Table des matières

[INTRODUCTION GENERALE 3](#_Toc89675147)

[Partie 1. NOTIONS ELEMENTAIRES 4](#_Toc89675148)

[1.1 DEFINITION 4](#_Toc89675149)

[1.2. Connexité 5](#_Toc89675150)

[1.2.1. Chaînes et chemins 5](#_Toc89675151)

[1.2.2. Graphes et composantes connexes et fortement connexes 6](#_Toc89675152)

[Partie 2. Représentation des graphes 8](#_Toc89675153)

[2.1. Matrices d’adjacence et d’incidence 8](#_Toc89675154)

[2.1.1. Matrice d’adjacence 8](#_Toc89675155)

[2.1.2. Matrice d’incidence sommets-arcs 8](#_Toc89675156)

[2.1.3. Matrice d’incidence sommets-arêtes 8](#_Toc89675157)

[2.2.1. A partir de la matrice d’adjacence 8](#_Toc89675158)

[2.2.2. A partir de la matrice d’incidence sommets-arcs ou sommets-arêtes 9](#_Toc89675159)

[Partie 3. Parcours des graphes 10](#_Toc89675160)

[3.1. Parcours en profondeur d’abord : DFS (Depth First Search) 10](#_Toc89675161)

[3.2. Détermination des composantes connexes 12](#_Toc89675162)

[3.2.1. Cas non oriente 12](#_Toc89675163)

[3.2.2. Cas orienté 13](#_Toc89675164)

[3.3. Détermination des composantes fortement connexes 13](#_Toc89675165)

[3.4. Tri topologique dans un graphe oriente sans circuit 15](#_Toc89675166)

[3.5. Fermeture transitive d’un graphe 17](#_Toc89675167)

[3.5.1. Algorithme de parcours de graphes 17](#_Toc89675168)

[Partie 4. Problèmes de meilleurs chemins 22](#_Toc89675169)

[4.1. Plus courts chemins d’origine fixée dans un graphe avec longueurs non négatives : algorithme de Dijkstra 22](#_Toc89675170)

[4.2. Mise en œuvre de l’algorithme de Dijkstra pour les graphes peu denses : algorithme de Johnson 24](#_Toc89675171)

[4.2.1. Présentation de la structure de données de tas (heap) 24](#_Toc89675172)

[4.2.2. Algorithme de Johnson 24](#_Toc89675173)

[4.3. Plus court chemin d’origine fixée dans un graphe sans circuit avec longueurs quelconques : algorithme de Bellman 25](#_Toc89675174)

[4.4. Détermination de plus courts chemins d’origine fixée dans un graphe avec longueurs quelconques 26](#_Toc89675175)

[4.4.2. Algorithme de Ford-Bellman 27](#_Toc89675176)

[Partie 5. Arbres couvrants 29](#_Toc89675177)

[5.1. Vocabulaire lié aux arbres couvrants 29](#_Toc89675178)

[5.2. Propriétés des arbres couvrants 29](#_Toc89675179)

[5.3. Arbre couvrant de poids minimal 30](#_Toc89675180)

[5.4. Aspects algorithmiques des arbres couvrants 32](#_Toc89675181)

[Partie 6. Problèmes de flots 38](#_Toc89675182)

[6.1. Amélioration d'un flot 39](#_Toc89675183)

[6.2. Recherche d'un flot maximal 40](#_Toc89675184)

[6.4. Complexité de l’Algorithme de Ford et Fulkerson 43](#_Toc89675185)

[Partie 7. Ordonnancement et graphes 50](#_Toc89675186)

[7.1. Représentation graphique 51](#_Toc89675187)

[7.2. Procédure d’ordonnancement des projets 54](#_Toc89675188)

[CONCLUSION GENERALE 56](#_Toc89675189)

[Références bibliographiques 57](#_Toc89675190)

# INTRODUCTION GENERALE

La théorie des graphes et l’algorithmique est un des outils privilégies de modélisation et de résolution de problèmes dans un grand nombre de domaines allant de la science fondamentale aux applications technologiques concrètes. Par exemple, les graphes déterministes et/ou aléatoires sont utilisés en physique, en sciences sociales (pour représenter des relations entre groupes d’individus), en mathématique combinatoire, en informatique (structures de données et algorithmique). Concernant les très nombreuses applications, on peut citer : les réseaux électriques et transport d’énergie, le routage du trafic dans les réseaux de télécommunications et les réseaux d’ordinateurs, le routage de véhicules et l’organisation des tournées ou rotations, les problèmes de localisation (d’entrepôts, d’antennes, . . .) et de placement, les problèmes d’ordonnancement de tâches et d’affectation de ressources… Ainsi la maitrise de l’outil théorie des graphes et algorithmique permettra de résoudre des problèmes d’ordre : **COMBINATOIRE**, **STOCHASTIQUE** et **CONCURRENTIEL**. Pour une meilleure compréhension de ce cours nous allons aborder les notions suivantes : les matrices associées à un graphe, la terminologie : graphe (orienté), connexité…, le problème de plus court chemin, le problème de coloration, parcours hamiltonien, Parcours eulériens, l’ordonnancement des tâches, flots et réseaux de transport.

# Partie 1. NOTIONS ELEMENTAIRES

## 1.1 DEFINITION

**Un graphe oriente** G= (X, U) est défini par :

– Un ensemble X={x1, x2,..., xn} dont les éléments sont appelés des sommets ou des nœuds. L’ordre du graphe G est le nombre de sommets n.

– Un ensemble U = {u1, u2,..., Um} dont les éléments, appelés des arcs, sont des couples ordonnés de X × X = {(x, y)|x ∈ X, y ∈ X}. Pour un arc u = (x, y) avec (x, y) ∈ X2, on dit que x est l’extrémité initial et y l’extrémité terminale de u, y est le successeur de x et x est le prédécesseur de y. On note |U| = m.

**Une boucle** est un arc ayant le même sommet comme extrémité initiale et terminale: u= (x, x) est appelé une boucle, ∀x ∈ X.

**Un p-graphe** est un graphe dans lequel il n’existe jamais plus de p arcs de la forme (x, y) entre x et y : **p = max(x, y)∈X2|{u ∈ U |u= (x, y)}|.**

Γ+(x**), l’ensemble des successeurs** de x, ∀x ∈ X, est une application multivoque de X dans une partie de X telle que : **Γ+(x) = {y ∈ X|(x, y) ∈ U}.** De même, Γ−(x), l’ensemble des prédécesseurs de x, ∀x ∈ X, est une application multivoque (réciproque) de X dans une partie de X telle que : **Γ−(x) = {y ∈ X |(y, x) ∈ U}.∀x ∈ X, Γ(x) = Γ+(x) ⋃ Γ−(x)** est **l’ensemble des voisins de x**. Un sommet x est dit **isolé** si Γ(x) = {x}. Un sommet x est adjacent`a Y si x /∈ Y et x **∈Γ−(Y) ⋃ Γ+(Y) ou Γ−(Y) =⋃Γ−(y) ∀y ∈ Y et Γ+(Y) = ⋃Γ+(y) ∀y ∈ Y**.

Si G est un 1-graphe, il est parfaitement défini par l’ensemble X et l’application multivoque Γ+ou Γ−. On peut alors le noter G= (X, Γ+) ou G= (X, Γ−).

**Le demi-degré extérieur (resp. intérieur)** du sommet x, note d+(x) (resp. d−(x)) désigne le nombre d’arcs ayant x comme extrémité initiale (resp. terminale), d+(x) = |Γ+(x)| et d−(x) =|Γ−(x)|.

Soit Y ⊂ X un sous-ensemble de sommets :

* ω+(Y) est l’ensemble des arcs ayant leurs extrémités initiale dans Y et leur extrémités terminales dans X \ Y.
* ω−(Y) est l’ensemble des arcs ayant leur extrémité terminale dans Y et leur extrémité initiale dans X \ Y.
* ω(Y) = ω+(Y) ⋃ ω−(Y) est appelé Co-cycle du graphe. ω(Y) est dit Co-circuit si ω+(Y) = ∅ ou ω−(Y) = ∅.

Un graphe G est dit c**omplet** si pour toute paire (x, y) de sommets (avec x ≠ y) il existe au moins un arc (x, y) ou (y, x). Un 1-graphe est complet si et seulement si : (x, y) ¢ U ⇒ (y, x) ∈ U.

Un graphe G est dit symétrique si (x, y) ∈ U ⇒ (y, x) ∈ U.

Un graphe G est dit antisymétrique si (x, y) ∈ U ⇒ (y, x) ¢ U.

Un graphe G est dit biparti si X1 ⋃ X2 = X, X1 ⋂ X2 = ∅ et X1 ⋂ Γ+(X1) = ∅ (i.e. Γ+(X1) ⊂ X2), X2 ⋂ Γ+(X2) = ∅ (i.e. Γ+(X2) ⊂ X1).

Un graphe non oriente G = (X, E) est défini par :

– Un ensemble X={x1, x2,..., xn} dont les éléments sont appelés des sommets ou des nœuds. L’ordre du graphe G est le nombre de sommets n

– Un ensemble E = {e1, e2,..., em} dont les éléments, appelés des arêtes, sont des couples non ordonnes de X∗X = {(x, y) |x ∈ X, y ∈ X}. On note |E| = m.

Un multigraphe est un graphe non oriente pour lequel il peut exister plusieurs arêtes entre deux sommets x et y donnes.

Un graphe non oriente est dit simple s’il est sans boucle et s’il n’existe pas plus d’une arête entre toute paire de sommets.

Deux arcs (arêtes) sont dit(e)s adjacent(e)s s’ils ont au moins une extrémité commune.

Le degré du sommet x, note d(x), est le nombre d’arcs ou d’arêtes ayant x comme extrémité. On a d(x) =d +(x) + d−(x) dans le cas d’un graphe oriente (attention : x est comptabilise deux fois encas de boucle).

Une clique est un graphe simple pour lequel il existe exactement une arête entre toute paire de sommets. On la note Kn si n est son nombre de sommets.

Le sous-graphe SG(Y) de G = (X, U) engendré par le sous-ensemble de sommets Y ⊂ X est le graphe dont les sommets sont les éléments de Y et les arcs les éléments de U ayant leurs deux extrémités dans Y: SG(Y) = (Y, UY) avec UY = {u ∈ U |u= (x, y), x, y ∈ Y2}.

Le graphe partiel GP(V) de G= (X, U) engendre par le sous-ensemble d’arcs V ⊂ U est le graphe dont les sommets sont ceux de X et les arcs ceux de V : GP(V) = (X,V). Le sous-graphe partiel SGP de G = (X, U) engendre par le sous-ensemble de sommets Y ⊂ X et le sous-ensemble d’arcs V ⊂ U est le graphe partiel GP(V) du sous-graphe SG(Y) ou le sous-graphe SG(Y) du graphe partiel GP(V) : SGP = (Y, V) avec Y ⊂ X et V ⊂ U.

Deux graphes G= (X, U) et G′= (X′, U′) sont dits isomorphes s’il existe deux bijections : g : X→X′ et h: U→U′ telles que ∀u = (x, y) ∈ U, h(u) = (g(x), g(y)) ∈ U′.

## 1.2. Connexité

### 1.2.1. Chaînes et chemins

**Chaînes et cycles** Une chaîne de longueur l allant du sommet x au sommet y est une suite d’arêtes μ(x, y) = (u1, u2,..., ul) tels que ∀i∈ {1, ..., l−1}, ui et ui+1 sont adjacents. Le sommet x (resp. y) est appelé l’extrémité initiale (resp. terminale) de la chaîne μ.

Une chaîne est dite élémentaire si elle ne passe jamais deux fois par le même sommet.

Une chaîne est dite simple si elle ne passe jamais deux fois par la même arête.

Remarque :

– Dans un graphe simple, si une chaîne est élémentaire, alors elle est simple.

– Dans le cas d’un graphe simple, une chaîne est parfaitement caractérisée par la suite des sommets qu’elle rencontre ou par l’extrémité initiale de la chaîne et la suite d’arêtes.

Une chaine simple de longueur m (i.e. maximale) est **une chaîne eulérienne**.

Un cycle est une chaîne dont les extrémités initiale et terminale coïncident. Un cycle élémentaire est un cycle minimal, au sens de l’inclusion, i.e. ne contenant strictement aucun cycle.

Chemins et circuits Un chemin de longueur l allant du sommet x au sommet y est une suite d’arcs μ[x, y] = (u1, u2,..., ul) tels que ∀i∈ {1, ..., l−1}, ui et ui+1 sont adjacents et l’extrémité terminale de ui est l’extrémité initiale de ui+1. Le sommet x (resp. y) est appelé l’extrémité initiale (resp. terminale) du chemin μ.

Un chemin est dit élémentaire s’il ne passe jamais deux fois par le même sommet.

Un chemin est dit simple s’il ne passe jamais deux fois par le même arc.

Remarque :

– Dans un graphe simple, si un chemin est élémentaire, alors il est simple.

– Dans le cas d’un graphe simple, un chemin est parfaitement caractérise par la suite des sommets qu’il rencontre ou par l’extrémité initiale du chemin et la suite d’arcs.

Un chemin élémentaire de longueur n−1 (i.e. maximale) est **un chemin hamiltonien.** Un circuit est un chemin dont les extrémités initiale et terminale coïncident. Un circuit élémentaire est un circuit minimal, au sens de l’inclusion, i.e. ne contenant strictement aucun circuit.

Un circuit hamiltonien est un chemin hamiltonien μ [xi1, xin] = (xi1, xi2, ..., xin) complété par (xin, xi1).

### 1.2.2. Graphes et composantes connexes et fortement connexes

**Connexité simple** Un graphe G est dit **connexe** si ∀(x, y) ∈ X2 x ≠ y, ∃μ (x, y).

**Proposition:** La relation R définie par xRy ⇔ x = y ou ∃μ (x, y) est une relation d’équivalence.

Preuve:

– Elle est réflexive : xRx.

– Elle est symétrique : xRy ⇒ yRx car il suffit d’inverser la chaîne.

– Elle est transitive : xRy et yRz ⇒ xRz car il suffit de concaténer les deux chaînes. Les classes d’équivalence induites sur X par R constituent une partition de X en X1, X2,..., Xp. Le nombre p de classes d’ ́équivalence est appelé le nombre de connexité graphe. Un graphe est dit connexe si et seulement si son nombre de connexité est égal a 1. Les sous-graphes G1, G2,..., G p engendres par les sous-ensembles X1, X2,..., Xp sont appelés les **composantes connexes** de G. Chaque composante connexe est un graphe connexe.

Connexité forte Un graphe G est dit **fortement connexe** si ∀(x, y) ∈ X2 x ≠ y, ∃μ[x, y]. Proposition: La relation R’ définie par xR’y ⇔ x = y ou ∃μ[x, y] et ∃μ[y, x] est une relation d’équivalence.

**Preuve:**

**–** Elle est réflexive : xR’x.

– Elle est symétrique : xR’y ⇒ yR’x.

– Elle est transitive : xR’y et yR’z ⇒ xR’z.

Les classes d’équivalence induites sur X par R’ constituent une partition de X en X1, X2,..., Xq. Le nombre q de classes d’équivalence est appelé le **nombre de connexité forte** du graphe. Un graphe est dit fortement connexe si et seulement si son nombre de connexité forte est égal a 1. Les sous-graphes G1, G2,..., Gq engendres par les sous-ensembles X1, X2, ..., Xq sont appelés **les composantes fortement connexes** de G.

On appelle **le graphe réduit**, note Gr, le graphe défini comme suit : les sommets de Gr représentent les composantes fortement connexes et il existe un arc entre deux sommets x et y si et seulement s’il existe au moins un arc entre un sommet de Xx et un sommet de Xy dans le graphe G.

Remarque : Un graphe réduit est nécessairement sans circuit.

