chemin élémentaire : {(1,2),(2,3),(3,4)}



Remarque :

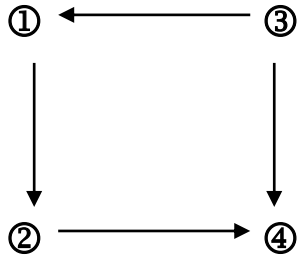
i. Un chemin, pour un 1-graphe, est complètement défini par la liste des sommets qui le constitue $C=\{x_1,\ldots,x_q\}$.
ii. S'il existe un chemin entre x et y alors on dit que y est un descendant de x ou inversement, on dit que x est un ascendant (ou ancêtre) de y.

6.4. Définition d'un cycle

On appelle *cycle* toute chaîne dont les extrémités coïncident.
On dit que le *cycle* est *élémentaire* si en le parcourant, on ne rencontre pas deux fois le même sommet en dehors de l'extrémité commune.

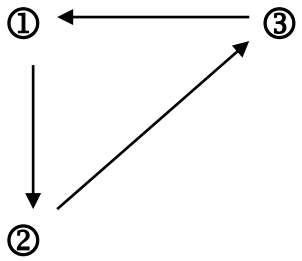
6.4.1. Exemple de cycle :

{(1,2},(2,4),(4,3),(3,1)}



6.4.2. Définition d'un circuit

Un circuit est un cycle dont tous les arcs sont dirigés dans le même sens. C'est aussi un chemin dont les extrémités coïncident.
Le circuit est élémentaire si en le parcourant, on ne rencontre pas deux fois le même sommet, en dehors de l'extrémité commune.
C={(1,2),(2,3),(3,1)}



7. Quelques propriètés remarquables

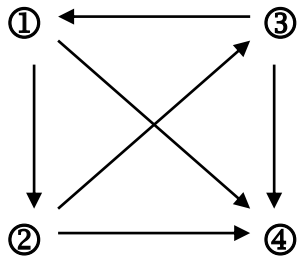
7.1. Faible connexité

Un graphe est dit (*faiblement*) *connexe*, si pour tout couple de sommet x_i, $x_j \in X$ ($i \neq j$),
Il existe une chaîne joignant x_i à x_j.

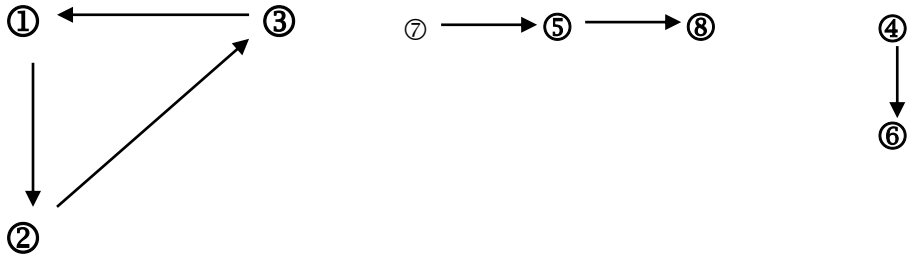
Proposition : Soit G=(X,U) un graphe. La relation binaire (R , X) définie par :
x_i **R** $x_j \Leftrightarrow x_i = x_j$ ou il existe une chaîne reliant x_i et x_j
est une **relation d'équivalence** (réflexive, symétrique, transitive).

L'ensemble des classes d'équivalences **X/R** partitionnent X . Les éléments de **X/R** sont dits *composantes faiblement connexes* du graphe **G**.

Exemple : le graphe G suivant ne contient qu'une seule composante connexe.



Le graphe G comporte trois composantes faiblement connexes :
$\{(x_1,x_3,x_2)\}$, $\{(x_7,x_5,x_8)\}$, $\{(x_4,x_6)\}$



7.2. Forte connexité.

On dit qu'un graphe est *fortement connexe* si pour tout couple de sommets $^{x}_{i}$, $x_j \in X$, $(i \neq j)$, il existe un chemin allant de x_i à x_j.

Proposition. Soit G=(X,U) un graphe. La relation binaire (R , X) définie par :
x_i **R** $x_j \Leftrightarrow x_i = x_j$ ou il existe un *chemin* reliant x_i et x_j *et* un *chemin* reliant x_j à x_i
est une **relation d'équivalence** (réflexive, symétrique, transitive).

L'ensemble des classes d'équivalences **X/R** partitionnent X . Les éléments de **X/R** sont dits *composantes fortement connexes* du graphe **G**.

