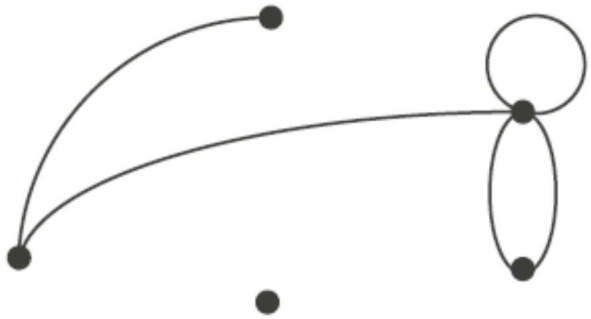
Chapitre I.

1. Définitions de base

1.1. Définition "intuitive" d'un graphe

Conceptuellement, un graphe est formé de sommets et d'arêtes reliant les sommets.



1.2. Définition mathématique d'un graphe

De manière plus formelle, on peut définir un graphe comme un couple G = (S,A) tel que

- S est un ensemble fini de sommets (Nœuds),

- A est un ensemble de couples de sommets $(s_i, s_j) \in S^2$.

On peut distinguer deux types de graphes :

Graphe orienté : les couples $(s_i, s_j) \in A$ sont orientés, c'est à dire que (s_i, s_j) est un couple ordonné où s_i est le sommet initial, et s_j le sommet terminal. Un couple (s_i, s_j) est appelé un arc, et est représenté graphiquement par $s_i \rightarrow s_j$.

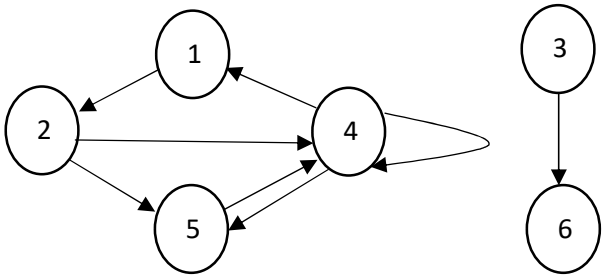


Figure 1 Graphe orienté

Le graphe orienté G1 = (S,A) avec S = {1, 2, 3, 4, 5, 6} et

A = {(1, 2), (2, 4), (2, 5), (4, 1), (4, 4), (4, 5), (5, 4), (6, 3)}.

Graphe non orienté : les couples $(s_i, s_j) \in A$ ne sont pas orientés, c'est à dire que (s_i, s_j) est équivalent à (s_j, s_i). Une paire (s_i, s_j) est appelée une arête, et est représentée graphiquement par $s_i — s_j$.

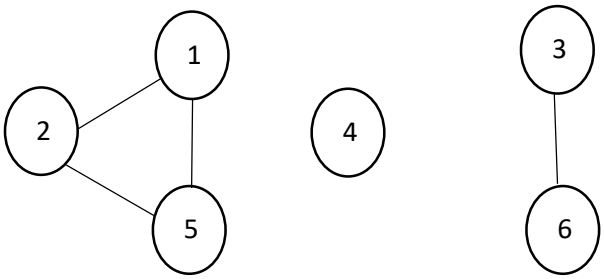


Figure 2 graphe non orienté

Le graphe non orienté G2 = (S,A) avec S = {1, 2, 3, 4, 5, 6} et

A = {(1, 2), (1, 5), (5, 2), (3, 6)}.

1.3. Terminologie

L'ordre d'un graphe |G| est le nombre de ses sommets. Exemple |G|=6.

– Une boucle est un arc ou une arête reliant un sommet à lui-même.

Exemple

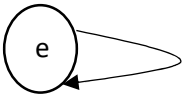


Figure 3 boucle

– Un graphe non-orienté est dit simple s'il ne comporte pas de boucle, et s'il ne comporte jamais plus d'une arête entre deux sommets. Un graphe non orienté qui n'est pas simple est un multigraphe. Dans le cas d'un multi-graphe, A n'est plus un ensemble mais un multi-ensemble d'arêtes. Exemple : Le graphe 2 est simple, le graphe 1 ne l'est pas (contient une boucle).

– Un graphe orienté est un p-graphe s'il comporte au plus p arcs entre deux sommets.

Exemple : 3-graphe, il existe trois arc entre (a,b)

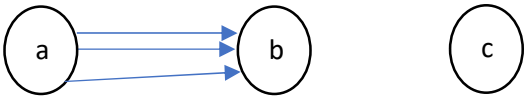


Figure 4 un 3-graphe

– Un graphe partiel d'un graphe orienté ou non est le graphe obtenu en supprimant certains arcs ou arêtes. Exemple d'un graphe partiel du graphe G1

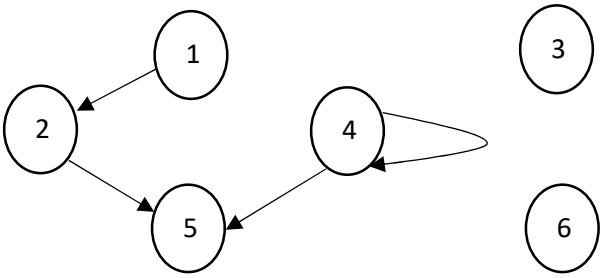


Figure 5 Un graphe partiel

– Un sous-graphe d'un graphe orienté ou non est le graphe obtenu en supprimant certains sommets et tous les arcs ou arêtes incidents aux sommets supprimés. Exemple d'un sous graphe de G1

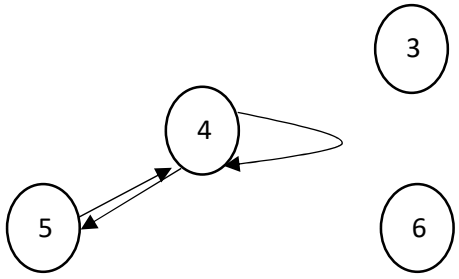


Figure 6 Un sous graphe

– Un graphe partiel d'un sous-graphe est un sous-graphe partiel. Exemple : un sous graphe du graphe partiel de la figure 5.

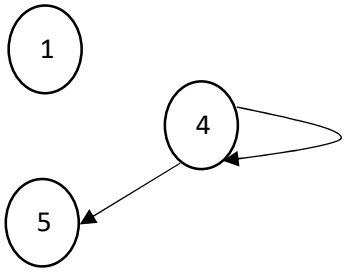


Figure 7 : un sous graphe du graphe partiel de la figure 5.

– Un graphe orienté est dit élémentaire s'il ne contient pas de boucle. Exemple le graphe G1 n'est pas élémentaire puisque il contient une boucle.

– Un graphe orienté est dit complet s'il comporte un arc (s_i, s_j) et un arc (sj , s_i) pour tout couple de sommets différents s_i, $s_j \in S^2$. De même, un graphe non-orienté est dit complet s'il comporte une arête (s_i, s_j) pour toute paire de sommets différents s_i, $s_j \in S^2$.

Exemple

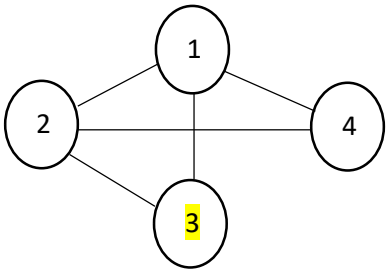


Figure 8 : Un graphe non orienté complet

1.4. Notion d'adjacence entre sommets :

– Dans un graphe non orienté, un sommet s_i est dit adjacent à un autre sommet s_j s'il existe une arête entre s_i et s_j . L'ensemble des sommets adjacents à un sommet s_i est défini par : $\Gamma(s_i)=adj(s_i) = \{s_j / (s_i , s_j) \in A$ ou $(s_j , s_i) \in A\}$

Exemple : dans le graphe 2, l'ensemble des sommets adjacents au sommet 2 sont {1,5}

– Dans un graphe orienté, on distingue les sommets successeurs des sommets prédécesseurs :

$\Gamma^+(s_i)=succ(s_i) = \{ s_j /(s_i, s_j) \in A\}$

$\Gamma^-(s_i)=pred(s_i) = \{ s_j /(s_j , s_i) \in A\}$

$\Gamma(s_i)= \Gamma^+(s_i) \cup \Gamma^-(s_i)$

1.5. Notion de degré d'un sommet :

On appelle degré d'un sommet le nombre d'arêtes dont ce sommet est une extrémité (les boucles étant comptées deux fois). Ce degré vaut 0 si le sommet est isolé.

Exemple : Dans le graphe G2, les degrés des sommets sont :

x	1	2	3	4	5	6
d(x)	2	2	1	0	2	1

Degré d'un sommet dans un graphe orienté. On note $d^+(s)$ le degré extérieur du sommet s, c'est-à-dire le nombre d'arcs ayant s comme extrémité initiale.

On note $d^-(s)$ le degré intérieur du sommet s, c'est-à-dire le nombre d'arcs ayant s comme extrémité finale.

Le degré du sommet s est : $d(s)=d^+(s)+d^-(s)$

Exemple : dans le graphe G1, les degrés des sommets sont :

X	$d^+(X)$	$d^-(X)$	d(X)
1	1	0	1
2	2	1	3
3	0	0	0
4	3	3	6
5	1	2	3
6	0	0	0

1.6. Représentation par matrice d'adjacence

Soit le graphe G = (S,A). On suppose que les sommets de S sont numérotés de 1 à n, avec n =|S|. La représentation par matrice d'adjacence de G consiste en une matrice booléenne M de taille $n \times n$ telle que M[i][j] = 1 si $(i, j) \in A$, et M[i][j] = 0 sinon.

Exemple : le graphe G1 est présenté par la matrice d'adjacence suivante

	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	0	1	1	0
3	0	0	0	0	0	1
4	1	0	0	1	1	0
5	0	0	0	1	0	0
6	0	0	0	0	0	0

Si le graphe est valué (par exemple, si des distances sont associées aux arcs), on peut utiliser une matrice d'entiers, de telle sorte que M[i][j] soit égal à la valuation de l'arc (i, j) si $(i, j) \in A$. S'il n'existe pas d'arc entre 2 sommets i et j, on peut placer une valeur particulière (par exemple 0 ou $-\infty$ ou null) dans M[i][j].

1.7. Représentation par listes d'adjacence

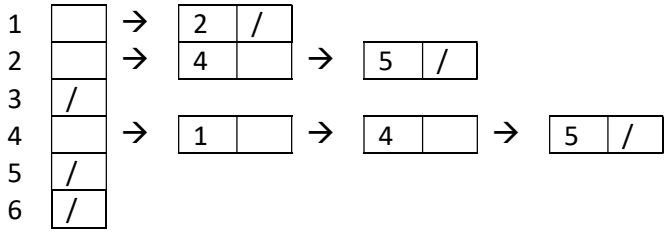
Soit le graphe G = (S,A). On suppose que les sommets de S sont numérotés de 1 à n, avec n = |S|. La représentation par listes d'adjacence de G consiste en un tableau T de n listes, une pour chaque sommet de S. Pour chaque sommet $s_i \in S$, la liste d'adjacence $T[s_i]$ est une liste chainée de tous les sommets s_j tels qu'il existe un arc ou une arête $(s_i, s_j) \in A$. Autrement dit, $T[s_i]$ contient la liste de tous les sommets successeurs de s_i. Les sommets de chaque liste d'adjacence sont généralement chainés selon un ordre arbitraire.

Si le graphe est valué (par exemple, si les arêtes représentent des distances), on peut stocker dans les listes d'adjacence, en plus du numéro de sommet, la valuation de l'arête.

Dans le cas de graphes non orientés, pour chaque arête (s_i, s_j), on aura s_j qui appartiendra à la liste chainée de $T[s_i]$, et aussi s_i qui appartiendra à la liste chainée de $T[s_j]$.