# Partie 2. Représentation des graphes

## 2.1. Matrices d’adjacence et d’incidence

### 2.1.1. Matrice d’adjacence

Soit G= (X, U) un 1-graphe d’ordre n, la matrice d’adjacence A = (aij(n×n)) a coefficients 0 ou 1 est définie comme suit :

* **axy = 1 ⇐⇒ (x, y) ∈ U‚**
* **axy= 0 sinon.**

### 2.1.2. Matrice d’incidence sommets-arcs

**Soit G** = (X, U) un graphe oriente sans boucle d’ordre n, la matrice d’incidence sommets-arcs est une matrice A = (aij(n×m)) a coefficients entiers 0, +1 ou −1 telle que chaque colonne correspond a un arc et chaque ligne a un sommet. Si u = (x, y) est un arc du graphe G alors la colonne u a tous ses termes nuls sauf :

* **axu = +1**
* **ayu = −1**

### 2.1.3. Matrice d’incidence sommets-arêtes

Soit G = (X, U) un graphe non oriente sans boucle d’ordre n, la matrice d’incidence sommets-arêtes est une matrice A= (aij(n×m)) a coefficients entiers 0 ou 1 telle que chaque colonne correspond a un arc et chaque ligne a un sommet. Si u = (x, y) est une arête du graphe G alors la colonne u a tous ses termes nuls sauf :

* **axu = 1**
* **ayu = 1**

**2.2. Représentation des graphes en machine**

### 2.2.1. A partir de la matrice d’adjacence

L’occupation mémoire d’une matrice d’adjacence est O(n2). Dans le cas de graphes peu denses, m << n2 pour les graphes orientes et m << n(n+ 1)/2 pour les graphes non orientes, il est plus avantageux d’écrire uniquement que les termes non nuls de la matrice d’adjacence.

**Représentation par liste d’adjacence** On mémorise pour chaque sommet la liste de ses prédécesseurs ou de ses successeurs en utilisant des listes chainées. Dans le cas oriente, cette représentation revient a décrire le graphe par son application multivoque Γ−ou Γ+.

**Remarque :**

* Les deux représentations seront différentes si le graphe n’est pas symétrique.
* Le désavantage d’une telle représentation est que le test d’appartenance d’un arc entre x et y peut atteindre O(n) puisque l’on peut trouver O(n) sommets sur la liste des successeurs de i.

### 2.2.2. A partir de la matrice d’incidence sommets-arcs ou sommets-arêtes

**La** matrice d’incidence permet de représenter un p-graphe ou un multigraphe (sans boucle). Comme il n’existe que deux termes non nuls par colonne dans cette matrice, on a toujours intérêt à représenter l’information sous une forme plus condensée. Par liste des arcs On d ́écrit la matrice d’incidence ligne par ligne: pour chaque sommet x ∈ X, on d ́écrit la liste ω+(x) des arcs ayant x comme extrémité initiale (ou comme extrémité dans le cas non oriente).

Par **liste des cocycles** On d ́écrit la matrice d’incidence colonne par colonne.

# Partie 3. Parcours des graphes

De nombreuxproblèmes fondamentaux en théorie des graphes concernent la connexité. On peut citer par exemple :

* Un sommet y est-il accessible par un chemin `a partir d’un autre sommet ?
* Quel est l’ensemble de tous les sommets accessibles par un chemin `a partir d’un sommet x?
* Le graphe est-il connexe, c’est-`a-dire tous les sommets sont-ils accessibles par une chaîne a partir d’un sommet donne x?
* Le graphe est-il fortement connexe, c’est-`a-dire existe-t-il un chemin joignant toute paire ordonnée (x, y) de sommets dans le graphe?

Pour répondre a toutes ces questions, **Tarjan** a propos ́e en 1972 un ensemble d’algorithmes efficaces (de complexité linéaire en fonction du nombre d’arcs ou d’arêtes du graphe) qui sont fondes sur une méthode d’exploration systématique des sommets d’un graphe connue sous le nom de parcours en profondeur d’abord.

## 3.1. Parcours en profondeur d’abord : DFS (Depth First Search)

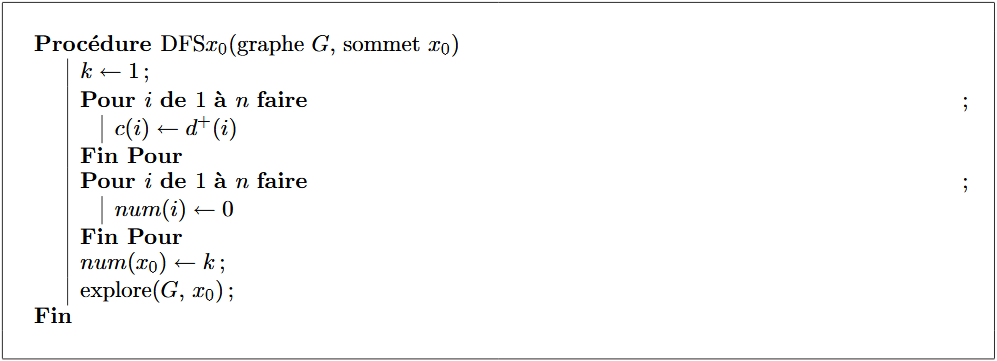
Il s’agit d’une généralisation du parcours préfixe des arbres. On explore G à partir d’un sommet x0 quelconque. Au cours de l’exploration, chaque sommet peut se trouver dans l’un des deux états suivants :

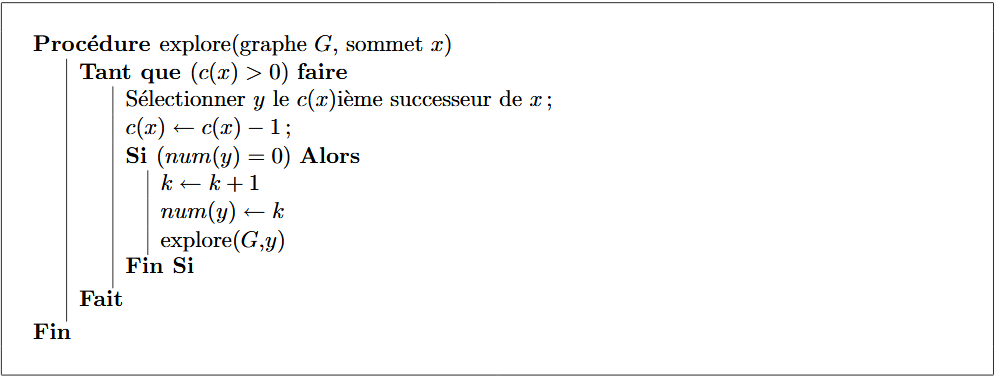
* E1: non explore ;
* E2: exploré

**Remarque:** Si un sommet est dans l’état E2 alors au moins un prédécesseur de ce sommet a déjà été explore. Un sommet x explore peut être dans l’un deux états suivants :

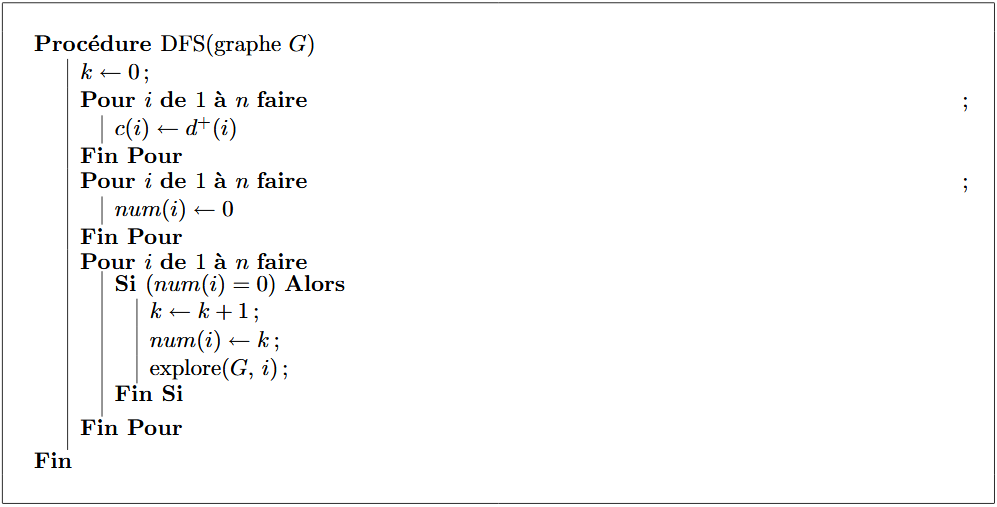
* S1: ferme. C’est le cas quand tous les successeurs y de x ont été explores ;
* S2: ouvert. C’est le cas quand certains successeurs y de x n’ont pas encore été explores ;

Pour chaque sommet x, on dispose d’un compteur c(x) indiquant a chaque instant le nombre de successeurs de x non encore explores. Initialement c(x) =d+(x) ∀x. Un sommet x est donc ouvert tant que c(x) > 0. A l’itération courante, on part du sommet explore x (explore a partir d’un sommet). Si x est ouvert, on sélectionne le c(x)ieme successeur non explore y de x (il faut alors décrémenter c(x) de 1), pour passer y a l’état explore et commencer une nouvelle itération a partir de y. Si x est ferme, il faut remonter au sommet s pour commencer une nouvelle itération a partir des. L’exploration `a partir de x0 se termine lorsque l’on est remonte en ce sommet et qu’il est ferme. On a donc pu atteindre tous les sommets pouvant être atteints par un chemin a partir de x0 dans le graphe. Rappelons qu’un sommet x relie par un chemin a tous les autres sommets d’un graphe est appelé racine. Voici l’algorithme en pseudo-langage de cette exploration par un parcours en profondeur d’abord :

  
 Algorithme 1 : Parcours en profondeur d’abord a partir du sommet x0



Algorithme 2 : Exploration en profondeur d’abord

DFSx0(G, x0) permet d’explorer tous les sommets de G s’il est possible de les atteindre a partir de x0. Si ce n’est pas le cas, il reste des sommets non explores. Pour continuer l’exploration et couvrir tous les sommets de G, on choisit un sommetx1parmi les sommets non explores (tels que num(x1) = 0) et on applique DFS a partir de x1; et ainsi de suite tant qu’il existe des sommets non explores. L’algorithme prend fin lorsque tous les sommets été explorés. 

Algorithme 3 : Parcours en profondeur d’abord

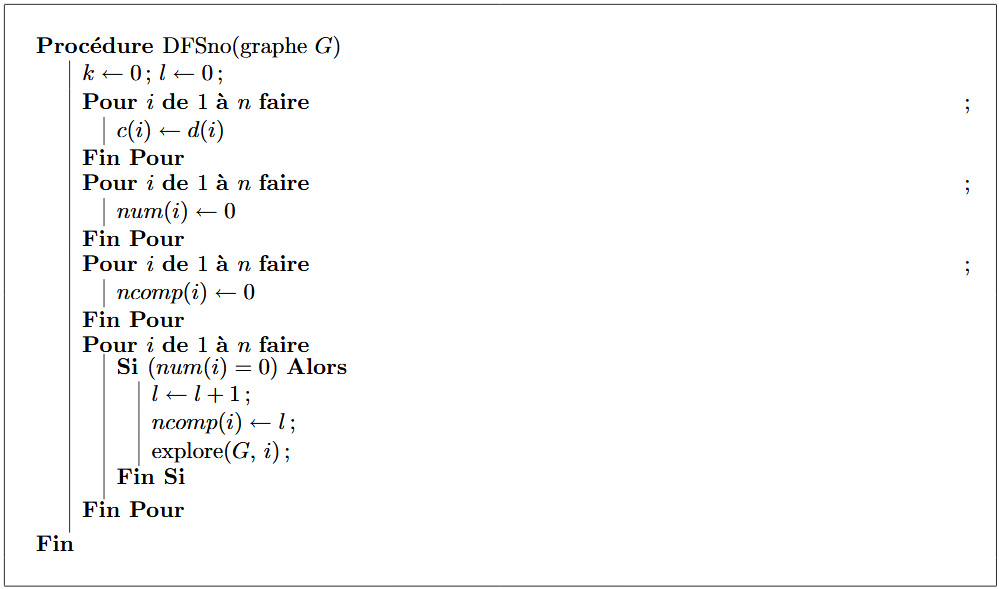
La complexité de cet algorithme est en O(m) +O(n) car chaque arc est examiné e au plus une fois et chaque sommet est explore au plus une fois. Or si G est connexe, m ≥ n−1 et le terme O(n) est absorbé.

On obtient alors une foret décrivant l’exploration de G par DFS, mais ́également une caractérisation des arcs du graphe. On distingue 4 types d’arc :

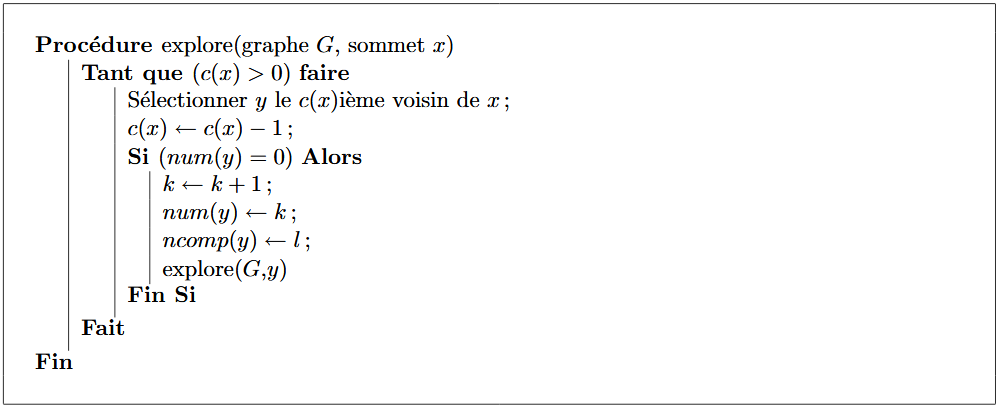
* Arc d’arbre: arc ayant permis de visiter de nouveaux sommets lors de l’exploration par DFS.
* Arc avant: (x, y) est un arc avant si et seulement si (x, y) n’est pas un arc d’arbre et y est un descendant de x dans l’un des arbres de la foret décrivant l’exploration par DFS (i.e. qu’il existe un chemin de x a y dans l’un des arbres).
* Arc arrière: (x, y) est un arc arrière si et seulement si (x, y) n’est pas un arc d’arbre et x est un descendant de y dans l’un des arbres de la foret décrivant l’exploration par DFS.
* Arc croise: (x, y) est un arc crois ́e si et seulement si y n’est un descendant de x et x n’est un descendant de y dans la foret décrivant l’exploration par DFS.