Exemple : Le graphe G' précédent comporte six composantes fortement connexes : $\{(x_1,x_3,x_2)\}$, $\{x_7\}$, {x5},{x8},{x4},{x6}

7.3. Symétrie.

On dit qu'un graphe G=(X,U) est ***symétrique*** si :

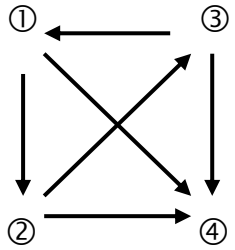
$$(x_i,x_j) \in U \Rightarrow (x_j,x) \in U,\ \forall x_i, \forall x_j \in X$$

7.4. Antisymétrie.

On dit qu'un graphe G=(X,U) est *antisymétrique* si :

$$(x_i,x_j) \in U \Rightarrow (x_j,xi) \notin U,\ \forall x_i, \forall x_j \in X,\ x_i \neq x_j$$

Exemple : *le graphe suivant est antisymétrique.*



7.5. Transitivité

On dit qu'un graphe G=(X,U) est *transitif* si :

$$(x_i,x_j)\in U \text{ et } (x_j,x_k)\in U \Rightarrow (x_i,x_k)\in U,\ \forall x_i,x_j,x_k\in X$$

7.6. Complètude.

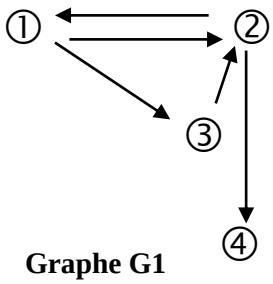
On dit qu'un graphe G=(X,U) est *complet* si :

$$(x_i,x_j)\notin U \Rightarrow (x_j,x_i)\in U,\ \forall x_i,x_j \in X,\ x_i\neq x_j$$

Remarquons qu'un graphe est complet si deux sommets sont au moins réunis par un arc. Le graphe G1 précédent est complet.

Chapitre 3 : Algorithmes basiques sur les graphes orientés

1 Structures de données utilisées pour représenter les graphes



Graphe G1

7.7. Utilisation de matrices d'adjacence

L'ensemble des arcs est représenté par un tableau de booléens M=[a_{ij}] appelé *matrice adjacente du graphe* définie par :

$$a_{ij}=1 \text{ ssi } (i,j)\in U$$

En langage Pascal, on représente une matrice d'adjacence par la structure de données suivante:
Type GRAPHE=array[1..n,1..n] of boolean
où n est le nombre de sommets.

L'exemple G se représente par la matrice d'adjacence M suivante :

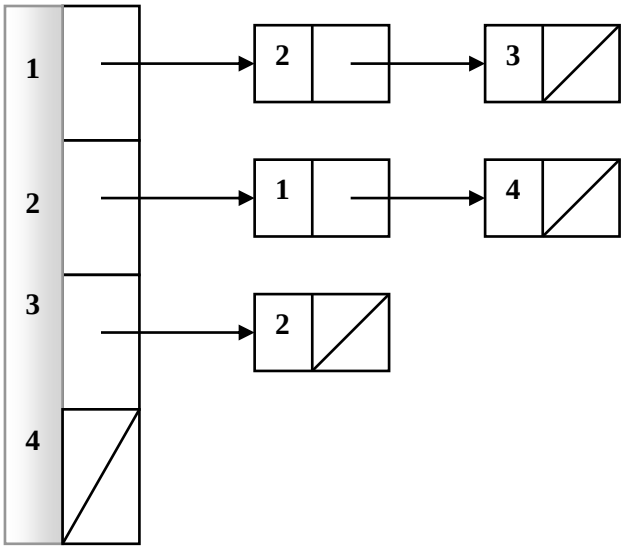
$$M=\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Remarquons que cette méthode requière un **temps d'ordre n^2** pour une consultation complète de la matrice. De plus, un objet de type graphe occupe en mémoire un **espace proportionnel à n^2 octets**, même si le graphe ne comporte que peu d'arcs.

7.8. Représentation des graphes par listes adjacentes.

Il est possible de représenter un graphe au moyen de listes.
Pour chaque sommet on associe la liste de ses successeurs rangés dans un certain ordre. Ces listes sont dites listes d'adjacence.

La tête de liste, pour chaque liste adjacente, est dans un tableau comprenant les n sommets.
Par exemple, le graphe G1 sera représenté par



Le type PASCAL correspond est :

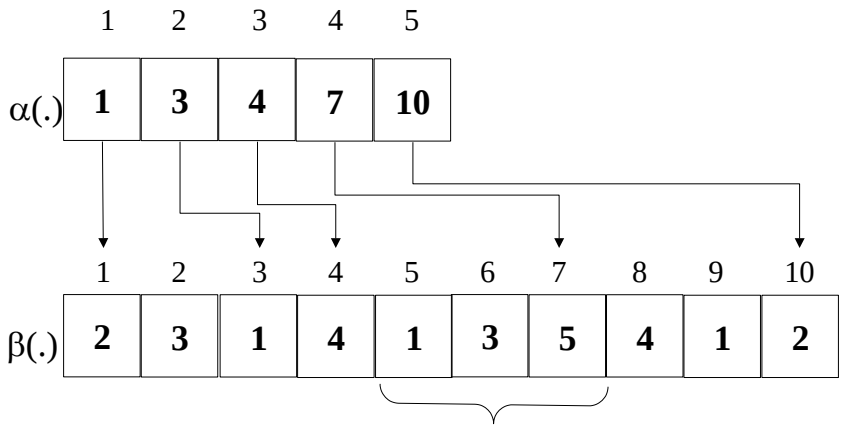
```
Type adr=↑ doublet;
GRAPHE= record
            no : 1..n;
            suiv : adr;
            end;

GRAPHE= array [1..n] of adr;
```

Cette représenter optimise la mémoire utilisée pour représenter un graphe.
Pour un graphe orienté qui possède n arcs et p sommets, l'espace mémoire utilisé est en $\Theta(n+p)$.