Exemple

	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	0	1	1	0
3	0	0	0	0	0	0
4	1	0	0	1	1	0
5	0	0	0	1	0	0
6	0	0	0	0	0	0



2. Cheminements et connexités

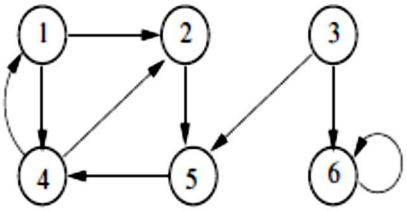
2.1 Chemin

Dans un graphe orienté, un **chemin** d'un sommet u vers un sommet v est une séquence $< s_0, s_1, s_2, ..., s_k >$ de sommets tels que $u = s_0$, $v = s_k$, et $(s_{i-1}, s_i) \in A$ pour $i \in [1..k]$. La longueur du chemin est le nombre d'arcs dans le chemin, c'est-à-dire k. On dira que le chemin contient les sommets $s_0, s_1, ..., s_k$, et les arcs $(s_0, s_1), (s_1, s_2), ..., (s_{k-1}, s_k)$. S'il existe un chemin de u à v, on dira que v est accessible à partir de u. Un chemin est élémentaire si les sommets qu'il contient sont tous distincts.

2.2. Circuit

Dans un graphe orienté, un **chemin** $< s_0, s_1, s_2, ..., s_k >$ forme un circuit si $s_0 = s_k$ et si le chemin comporte au moins un arc ($k \geq 1$). Ce circuit est élémentaire si en plus les sommets $s_1, s_2, ..., s_k$ sont tous distincts. Une boucle est un circuit de longueur 1.

Considérons par exemple le graphe orienté suivant :



Un chemin élémentaire dans ce graphe est < 1, 4, 2, 5 >.

Un chemin non élémentaire dans ce graphe est < 3, 6, 6, 6 >.

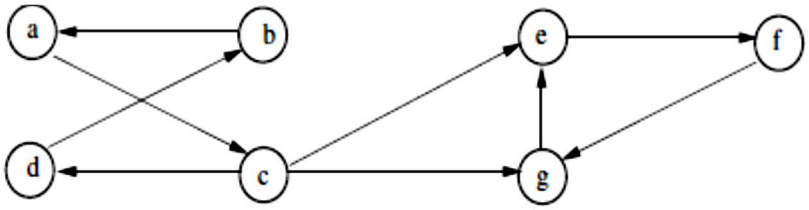
Un circuit élémentaire dans ce graphe est < 1, 2, 5, 4, 1 >.

Un circuit non élémentaire dans ce graphe est < 1, 2, 5, 4, 2, 5, 4, 1 >.

On retrouve ces différentes notions de cheminement dans les graphes non orientés. Dans ce cas, on parlera de chaine au lieu de chemin, et de cycle au lieu de circuit. Un graphe sans cycle est dit acyclique.

2.3. Le nombre de chemins

Une matrice d'adjacence à la puissance n permet de connaître le nombre de chemins de longueurs n entre n'importe quel couple de point du graphe.



La matrice d'adjacence associée à ce graphe est la matrice M suivante :

	a	b	c	d	e	f	g
a	0	0	1	0	0	0	0
b	1	0	0	0	0	0	0
c	0	0	0	1	1	0	1
d	0	1	0	0	0	0	0
e	0	0	0	0	0	1	0
f	0	0	0	0	0	0	1
g	0	0	0	0	1	0	0

En multipliant cette matrice par elle-même, on obtient la matrice M^2 des chemins de longueur 2 :

	a	b	c	d	e	f	g
a	0	0	0	1	1	0	1
b	0	0	1	0	0	0	0
c	0	1	0	0	1	1	0
d	1	0	0	0	0	0	0
e	0	0	0	0	0	0	1
f	0	0	0	0	1	0	0
g	0	0	0	0	0	1	0

En additionnant M et M^2, on obtient la matrice M +M^2 des chemins de longueur inférieure ou égale à 2 :

	a	b	c	d	e	f	g
a	0	0	1	1	1	0	1
b	1	0	1	0	0	0	0
c	0	1	0	1	1	1	1
d	1	1	0	0	0	0	0
e	0	0	0	0	0	1	1
f	0	0	0	0	1	0	1
g	0	0	0	0	1	1	0

2.4. Notions de connexité

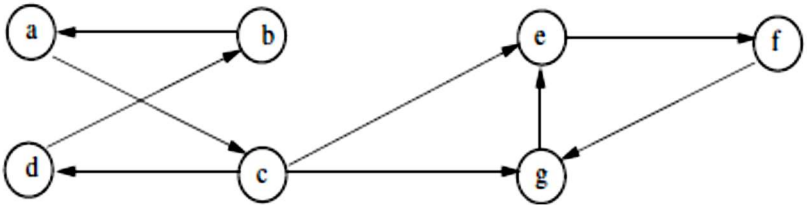
Un graphe non orienté est connexe si chaque sommet est accessible à partir de n'importe quel autre.

Par exemple, le graphe non orienté suivant n'est pas connexe.

Car il n'existe pas de chaîne entre les sommets a et e. En revanche, le sous-graphe défini par les sommets {a, b, c, d} est une **composante connexe.**

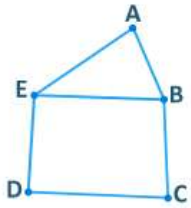
Un graphe orienté est fortement connexe si chaque sommet est accessible à partir de n'importe quel autre. Par exemple, le graphe de gauche ci-dessous est fortement connexe, tandis que celui de droite ne l'est pas :

Une composante fortement connexe d'un graphe orienté G est un sous-graphe G^0 de G qui est fortement connexe et maximal (c'est à dire qu'aucun autre sous-graphe fortement connexe de G ne contient G^0). Par exemple, le graphe orienté suivant :

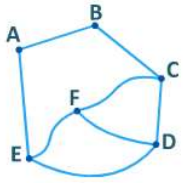


2.5. Un graphe eulérien

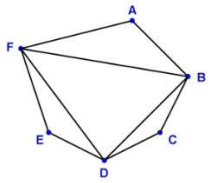
Une chaine eulérienne est une chaine qui emprunte une et une seule fois chaque arête du graphe. Un cycle eulérien est un cycle qui emprunte une et une seule fois chaque arête du graphe.
Un graphe comportant une chaine ou un cycle eulérien est appelé graphe eulérien.
Théorème : Un graphe (simple ou multiple) connexe admet un cycle eulérien si et seulement s'il n'a pas de sommet de degré impair.
Théorème : Un graphe (simple ou multiple) connexe admet une chaine eulérienne entre deux sommets u et v si et seulement si le degré de u et le degré de v sont impairs, et les degrés de tous les autres sommets du graphe sont pairs.
Exemples



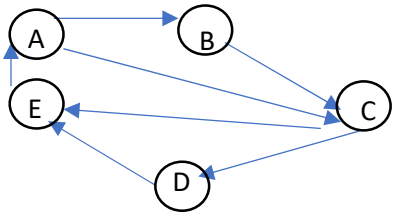
Ce graphe possède la chaine eulérienne suivante :B,A,E,D,C,B,E



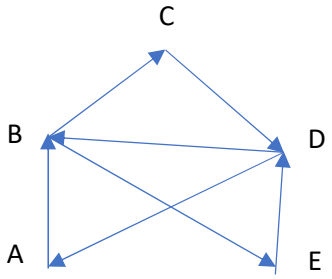
Ce graphe ne contient pas une chaine eulérienne puisqu'il existe trois sommets d'ordres impair



Ce graphe contient le cycle eulérien suivant : A,B,C,D,B,F,D,E,F,A
Un chemin eulérien est un chemin qui emprunte une et une seule fois chaque arc du graphe.
Un circuit eulérien est un circuit qui emprunte une et une seule fois chaque arc du graphe.
Théorème : Un multigraphe orienté fortement connexe admet un circuit eulérien si et seulement si $d^+(s_i) = d^-(s_i)$ pour tout sommet $s_i \in S$.
Théorème : Un graphe orienté connexe admet un chemin eulérien de u vers v si et seulement si
- $d^+(u) = d^-(u) + 1$
- $d^+(v) = d^-(v) - 1$
- $d^+(s_i) = d^-(s_i)$ pour tout autre sommet s_i $s_i \in S - \{u, v\}$



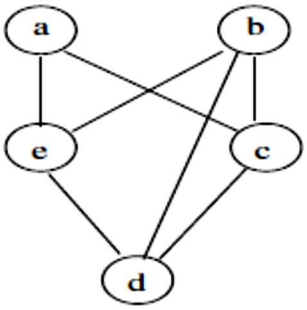
Le chemin A,B,C,D,E,A,C,E est chemin eulérien de longueur 7.



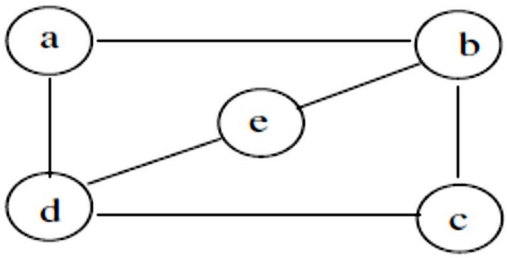
A,B,E,D,B,C,D,A est un circuit eulérien

2.6. Un graphe hamiltonien

Une chaine hamiltonienne passe une et une seule fois par chacun des n sommets du graphe. On appelle cycle hamiltonien un cycle élémentaire de longueur n. Un graphe possédant un cycle ou une chaine hamiltonien sera dit graphe hamiltonien.
Par exemple, le graphe suivant possède un cycle hamiltonien (< a, e, b, d, c, a >)



En revanche, le graphe suivant ne possède pas de cycle hamiltonien, mais possède une chaine hamiltonienne (< a, b, e, d, c >).

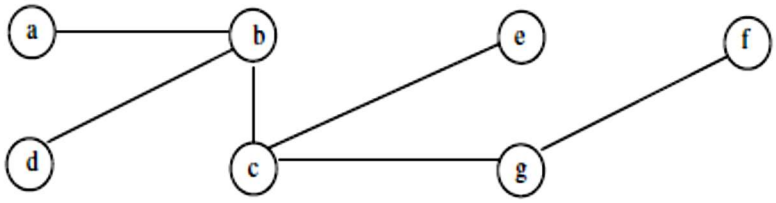


3. Arbres et arborescences

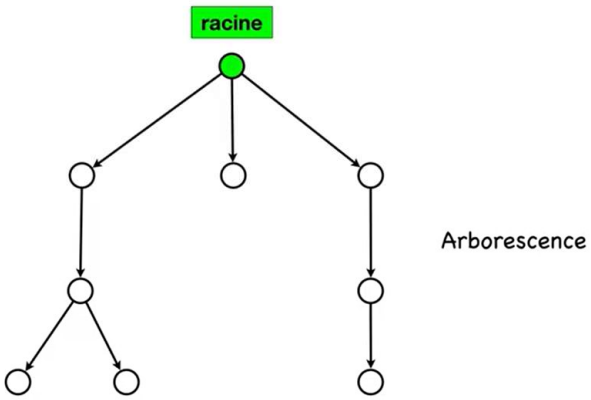
Les arbres et les arborescences sont des graphes particuliers très souvent utilisés en informatique pour représenter des données.
Etant donné un graphe non orienté comportant n sommets, les propriétés suivantes sont équivalentes pour caractériser un arbre :

1. G est connexe et sans cycle,
2. G est sans cycle et possède n - 1 arêtes,
3. G est connexe et admet n - 1 arêtes,
4. G est sans cycle, et en ajoutant une arête, on crée un et un seul cycle élémentaire,
5. G est connexe, et en supprimant une arête quelconque, il n'est plus connexe,
6. Il existe une chaine et une seule entre 2 sommets quelconques de G.