## 3.2. Détermination des composantes connexes

### 3.2.1. Cas non oriente

Pour déterminer les composantes connexes d’un graphe G= (X, U) non oriente, il suffit d’appliquer l’algorithme de parcours du graphe en profondeur d’abord en modifiant la procédure DFS en initialisant les compteurs c(x) par les degrés d(x) et en explorant, non pas les successeurs des sommets, mais les voisins. Chaque arbre de la foret obtenue couvre des sommets appartenant à une même composante connexe et le nombre d’arbres de la foret est donc le nombre de connexité de G. Ce nouvel algorithme, appelé DFSno, est présente ci-dessous

Algorithme 4 : Parcours en profondeur d’abord pour la détermination des composants connexe



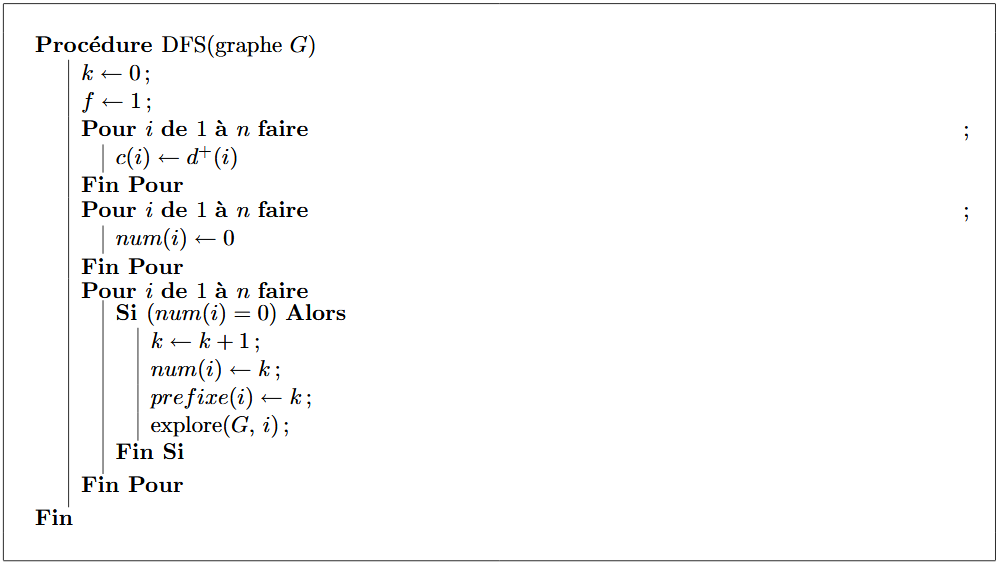
Algorithme 5 : Exploration en profondeur d’abord pour la détermination des composantes connexes

### 3.2.2. Cas orienté

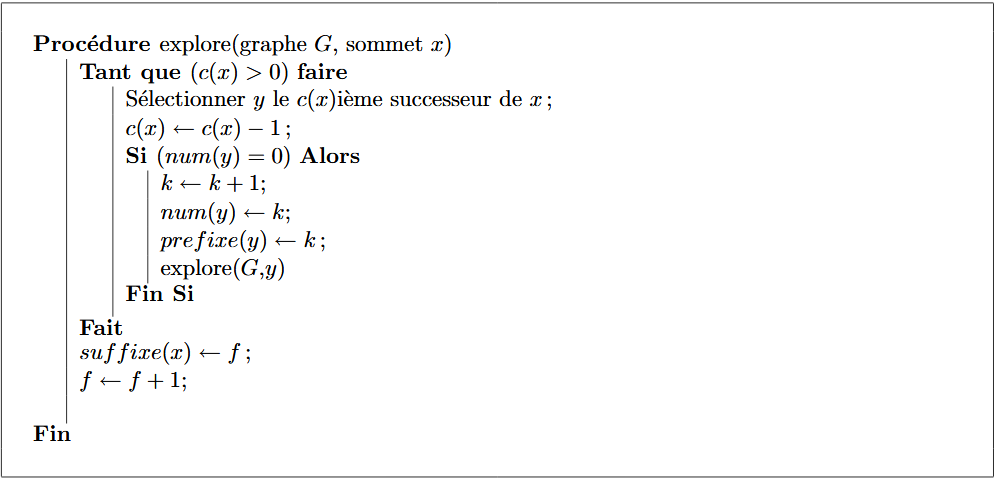
Pour déterminer les composantes connexes d’un graphe G = (X, U) oriente, il suffit de transformer G en un graphe G′ non orient ́e et appliquer DFSno(G′). Chaque arbre de la foret obtenue couvre des sommets appartenant a une même composante connexe et le nombre d’arbres de la foret est donc le nombre de connexité de G.

## 3.3. Détermination des composantes fortement connexes

L’exploration d’un graphe par DFS permet d’obtenir une numérotation préfixe et une numérotation suffixe des sommets comme suit :



Algorithme 6 : Parcours en profondeur d’abord



Algorithme 7 : Exploration en profondeur d’abord

L’algorithme de Kosaraju-Sharir utilise la numérotation suffixe pour déterminer les composantes fortement connexes d’un graphe G= (X, U) :

1. Appliquer DFS(G) en numéro tant les sommets dans l’ordre suffixe.

2. Construire le graphe G′= (X, U′) obtenu en inversant le sens des arcs de G.

3. Appliquer DFS(G′) en démarrant par le sommet de plus grand numéro suffixe et itérer le processus a partir du sommet non marque de plus grand numéro suffixe.

Théorème 1 Chaque arbre de la foret construite avec DFS(G′) couvre les sommets constituant une composante fortement connexe de G.

Preuve:

1. Il faut commencer par montrer que si x et y, deux sommets de G, appartiennent a une même composante fortement connexe, alors ils appartiennent au même arbre de la foret d’exploration en profondeur de G′. Si x et y appartiennent a la même composante fortement connexe, il existe un chemin allant de x a y et un autre allant de y a x. Ainsi, si on effectue une recherche en profondeur d’abord dans G′ a partir d’un sommet z et que l’on atteint x alors on atteindra nécessairement y car x et y sont mutuellement accessibles et inversement.

2. Il faut également montrer que si x et y appartiennent au même arbre de la foret d’exploration en profondeur d’abord de G′, alors x et y appartiennent a la même composante fortement connexe de G. Soit z la racine de l’arbre contenant x et y. Il existe nécessairement un chemin allant de z a x dans G′ et donc de x a z dans G. Par ailleurs, on a par construction : suffixe(z) > suffixe(x) ce qui signifie que dans DFS(G), l’appel récursif en x s’est termine avant celui en z. Deux cas doivent alors être considérés : soit préfixe(z) > préfixe(x), soit préfixe(x) > préfixe(z). Si préfixe(z) > préfixe(x), cela signifie que l’on explore x avant z dans G mais, comme il existe un chemin de x a z, on doit atteindre z par x ce qui contredit suffixe(z) > suffixe(x). Ainsi préfixe(x) > préfixe(z) et suffixe(z) > suffixe(x) impliquent que x est un descendant de z dans un arbre de la foret d’exploration en profondeur d’abord de G. Il existe donc un chemin allant de z a x dans G ce qui implique que z et x appartiennent a la même composante fortement connexe. Un raisonnement analogue permet de montrer que z et y appartiennent aussi a la même composante fortement connexe. Donc x et y appartiennent à la même composante fortement connexe.

On peut ainsi construire le graphe réduit associe a G.

## 3.4. Tri topologique dans un graphe oriente sans circuit

On rencontre fréquemment les graphes sans circuits dans les applications. Par exemple, dans un problème d’ordonnancement, le graphe potentiels-taches G= (X, U), défini de la façon suivante : les sommets représentent les taches a exécuter et (x, y) est un arc si et seulement si l’exécution de la tache x doit précéder celle de la tache y et la longueur de l’arc (x, y) représente la durée de la tache x; doit par construction être un graphe sans circuit.

Dans un graphe sans circuit, on peut définir les notions d’ordre topologique et de fonction rang.

**Définition 1** Etant donne un graphe G= (X, U) d’ordre n, on appelle ordre topologique une numérotation v des sommets de 1 a n telle que les numéros d’ordre v(y) associes aux sommets y vérifient :

∀(x, y) ∈ U : v(x) < v(y).

En d’autres termes, si l’on parcourt la liste des sommets dans l’ordre défini par une telle numérotation, un sommet y ne peut être rencontre que si l’on a, au préalable, rencontre tous ses prédécesseurs.

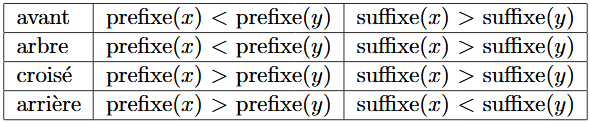
**Remarque 1** Il n’y a pas unicité : plusieurs ordres topologiques peuvent être définis sur un même graphe.

**Propriété 1** Un graphe G est sans circuit si et seulement si il n’existe pas d’arc arrière dans le parcours en profondeur d’abord de G.

**Preuve:** Montrons que s’il existe un circuit, alors il existe un arc arrière dans le parcours en profondeur d’abord de G. Soit le circuit μ= (x, x0,..., xk, x). Si x est le premier sommet du circuit explore par DFS, x sera la racine d’un arbre comprenant au moins tous les autres sommets du circuit. Si un arc du circuit n’est pas un arc d’arbre, alors il s’agit d’un arc arrière et il existe au moins un arc du circuit qui ne sera pas un arc d’arbre. La réciproque est triviale en utilisant un raisonnement similaire, en effet, s’il existe un arc arrière alors le graphe contient au moins un circuit.

**Propriété 2** L’ordre suffixe inverse est un ordre topologique.

**Preuve:** Le tableau ci-dessous présente, pour un type d’arc (x, y), les relations entre les numérotations suffixes et préfixes des sommets x et y.



Comme il n’existe par d’arc arrière lorsque le graphe est sans circuit, l’ordre suffixe inverse est bien un ordre topologique.

**Définition 2** Si l’on note R l’ensemble des sommets d’un graphe sans circuit G de demi-degré intérieur nul, on appelle fonction rang associée a G l’application rang qui a tout sommet y associe un nombre entier défini de la fa ̧con suivante :

* **rang(y) = 0 ∀y ∈ R**
* rang(y) = nombre d’arcs dans un chemin de cardinal maximal joignant un sommet quelconque de l’ensemble R et le sommet y, ∀y ¢ R.

La fonction rang est unique et elle définit une partition de l’ensemble des sommets en niveaux, chaque niveau k étant forme par le sous-ensemble de sommets :

**Xk = {x ∈ X : rang(x) =k}**

## 3.5. Fermeture transitive d’un graphe

Considérons un 1-graphe oriente connexe G= (X, U). La fermeture transitive d’un graphe G est un graphe F= (X, UF) défini sur le même ensemble de sommets X et dont l’ensemble des arcs est défini par :

**UF = {(x, y) ∈ X2 : ∃μ[x, y]}**

Le problème de la fermeture transitive d’un graphe a été beaucoup étudié. Il existe principalement deux types d’approches, la première se basant sur l’algorithme de Kosaraju-Sharir notamment et la seconde utilise les multiplications matricielles de la matrice d’adjacence du graphe.

### 3.5.1. Algorithme de parcours de graphes

Pour trouver la fermeture transitive d’un graphe G, on peut décomposer le problème en trois étapes :

1. Déterminer les composantes fortement connexes de G.

2. Déterminer la fermeture transitive du graphe réduit Gr qui est sans circuit.

3. Déduire la fermeture transitive de G de la fermeture transitive de Gr.

**Remarque 2** Si G = (X, U) est fortement connexe, sa fermeture transitive est un 1-graphe plein sans boucle, c’est-`a-dire ∀(**x, y) ∈ X2, (x, y) ∈ UF et (y, x) ∈ UF.**

* **Etape 1:** Recherche des composantes fortement connexes de G par l’algorithme de Kosaraju-Sharir.
* **Etape 2:** Recherche de la fermeture transitive du graphe Gr = (Xr, Ur) (sans circuit) d’ordre k. Il faut en premier lieu définir un ordre topologique sur les sommets de Gr. Les sommets sont alors numérotés de 1 a k selon cet ordre topologique. Pour tout sommet x de Xr, Lx désigne la liste des sommets pouvant être atteints par un chemin partant de x (listes des descendants). Pour chaque sommet x, la liste Lx est initialisée par l’ensemble de ses successeurs. Puis, pour chaque sommet y allant de k a 1 (ordre topologique inverse), on établit sa liste des descendants de la fa ̧con suivante :

**Ly ← Ly ⋃z∈Γ+(y) Lz**

Comme les sommets sont examines dans l’ordre topologique inverse, lorsque le sommet y est examine, les listes définitives Lz de tous ses successeurs directs z ont déjà été constituées. La fermeture transitive Fr = (Xr, UFr) de Gr est constituée des arcs suivants : ∀x ∈ Xr, ∀y ∈ Lx, (x, y) ∈ UFr.

**Etape 3:** Déduire la fermeture transitive de G de la fermeture transitive de Gr.

1. Pour chaque composante fortement connexe Xi de cardinal ni, établir la liste des ni(ni−1) arcs du sous-graphe sans boucle définis sur Xi.

2. Si k et l sont deux sommets de Gr (correspondant aux composantes fortement connexes Xk et Xl de G) tels que l’arc (k, l) appartienne a la fermeture transitive de Gr, alors établir la liste des nk × nl arcs de la forme (i, j) avec i ∈ Xk et j ∈ Xl.

**3.5.2. Puissances de la matrice D’adjacence**

Calcul de chemins Une matrice d’adjacence A est une matrice carrée d’ordre n telle que :

**aij = 1 si (xi, xj) ∈ U**

**0 sinon.**

On définit également **A2= (a2ij)** par : **a2ij = ∑nl = 1 ail.alj**Ainsi, **ail.alj = 1** ⇔ il existe un chemin de xi a xj de longueur 2 transitant par xl. De ce fait, le terme a2ij représente le nombre de chemins de longueur 2 avec xi Pour extrémité initiale et xj pour extrémité terminale.

De la même manière, on défini **Ak = (akij)** par :

**akij****= ∑nl=1ak−1.alj**

**ak−1 il.alj** représente le nombre de chemins de xi a xj de longueur k transitant par xl si

**alj = 1.** Ainsi, **akij** représente le nombre de chemins de longueur k avec xi pour extrémité initiale et xj pour extrémité terminale.

On notera que les boucles sont prises en compte. On peut donc repérer sur la diagonale les nombres de circuits de longueur k.

Existence de chemins **A[k] = (a[k]ij)** est une matrice booléenne dont chaque terme est défini par :

**a[k]ij = ∨nl=1 (a[k−1]il ∧ alj)**

Ainsi, **a[k]ij** représente l’existence de chemins de xi à xj de longueur k. Soit :̂

A = A ∨ A [2] ∨ A[3] ∨...∨ A[n−1]̂

A représente **la fermeture transitive de Γ**

∀x ∈ X, Γ(x) = Γ(x) ∪ Γ2(x) ∪...∪ Γn−1(x)

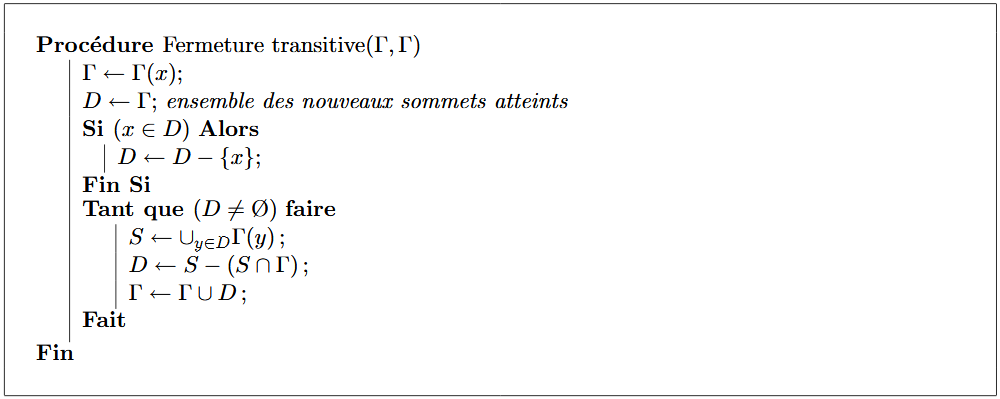
Dans ce paragraphe, nous définissons la fermeture transitive d’une autre manière qu’au paragraphe 4.5.