Remarque :

- On peut représenter la structure de données précédentes avec 2 tableaux : un de taille n et un autre de taille m.



Successeurs du sommet 4

- Il est parfois utile de représenter le graphe à la fois par des listes de successeurs et de prédécesseurs.

7.9. Comparaison des deux structures de données précédente

Beaucoup d'algorithmes font appel à la sous-structure suivante :

```
pour tous les sommets x
        pour tous les successeurs y de x
                ….
        fin Pour
fin Pour
```

La complexité de cette sous structure est en $\Theta(m+n)$ pour les listes d'adjacence alors qu'elle est en $\Theta(n^2)$ pour la matrice d'adjacence. Ceci justifie que dans la plupart des algorithmes de ce cours, on représentera les graphes par des listes d'adjacence.

Exemple : écrire l'algorithme qui consiste à calculer pour chaque sommet, la somme des numéros de ces successeurs. Calculer sa complexité pour les deux structures de données citées précédemment.
Cet algorithme s'écrit, de manière générique, de la façon suivante :

```
Calcul_somme_successeur(G)

Pour tout sommet x
  s[x]:=0
        pour tout successeur y de x
          s[x]:=s[x]+numero de y
  fin pour
fin pour
```

La complexité de cet algorithme est dépendant de la structure de donnée choisie pour représenter le graphe G. Nous allons détailler cet algorithme dans le cas où l'on utilise des listes d'adjacence ou une matrice d'adjacence

Calcul_somme_successeurs(α,β)	***Nombre d'exécutions***
pour *tout i:=1 à n*	$n+1$
s[i]=0	n
pour *tout j=α(i-1)+1 à α(i)*	$\sum_{i=1 \text{ à } n} d^+(i)+1=m+n$
s[i]=s[i]+ β[j]	$\sum_{i=1 \text{ à } n} d^+(i)=m$
fin pour	
fin pour	

Chaque ligne s'exécute en $\Theta(1)$ (c'est à dire en temps indépendant du nombre de sommets n et d'arcs m). La complexité est donc en $\Theta(m+n)$

Calcul_somme_successeurs(m)	***Nombre d'exécutions***
pour *tout i:=1 à n*	$n+1$
s[i]=0	n
pour *tout j:=1 à n*	$n(n+1)$
si *m(i,j)=1* ***alors***	$n(n+1)$
s[i]=s[i]+j	m
fin pour	
fin pour	

En adoptant le même raisonnement que précédemment, on montre que la complexité est en $\Theta(m+n^2)=\Theta(n^2)$ (car $m \leq n^2$).

8. Parcours de graphes

Dans la plupart des algorithmes rencontrés dans la théorie des graphes, il est nécessaire d'effectuer un examen exhaustif des sommets et des arcs (ou arêtes) du graphe étudié (par exemple, étude de la connexité d'un graphe...)

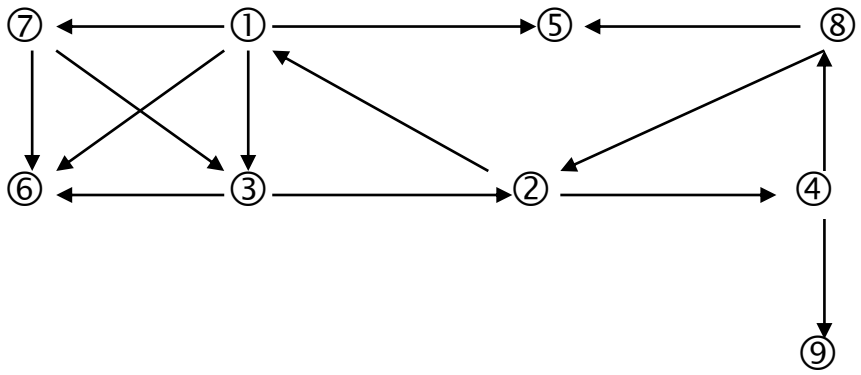
8.1. Parcours en largeur.

8.1.1. Introduction

A partir d'un sommet donné, l'exploration du graphe se fait niveau par niveau. Ce type de parcours est *intrinsèquement itératif.* Le principe du parcours en largeur est le suivant : pour un sommet de départ s, on commence à visiter tous les successeurs de s avant de visiter les autres descendants de *s*. Ainsi, le parcours en largeur, dont le sommet de départ est s, consiste à visiter tous les chemins de longueur 1, puis tous les chemins de longueurs 2 non déjà visités, …

Définition. Etant donné deux sommets s_1 et s_2, on appelle *distance de s_1 à s_2*, la longueur du plus court chemin allant de s_1 vers s_2.

Exemple :



Sommet de départ : s_1
s_3, s_5, s_6 et s_7 : distance 1 de s_1
s_2 : distance 2
s_4 :distance 3
s_8, s_9 : distance 4

8.1.2. Algorithme

La procédure *PL*, appelée pour un sommet s, visite tous les sommets qui peuvent être atteints à partir de *s* et qui n'ont pas été marqués, et seulement ceux-ci.
Elle utilise une structure de file. Rappelons que pour le type File, les adjonctions se font à une extrémité (procédure *Enfile*) et les accès (fonction *tête* qui permet d'accéder à la tête de la file) ou suppression (*Défile*) à l'autre extrémité. Utilisation d'une structure de file : lorsque l'on visite les successeurs non marqués d'un sommet s, il faut les ranger successivement dans une file Fifo.
La recherche au niveau suivant repartira des chacun des successeurs de s, à partir du premier de la file.