Par exemple, le graphe suivant est un arbre :

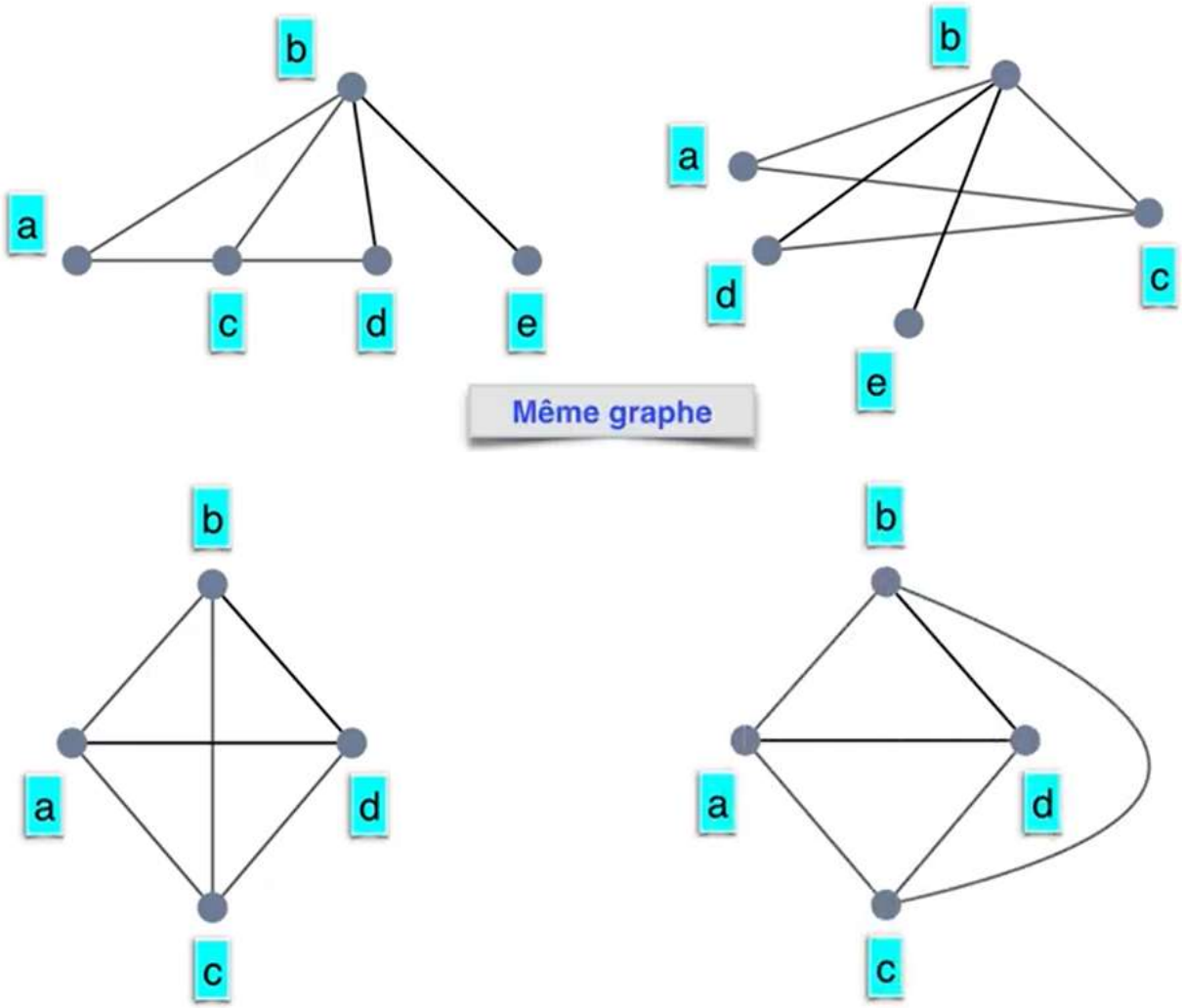


Une arborescence est un graphe orienté sans circuit admettant une racine $s_0 \in S$ telle que, pour tout autre sommet $s_i \in S$, il existe un chemin unique allant de s_0 vers s_i. Si l'arborescence comporte n sommets, alors elle comporte exactement n - 1 arcs.

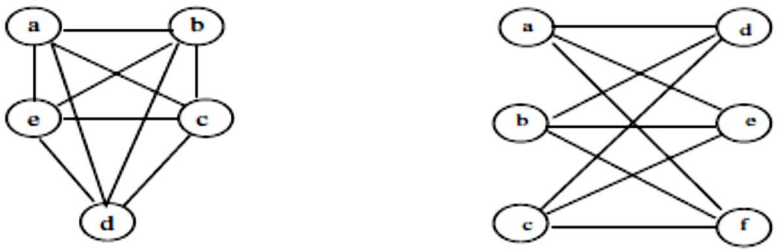


4. Graphes planaires

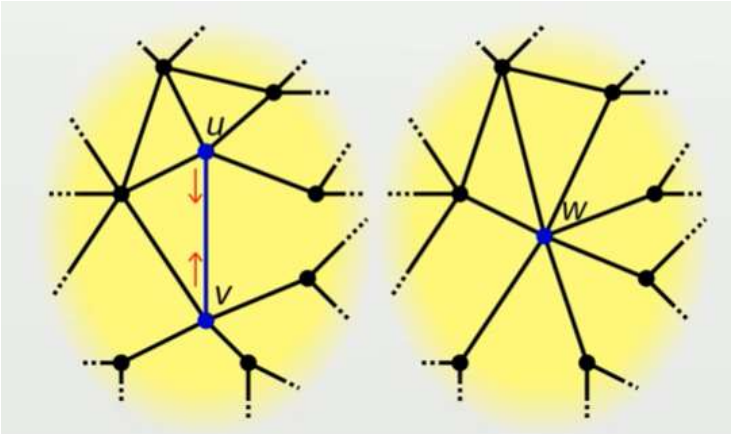
On appelle graphe planaire tout graphe non orienté pouvant être dessiné sur un plan de telle sorte que les sommets soient des points distincts, et que les arêtes ne se rencontrent pas en dehors de leurs extrémités.



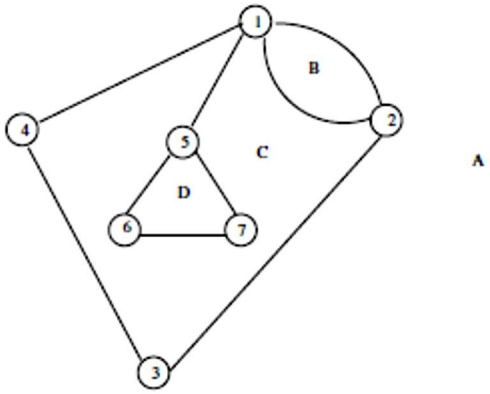
Caractérisation des graphes planaires : Les deux graphes non planaires les plus simples sont K_5 et $K_{3,3}$:



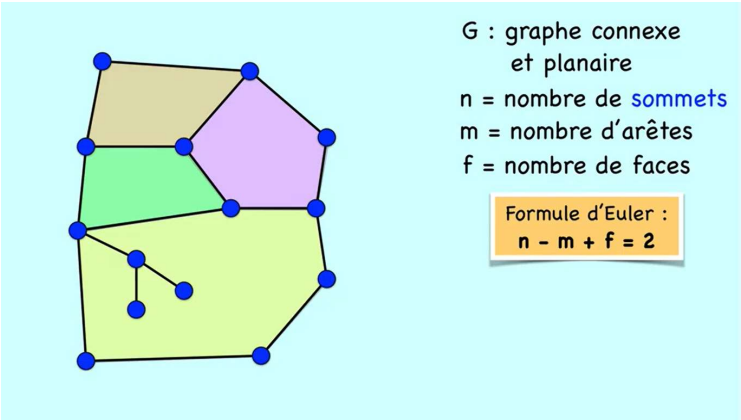
4.1. Théorème de Kuratowski : un graphe est planaire si et seulement s'il ne possède aucun mineur isomorphe à K_5 et $K_{3,3}$. (Un mineur d'un graphe est obtenu par une succession d'opérations de suppression et de "fusion" de sommets). Autrement dit, tout graphe non planaire contient une "copie" d'au moins un de ces deux graphes.



Faces d'un graphe planaire : Etant donnée une représentation planaire d'un graphe G, le plan se retrouve divisé en un certain nombre de régions qu'on appelle les faces de la représentation planaire. Par exemple, le graphe suivant possède 4 faces (notées A, B, C et D). On dira que les arêtes (1, 2), (1, 4), (4, 3), (3, 2), (5, 6) et (5, 7) constituent des frontières entre des faces différentes, tandis que l'arête (5, 1) constitue un isthme.



4.1. Formule d'Euler : Soit G un graphe planaire connexe possédant n sommets, m arêtes et f faces, on a : n-m+f=2.

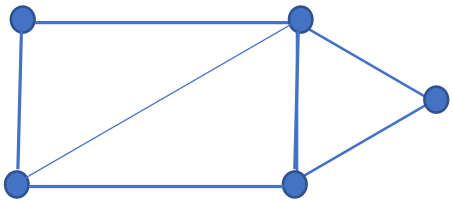


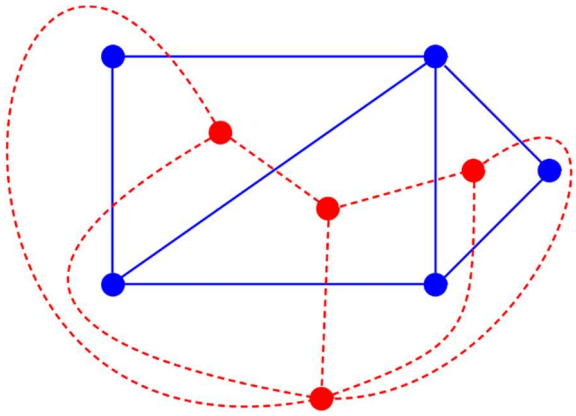
F=5, N=14, M=17 ; 14-17+5= ?2

Graphe Dual : On appelle dual d'un graphe planaire —appelé primal—le graphe obtenu de la façon suivante :

- dans toute face du primal on dessine un sommet du dual,
- pour toute arête séparant deux faces du primal, on dessine une arête joignant les deux sommets correspondants du dual (et qui traverse l'arête correspondante du primal).

Remarquons que cette relation est symétrique : si G_2 est le dual de G_1, alors G_1 est le dual de G_2.