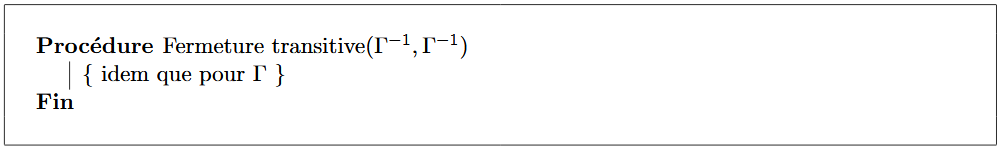
∀xi ∈ X, a Γ (xi), on associe la ligne i de la matricê A telle que :

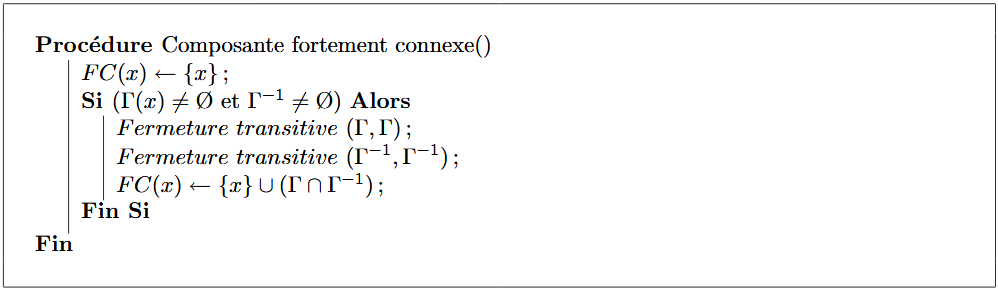
**aij= 1 si ∃ μ[xi, xj]**

**0 sinon.**

On notera que la construction de la composante fortement connexe associée a un sommet x donne peut être réalisée par le calcul de **Γ(x) ∩ Γ−1(x)** (cet ensemble correspond aux sommets qui se situent sur un circuit contenant x).







On associe â A un graphe τ(G) = (X, τ(U)) de la fermeture transitive. Deux graphes G = (X, U) et G′ = (X, U′) sont dit τ- ́équivalent s’ils possèdent la même fermeture transitive, c’est-`a-dire τ(G) = τ(G′) (⇔ τ(U) = τ(U′)).

Un graphe G= (X, U) est dit τ-minimal

si ∀u ∈ U, G′ = (X, U− {u}) n’est pas τ- ́équivalent a G.

i.e. si ∀u ∈ U, τ(U − u) ⊂ τ(U) (l’inclusion est stricte)

I.e. τ(G′) est un graphe partiel de τ(G).

**3.5.3. Algorithmes par multiplications matricielles**

Pour trouver la fermeture transitive d’un graphe G d’ordre n, il suffit d’élever la matrice d’adjacence de G` à la puissance n−1.

**Algorithme trivial en** O(n4) L’algorithme trivial correspond au

produit de matrices en O(n3) avec n−2 produits

addition de matrices en O(n2) avec n−2 additions

**Algorithme de Warshall en O(n3)**

A → A(1) → A(2) →...→ A(k) →...→ A(n)

Soit k ∈{1,2,...,n} donné, on pose a(k)ij = 1 s’il existe au moins un chemin de xi a xj transitant uniquement par des sommets de numéros appartenant a{1,2,...,k}:

**Ou bien**

il existe un chemin de xi a xj transitant seulement par des sommets de numéros appartenant a {1, 2 ,..., k−1} i.e. **a(k−1)ij.**

**Ou bien**

il existe un chemin de xi a xk transitant seulement par des sommets de numéros appartenant `a{1, 2, ..., k−1} **i.e. a(k−1)ik.**

**et**, il existe un chemin de xk a xj transitant seulement par des sommets de numéros appartenant a {1, 2,..., k−1**} i.e. a(k−1)kj.**

⇒ **a(k)ij = a(k−1)ij ∨ (a(k−1)ik ∧ a(k−1)kj).** Avec

initialement, **a(1)ij = aij ∨ (ai1 ∧ a1j)**



La complexité temporelle est en O(n3).On notera qu’a chaque itération k, on fait un produit de la colonne k de A ”transformée” par la ligne k de A ”transformée” ; c’est une matrice anti-scalaire qui se calcule en O(n2).

**i.e.**

**A(1) = A ∨ (A1 ∧ A1)**

**A(2) = A(1) ∨ (A(1)2 ∧ A(1)2)**

**. . .**

**A(k) = A(k−1) ∨ (A(k−1)k ∧ A(k−1)k)**

# Partie 4. Problèmes de meilleurs chemins

Les problèmes de cheminement dans les graphes (en particulier la recherche d’un plus court chemin) comptent parmi les plus classiques de la théorie des graphes et les plus importants dans leurs applications. Le problème du plus court chemin (pcch) peut être pose de la façon suivante : étant donne un graphe G = (X, U), on associe `a chaque arc u = (i, j) un nombre réel, note l(u) ou lij, appelé longueur de l’arc. Le problème du pcch entre deux sommets i0 et j0 du graphe consiste a déterminer, parmi tous les chemins allant de i0 a j0 celui, note μ∗ dont la longueur totale :

**l( μ∗ ) = ∑u∈μ∗ l(u)**

Soit minimale.

Condition nécessaire Le problème du pcch a une solution si et seulement si il n’existe pas dans le graphe de circuit de longueur strictement négative pouvant être atteint a partir de i0.

Si cette condition nécessaire est vérifiée, il existe toujours un chemin de longueur minimale qui soit élémentaire. En effet, lorsque tous les circuits du graphe pouvant être atteint a partir de i0 ont une longueur strictement positive, tout pcch est nécessairement élémentaire.

Lorsque l’on cherche un pcch entre deux sommets, on doit déterminer d’autres ppch entre i0 et d’autres sommets du graphe. Aussi, les algorithmes existant se divisent en deux catégories : ceux dans lesquels on recherche un pcch d’un sommet spécifié i0 a tous les autres sommets du graphe ; ceux qui procèdent directement de la recherche de tous les pcch entre i et j pour tous les couples (i, j) du graphe.

Il existe un grand nombre d’algorithmes permettant de déterminer un pcch d’un sommet particulier i0`a tous les autres sommets du graphe. Les plus efficaces d’entre eux réalisent un marquage des sommets c’est-`a-dire qu’a chaque sommet i est associe une marque λ(i) représentant `a la fin de l’algorithme la longueur du pcch allant de i0 a i.

## 4.1. Plus courts chemins d’origine fixée dans un graphe avec longueurs non négatives : algorithme de Dijkstra

Soit G = (X, U) un graphe dont les arcs sont munis de longueurs réelles positives ou nulles. On cherche les pcch de i0 a tous les autres sommets du graphe.

L’algorithme de Moore-Dijkstra procède en n−1 itérations. A l’initialisation, λ(i0) ← 0 et λ(i) ← ∞ pour tout i6 = i0. A une itération quelconque de l’algorithme, l’ensemble des sommets est partage en deux sous-ensembles S et X\S. Le sous-ensemble S contient l’ensemble des sommets définitivement marques c’est-`a-dire les sommets i pour lesquels la marque λ(i) représente effectivement la longueur d’un pcch allant de i0 a i (a l’initialisation S ← {i0}). X\S contient les sommets i ayant une marque provisoire vérifiant:

**λ(i) = mink∈S∩Γ−(i){λ(k) + lki}**

L’algorithme est base sur le lemme suivant.

Lemme: Si i est un sommet de X\S de marque provisoire λ(i) minimale :

**λ(i) = minj∈X\Sλ(j)**

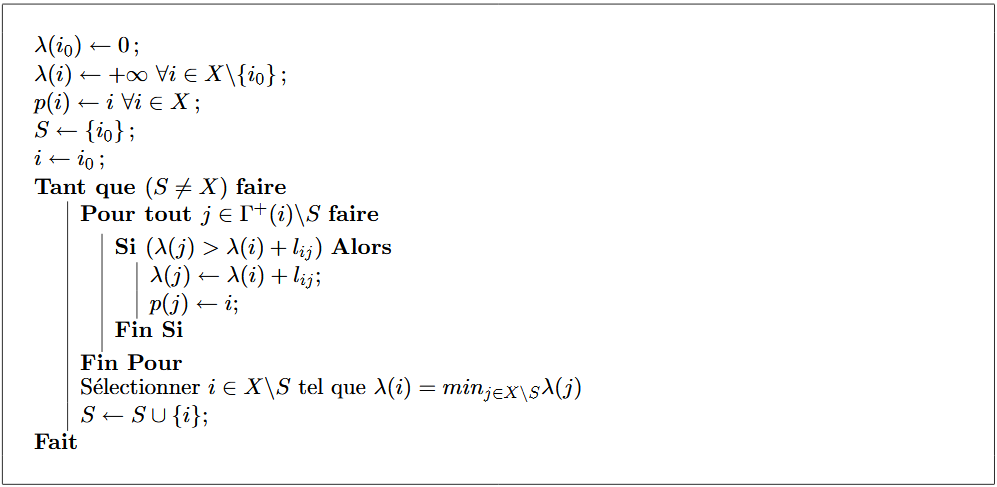
Alors λ(i) est la longueur d’un pcch chemin allant de i0 à i.

Ainsi a chaque itération, on sélectionne le sommet i de plus petite marque provisoire, on l’inclut dans l’ensemble S des sommets définitivement marques et on remet a jour les marques de ces successeurs non définitivement marques de la façon suivante :

**∀j ∈ Γ+(i)\S, λ(j) = min {λ(j); λ(i) + lij}**

Lorsque tous les sommets du graphe sont dans l’ensemble S, les marques représentent les longueurs des pcch allant de i0 a tous les autres sommets du graphe.

Comme a chaque itération, on attribue une marque définitive a un nouveau sommet, on détermine n−1 pcch entre i0 et les autres sommets du graphe en au plus n−1 itérations.



Complexité : A chaque itération, on sélectionne le sommet j de plus petite marque en O(n) opérations dans le pire cas, et on remet a jour les marques des successeurs de j en O(d+(j)) opérations. En tout, il y a n itérations pour marquer tous les sommets du graphe. La complexité totale est donc en O(n2) + O(m) ≈ O(n2).

Remarquons que lorsque le graphe est peu dense, le terme O(n2) l’emporte sur O(m) et l’opération la plus couteuse consiste a recherche le sommet de plus petite marque. Pour diminuer la complexité de cette recherche, on peut utiliser une structure de données appelée tas.

## 4.2. Mise en œuvre de l’algorithme de Dijkstra pour les graphes peu denses : algorithme de Johnson

### 4.2.1. Présentation de la structure de données de tas (heap)

Un maximier est un arbre valu ́e partiellement ordonne, c’est-`a-dire que la valuation d’un sommet est inferieure ou égale a celle de ses fils, aussi ́équilibré que possible (les seules feuilles manquantes doivent se trouver `a droite des feuilles situées au plus bas niveau de l’arbre). Le sommet de plus petite valuation est le sommet racine. On peut donc le trouver en O(1) opération. Si on le supprime, on détruit la structure d’arbre. Pour recomposer un maximier, il suffit de prendre la feuille la plus a droite du niveau le plus bas et on la place temporairement `a la racine. Puis, il faut pousser cet élément aussi bas que possible en l’ ́échangeant avec celui de ses fils ayant la plus petite valuation inferieure. On s’arrêté lorsque l’élément est, soit, devenu une feuille, soit, ases fils de plus grande valuation. La suppression de la racine a une complexité en 0(log2n) car aucun chemin de l’arbre ne contient plus de log2n arêtes.

Pour insérer un nouvel élément tout en preservant la structure du maximier, on réalise la procédure ”inverse”. On commence par placer l’élément en question aussi loin que possible `à gauche au plus bas niveau de l’arbre. Puis, on le pousse aussi haut que possible dans l’arbre de la façon suivante : si son père a une valuation supérieure a la sienne, il faut les ́échanger, et on réitère les comparaisons jusqu’`a ce que l’ ́élément inséré se trouve `a la racine, ou bien, ait une priorité inferieure a celle de son père. L’insertion d’un nouvel élément a ́également une complexité en O(log2n).

Le tas est une structure de données pouvant être utilisée pour représenter en machine un maximier. Soit un maximier contenant n sommets. Dans le tas, on utilise les n premières cellules d’un tableau un i dimensionnel T de la fa ̧con suivante : les sommets du maximier remplissent les cellules T[1], T[2],...,T[n] niveau par niveau `a partir du haut, et `a l’intérieur d’un même niveau de la gauche vers la droite. Ainsi, le fils gauche, s’il existe, du sommet T[i] se trouve en T[2i] et le fils droit, s’il existe, en T[2i+ 1] ; le père de T[i] se trouve en T[i/2].

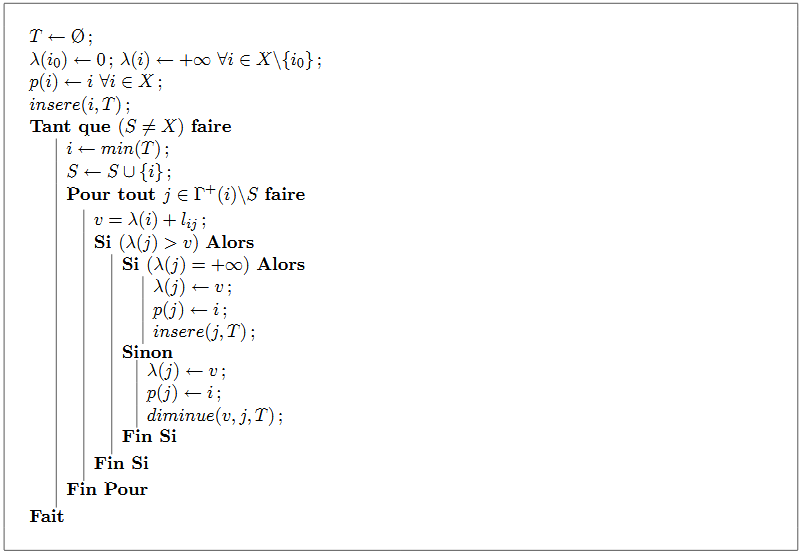
### 4.2.2. Algorithme de Johnson

Les sommets i de G et leur marque λ(i) sont stockées dans un maximier ce qui permet de trouver le sommet de marque initiale en O(1) opération. Dans l’algorithme de Johnson, on fait appel aux fonctions suivantes :

– insère (i, T) : insère le sommet i dans le tas T

– min(T) : retourne et supprime le sommet racine (de plus petite valuation) de T et recompose la structure

– diminue (valeur, i, T) : modifie la valuation du sommet i dans T (elle passe `a valeur) et recompose la structure Le tas est constitue des sommets ayant des marques temporaires finies.



A chaque itération, on sélectionne le sommet i de plus petite marque et on recompose le tas : la complexité est en O(1) + O(log2n). Puis, on remet a jour les marques de tous les successeurs du sommet i sélectionné en O(d+(i) × log2n). Comme il y a n itérations, on a une complexité totale de O(nlog2n) +O(mlog2n) ≈ O(mlog2n).

## 4.3. Plus court chemin d’origine fixée dans un graphe sans circuit avec longueurs quelconques : algorithme de Bellman

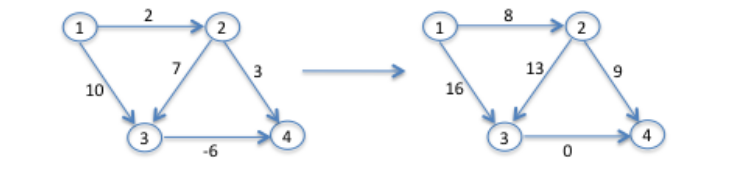
Soit G= (X, U) un graphe sans circuit dont les arcs sont munis de longueurs réelles quelconques. On cherche les pcch allant de i0 a tous les autres sommets du graphe. Condition d’optimalité Un ensemble de valeurs λ∗(i) pour i= 1,..., n, avec λ∗(i0) = 0, représente les longueurs des pcch allant de i0 aux autres sommets du graphe si et seulement si : ∀(i, j) ∈ U : λ∗(j) ≤ λ∗(i) + lij Lorsque le graphe ne présente pas de circuit, il faut déterminer une numérotation des sommets allant de 1 a n qui constitue un ordre topologique (le sommet de départ i0 ayant le numéro 1) et marquer les sommets dans cet ordre.



## 4.4. Détermination de plus courts chemins d’origine fixée dans un graphe avec longueurs quelconques

Soit G = (X, U) un graphe dont les arcs sont munis de longueurs réelles quelconques. On cherche les pcch de i0 à tous les autres sommets du graphe.

Remarque préliminaire Soit un graphe G présentant sur certains arcs une valuation négative et, envisageons de rendre les valuation toutes positives ou nulles en ajoutant a chaque valeur la valeur absolue maximale des valeurs négatives.