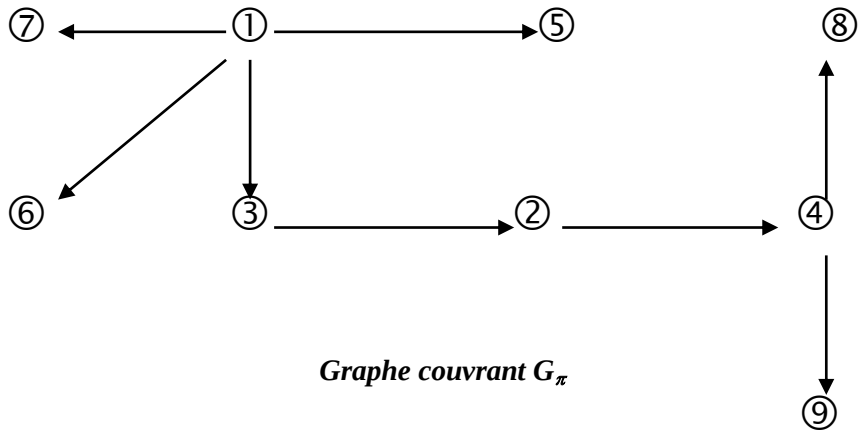
```
PL(G,s)
    Pour chaque sommet u∈S[G]
        d[u]:=∞
        π[u]:=nil
    fin pour
    d[s]:=0
    File:={s}
    Tant que Fifo≠∅
        u:=tête(Fifo)
        pour chaque successeur v de u
            si d[v]=∞ alors
                    d[v]:=d[u]+1
                    π[v]:=u
                    Enfile(Fifo,v)
```

fin si
fin pour
Défile(Fifo)
fin tant que

8.1.3.Plus courts chemins

Nous pouvons montrer que la valeur d[u], pour tout sommet u, correspond à la valeur du plus court chemin de s à u. S'il n'existe pas de chemin entre s et u alors d[u] reste infini. On appelle G_π le graphe couvrant défini à partir du dictionnaire des prédécesseurs π. G_π possède la propriété suivante :
s'il existe un chemin entre s et u dans G alors il existe un et un seul chemin de s à u dans G_π et c'est le plus court (sa distance étant égale à d[u]).

Exemple. *En reprenant l'exemple de début de section, un appel à la procédure PL(G,1) conduirait à visiter les sommets situés à une distance de 1, puis de 2, etc…jusqu'au sommet 5 situé à une distance de 5 du sommet 1*



Graphe couvrant G_π

8.1.4.Complexité

La première boucle s'exécute en $\Theta(n)$. Pour la suite, il suffit de remarquer qu'un élément u entre au plus une seule fois dans la file *Fifo.* Lorsque u arrive en tête de file alors ses successeurs sont visités et ensuite u est supprimé de la file. Aussi nous pouvons écrire que la procédure PL s'exécute en

O(m+n)

Comme nous ne pouvons que majorer le temps d'exécutions, nous devons utiliser la notation O(.).

8.2. Parcours en profondeur

8.2.1.Algorithme

A partir d'un sommet donné, il s'agit de suivre un chemin le plus loin possible, puis à faire des retours en arrière pour reprendre tous les chemins ignorés précédemment. De tels parcours se traitent naturellement de façon *récursive.*
On choisit un sommet de départ u et on appelle la procédure Visiter-PP(G,u). Le sommet u est alors *marqué*, et le principe utilisé est de suivre un chemin issu de u, aussi loin que possible, en marquant les sommets au fur et à mesure que l'on les atteints. Pour marquer un sommet on affecte la valeur temps à la variable d[u] (sa valeur était précédemment à ∞). En fin de chemin, on revient au *dernier choix* fait et on reprend une autre direction. Tous les sommets qui n'ont pas été marqués avant d[u] et qui peuvent être atteints à partir de u sont *visités*, et seulement eux. Une fois que tous les sommets visitables à partir de u ont été visités, on affecte la valeur temps à la variable f[u].
Le parcours total du graphe est obtenu par le programme principal qui appel la procédure de parcours pour un sommet non marqué, et qui recommence à l'issue de ce parcours, et ceci tant qu'il reste des sommets non marqués.