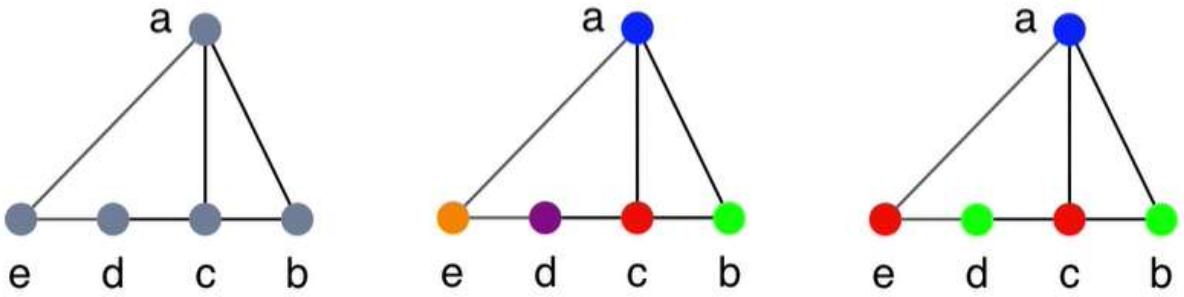




5. Coloriage de graphes

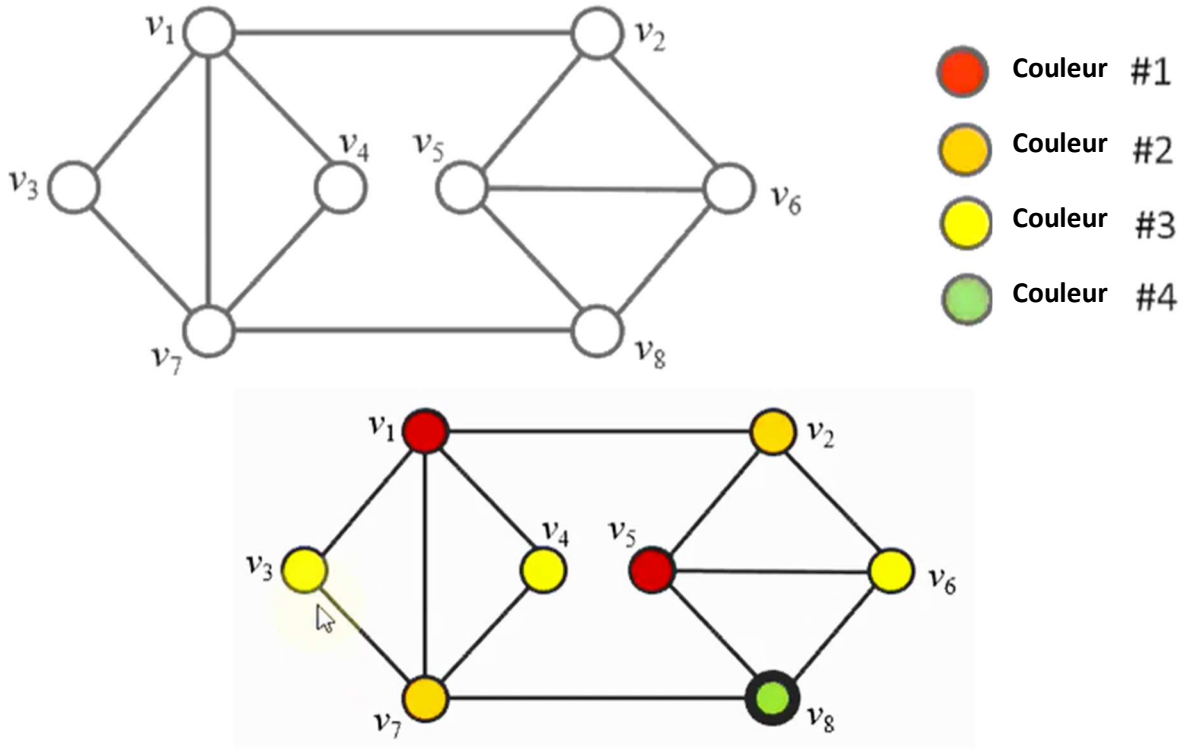
Pour un graphe non orienté G, le Coloriage consiste à attribuer une couleur à chaque sommet, de telle sorte qu'une même couleur ne soit pas attribuée à deux sommets adjacents.

Le nombre minimum de couleurs nécessaires pour colorier un graphe G est appelé le nombre chromatique de G, et noté X(G).



5.1. L'algorithme de Brélaz (DSATUR)

Tant qu'il existe un sommet non colorié, on choisit à chaque itération le sommet non colorié ayant le plus grand nombre de voisins coloriés avec des couleurs différentes, les ex aequo étant départagés en choisissant le sommet de plus fort degré. Le sommet choisi est alors colorié par la plus petite couleur possible (en partant du principe que les couleurs sont triées selon un ordre donné), si toutes les couleurs existantes sont utilisées par au moins un voisin du sommet à colorier, alors une nouvelle couleur est ajoutée à l'ensemble des couleurs disponibles.



Conjecture des 4 couleurs : Le nombre chromatique d'un graphe planaire est inférieur ou égal à 4.

6. Parcours de graphes

On étudie dans la suite les deux principales stratégies d'exploration : le parcours en largeur et le parcours en profondeur.

Dans les deux cas, l'algorithme procède par coloriage des sommets :

– Initialement, tous les sommets sont blancs. Lorsqu'un sommet est "découvert", il est colorié en gris. Le sommet reste gris tant qu'il reste des successeurs de ce sommet qui sont blancs. Un sommet est colorié en noir lorsque tous ses successeurs sont gris ou noirs.

6.1. Arborescence couvrante

On parcourt un graphe à partir d'un sommet donné s_0. Cette arborescence contient un arc (s_i, s_j) si et seulement si le sommet s_j a été découvert à partir du sommet s_i.

L'arborescence associée à un parcours de graphe sera mémorisée dans un tableau π tel que $\pi[s_j] = s_i$ si s_j a été découvert à partir de s_i, et $\pi[s_k]$ = nil si s_k est la racine, ou s'il n'existe pas de chemin de la racine vers s_k.

6.2. Parcours en largeur (Breadth First Search = BFS)

Le parcours en largeur est obtenu en gérant la liste d'attente au coloriage comme une file d'attente (FIFO = First In First Out). Autrement dit, on enlève à chaque fois le plus vieux sommet gris dans la file d'attente, et on introduit tous les successeurs blancs de ce sommet dans la file d'attente, en les coloriant en gris.

Structures de données utilisées :

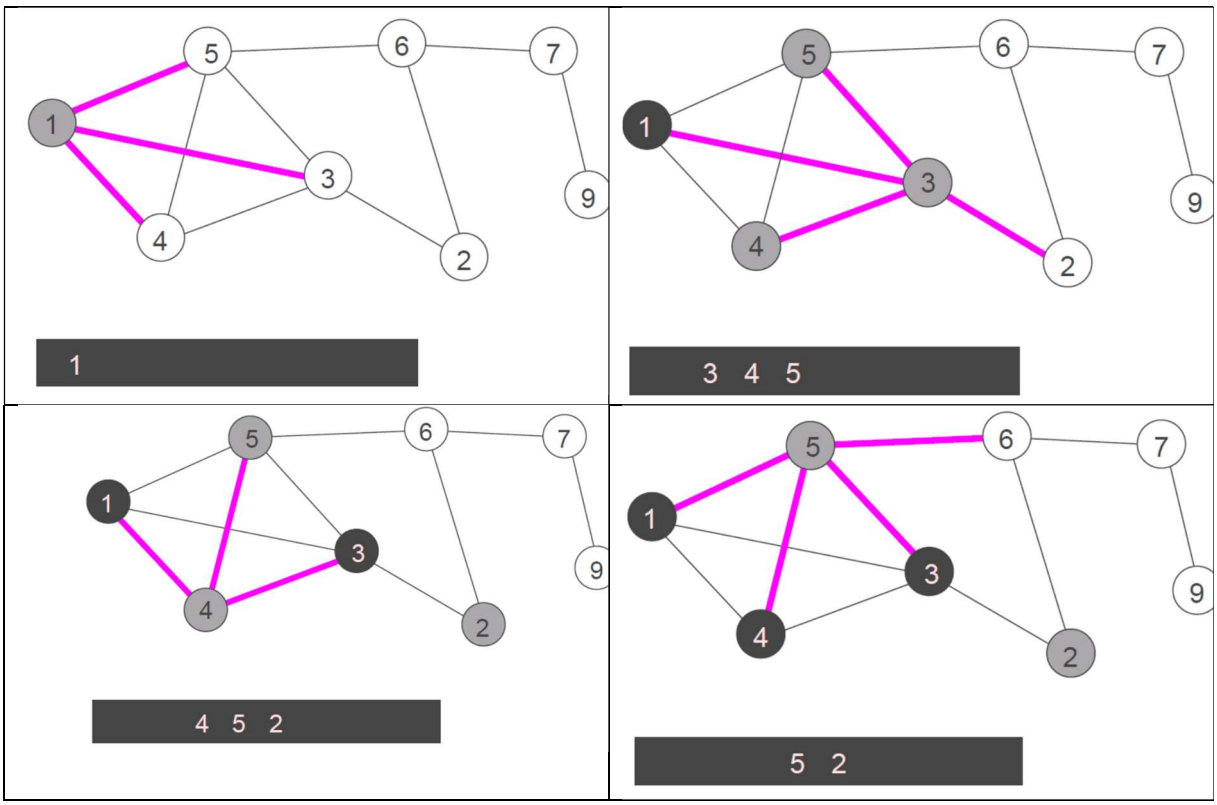
– On utilise une file F, pour laquelle on suppose définies les opérations init_file(F) qui initialise la file F à vide, ajoute_fin_file(F,s) qui ajoute le sommet s à la fin de la file F, est_vide(F) qui retourne vrai si la file F est vide et faux sinon, et enleve_debut_file(F,s) qui enlève le sommet s au début de la file F.

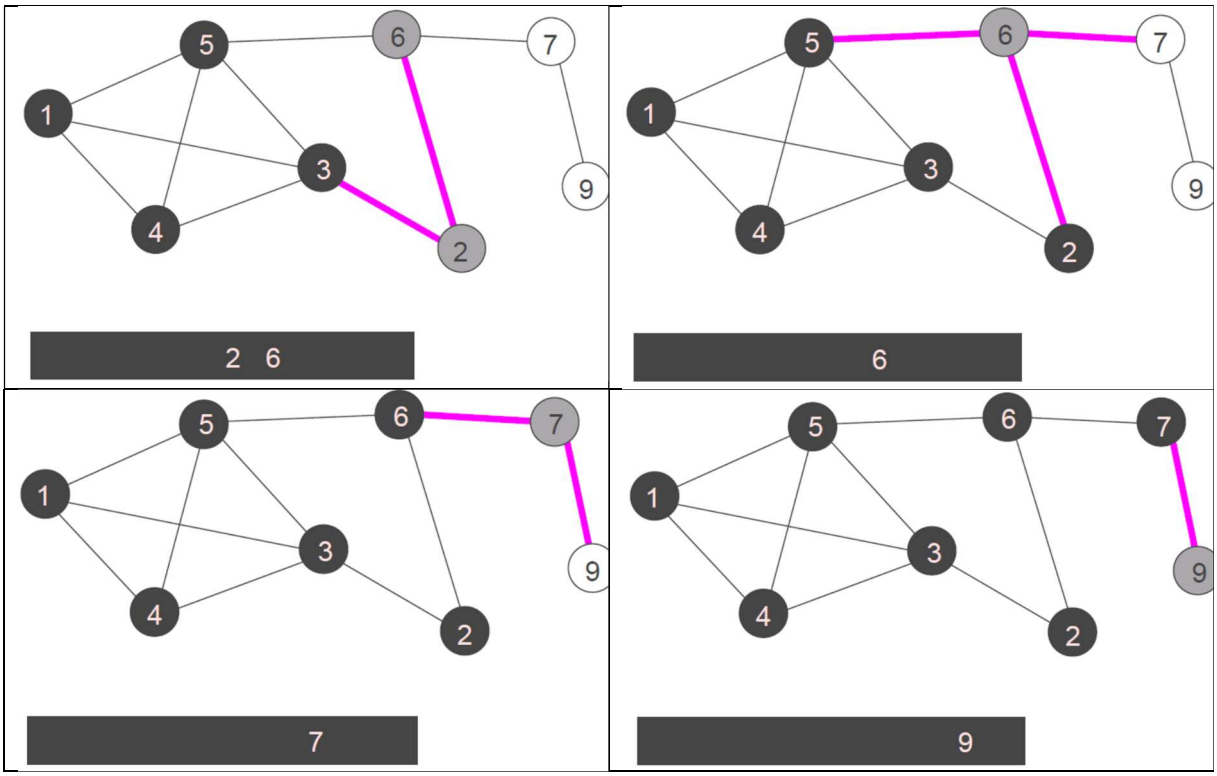
– On utilise un tableau π qui associe à chaque sommet le sommet qui l'a fait entrer dans la file, et un tableau couleur qui associe à chaque sommet sa couleur (blanc, gris ou noir).