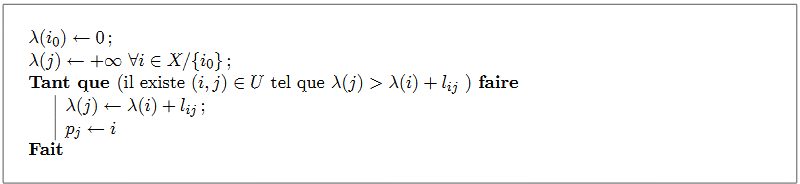


Ceci change la nature du problème puisque, comme on le voit dans l’exemple ci-dessus, le plus court chemin allant de 1 a 4 dans le graphe initial (de valeur 3) n’est plus optimal dans le graphe transforme.

**4.4.1. Algorithme de Ford (1956)**

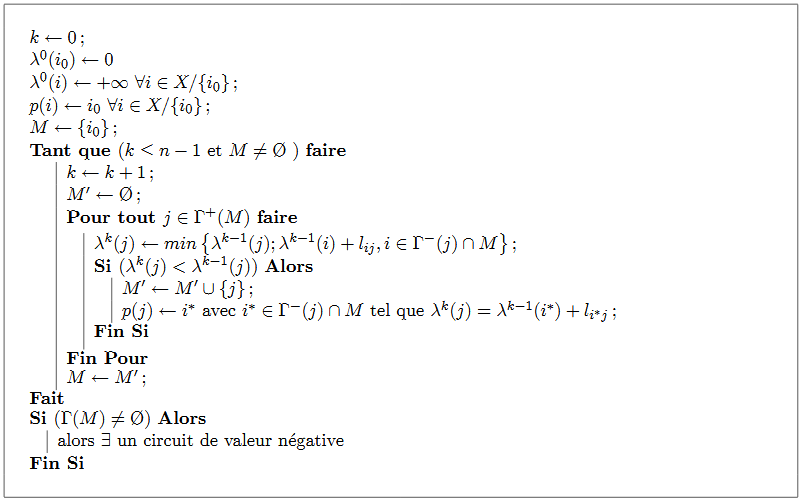
Dans l’algorithme de Ford, les marques λ(i) des sommets sont modifiées itérativement de fa ̧con`a converger vers la condition d’optimalité présentée en section 5.3.

L’algorithme consiste alors, a partir d’un ensemble de valeurs provisoires, a parcourir séquentiellement la liste des arcs du graphe de fa ̧con a vérifier que la condition d’optimalité est satisfaite. Si cette condition est satisfaite pour tous les arcs (i, j), le problème est résolu ; sinon, il existe un arc(i, j) tel que : λ(j) > λ(i) + lij et , dans ce cas, on remet a jour la marque de j de la fa ̧con suivante : λ(j) ← λ(i) + lij (ce qui signifie que l’on amélioré la marque provisoire de j en empruntant le chemin de longueur λ(i) entre i0 et i suivi de l’arc (i, j)).



### 4.4.2. Algorithme de Ford-Bellman

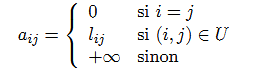
Cet algorithme est une variante ”optimisée” de l’algorithme de Ford. Pour parcourir la liste des arcs du graphe, on regarde pour chaque sommet i l’ensemble de ses prédécesseurs. Et, à une itération k donnée, on ne va pas s’intéresser a tous les sommets du graphe mais seulement a ceux dont la marque a été modifiée au cours de l’itération précédente. L’algorithme calcule donc a chaque itération k un ensemble de marques, notées λkj pour tout j ∈ X. On note M = {j ∈ X |λkj < λk−1j}, l’ensemble des sommets dont les marques ont et ́e modifiées a l’itération k, et seuls les sommets appartenant a Γ+(M) peuvent voir leurs marques modifiées au cours de l’itération k+1. En fait les marques λkj ont une interprétation très précise : c’est la valeur du meilleur chemin de i0 a j ne contenant pas plus de k arcs. Ainsi, en l’absence de circuit absorbant dans le graphe, l’algorithme termine nécessairement à l’issue de l’itération n car tout chemin élémentaire a une longueur maximale égale à n−1. Si une ou plusieurs marques sont modifiées a l’itération, cela signifie que le graphe présente un circuit de valeur négative (car il existe un pcch de longueur > n−1 qui emprunte donc nécessairement un circuit de valeur < 0 !). L’algorithme de Ford-Bellman peut donc s’écrire comme suit :



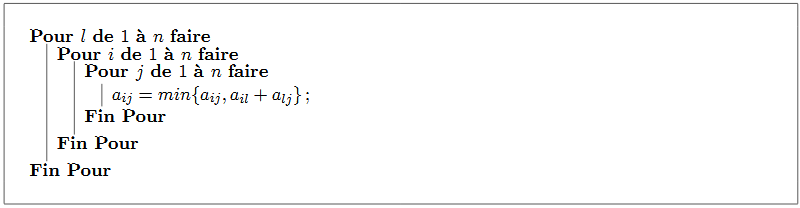
Complexité A chaque itération, on remet a jour, dans le pire cas, les marques de tous les sommets en regardant l’ensemble des arêtes : O(m) opérations. Le nombre maximal d’itérations étant n, la complexité totale est en O(nm).

**4.5. Plus courts chemins entre toutes les paires de sommets : algorithme de Floyd**

Soit la matrice A = {aij} de taille n × n avec :



L’algorithme de Floyd permet de calculer les pcch entre tous les couples de sommets de la façon suivante. A la première itération, on cherche le pcch entre chaque couple (i, j) passant éventuellement par le sommet 1 ; a l’itération l (avec l >1), on cherche le pcch entre chaque couple (i, j) passant par des sommets d’indice inferieur ou égal a l. Une description formelle de l’algorithme est donnée ci-dessous.



La complexité totale est en O(n3).

# Partie 5. Arbres couvrants

En [théorie des graphes](https://fr.wikipedia.org/wiki/Th%C3%A9orie_des_graphes), étant donné un [graphe non orienté](https://fr.wikipedia.org/wiki/Graphe_non_orient%C3%A9) [connexe](https://fr.wikipedia.org/wiki/Connexit%C3%A9_(math%C3%A9matiques)) dont les arêtes sont pondérées, un arbre couvrant de poids minimal (ACM) de ce graphe est un [arbre couvrant](https://fr.wikipedia.org/wiki/Arbre_couvrant) ([sous-ensemble](https://fr.wikipedia.org/wiki/Sous-ensemble) qui est un [arbre](https://fr.wikipedia.org/wiki/Arbre_(th%C3%A9orie_des_graphes)) et qui connecte tous les sommets ensemble) dont la somme des poids des arêtes est minimale (c'est-à-dire de poids inférieur ou égal à celui de tous les autres arbres couvrants du graphe). L'arbre couvrant de poids minimal est aussi connu sous certains autres noms, tel qu’arbre couvrant minimum ou encore arbre sous-tendant minimum.

L'arbre couvrant minimum peut s'interpréter de différentes manières selon ce que représente le graphe. De manière générale si on considère un réseau où un ensemble d'objets doivent être reliés entre eux (par exemple un [**réseau électrique**](https://fr.wikipedia.org/wiki/R%C3%A9seau_%C3%A9lectrique) et des habitations), l'arbre couvrant minimum est la façon de construire un tel réseau en minimisant un coût représenté par le poids des arêtes (par exemple la longueur totale de câble utilisée pour construire un réseau électrique).

## 5.1. Vocabulaire lié aux arbres couvrants

* **Un arbre** : est graphe connexe, non orienté et acyclique. Remarque : dans un arbre, le **nombre de sommets = nombre d’arêtes + 1 ;**
* **Un graphe partiel G’ (V, E’) d’un graphe G (V, E)** : est un graphe qui a les mêmes sommets que **G** ou mieux un graphe dont l’ensemble des arêtes E**’** est inclus dans **E ;**
* **Un arbre couvrant T d’un graphe**: est un graphe partiel, sans cycle ;
* **Un graphe pondéré**: est un graphe où un entier positif est affecté à chaque arête, cet entier est appelé **poids ;**
* **Le poids d’un graphe**: est la somme des poids des arêtes du graphe

## 5.2. Propriétés des arbres couvrants

* **Multiplicité ou unicité:**

Comme le montre la figure ci-contre, un graphe connexe peut comporter plusieurs arbres couvrants minimum différents, mais si tous les poids sont différents, alors il est unique. Un graphe non orienté et général possède une forêt couvrante de poids minimal ;

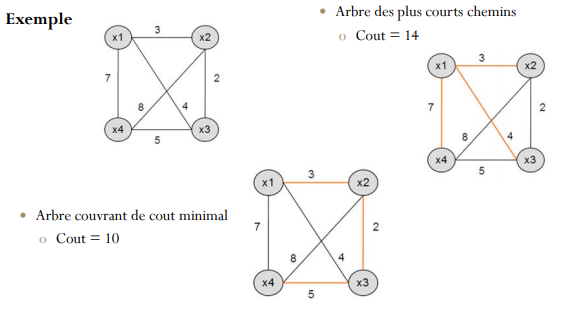
* **Propriétés des cycles et des coupes :**

Pour tout [cycle](https://fr.wikipedia.org/wiki/Cycle_(th%C3%A9orie_des_graphes)) dans le graphe, si une arête est de poids strictement plus grand que les autres, alors cette arête n'est pas dans l'arbre couvrant de poids minimal. Autrement dit, toute arête *(u, v)* du graphe est de poids supérieur ou égal à l'arête de poids maximum sur le chemin reliant *u* à *v* dans l'arbre. Pour toute [coupe](https://fr.wikipedia.org/wiki/Coupe_(th%C3%A9orie_des_graphes)) du graphe, si une arête est de poids strictement inférieur aux autres, alors elle appartient à l'arbre couvrant de poids minimal ;

* **Poids de l'arbre d'un ensemble de points :**

Un problème classique est de savoir, étant donné un ensemble de points dans **Rd r** avec la norme euclidienne, quel est le poids de l'arbre couvrant minimal. Il est de l'ordre de **{\displaystyle n^{\frac {d-1}{d}}}n** moyenne et avec probabilité **1.**

Un arbre couvrant de cout minimal et arbre des plus courts chemins. En général l’arbre couvrant de cout minimal est différent de l’arbre des plus courts chemins :



## 5.3. Arbre couvrant de poids minimal

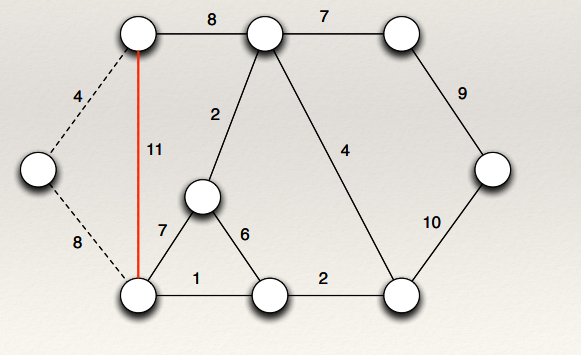
Soit G= (V, E, ω) un graphe non-orienté pondéré, on appelle arbre couvrant de poids minimal ou maximal un arbre dont la somme des poids des arêtes le constituant est minimale ou (maximale).

***Problématique***: on doit connecter les sommets de la façon la plus économique possible. Tout le problème est de trouver l’ACPM.

Pour ce faire on va construire un algorithme basé sur la bi-coloration rouge et bleu des arêtes de G. Au départ les arêtes ne sont pas colorées pour les colorer nous allons utiliser deux règles :

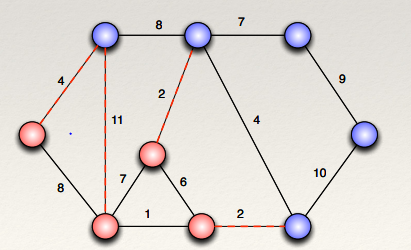
***La règle rouge de TARJAN***:

* Choisir un cycle µ disponible c'est-à-dire qui ne contient pas d’arête rouge ;
* Dans µ choisir une arête ***e*** non- coloriée de poids maximal, colorie l’arête en rouge.

***Remarque :***l’arête de poids maximal d’un cycle ne fait partie d’aucun arbre couvrant

Exemple :

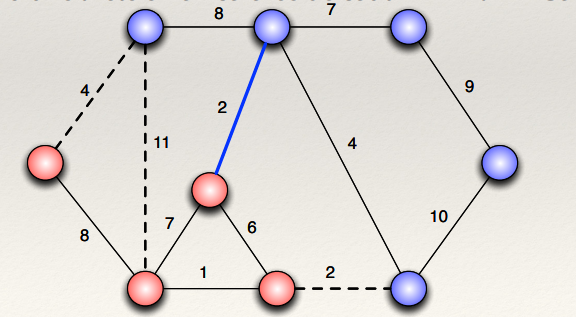
**Une coupe ou cocycle** : est un ensemble d’arêtes dont l’une des extrémités est dans un sous-ensemble A et l’autre extrémité étant dans le complément de A.



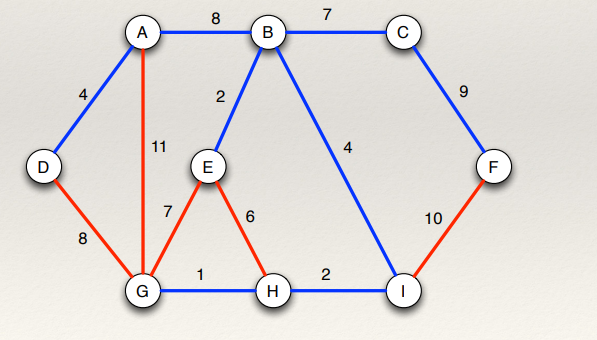
Exemple :

***Règle bleu de Tarjan :***

* Une coupe Ɵ qui ne contient pas d’arête bleu est dite disponible ;
* Choisir une couple Ɵ disponible ;
* Choisir dans Ɵ une arête e non coloriée de poids minimal et colorie en bleu.

Exemple :

***Remarque :*** l’arête de poids minimal d’une coupe fait partie d’un arbre couvrant du graphe. Sur le schéma ci-contre : en bleu on a l’arbre couvrant de poids minimal T et celui en rouge est le co-arbre de T :



## 5.4. Aspects algorithmiques des arbres couvrants

Il existe de nombreux algorithmes de construction d'un arbre couvrant de poids minimal. Par exemple l'[algorithme de Borůvka](https://fr.wikipedia.org/wiki/Algorithme_de_Bor%C5%AFvka) (le premier algorithme inventé pour ce problème), l'[algorithme de Prim](https://fr.wikipedia.org/wiki/Algorithme_de_Prim) et l'[algorithme de Kruskal](https://fr.wikipedia.org/wiki/Algorithme_de_Kruskal). Ces algorithmes sont tous des [algorithmes glouton](https://fr.wikipedia.org/wiki/Algorithme_glouton).

**l'**[**Algorithme de Borůvka**](https://fr.wikipedia.org/wiki/Algorithme_de_Bor%C5%AFvka)

***Principe***

Le principe est de réduire ***G*** en [contractant des arêtes](https://fr.wikipedia.org/wiki/Contraction_d%27ar%C3%AAte) : on choisit peu à peu les arêtes qui seront dans l'arbre, et à chaque fois que l'on en choisit une, on fusionne les nœuds que cette arête relie. Ainsi, il ne reste plus qu'un sommet à la fin. On peut aussi décrire l'algorithme sans parler de contraction : on construit une forêt dont les arbres fusionnent peu à peu pour former un arbre couvrant de poids minimum.

***Pseudo code***

On prend ***G*** le graphe et ***F*** l'ensemble des arêtes choisies (on le remplit peu à peu).