L'algorithme ainsi que ses application seront développés au cours du TD n°=1. La seule différence avec l'algorithme présenté est que la variable couleur a été supprimée.

```
PP(G)
    pour chaque sommet u∈S(G)
        π[u]:=nil
        d[u]:=0
    fin pour
    temps:=0
    pour chaque sommet u∈S(G)
        si d[u] = nil alors
            Visiter-PP(G,u)
      fin si
    fin pour

Visiter-PP(G,u)
    temps:=temps+1
    d[u]:=temps
    pour chaque v∈Γ+[u]
        si d[v] =0 alors
            π[v]:=u
            Visiter-PP(G,v)
        fin si
    fin pour
    temps:=temps+1
    f[u]:=temps
```

8.2.2. Complexité

Pour calculer la complexité de cet algorithme, il faut remarquer que la procédure Visiter-PP(G,u) est appelé exactement une fois pour chaque sommet. Il reste à remarquer que la complexité de la procédure Visiter-PP(G,u) est en $\Theta(d^+(u)+1)$. Aussi le temps total passé dans la procédure Visiter-PP est en

$$\sum_{u\in X}\Theta(1+d^+(u))= \Theta(\sum_{u\in X}(d^+(u)+1)) = \Theta(m+n)$$

On conclut en notant que le temps passé dans la procédure PP est en $\Theta(n)$. La complexité de l'algorithme de parcours en profondeur est donc en

$$\Theta(m+n)$$

8.2.3. Foret couvrante

On appelle foret couvrante, le graphe partiel de G généré par l'ensemble des arcs visités pendant le parcours en profondeur. Ce graphe partiel, appelé parfois foret couvrante couvrant ou graphe des liaisons est représenté dans le dictionnaire des prédécesseurs π ; chaque sommet u possède au plus un prédécesseur $\pi[u]$, propriété qui justifie la dénomination de foret couvrante pour ce graphe. Les arcs de cette foret couvrante sont appelés arcs couvrants ou arcs de liaisons. Plus généralement on peut distinguer 4 types d'arcs :

i. Les **arcs couvrants**
ii. Les arcs (x,y) tel que x est un descendant de y dans G_π, sont appelés **arcs en arrière**
iii. Les arcs (x,y) sont tels que y est descendant de x dans G_π sont appelés **arcs en avant**
iv. Tous les autres sont appelés **arcs croisés**.

Proposition. les trois assertions suivantes sont équivalentes :

i. v est un descendant de u dans le graphe couvrant
ii. $d[u]<d[v]<f[v]<f[u]$

iii. à la date d[u], il existe un chemin de sommets non marqués de u à v.

Dans les sections suivantes, nous verrons des applications de cet algorithme pour la détection de circuits et trouver les composantes fortement connexes.

8.3. Détection de circuits

De nombreux algorithmes (tels par exemple la pluspart des algorithmes d'optimisation) supposent que les graphes ne possèdent pas de circuits. De sorte qu'il est fréquent de tester l'absence de circuits dans un graphe.
Un premier algorithme, vu au cours du TD 1, consiste à faire un parcours en profondeur et à détecter des arcs en arrière. La complexité de ce premier algorithme de détection de circuits est en $\Theta(n+m)$.
Nous nous proposons de montrer un algorithme plus simple et plus efficace en pratique même si sa complexité est aussi en $\Theta(n+m)$. Cet algorithme s'appuie sur le résultat suivant.

8.3.1. Résultats préliminaires

Théorème : un graphe est sans circuit si et seulement s'il existe un sommet x sans successeur et si le sous-graphe engendré par X\{x} est sans circuits.

Preuve :

$\Rightarrow$ Soit G={X,U} un graphe sans circuit. Raisonnons par l'absurde et supposons que tous les sommets i de X possèdent un successeur. Construisons une suite de sommets $i_1, i_2,\ldots,i_n, i_{n+1}$ tels que $i_{j+1} \in \Gamma^+(i_j)$. Il existe donc deux entiers p et q tels que $i_p=i_q$. La séquence $i_p, i_{p+1},\ldots,i_q$ est donc un circuit.
Aussi X possède un sommet x sans successeur et trivialement le sous-graphe engendré par X\{x} ne possède pas de circuits (sinon G en possèderait un!)

$\Leftarrow$ Soit G={X,U} un graphe tel qu'il existe un sommet x sans successeur et tel que le sous-graphe engendré par X\{x} soit sans circuit. Supposons que G possède un circuit C. le sommet x n'ayant pas de successeur, $x \notin C$. Donc C est un circuit du sous-graphe engendré par X\{x} ce qui contredit l'hypothèse. G ne possède donc pas de circuit.

□

8.3.2. Algorithme

Du théorème précédent, on peut écrire un algorithme de complexité $\Theta(n+m)$ permettant de détecter le présence ou non de circuits. Le principe consiste à trouver un sommet sans successeurs, à l'enlever du graphe et à recommencer ceci tant que ceci est possible. Si tous les sommets peuvent être enlevés alors le graphe ne possède pas de circuits ; dans le cas contraire, il en possède un. En fait, plutôt que d'enlever réellement un sommet (opération coûteuse en O(m+n)), on maintient à jour les degrés de chaque sommets dans un tableau *degré* ; un sommet sans successeur est un sommet de degré nul. Les sommets de degré nul sont stockés dans un ensemble F. Lorsqu'un sommet est enlevé du graphe, on l'enlève de F.