– On va en plus utiliser un tableau d qui associe à chaque sommet son niveau de profondeur par rapport au sommet de départ s_0 (autrement dit, $d[s_i]$ est la longueur du chemin dans l'arborescence π de la racine s_0 jusque s_i). Ce tableau sera utilisé plus tard.

Algorithme de parcours en largeur (BFS)(S, A, s_0)
init_file(F)
pour tout sommet $s_i \in S$ faire
    $\pi[s_i] \leftarrow$ nil
    $d[s_i] \leftarrow 1$
    couleur$[s_i] \leftarrow$ blanc
fin pour
$d[s_0] \leftarrow 0$
ajoute_fin_file(F, s_0)
couleur$[s_0] \leftarrow$ gris
tant que est_vide(F) = faux faire
    enleve_debut_file(F, s_i)
    pour tout $s_j \in$ succ(s_i) faire
        si couleur$[s_j]$ = blanc alors
            ajoute_fin_file(F, s_j)
            couleur$[s_j] \leftarrow$ gris
            $\pi[s_j] \leftarrow s_i$
            $d[s_j] \leftarrow d[s_i] + 1$
        finsi
    fin pour
    couleur$[s_i]$ noir
fin tant
fin BFS

Exemple d'application de l'algorithme sur un graphe non orienté. L'application de l'algorithme pour un graphe orienté est la même, il faut tout simplement appliquer la notion de successeur c.à.d. suivre l'orientation des arecs.





6.3. Parcours en profondeur (Depth First Search = DFS)

Le parcours en profondeur est obtenu en gérant la liste d'attente au coloriage en noir comme une pile (LIFO = Last In First Out). Autrement dit, on considère à chaque fois le dernier sommet gris entré dans la pile, et on introduit devant lui tous ses successeurs blancs.

Structures de données utilisées :

– On utilise une pile P, pour laquelle on suppose définies les opérations init_pile(P) qui initialise la pile P à vide, empile(P,s) qui ajoute s au sommet de la pile P, est_vide(P) qui retourne vrai si la pile P est vide et faux sinon, sommet(P) qui retourne le sommet s au sommet de la pile P, et depile(P,s) qui enlève s du sommet de la pile P.

– On utilise, comme pour le parcours en largeur, un tableau π qui associe à chaque sommet le sommet qui l'a fait entrer dans la pile, et un tableau couleur qui associe à chaque sommet sa couleur (blanc, gris ou noir).

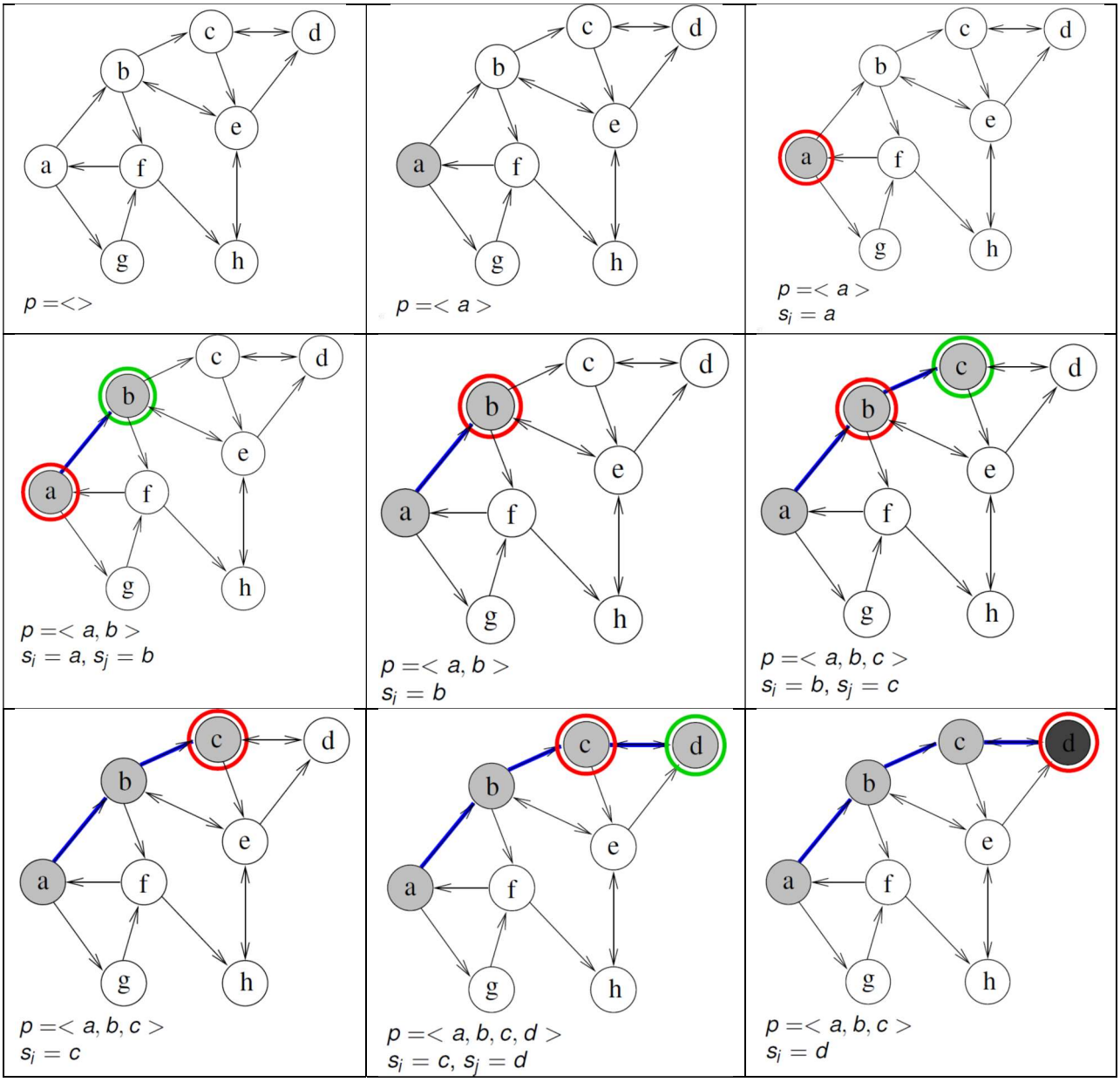
– On va en plus mémoriser pour chaque sommet s_i :

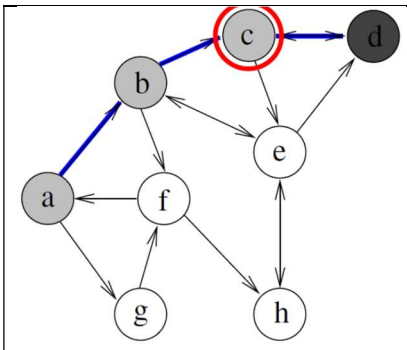
- dec[s_i] = date de découverte de s_i (passage en gris)
- fin[s_i] = date de fin de traitement de s_i (passage en noir) où l'unité de temps est une itération. La date courante est mémorisée dans la variable tps.

Algorithme de parcours en profondeur des sommets accessibles depuis s_0DFSinit_pile(P)
pour tout sommet $s_i \in S$ faire
    $\pi[s_i] \leftarrow$ nil
    couleur[s_i] ←blanc
fin pour
tps ← 0
dec[s_0] ← tps
empile(P, s_0)
couleur[s_0] ← gris
tant que est_vide(P) = faux faire
    tps ← tps + 1
    s_i ← sommet(P)

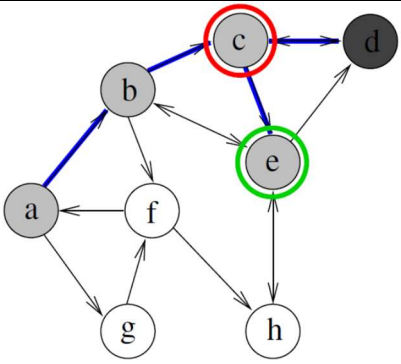
si $\exists s_j \in succ(s_i)$ tel que couleur[s_j]=blanc alors
empile(P, s_j)
couleur[s_j] ← gris
π[s_j] ← s_i
dec[s_j] ← tps
sinon /* tous les successeurs de s_i sont gris ou noirs */
depile(P, s_i)
couleur[s_i] ← noir
fin[s_i] ← tps
finsi
fin tant
fin DFS

EXEMPLE

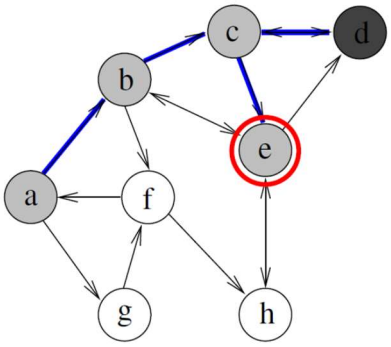




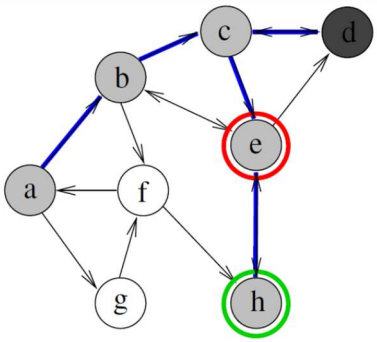
$p =< a, b, c >$
$s_i = c$



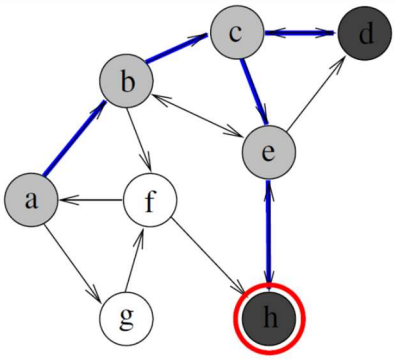
$p =< a, b, c, e >$
$s_i = c, s_j = e$



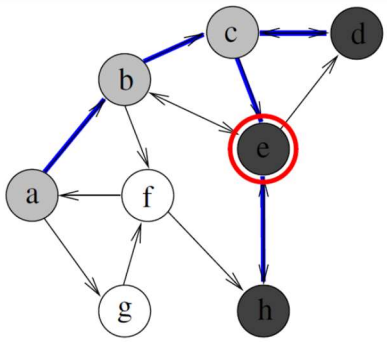
$p =< a, b, c, e >$
$s_i = e$



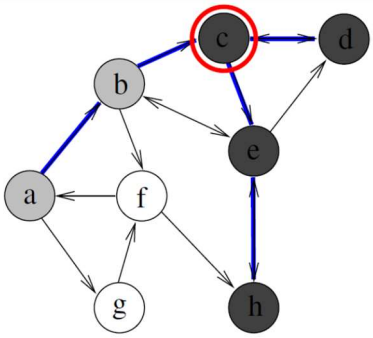
$p =< a, b, c, e, h >$
$s_i = e, s_j = h$