*1 F ← vide*

*2 Tant que G n'est pas réduit à un sommet faire*

*3 Détruire les boucles de G (\*)*

*4 Remplacer les arêtes multiples entre deux sommets par une seule dont le poids est le minimum*

*5 Pour tout x, sommet de G, faire*

*6 Trouver l'arête e\_x de poids minimum adjacente à x*

*7 F ← F union e\_x*

*8 Contracter e\_x (\*\*)*

*9 fin pour*

*10 fin tant que*

*11 renvoyer F*

***Complexité :***

L'algorithme de Borůvka admet une [complexité](https://fr.wikipedia.org/wiki/Complexit%C3%A9_en_temps) en **o{\displaystyle O(A\ln(S))}(In(S))** {\displaystyle O(A\ln(S))}, où **{\displaystyle A}A** est le nombre d'arêtes et **{\displaystyle S}S** le nombre de sommets du graphe considéré

**Algorithme de Prim**

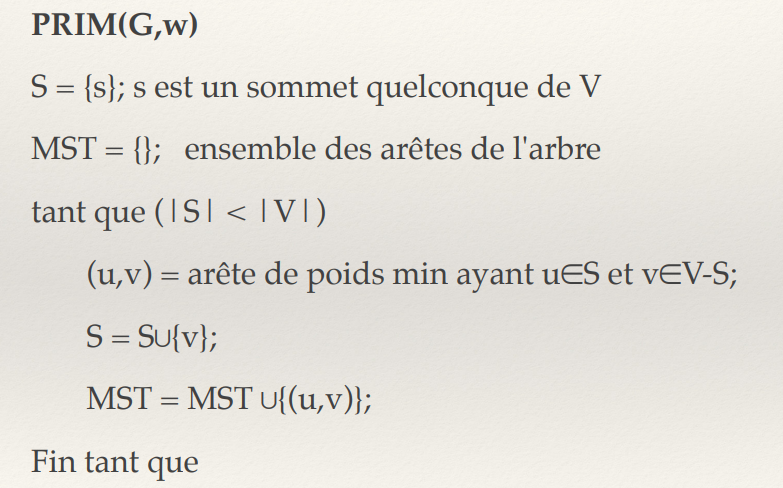
L'algorithme de Prim est un des  [algorithmes de glouton](https://fr.wikipedia.org/wiki/Algorithme_glouton) qui calcule un [arbre couvrant minimal](https://fr.wikipedia.org/wiki/Arbre_couvrant_minimal) dans un [graphe connexe](https://fr.wikipedia.org/wiki/Graphe_connexe) valué et non orienté. En d'autres termes, cet algorithme trouve un sous-ensemble d'arêtes formant un [arbre](https://fr.wikipedia.org/wiki/Arbre_(informatique)) sur l'ensemble des sommets du graphe initial e, tel que la somme des poids de ces arêtes soit minimale. Si le graphe n'est pas connexe, alors l'algorithme détermine un arbre couvrant minimal d'une composante connexe du graphe.

***Idée principale de l’algorithme*** : maintenir un sous graphe partiel connexe

***Principe :***

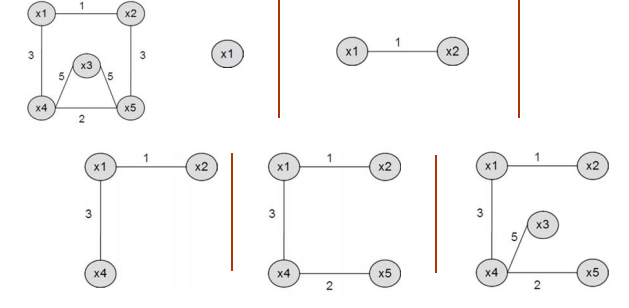
L'algorithmeconsiste à faire croître un arbre depuis un sommet. On part d'un sous-ensemble contenant un sommet unique. À chaque itération, on agrandit ce sous-ensemble en prenant l'arête incidente à ce sous-ensemble de coût minimum. En effet, si l'on prend une arête dont les deux extrémités appartiennent déjà à l'arbre, l'ajout de cette arête créerait un deuxième chemin entre les deux sommets dans l'arbre en cours de construction et le résultat contiendrait un cycle.

***Pseudo code***



Au début tous les sommets sont dans la file de priorité. La priorité est donnée par cout. Autrement dit, le sommet possédant la plus faible valeur sortira en premier de la file. On retire un à un les sommets de la file de priorité. L'algorithme retourne l'arbre couvrant de poids minimum

**Exemple**



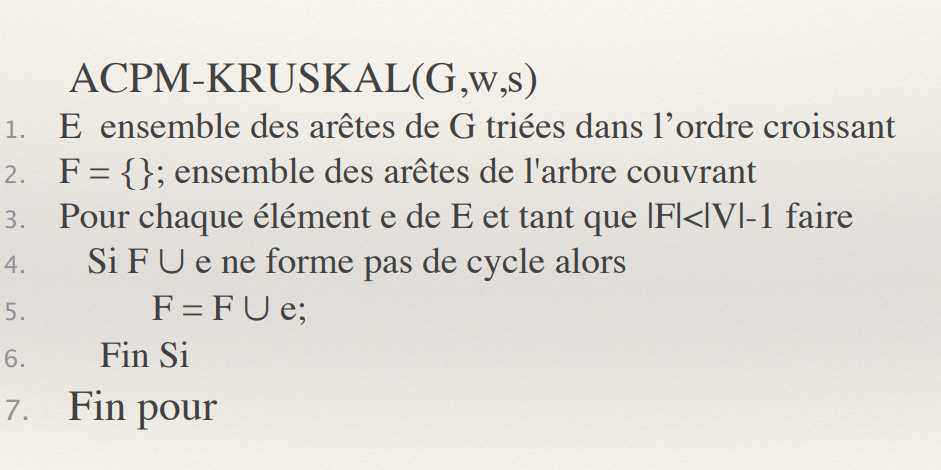
***Convergence***: On détermine un arbre couvrant si le graphe initial est connexe arrêt lorsque tous les sommets sont « marqués »

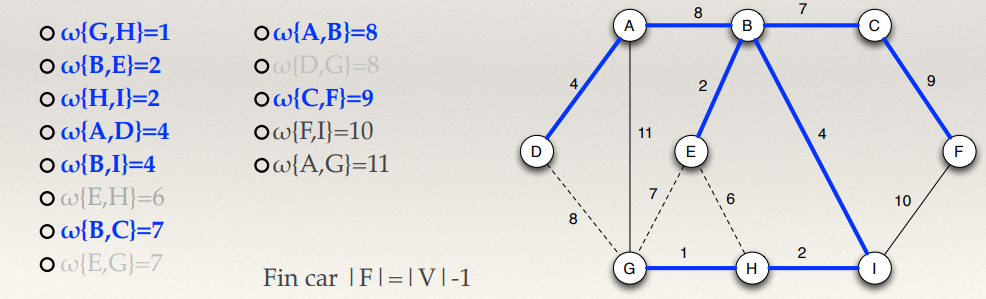
***Optimalité*** : Admise

***Complexité* :** l’algorithme contient |V| étapes (Entrées des sommets dans S), à chaque étape il faut déterminer l’arête de poids minimal, vérifier toutes les arêtes incidentes à un sommet S, déterminer si elles sont sortantes, garder celles de poids minimum. Cette implémentation conduit à une complexité d’ordre de **la somme des degrés** pour chaque sommet étape et on sait que la somme des degrés d’un sommet est égale à deux fois le nombres d’arêtes. La complexité totale nous donne : ***o (|V|. |E|)***, en utilisant la tas de Fibonacci on peut obtenir ***o*(|V|+|E|x log (|V|),**

**Algorithme de Kruskal**

***Principe*** : Algorithme de Kruskal se base sur la caractérisation des arbres comme étant des graphes acycliques maximaux. Il va ajouter au fur et à mesure des arêtes en s’assurant que le graphe partiel reste une forêt. L’algorithme s’arrête quand la forêt devient un arbre.



**Exemple**

***Convergence***: On détermine un arbre couvrant si le graphe initial est connexe arrêt si le nombre requis d’arêtes est sélectionné.

***Optimalité*** : Admise.

***Complexité***

le nombre d’itérations : N- 1 + initialisation tri des arêtes. A chaque étape : on cherche l’arête minimale ne créant pas de cycle et maintient les composantes connexes de la forêt en construction ; Si l’arête sélectionné relie 2 sommets de la même composante elle crée un cycle, on Fusionne les composantes connexes reliées par l’ajout d’une arête. La première phase de l’algorithme consiste à trier les arêtes : c’est **la phase la plus couteuse de l’algorithme**; le temps est de ***o(|E|.log |E|)*,**la seconde phase consiste à détecter si l’ajout d’une arête crée un cycle avec de ***o(|V|.log |V|)*,**ce temps est négligeable.

**Applications des arbres couvrants à coût**

Soit un graphe **G** représente le réseau de rues d’une ville que l’on cherche à câbler. Pour câbler de façon optimale cette ville il faut trouver un graphe connexe tel que :

* Chaque sommet doit avoir accès au réseau ;
* Le coût du câblage doit être minimum.

# Partie 6. Problèmes de flots

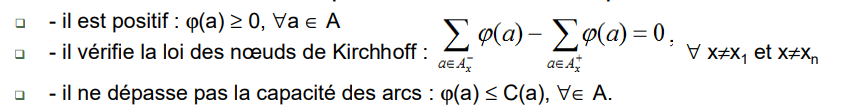
**Vocabulaire lié aux flots**

Un réseau est composé de nœuds et de liaisons joignant entre eux certains nœuds. On peut naturellement lui associer un graphe de la façon suivante : A chaque nœud i, on associe un sommet xi A chaque liaison (i, j), on associe un arc (xi , xj) si le mouvement partiel entre i et j ne peut se faire que de i vers j, et deux arcs de sens contraires si le mouvement p

* **Un réseau** : est un graphe fortement connexe, sans boucle et ayant plus d’un sommet ;
* **Un flot dans un réseau** : est l’affectation d’une valeur réelle à chaque arc, représentant une quantité transportée sur cet arc (vérifiant la loi de Kirchhoff en chaque nœud : conservation des flux) ;
* Le problème de la recherche d’un flot maximal est le plus important : **on se donne une capacité maximale sur chaque arc (borne supérieure). Il s'agit de déterminer un flot dont la valeur en un certain lieu est maximale**;
* On appelle **réseau de transport**: un graphe orienté antisymétrique valué **G = (X, A, C),** sans boucle et dans lequel il existe : un sommet **x1** sans prédécesseur (c’est-à-dire **Γ-1 (x1) = φ)** nommé **entrée ou source du réseau**, un sommet **xn** sans successeur (c’est à dire **Γ(xn) = φ) nommé sortie ou puits du réseau,** et tel qu’au moins un chemin unisse **x1** à **xn** dans G.

**Propriétés**

La fonction de pondération **C** est supposée positive et l’on nomme capacité de l’arc **a** le nombre **C(a).**

* Si**Ax-**est l’ensemble des arcs sortants et**Ax+** l’ensemble des arcs entrants du sommet **x,** on dit qu’une fonction**ϕ(a)** définie sur **A** et à valeurs réelles est **un flux** pour le réseau de transport si :
* Si x n’est ni **x1**, ni**xn,** la quantité entrante en **x** doit être égale à la quantité sortante que nous désignons par :



* Si **φ** est un flot sur un réseau de transport G, alors on a : ; cette quantité s’appelle **la valeur du flot ;**
* Pour un flot ϕ dans un réseau de transport **G = (X, A, C),** on dit qu’un arc est saturé si on a **φ (a) = C(a).**
* Le flot est dit complet si tout chemin allant de **x1** à **xn** contient au moins un arc saturé.

## 6.1. Amélioration d'un flot

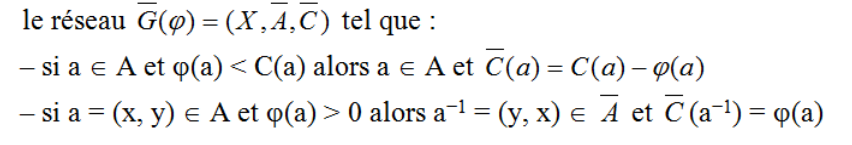
Soit **φ** un flot complet. On utilise une procédure itérative pour identifier et marquer tous les sommets du graphe où il est possible de faire transiter une unité de flot supplémentaire. On définit un processus d’étiquetage de certains sommets du graphe. En **x1**, on place une étiquette **+**. Soit **x** un sommet déjà marqué :

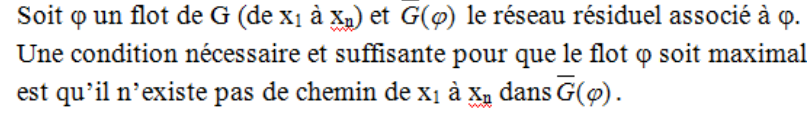
On marque avec une étiquette **+x** tout successeur **y** non marqué de **x** pour lequel le flot n’est pas à son maximum **((x, y) ∈ A et φ(x, y) < C (x, y)).**On marque avec une étiquette **−x** tout prédécesseur **y** non marqué de **x** pour lequel le flot n’est pas **nul ((x, y) ∈ A et φ (y, x) > 0).** L’étiquette a pour rôle de donner le nom d’un prédécesseur ou d’un successeur d’un sommet donné en indiquant si le flot peut être augmenté dans le sens de parcours **(étiquette +x)** ou diminué s’il est dans le sens contraire **(étiquette −x).**

Si l’on parvient jusqu’au marquage du sommet **xn** avec cette procédure, c’est qu’il existe une chaîne**µ** de x1 à **xn** dont tous les sommets sont marqués avec l’indice du sommet précédent au signe près. Notons que l’on n’est pas obligé de marquer tous les sommets (tous les successeurs ou prédécesseurs d’un sommet donné) ; l’objectif étant simplement d’élaborer une chaîne marquée de x1 à **xn**.

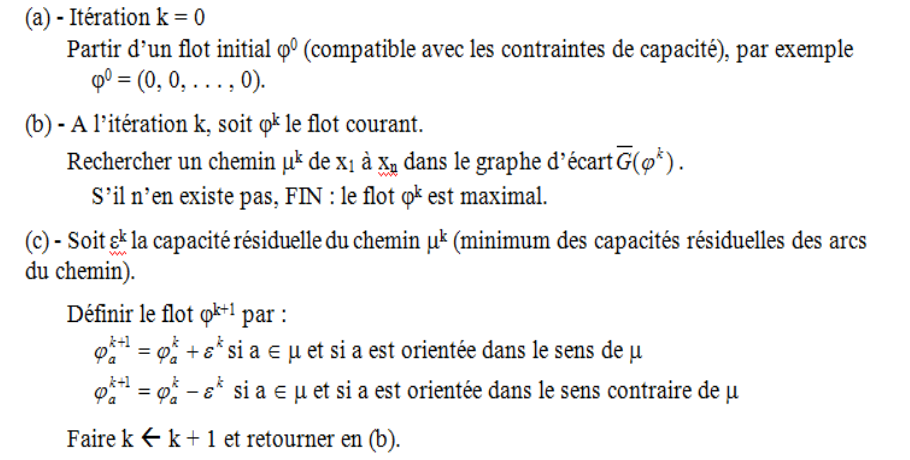
Soit alors**: ϕ′(a) = ϕ (a) si a ∉µϕ′(a) = ϕ (a) + 1 si a ∈µ**et si a est orientée dans le sens de**µϕ′(a) = ϕ (a) − 1 si a∈µ**et si a est orientée dans le sens contraire de **µ.**On vérifie aisément que **ϕ′**est encore un flot: l : la valeur du flot**ϕ′** est supérieure, donc meilleure que celle de **ϕ**. On systématise l’idée précédente avec la notion de graphe d’écart.