Détection_de_circuits (G)
Pour *tout sommet x*
degré[x]:=0
traité[x]:=faux
fin pour
Pour *tout sommet x,* ***Pour*** *tout prédécesseur y de x, degré[y]:=degré[y]+1*

$F=\varnothing$
Pour *tout sommet x,* ***si*** *degré[x]=0* ***alors*** *$F=F\cup\{x\}$*

Tant que *$F\neq\varnothing$*

Soit x∈F
traité[x]:=vrai
***Pour** tout prédécesseur y de x tel que traité[y]=faux*
degré[y]:=degré[y]-1
***si** degré[y]=0 **alors** F:=F∪{y}*
Fin pour
Enlever x de F
Fin Tant que

***si** traité[x]:=vrai pour tout x, **alors retourner** pas de circuit*
***sinon retourner** circuit*

2.3.3 Complexité.

En supposant que le graphe soit représenté par deux tableaux de listes d'adjacence (un pour les successeurs et un pour les prédécesseurs), et en remarquant qu'un sommet ne peut "rentré" qu'une seule fois dans la file F, la complexité de cet algorithme est en

$$\Theta(m+n)$$

3. Décomposition en niveaux

La décompostion en niveaux consiste à partitionner les sommets. Cette décomposition, utile dans quelques problèmes d'ordonnancement, n'est possible que si le graphe G est sans circuits.

3.1. Définition

Considérons un graphe G sans circuit. Nous classons les sommets du graphe G en niveaux. On définit le niveau N(0) comme étant l'ensemble des sommets qui ne possèdent pas de successeurs. Nous dirons qu'un sommet u appartient au niveau N(i) si le plus long chemin entre u et un sommet v∈N(0) est de longueur i. On peut réécrire la définition précédente par une formule équivalence de réccurence.

$N(0)=\{x / x\in X, \Gamma^+(x)=\emptyset\}$
$N(i)=\{x/ x\in X\setminus\cup_{j<i}N(j), \Gamma^+(x)\cap X\setminus\cup_{j<i}N(j)=\emptyset\}$

***Exercice**. Montrer que si G possède un circuit, alors cette décomposition est impossible.*

3.2. Algorithme

Une légére modification de l'algorithme précédent (Détection_de_circuits) permet calculer les différents niveaux. Comme pour la détection de circuits, on enlève (en fait on maintient à jour les degrés de chaque sommets) les sommets sans successeurs, on les mets dans le niveau courant et on recommence sur le sous-graphe courant. l'algorithme "*Décomposition_niveaux"* décompose le graphe en niveaux si G ne possède pas de circuits et renvoie faux si G possède un circuit.

Décomposition niveaux (G)

***Pour** tout sommet x*
degré[x]:=0
traité[x]:=faux
fin pour
***Pour** tout sommet x, **Pour** tout prédécesseur y de x, degré[y]:=degré[y]+1*

i:=0
N(0)=∅

***pour** tout sommet x, **si** degré[x]=0 **alors** N(0=N(0)∪{x} et niveau[x]=0*

***Tant que** fin=faux*

Tant que *N(i)≠∅*
Soit x∈N(i)
E=∅
traité[x]:=vrai
Pour *tout prédécesseur y de x tel que traite[y]=faux*
degré[y]:=degré[y]-1
si *degré[y]=0* ***alors*** *N(i+1):=N(i+1)∪{y} et niveau[y]=i+1*
Fin pour
Enlever x de N(i)
fin tant que
i:=i+1
si *N(i+1)=∅* ***alors*** *fin=vrai*
fin tant que

si *traité[x]=vrai pour tout x,* ***alors*** *retourner niveau*
sinon *retourner présence circuit*

3.3. Complexité

La complexité de cet algorithme est exactement la même que celle de l'algorithme précédent, i.e. en

Θ(m+n)

3.4. Numérotation topologique

Nous dirons que les sommets sont numérotés avec une numérotation topologique si pour tout arc (i,j) le numéro de l'extrémité initiale est inférieur à celui de l'extrémité finale, i.e. i<j.

Proposition. Soit G un graphe. Si G possède un circuit alors il n'existe pas de numérotation topologique des sommets

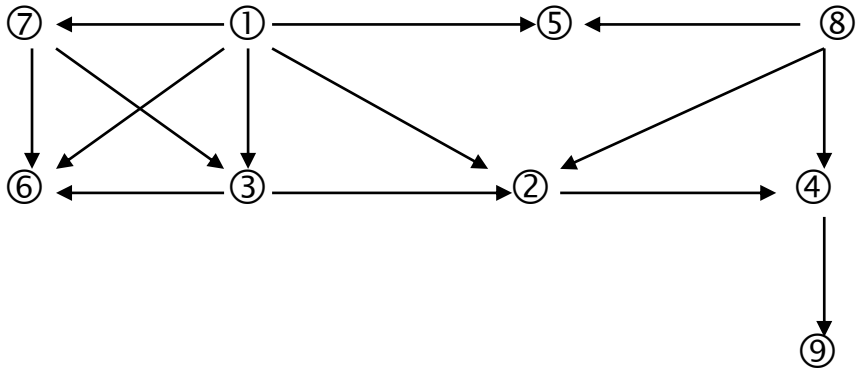
Preuve : soit $(x_1, x_2, \ldots, x_p, x_1)$ *un circuit. Raisonnons par l'absurde et supposons que la numérotation des sommets soit topologique. Par définition, ceci implique que* $x_1<x_2<\ldots<x_p<x_1$
D'où contradiction.

Pour les algorithmes sans circuits, la décomposition en niveaux va permettre de définir une numérotation topologique grâce à la proposition suivante.