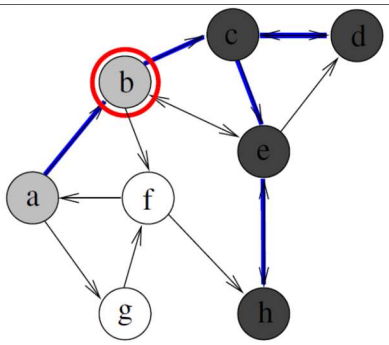
$p =< a, b, c, e >$
$s_i = h$



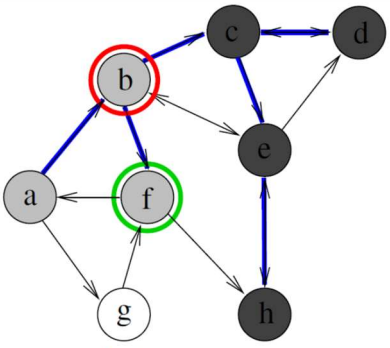
$p =< a, b, c >$
$s_i = e$



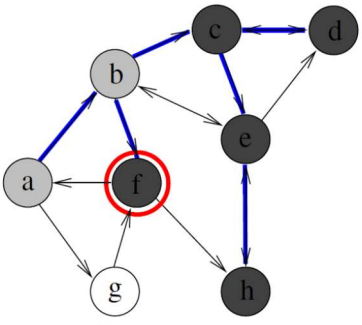
$p =< a, b >$
$s_i = c$



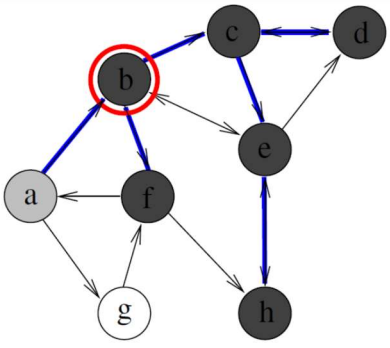
$p =< a, b >$
$s_i = b$



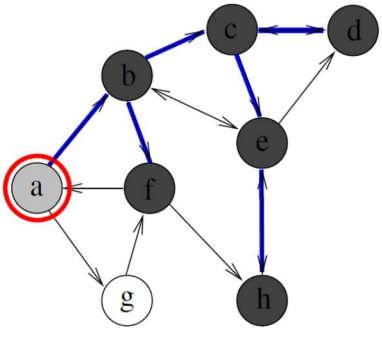
$p =< a, b, f >$
$s_i = b, s_j = f$



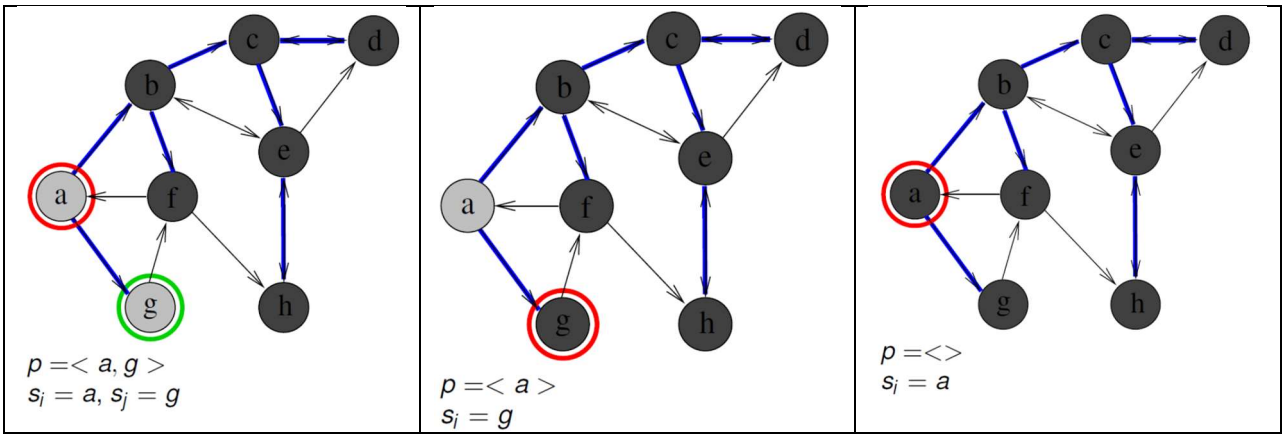
$p =< a, b >$
$s_i = f$



$p =< a >$
$s_i = b$



$p =< a >$
$s_i = a$



7. Plus courts chemins

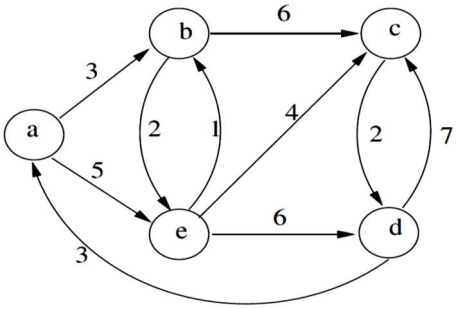
Soit G = (S,A) un 1-graphe orienté valué tel que la fonction cout : A →R associe à chaque arc (s_i, s_j) de A un coût réel cout(s_i, s_j).

Le coût d'un chemin p =< s_0, s_1, s_2, : : : , s_k > est égal à la somme des coûts des arcs composant le chemin, c'est à dire,

$$cout\ (p) = \sum_{i=1}^{k} cout\ (s_{i-1}, s_i)$$

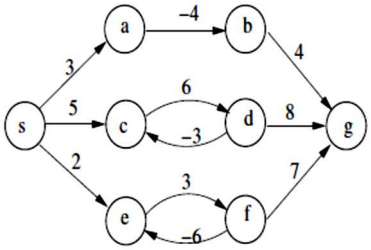
Le coût d'un chemin sera aussi appelé poid du chemin.
Le coût (ou poid) d'un plus court chemin entre deux sommets s_i et s_j est noté $\delta(s_i, s_j)$ et est défini par
$\delta(s_i, s_j) = +\infty$, s'il n'existe pas de chemin entre s_i et s_j
$\delta(s_i, s_j)$= min{cout(p)/p = chemin de s_i à s_j , s'il existe un chemin entre s_i et s_j

Exemple



Conditions d'existence d'un plus court chemin : s'il existe un chemin entre deux sommets u et v contenant un circuit de coût négatif, alors $\delta(u, v) = -\infty$, et il n'existe pas de plus court chemin entre u et v. Un circuit négatif est appelé un circuit absorbant.

Exemple

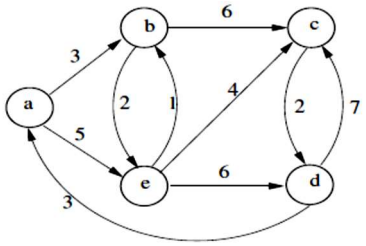


Le chemin < s, e, f, e, f, g > contient le circuit < e, f, e > de coût négatif -3. Autrement dit, à chaque fois que l'on passe dans ce circuit, on diminue de 3 le coût total du chemin. Par conséquent, $\delta(s, g) = -\infty$ et il n'existe pas de plus court chemin entre s et g.

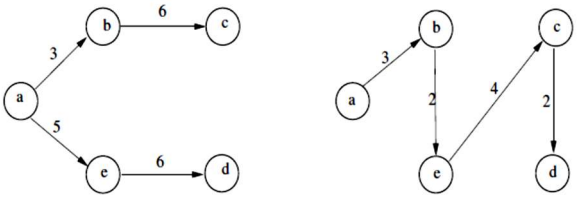
Définition du problème des plus courts chemins à origine unique : Etant donné un graphe orienté G = (S,A), une fonction cout : A → R et un sommet origine $s_0 \in S$, on souhaite calculer pour chaque sommet $s_j \in S$ le coût $\delta(s_0, s_j)$ du plus court chemin de s_0 à s_j. On supposera que le graphe G ne comporte pas de circuit absorbant.

Arborescence des plus courts chemins : l'arborescences couvrante calculée lors d'un parcours d'un graphe est mémorisée dans un tableau π tel que $\pi[s_0] = nil$ et $\pi[s_j] = s_i$ si $s_i \rightarrow s_j$ est un arc de l'arborescence. Pour connaitre le plus court chemin entre s_0 et un sommet s_k donné, il faudra alors "remonter" de s_k jusque s_0 en utilisant π.

Remarque : $\delta(s_0, s_i) = \delta(s_0, \pi[s_i]) + cout(\pi[s_i], s_i)$.
Considérons par exemple le graphe valué orienté suivant :



Ce graphe possède plusieurs arborescences des plus courts chemins dont l'origine est a, par **Exemple**



La première de ces 2 arborescences est représentée par le tableau π tel que $\pi[a] = nil$, $\pi[b] = a$, $\pi[c] = b$, $\pi[d] = e$ et $\pi[e] = a$.

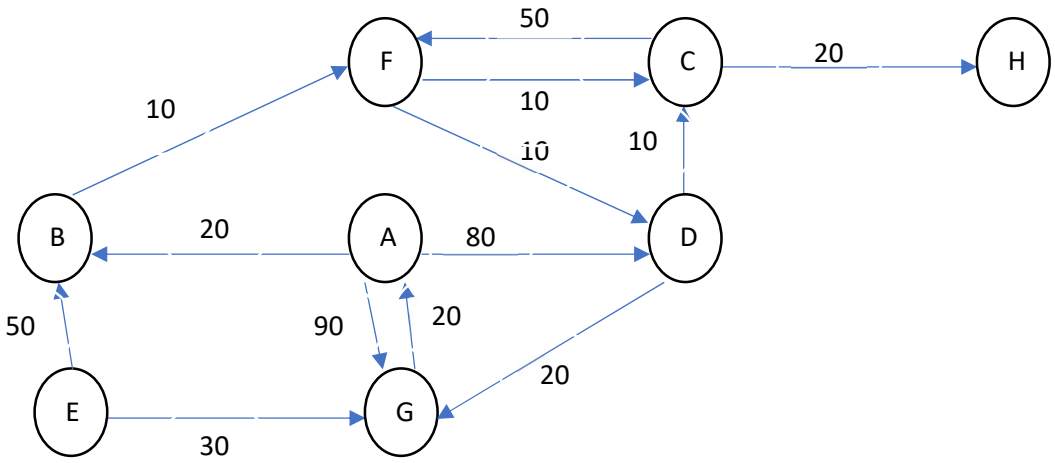
7.1. Algorithme de Dijkstra

On maintient 2 ensembles disjoints E et F tels que $E \cup F = S$. L'ensemble E contient chaque sommet s_i pour lequel on connait un plus court chemin depuis s_0 (c'est-à-dire pour lequel $d[s_i] = \delta(s_0, s_i)$). L'ensemble F contient tous les autres sommets. A chaque itération de l'algorithme, on choisit le sommet s_i dans F pour lequel la valeur $d[s_i]$ est minimale, on le rajoute dans E, et on relâche tous les arcs partant de ce sommet s_i.

fonc Dijkstra(G = (S,A), cout : A→R^+, $s_0 \in S$)
retourne une arborescence des plus courts chemins d'origine s
    pour chaque sommet $s_i \in S$ faire
        $d[s_i] \leftarrow +\infty$
        $\pi[s_i] \leftarrow$ nil
    fin pour
    $d[s_0] \leftarrow 0$
    $E \leftarrow \emptyset$
    $F \leftarrow S$
    tant que $F \neq \emptyset$, faire
        soit s_i le sommet de F tel que $d[s_i]$ soit minimal
        /* $d[s_i] = \delta(s_0, s_i)$ */
        $F \leftarrow F - \{s_i\}$
        $E \leftarrow E \cup \{s_i\}$

pour tout sommet $s_j \in succ(s_i)$ faire : relacher(s_i, s_j)
fin tant
retourner(π)
fin Dijkstra

Exemple : les plus courts chemins à partir de A



Itération		s_i	E	B	C	D	E	F	G	H
1	d	A	∅	20	∞	80	∞	∞	90	∞
	pred			A		A			A	
1	d	B	{A}	20	∞	80	∞	30	90	∞
	pred			A		A		B	A	
2	d	F	{A,B}	20	40	70	∞	30	90	∞
	pred			A	F	F		B	A	
3	d	C	{A,B,F}	20	40	50	∞	30	90	60
	pred			A	F	C		B	A	C
4	d	D	{A,B,F,C}	20	40	50	∞	30	70	60
	pred			A	F	C		B	D	C
4	d	H	{A,B,F,C,H}	20	40	50	∞	30	70	60
	pred			A	F	C		B	D	C
4	d	G	{A,B,F,C,H,G}	20	40	50	∞	30	70	60
	pred			A	F	C		B	D	C

7.2. Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford permet de trouver les plus courts chemins à origine unique dans le cas où le graphe contient des arcs dont le coût est négatif, sous réserve que le graphe ne contienne pas de circuit absorbant (dans ce cas, l'algorithme de Bellman-Ford va détecter l'existence de circuits absorbants).