**Graphe d'écart ou réseau résiduel**

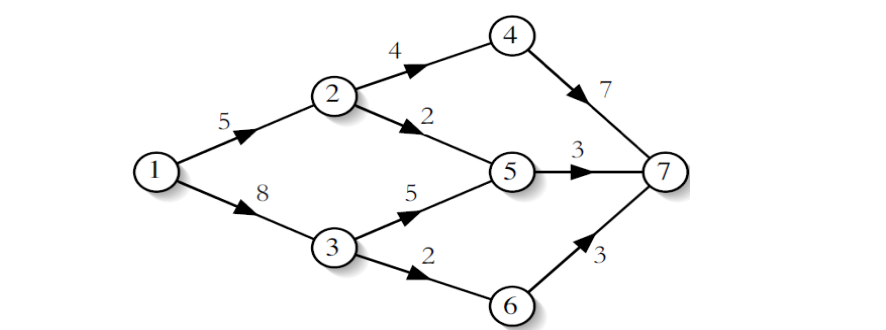
Soit un réseau de transport **G = (X, A, C)** possédant un flot complet **ϕ**. On appelle graphe d’écart (réseau résiduel) :

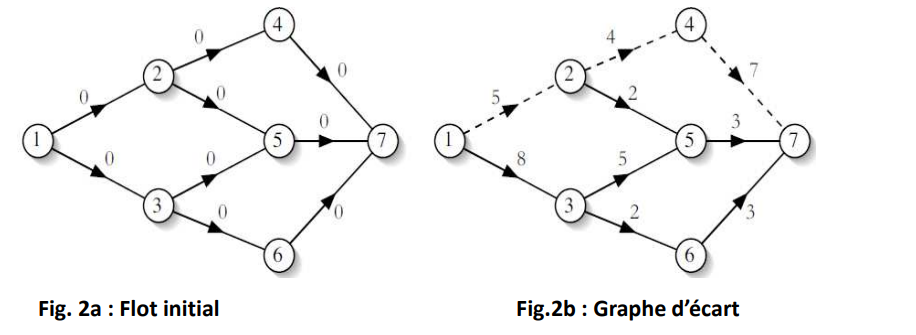
Le réseau résiduel indique le long de quels arcs on peut augmenter ou diminuer le flot. L’intérêt du graphe d’écart apparaît dans **le théorème** suivant :

## 6.2. Recherche d'un flot maximal

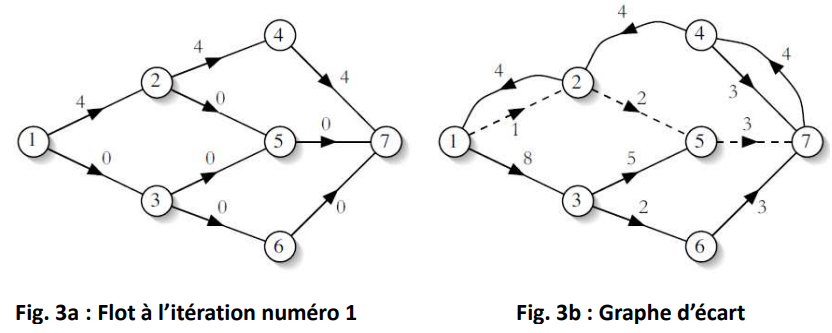
**Algorithme de Ford et Fulkerson**

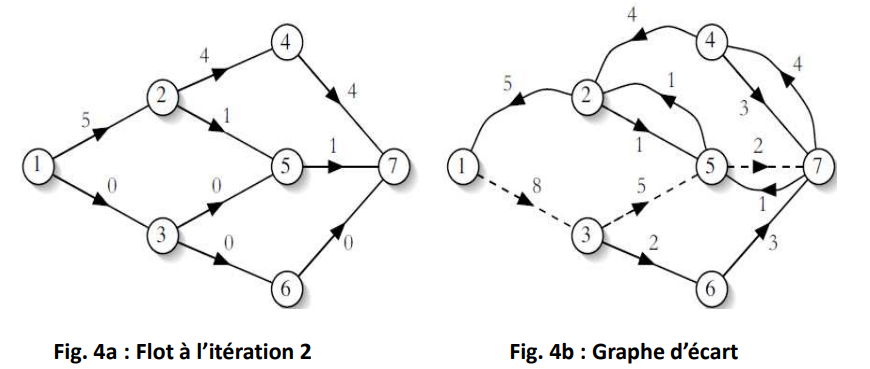
* Par définition du graphe d’écart, les flots **ϕk** sont tous compatibles avec les capacités.
* Par ailleurs, **εk** est positif à chaque itération et, par suite, l’algorithme produit une séquence de flots compatibles de valeurs strictement croissantes.
* A chaque étape, la méthode consiste donc à rechercher une chaîne joignant **x1 à xn.** Il est évident que le choix d’une telle chaîne n’est pas unique.
* Plusieurs travaux ont tenté de systématiser ce choix afin d’améliorer les performances moyennes de l’algorithme.

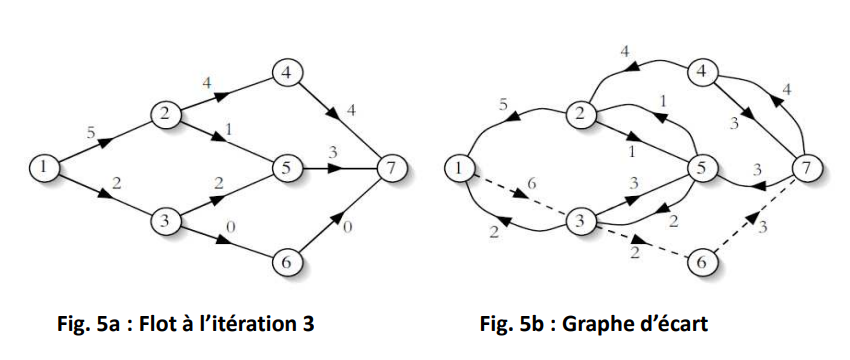
**Exemple** : On considère le graphe de la figure 1 (les nombres associés aux arcs représentent les capacités), pour lequel on cherche à déterminer un flot maximal entre le sommet 1 et le sommet 7.

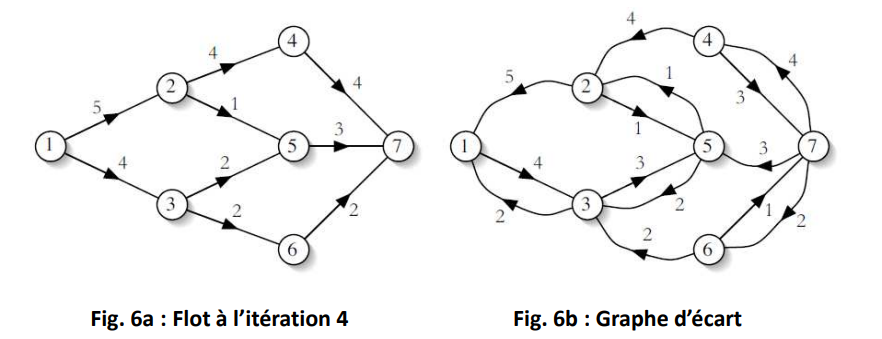
Toutes les composantes du flot initial sont considérées nullesϕ0 = (0, 0, . . . , 0) (figure 2a) ; le graphe d’écart est représenté figure 2b. Le chemin choisi est tracé en pointillés, sa capacité résiduelle vautε**1= min (5, 4, 7) = 4**.

On augmente donc de quatre le flot sur les arcs composant le chemin (ils sont tous orientés positivement). On obtient un nouveau flotϕ1, ainsi que son graphe d’écart, tous deux représentés aux figures 3a et 3b.



Le nouveau chemin est toujours représenté en traits pointillés. Cette fois, sa capacité résiduelle vaut ε2 = min (1, 2, 3) = 1. Le nouveau flot ϕ2 et le graphe d’écart correspondant sont donnés à la figure 4.

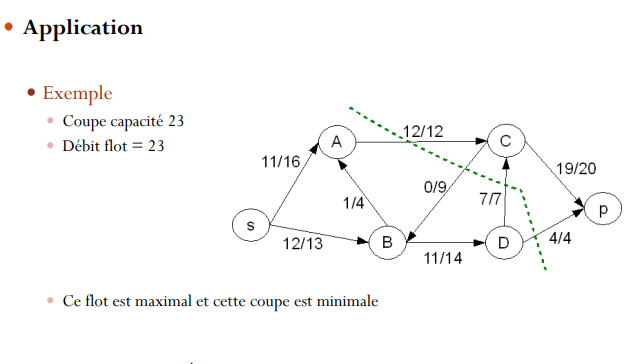
En poursuivant ainsi l’algorithme, on obtient les situations décrites aux figures suivantes.

Après l’itération numéro 4, on constate qu’il n’existe plus de chemin joignant le sommet 1 au sommet 7 dans le graphe d’écart. L’algorithme s’achève donc et le flot maximal est celui obtenu lors de cette dernière étape ϕ1=ϕ7= 9.

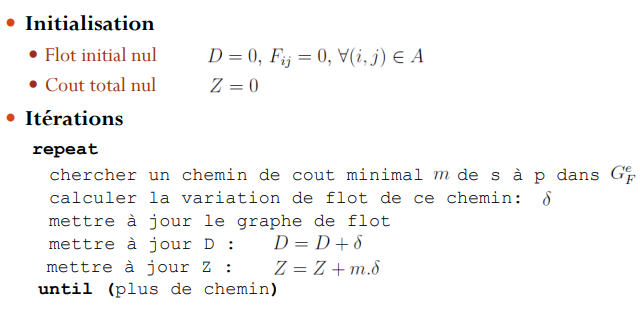
## 6.4. Complexité de l’Algorithme de Ford et Fulkerson

**Problème de flot maximal à cout minimal**

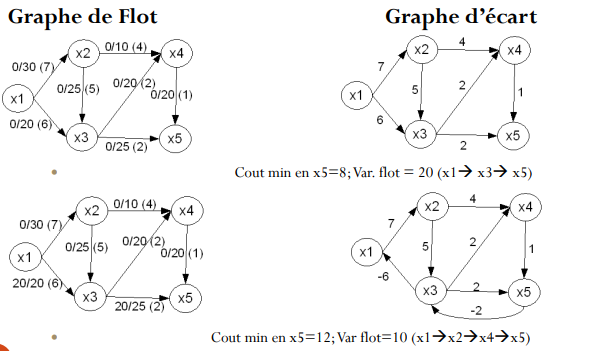
* Un coût de transport est associé à chaque lien du réseau (en plus des capacités). On déterminer un flot maximal de cout minimal entre un sommet source et un sommet puits en respectant les contraintes de capacité des arcs.

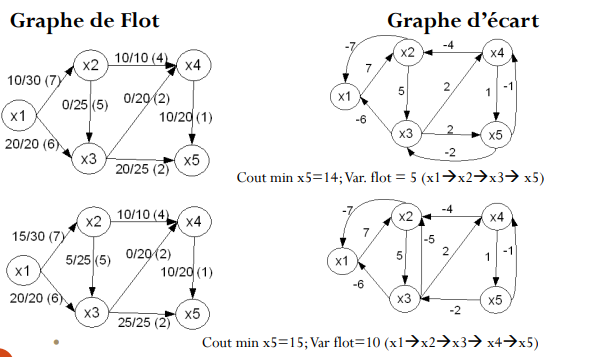


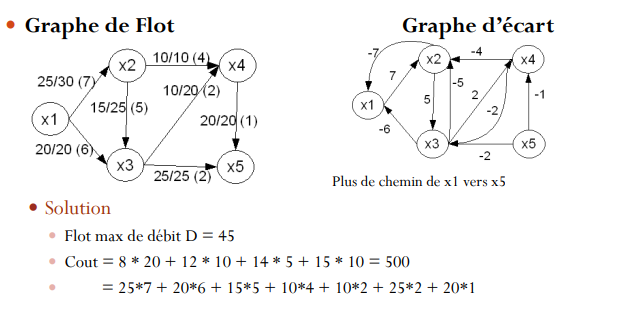
**Algorithme de Busacker Gowen**

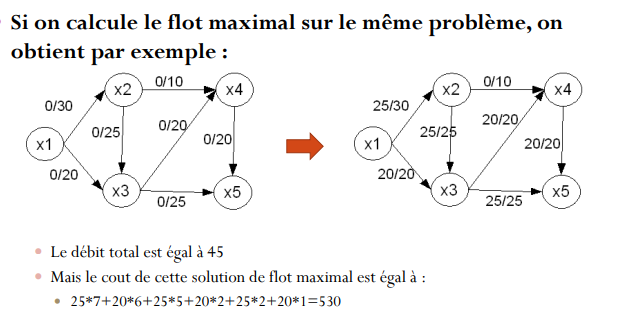


**Exemple Algorithme de Busacker Gowen**



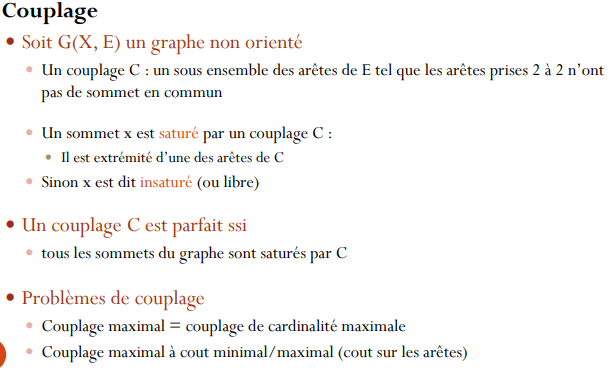


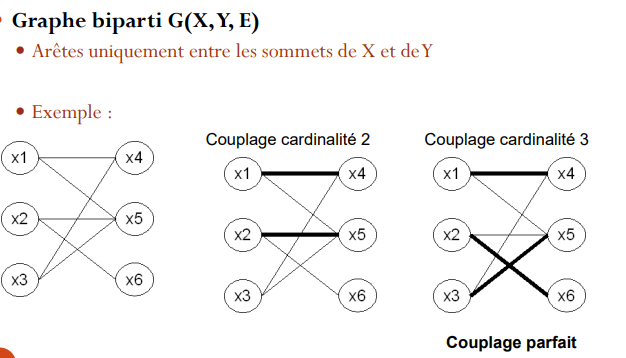




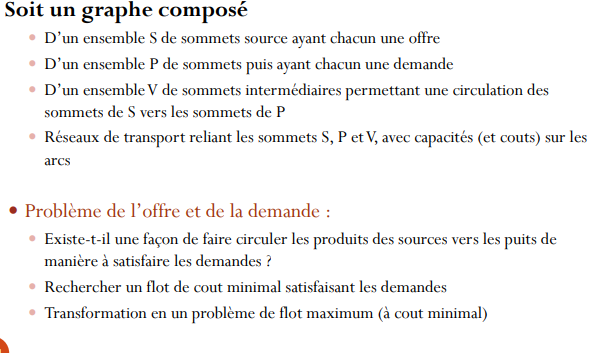
**Applications des flots**

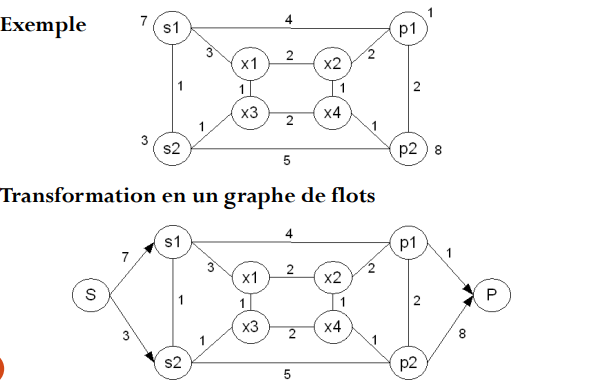
* **Couplage biparti**





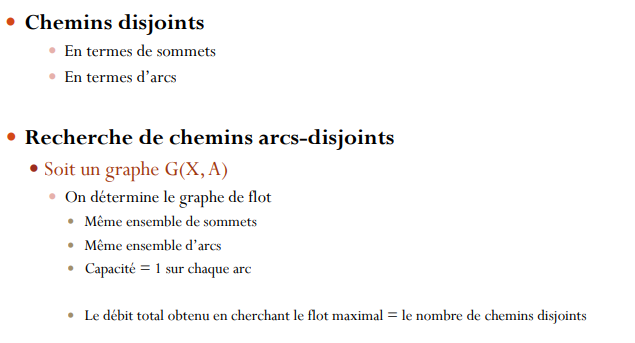
* **Applications des flots :Réseaux d’offres et de demandes**

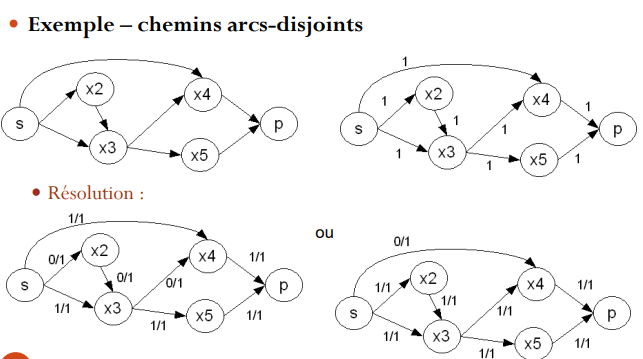




**Résolution** : Chercher un flot max de S vers P , Si la solution sature les arcs des sommets Pi vers P alors les demandes sont satisfaites Sinon les demandes ne peuvent être satisfaites.