Proposition. Soit G un graphe sans circuit. Appelons n_i le cardinal du niveau N(i). Alors la numérotation suivante est topologique : les sommets de N(0) sont numérotés de 1 à n_0 et ceux de N(i), i>0 de $n-(n_0+\ldots+n_i-1)$ à $n-(n_0+\ldots+n_{i-1})$. La numérotation obtenue est topologique.

Preuve :
Soit (u,v) un arc. Supposons que v∈N(i) pour i donné. Par définition, ceci signifie que la longueur de plus long chemin entre v et un sommet de N(0) (ensemble des sommets sans successeurs) est i. Soit C ce chemin. Aussi en concaténant l'arc (u,v) et le chemin C, on construit un chemin de longueur i+1 entre u et un sommet de N(0). Le niveau de u est donc supérieur ou égal à i+1. La numérotation proposée garantit que le numéro de u est inférieur au numéro de v. La numérotation est donc topologique.

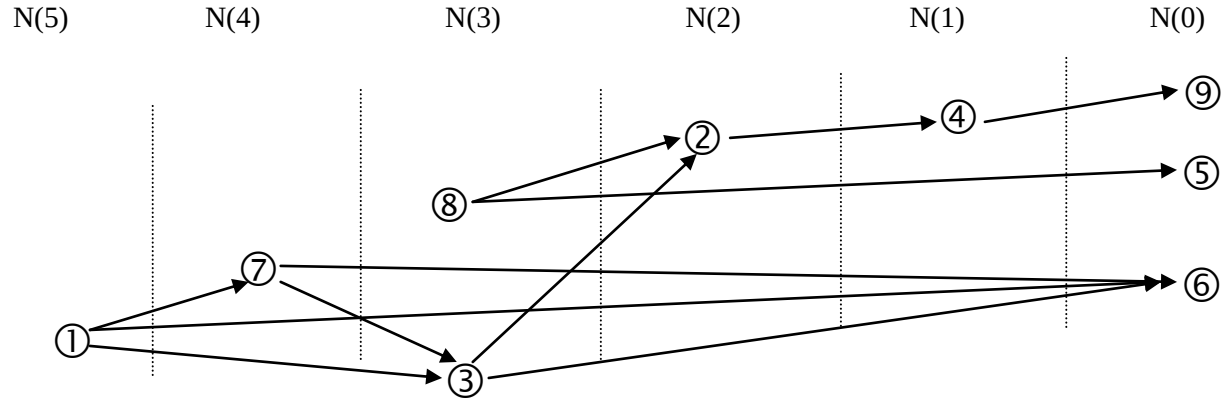
Exemple. Considérons le graphe suivant



Ce graphe est sans circuit, nous pouvons donc le décomposer en niveaux. En appliquant l'algorithme précédent, nous obtenons la décomposition en niveau suivante

N(0)={9,6,5}
N(1)={4}
N(2)={2}
N(3)={3,8}
N(4)={7}
N(5)={1}

On peut ainsi représenter le graphe en mettant les sommets de gauche à droite par ordre de niveau décroissant. On constate dans ce cas, que tous les arcs sont dirigés vers la droite.



Pour obtenir une numération topologique, il suffit d'appliquer la proposition précédente. On pourrait par exemple renuméroter les sommets de la manière suivante.
$1 \to 1$; $7 \to 2$; $3 \to 3$; $8 \to 4$; $2 \to 5$; $4 \to 6$; $9 \to 7$; $6 \to 8$; $5 \to 9$.

4. Composantes fortement connexes

Nous allons, dans cette section, appliquer l'algorithme de parcours en profondeur à la décomposition d'un graphe en composantes fortement connexes. La décomposition d'un graphe en composantes fortement connexes permet parfois de diviser un problème en sous-problèmes, un par composante fortement connexe. La combinaison de ces "sous-solutions" suit la structure des connections entre les composantes fortement connexes. Cette structure peut être représentée par un graphe dit "réduit" (voir TD n°=4).

4.1. Algorithme basique

Nous rappelons que deux sommets u et v appartiennent à la même composante fortement connexe si et seulement s'il existe un chemin de u à v et de v à u. Aussi la composante fortement connexe d'un

sommet u est l'ensemble des sommets v qui sont à la fois ascendants et descendants de u. Ceci est à la base de l'algorithme suivant.

```
Composante_fortement_connexe_1(G)
    numéro:=1
    pour tout sommet u
        choisi[u]:=0
    fin pour
    tant qu'il existe un sommet u tel que choisi[u]=0
        calculer les ascendants de u pour lesquels choisi[u]=0
        calculer les descendants de u pour lesquels choisi[u]=0
        pour tous les sommets v à la fois descendants et ascendants de u
            choisi[v]:=1
            numéro_composante[v]:=numéro
        fin pour
        numéro:=numéro+1
    fin tant que
```

La validité de cet algorithme est évidente au vu de ce qui a été dit précédemment. Le calcul des descendants de u peut se faire à l'aide d'un parcours en profondeur ou en largeur. Le calcul des ascendants peut se faire avec le même algorithme en inversant le sens des arcs (on note G^{-1} le graphe ou les arcs de G sont inversés).
Le calcul des ascendants (ou des descendants) peut se faire par un parcours en largeur ou en profondeur. Sa complexité est donc en O(m+n). La boucle tant que s'exécute au plus n fois. La complexité de l'algorithme précédent est donc en O(n(m+n))

4.2. Amélioration de l'algorithme précédent

L'algorithme précédent peut être améliorée si lors du parcours en profondeur, plus d'informations est utilisée. L'idée de l'algorithme suivant est d'utilisée les dates f[u] de fin de traitement d'un sommet de la procédure de parcours en profondeur PP. Nous ne démontrerons pas la validité de cet algorithme qui peut paraître à première vue, un peu mystérieuse.