Principe : on associe à chaque sommet s_i une valeur d[s_i] qui représente une borne maximale du coût du plus court chemin entre s_0 et s_i. L'algorithme diminue alors progressivement les valeurs d[s_i] en relâchant les arcs. Chaque arc va être relâché plusieurs fois : on relâche une première fois tous les arcs, après quoi, tous les plus courts chemins de longueur 1, partant de s_0, auront été trouvés. On relâche alors une deuxième fois tous les arcs, après quoi tous les plus courts chemins de longueur 2, partant de s_0, auront été trouvés... et ainsi de suite... Après la $k^{ième}$ série de relâchement des arcs, tous les plus courts chemins de longueur k, partant de s_0, auront été trouvés. Si le graphe contient un circuit absorbant, au bout de n - 1 passages, on aura encore au moins un arc (s_i, s_j) pour lequel un relâchement

permettrait de diminuer la valeur de d[s_j]. L'algorithme utilise cette propriété pour détecter la présence de circuits absorbants.

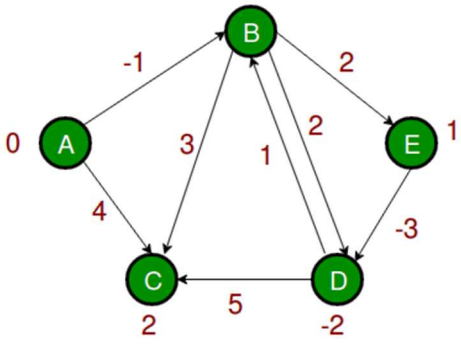
fonc Bellman-Ford(G = (S,A), cout : S → R, $s_0 \in S$)
retourne une arborescence des plus courts chemins d'origine s_0

```
pour chaque sommet s_i ∈ S faire
        d[s_i] ← +∞
        π[s_i] ← nil
fin pour
d[s_0] ← 0
pour k variant de 1 à |S|-1 faire
        pour chaque arc (s_i, s_j) ∈ A faire relacher(s_i, s_j) fin pour
fin pour
pour chaque arc (s_i, s_j) ∈ A faire
        si d[s_j] > d[s_i] + cout(s_i, s_j) alors afficher("circuit absorbant") finsi
fin pour
retourner(_)
```

fin Bellman-ford

Exemple :

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



8. Arbres couvrants minimaux (ACM)

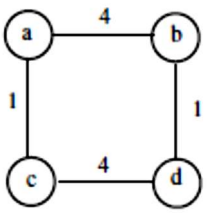
8.1. Arbre couvrant minimal

Pour définir le réseau câblé de telle sorte que la longueur totale de câble soit minimale et qu'un certain nombre de lieux soient desservis. On peut modéliser ce problème de câblage à l'aide d'un graphe non orienté connexe G = (S,A), où S associe un sommet à chaque lieu devant être desservi, et A contient une arête pour chaque portion de route entre 2 lieux. Ce graphe est valué par une fonction coût qui spécifie pour chaque (s_i, s_j) la longueur de câble nécessaire pour connecter s_i à s_j. Il s'agit alors de trouver un arbre qui recouvre l'ensemble des sommets du graphe. Ce graphe est appelé arbre couvrant. On cherche à minimiser le poids total des arêtes de l'arbre. On dira qu'on cherche l'arbre couvrant minimal, abrégé par ACM.

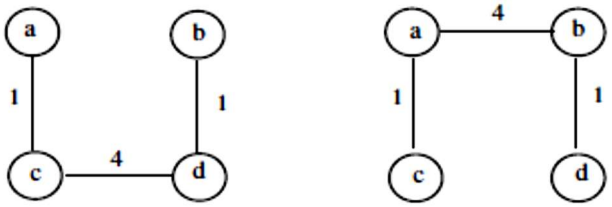
De façon plus formelle, un ACM d'un graphe G = (S,A) est un graphe partiel G_0 = (S,A_0) de G tel que G_0 est connexe et sans cycle (G_0 est un arbre), et la somme des coûts des arêtes de A_0 est minimale.

Remarque : il peut exister plusieurs ACM, de même coût, associés à un même graphe.

Par exemple, le graphe

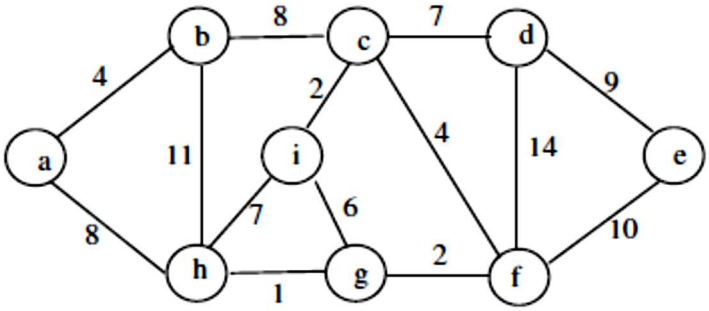


possède les 2 ACMs suivants :



8.2. Principe d'algorithme de Kruskal

Principe de l'algorithme de Kruskal : On commence par trier l'ensemble des arêtes du graphe par ordre de coût croissant. On va sélectionner de proche en proche les arêtes devant faire partie de l'ACM. Au début, cet ensemble est vide. On considère ensuite chacune des arêtes du graphe selon l'ordre que l'on vient d'établir (de l'arête de plus faible coût jusqu'à l'arête de plus fort coût). A chaque fois, si l'arête que l'on est en train de considérer peut-être ajoutée à l'ensemble des arêtes déjà sélectionnées pour l'ACM sans générer de cycle, alors on la sélectionne, sinon on l'abandonne.
Considérons par exemple le graphe suivant :



On trie les arêtes du graphe. On obtient l'ordre suivant :
hg < ic = gf < ab = cf < ig < hi = cd < bc = ah < de < fe < bh < df
On ajoute alors successivement dans l'ACM les arêtes : hg, ic, gf, ab, cf, cd, bc, de

8.3. Algorithme de Kruskal

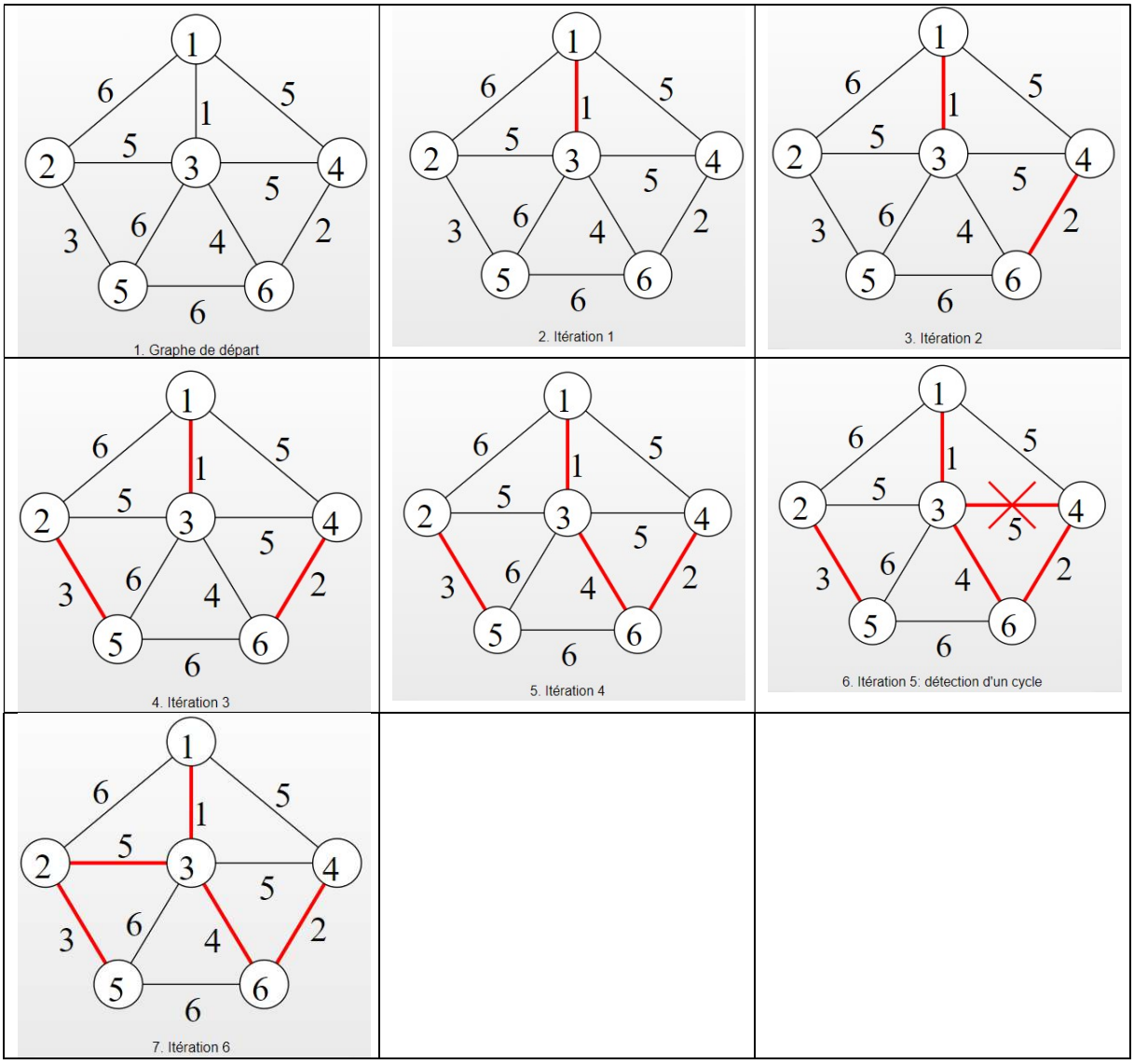
fonc Kruskal(G = (S,A), cout : A → R)
retourne un ACM G_0 = (S,K)

pour chaque sommet $s_i \in S$ faire $\pi[s_i] \leftarrow$ nil
trier les arêtes de A par ordre de coût croissant
K←∅ ,
tant que |K| < |S| -1 faire
soit (s_i, s_j) la k^{ieme} plus petite arête de A
/* recherche de la racine r_i de la composante connexe de s_i */
$r_i \leftarrow s_i$
tant que $\pi[r_i] \neq$ nil faire $r_i \leftarrow \pi[r_i]$

/* recherche de la racine r_j de la composante connexe de s_j */
$r_j \leftarrow s_j$
tant que $\pi[r_j] \neq$ nil faire $r_j \leftarrow \pi[r_j]$
si $r_i \neq r_j$ alors
/* on ajoute (s_i, s_j) à l'ACM */
$K \leftarrow K \cup \{(s_i, s_j)\}$
/* on fusionne les deux composantes connexes */
$\pi[r_i] \leftarrow r_j$
fin si
fin pour
retourne(S,K)
fin Kruskal

Exemple

L'exemple détaille les différentes itérations de l'algorithme



9. Réseaux de transport

9.1. Définition

Un réseau de transport désigne le fait qu'un "matériau" (de l'eau, de l'électricité, de l'information, ...) doit s'écouler depuis une **source**, où il est produit, jusqu'à un **puits**, où il est consommé. Un réseau de transport sera défini par un quadruplet (G, c, s, p) tel que
- G = (S,A) est un graphe orienté,
- c : A → R^+ est une fonction qui associe à chaque arc sa capacité,
- s ∈ S est la source, et p ∈ S est le puits.