**Applications des flots : Chemins disjoints**



****

**Les exemples d’application des flots**

* Un problème de transport (ou de circulation) a pour objet d'optimiser l'exécution d'un certain mouvement de matière, sur un réseau donné. Exemples :
* Expédition du pétrole brut depuis les régions productrices vers les raffineries des régions consommatrices;
* Déplacement des individus dans une ville pour se rendre à leur travail.
* Acheminement de moyens militaires en hommes et en matériel
* Conversations téléphoniques dans un réseau téléphonique.
* Circulation de gaz ou d'eau dans un réseau de canalisation,
* Réseau de rues, de routes, réseau métropolitain, réseau de transports aériens (ou maritimes),

# Partie 7. Ordonnancement et graphes

* Il s’agit de savoir planifier l’exécution de tâches qui ont une certaine durée, et qui ont entre elles des relations d’antériorité ;
* Tout projet comporte un certain nombre d'opérations élémentaires ;
* On regroupe celles-ci en tâches qui peuvent être soit une opération élémentaire soit un ensemble d'opérations élémentaires selon le degré désiré ;
* Le choix du découpage d'un projet est une opération délicate car il faut trouver un compromis entre un découpage trop fin qui risque de devenir inextricable (trop complexe et embrouillé) et un découpage trop lâche qui risque de masquer le caractère fondamental d'une opération élémentaire ;
* La plupart du temps, chaque tâche identifiée est, en fait, un sous projet plus ou moins complexe dont l'ordonnancement peut être traité à un niveau qui en fournit les caractéristiques globales.

Ces caractéristiques ne vont pas sans contraintes et exigences. On distingue :

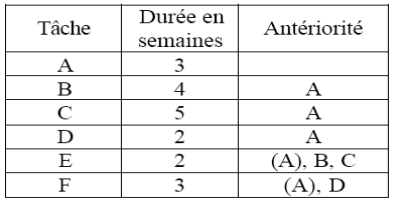
* **Contraintes de type potentiel**: elles portent sur le temps **ti**. On peut les séparer en deux classes :
* **Localisation dans le temps** : tâches ne pouvant pas commencer avant une date donnée ; tâche devant se terminer avant une certaine date
* **Succession entre tâches** : tâche **j** ne pouvant commencer avant que la tâche i ne soit terminée ; tâche j devant commencer exactement **aij** après le début de la tâche **i**.
* **Contraintes de type cumulatif** : La nécessité d'utiliser certains moyens (main d' œuvre, machines, matière première, financement, etc.) impose sur l'ordonnancement d'autres exigences provenant du fait que ces moyens ne sont disponibles qu'en quantité limitée. Ce type de contrainte est moins strict que le premier puisqu'il est toujours possible d'augmenter la quantité disponible (heures supplémentaires, sous-traitance de certaines opérations, etc.) ;
* **Contraintes de type disjonctif** : C'est lorsque deux tâches, par exemple, i et j, n'ont aucun intervalle de temps et particulièrement lorsque deux tâches nécessitent l'utilisation d'un même moyen unique. Si l'ordre de succession des tâches est imposé, une telle contrainte disjonctif n'est donc pas nécessaire.

## 7.1. Représentation graphique

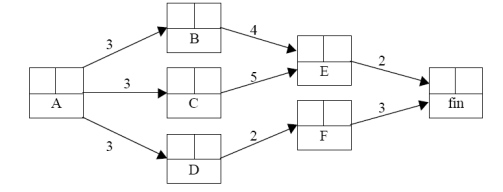
On se propose de représenter graphiquement et sous la forme d'un graphe les tâches que l'on vient de définir et, si possible, les contraintes correspondantes. Cette représentation sous forme d'un graphe orienté donne la répartition des différentes tâches, la direction et la longueur des arcs étant quelconques. On distingue deux types fondamentaux de représentation : potentiels-tâches (MPM) et potentiels-étapes (PERT).

**Exemple :**

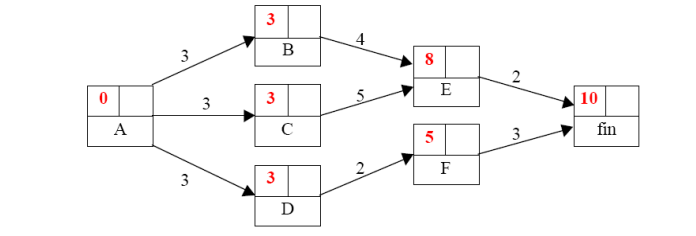
On remplira la colonne « antériorité » avec les tâches qui doivent être exécutées avant celle considérée. On n’utilisera que les « antériorités immédiates », c’est-à-dire que si la tâche E doit être traitée après la tâche B et que la tâche B doit être traitée après la tâche A, on ne marquera pas l’antériorité A dans la ligne consacrée à E. Les antériorités non immédiates ont été écrites entre parenthèses.



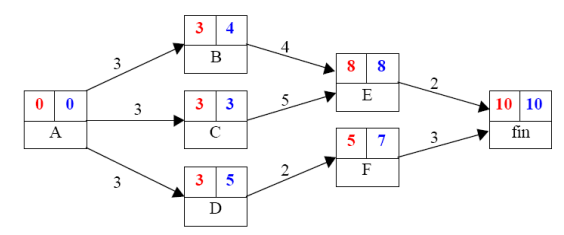
**Méthode Potentiel Métra : MPM**

**Modélisation** : On crée un graphe orienté dont les sommets sont les tâches ; on crée une tâche fictive qui est la tâche « fin » (sous-entendu « du processus »). Les arcs sont les relations d’antériorité immédiate ; ils sont values par la durée de la tâche source.

Des algorithmes de parcours de graphes permettent de calculer des chemins vérifiant certaines propriétés :

* **Dates « au plus tôt »** : On traite les sommets par niveaux en partant du début. Pour chaque sommet **i**on note la date **ti** qui est la longueur du plus long chemin de la tâche initiale à la tâche **i ;**

Le travail ne pourra donc pas être terminé avant 10 semaines. La tâche E ne pourra pas commencer avant 8 semaines, la tâche F avant 5 semaines, etc.

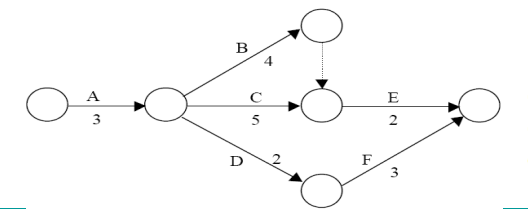
* **Dates « au plus tard »:**On traite les sommets en partant de la fin (en marquant 10 pour le sommet « fin »). Pour chaque sommet on note la date **ti\***qui est la longueur du plus court chemin de la tâche **i**à la tâche « fin ».

Pour effectuer l’ensemble des tâches en 10 semaines, il faudra avoir commencé la tâche E au bout de 8 semaines, commencé la tâche F au bout de 7 semaines, etc**.**

Il y a des tâches critiques, celles pour lesquelles on a : **ti\* = ti** : la tâche E devra être effectuée en 8 semaines (ni plus ni moins) pour que le processus soit achevé au bout des 10 semaines. Les tâches critiques définissent un ou plusieurs chemins critiques composés de tâches dont l’exécution ne doit connaître aucun retard pour que le projet soit achevé au plus tôt. Par contre il y a de la latitude pour les tâches qui ne sont pas critiques : la tâche F pourra être démarrée entre la semaine 5 et la semaine 7. De même il y a un chemin critique : **A – C – E – fin** (il y a toujours un chemin critique dans un graphe MPM).

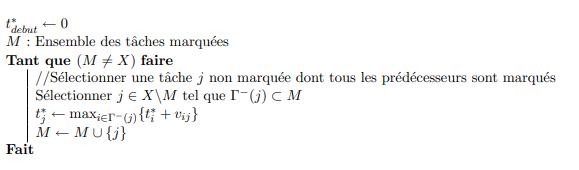
**Méthode américaine : PERT**

**PERT s**ignifie **Program Evaluation and Review Technical**:

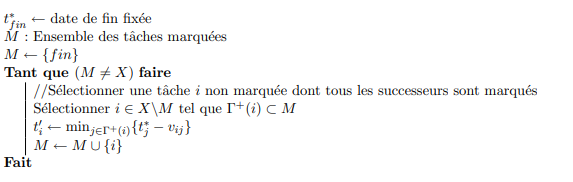
**Modélisation** : On construit un graphe dont les arcs représentent les tâches ; ils sont valués par la durée. Intuitivement, un sommet ayant un arc entrant étiqueté X correspond à « la tâche X est terminée », un sommet ayant deux arcs entrants X et Y a « les tâches X et Y sont terminées ». Selon les relations d’antériorité immédiate, on peut être amené à rajouter des arcs fictifs de coût 0 (voir exemple ci-dessous).

On est obligé de rajouter un arc fictif de coût 0 (en pointillés), car la tâche E ne peut démarrer que lorsque B et C sont terminées.

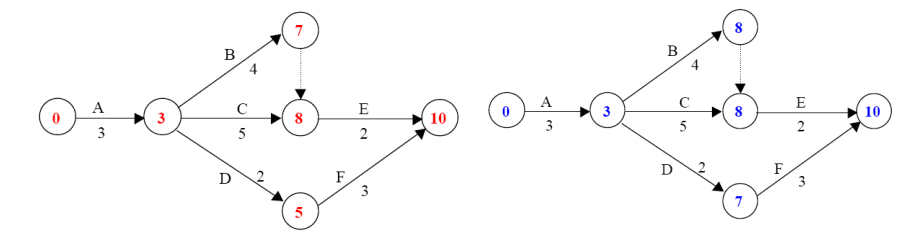
* **Dates "au plus tôt" & "au plus tard"** :

Au niveau de chaque sommet, on calcule la longueur du plus long chemin du sommet source à ce sommet. On a donc les dates de fin « au plus tôt ». À l’envers, on calcule la longueur du chemin le plus court conduisant au sommet « fin ». On a donc les dates « au plus tard ».

**Algorithme Dates "au plus tôt"**



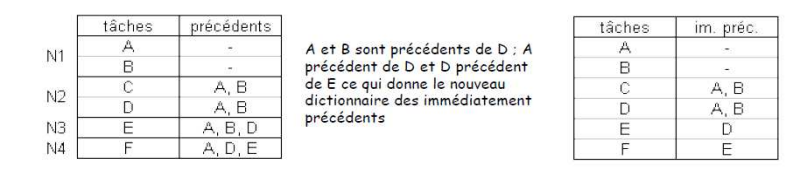
**AlgorithmeDates "au plus tard"**



## 7.2. Procédure d’ordonnancement des projets

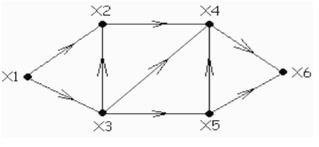
Il s’agit :

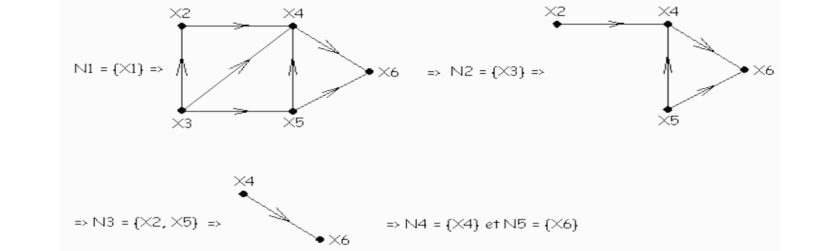
* Décomposer le graphe en niveaux (voir ci-dessous)
* Présenter le graphe sous forme de dictionnaire des précédents dans l'ordre des niveaux
* Déterminer les tâches immédiatement précédentes en supprimant les tâches redondantes de la manière suivante :
* Si deux ou plusieurs tâches sont telles qu’une ou plusieurs sont précédentes d'une tâche, on supprime de la colonne des précédents les tâches redondantes.



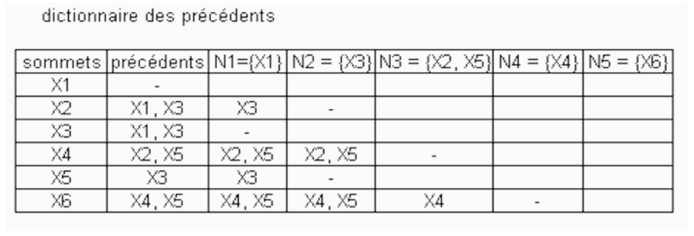
La décomposition d'un graphe en niveaux s'applique aux **graphes connexes sans circuits**. Elle consiste à ordonner les sommets d'un graphe en niveaux de façon à ce qu'aucun retour vers l'arrière ne soit possible. Nous présentons deux techniques :

* A partir du graphe : en éliminant à chaque itération "i" tous les sommets qui n'ont pas de précédents. Ces sommets constitueront donc le niveau Ni.
* A partir du dictionnaire des précédents : Les sommets n'ayant pas de précédent constituent le premier niveau, puis, à chaque itération, on élimine du dictionnaire des précédents les sommets appartenant aux niveaux précédents, les sommets qui se trouvent, à cette itération **"i",** sans précédents feront partie du niveau recherché **Ni.**

**Exemple** : Considérons le graphe suivant

Décomposition à partir du graphe :

Décomposition à partir du dictionnaire des précédents :



Dans tous les deux cas, on obtient le graphe suivant :

# CONCLUSION GENERALE

Il était question, dans ce cours, de faire une étude les algorithmes des graphes. Nous avons énuméré et décrit entre autre les algorithmes de parcours (largeur, profondeur et du tri topologique), les algorithmes de recherche de l’arbre couvrant minimal basé sur les algorithmes de Kruskal et de Prim, les algorithmes sur la recherche du plus court chemin basé sur les algorithmes de Bellman-Ford et Dijkstra, la recherche du flot maximal et l’ordonnancement tout en donnant la complexité de chacun d’eux. Il en ressort que ses méthodes sont des solutions optimales pour la résolution de nombreux problèmes dans la vie courante notamment dans les disciplines telles que : la mathématique, la physique, l’économie, la chimie, sociologie, bio-informatique, recherche opérationnelle, réseaux de communication, fonctionnement de systèmes.

# Références bibliographiques

[1] <https://fr>.m.wikipedia.org/wiki/probl%C3%A8me\_de\_flot\_maximum

[2] <https://www.pairform.fr/doc/1/32/180/web.co/module_src_11.html>

[3] <https://fr.m.wikipedia.org/wiki/Algorithme_de_tarjan>

[4] <https://devellopement-informatique.com/article/181/algorithme-de-chemin-le-plus-court-de-bellman-ford>

[5] Algorithme de Graphes Lucas Létocart Université de Paris 13