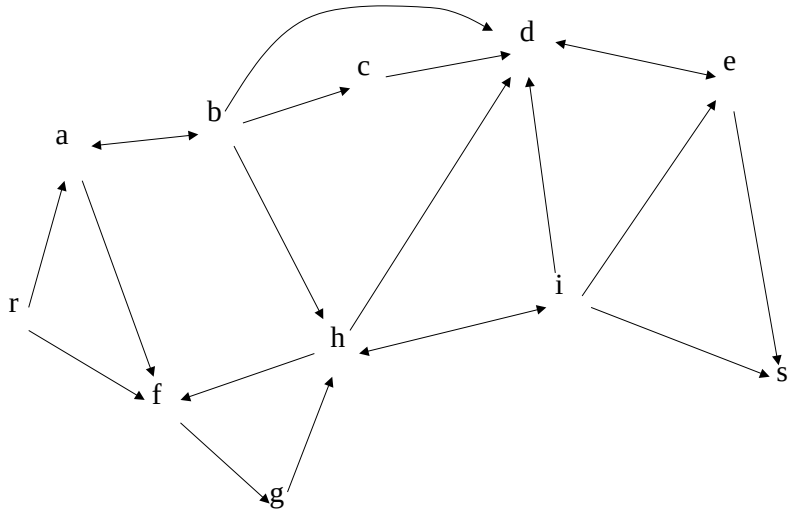
Composante_fortement_connexe_2(G)
PP(G)
PP(G^{-1}) //en considérant les sommets v en ordre décroissant de f[v] dans la boucle principale de PP
Une composante fortement connexe est constituée de tous les sommets découverts lors de l'exécution de Visiter-PP(G,u) pour un sommet u choisi dans la procédure PP (lancée sur le graphe G^{-1}). Autrement dit, Les composantes fortement connexes de G sont les composantes faiblement connexes du graphe de liaison G_π obtenue lors du second parcours en profondeur.

Exercice. *Détailler l'algorithme précédent.*

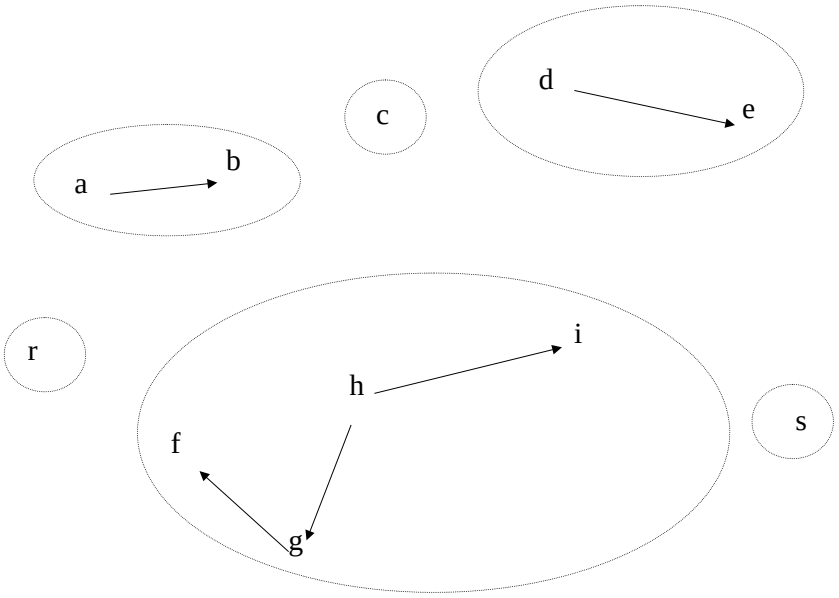
La complexité de cet algorithme est égale à la complexité de l'algorithme PP, i.e. en O(m+n). L'amélioration est donc significative par rapport à l'algorithme précédent.

4.3. Exemple.

Considérons le graphe ci-dessous. Exécutons l'algorithme **Composante_fortement_connexe_2(G)** sur ce graphe. Le premier parcours en profondeur PP peut conduire (ce n'est pas la seule possibilité) aux dates de parcours f suivantes.
f[r]=21; f[a]=20;f[b]=19; f[c]=10; f[d]=9; f[e]=8; f[f]=17; f[g]=16; f[h]=18; f[i]=13; f[s]=7.
Construisons ensuite le graphe G_π obtenu lors du second parcours en profondeur, i.e sur le graphe G^{-1} en choisissant, dans la boucle principale PP, les sommets u par ordre décroissants sur leur date de fin de traitement f[u] lors du premier parcours en profondeur.



Le graphe G_π obtenu est le suivant :



Les composantes fortement connexes sont donc : {r}, {a,b}, {c}, {d,e}, {f,g,h,i}, {s}