9.2. Définition du problème du flot maximal

: Etant donné un réseau de transport (G, c, s, p), il s'agit de trouver un flot f tel que |f| soit maximal.
Modélisation en programmation linéaire : Le problème du flot maximal peut être exprimé comme un problème de programmation linéaire, c'est-à-dire comme une fonction linéaire à maximiser tout en respectant un certain nombre de contraintes linéaires. Etant donné le réseau de transport (G = (S,A), c, s, p), il s'agit de résoudre le problème linéaire suivant :
Maximiser $\sum_{s_i \in S} f(s, s_i)$
Telque $f(s_i, s_j) \leq c(s_i, s_j) \quad \forall (s_i, s_j) \in A$
$f(s_i, s_j) = -f(s_j, s_i) \quad \forall (s_i, s_j) \in S^2$
$\sum_{s_j \in S} f(s_i, s_j) = 0 \quad \forall s_i \in S$

9.3. Principe de l'algorithme de Ford-Fulkerson

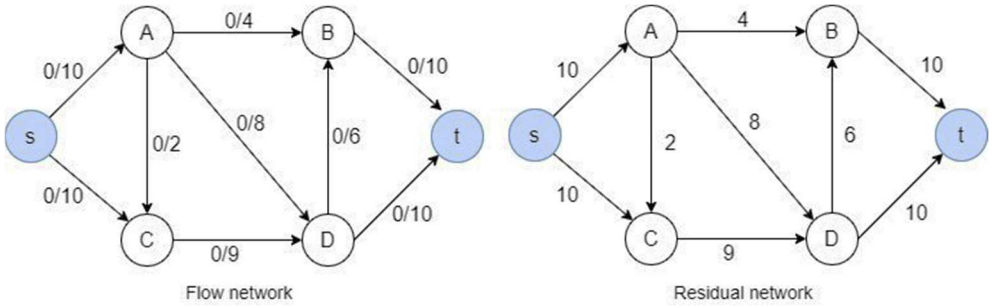
– Au départ, le flot est nul, c'est-à-dire que f(s_i, s_j) = 0 pour tout couple de sommets (s_i, s_j)∈S^2.
– On augmente ensuite itérativement le flot f en cherchant à chaque fois un "chemin améliorant", c'est-à-dire un chemin allant de la source s jusqu'au puits p et ne passant que par des arcs dont le flot actuel est inférieur à la capacité. Pour cela, à chaque itération, on calcule la "capacité résiduelle" de chaque arc, c'est-à-dire la quantité de flot pouvant encore passer.

9.4. Algorithme de Ford-Fulkerson

```
fonc Ford-Fulkerson(G = (S,A), c : S × S → R+, s ∈ S, p ∈ S)
retourne un flot maximal
    pour chaque (si, sj) ∈ S² faire
        f(si, sj)←0
        cf (si, sj)←c(si, sj)
    fin pour
    tant que il existe un chemin améliorant ch dans le graphe résiduel faire
        /* calcul de la capacité résiduelle du chemin améliorant */
        cf (ch)←min(si, sj)∈ch(cf(si, sj))
        /* mise à jour du flot et de la capacité résiduelle le long des arcs de ch */
        pour tout arc (si, sj) du chemin améliorant ch faire
            f(si, sj) ← f(si, sj) + cf (ch)
            f(sj, si) ← f(sj, si) - cf (ch)
            cf (si, sj) ← cf(si, sj) - cf (ch)
            cf (sj, si) ← cf (sj, si) + cf (ch)
        fin pour
    fin tant
    retourne(f)
fin Ford-Fulkerson
```

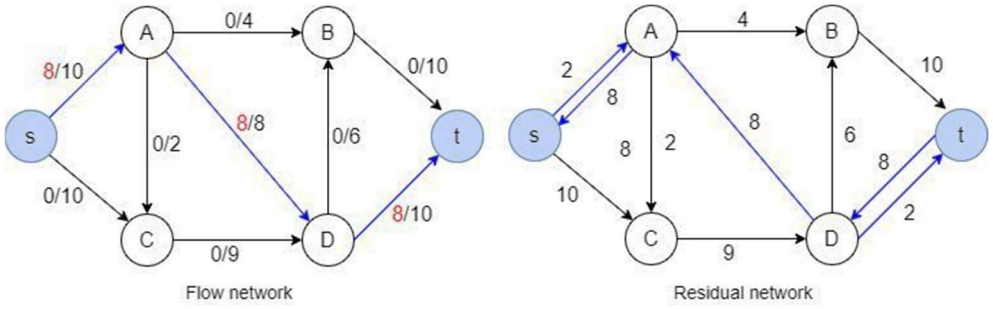
Exemple. Déterminer le flot maximal à partir du graphe de flot ci-dessous

Étape 1: mettre le flux de chaque arc sur 0

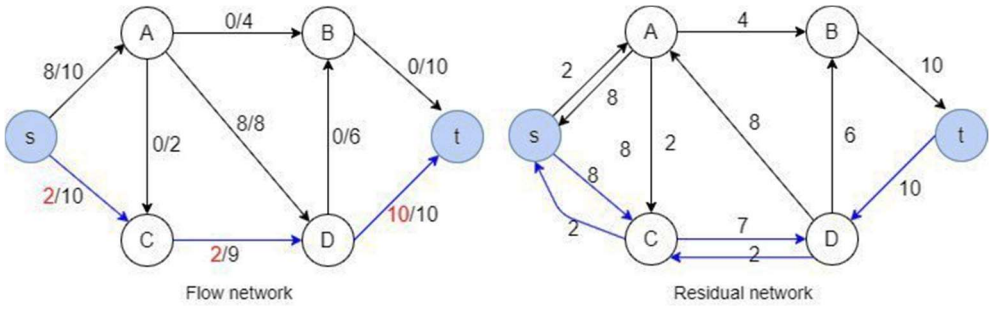


Maintenant, trouvez un chemin améliorant dans le réseau résiduel. Ici, on sélectionne le chemin

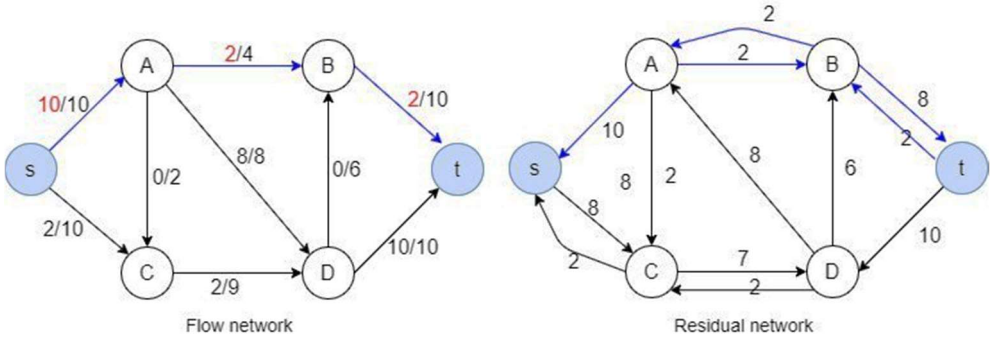
s -> A -> D -> t. mettez à jour les valeurs de flux des arêtes



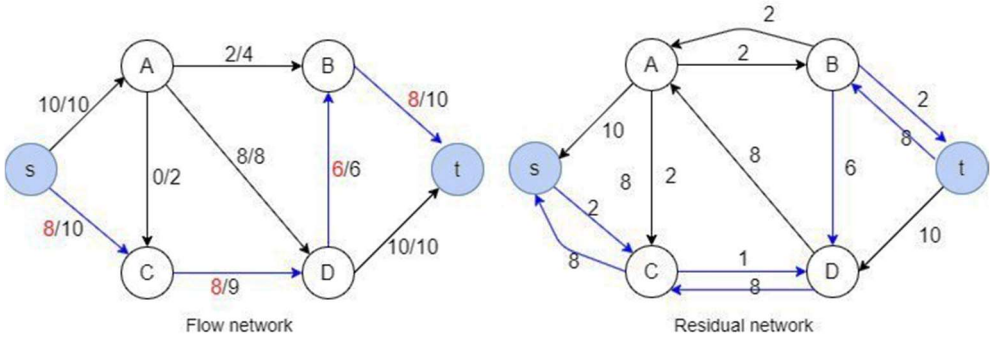
sélectionnez le chemin améliorant s -> C -> D -> t. Désormais, la capacité résiduelle est de 2.



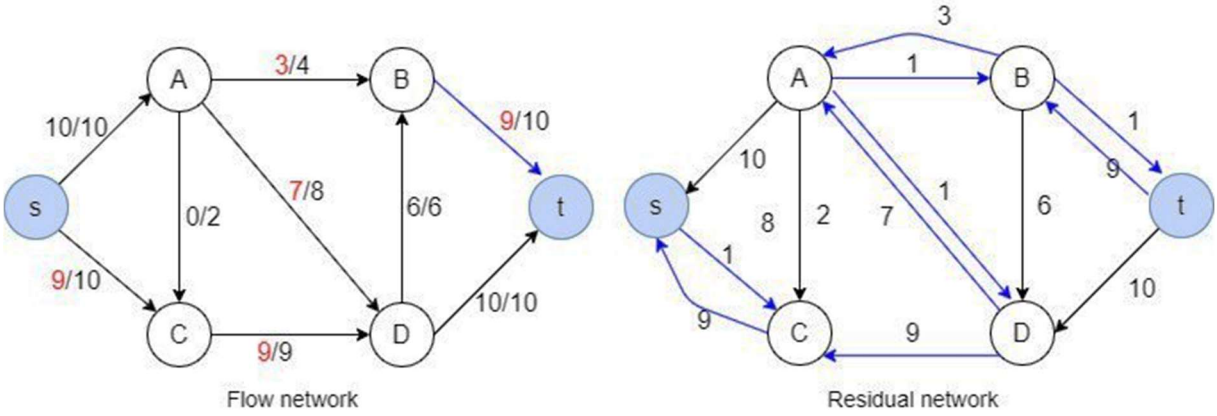
sélectionnez le chemin améliorant s -> A -> B->t, Désormais, la capacité résiduelle est de 2.



sélectionnez le chemin améliorant s -> C -> D -> B->t. Désormais, la capacité résiduelle est de 6.



sélectionnez le chemin améliorant s -> C -> D -> A-> B->t. Désormais, la capacité résiduelle est de 1.



Il n'existe un chemin améliorant ch dans le graphe résiduel. Cela signifie que la méthode Ford Fulkerson est terminée. Le débit maximum étant égal au débit sortant de la source, dans cet exemple, le débit maximum est de 10 + 9 = 19.

Références

Théorie des graphes et optimisation dans les graphes, Christine Solnon

Cours de Théorie des graphes, Kalla Salim

Network Flow Problems, Jaehyun Park

À la découverte des graphes et des algorithmes de graphes, Christian Laforest