Initiation
à la
théorie des graphes
2

Circuits Hamiltoniens

Un chemin Hamiltonien est un chemin qui contient chaque sommet exactement une fois, sauf que le premier et le dernier sommet peuvent être identiques. **Un circuit (cycle) Hamiltonien est un chemin Hamiltonien qui est un cycle.**

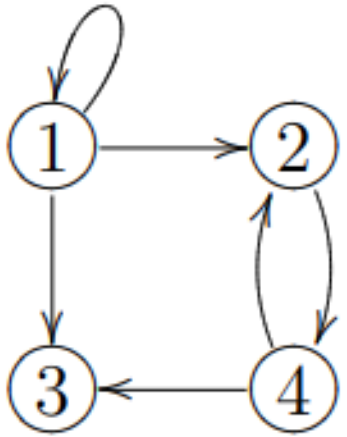
Le graphe de gauche ci-dessous n'a pas de circuit Hamiltonien, mais contient le chemin Hamiltonien <1, 2, 4, 3>.

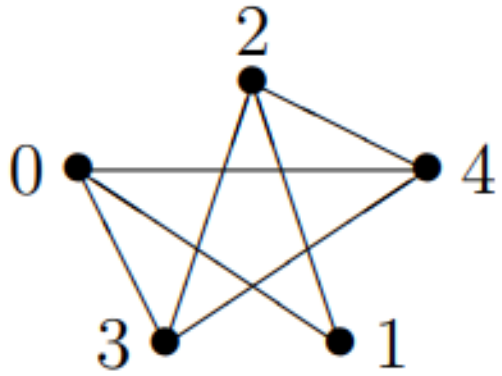
Le graphe de droite contient les circuits Hamiltoniens <0, 1, 2, 3, 4, 0> et <0, 1, 2, 4, 3, 0>

Rappel

Une chaîne eulérienne est formée de toutes les arêtes d'un graphe, chacune des arêtes n'apparaissant qu'une seule fois

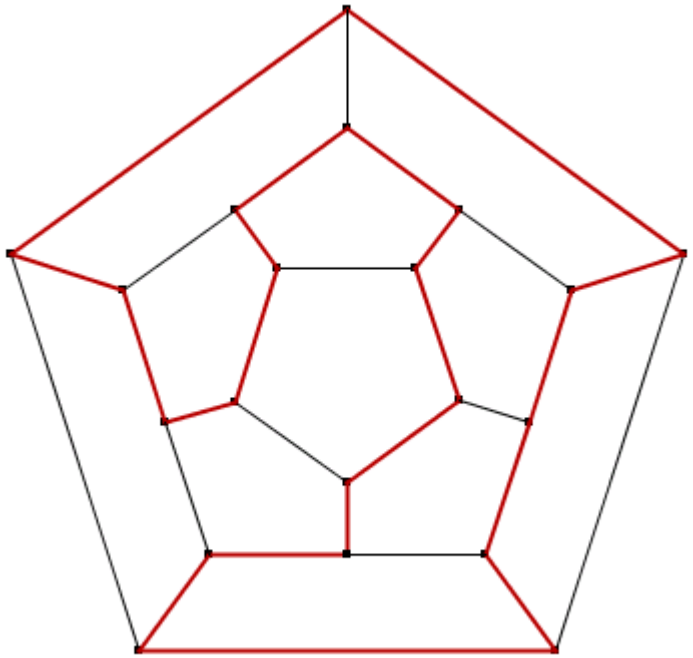
Un cycle eulérien est un cycle formé de toutes les arêtes d'un graphe, chacune des arêtes n'apparaissant qu'une seule fois.





Cycle hamiltonien

Voici un autre graphe hamiltonien sur lequel est dessiné en rouge un cycle hamiltonien :



Le cycle hamiltonien est donc l'analogue du cycle eulérien, où on demande à un cycle de passer une, et une seule fois, par chaque arête.

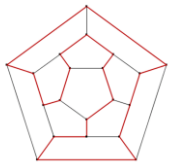
Existence d’un cycle Hamiltonien

Étonnamment, alors qu'on connait une caractérisation simple des graphes possédant un cycle eulérien, comme nous l’avons vu, c’est le théorème d'Euler, ce n'est pas le cas pour les cycles hamiltoniens!

Théorème d'Euler

- Un graphe connexe admet une chaîne eulérienne si et seulement si il ne possède aucun, ou exactement deux sommets de degré impair.
- Un graphe connexe admet un cycle eulérien si et seulement si il ne possède que des sommets de degré pair.

Le nom de cycle hamiltonien vient d'un puzzle inventé par Sir William Rowan Hamilton (l'inventeur des quaternions) en 1859. Ce jeu consistait en un dodécaèdre dont les 20 sommets étaient étiquetés par le nom d'une ville. Le but du jeu était de trouver un chemin passant une seule fois par chaque ville et revenant à la ville de départ. La version "dans le plan" de ce jeu correspond au dessin précédent.

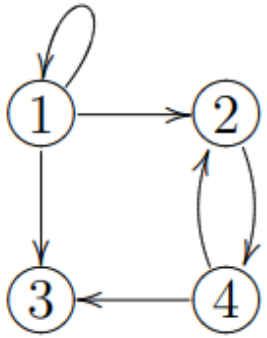


Opérations sur les graphes et les matrices

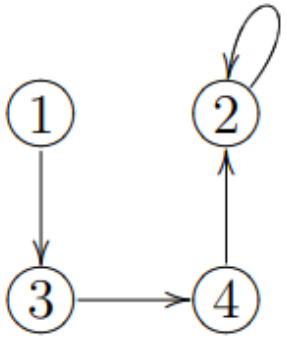
Nous allons définir cinq opérations sur les graphes et sur les matrices d'adjacence de ces graphes. Ce sont des opérations qui s'appliquent aussi aux relations.

Soient donc les graphes orientés G = <S, A> , G1 = <S, A1> et G2 = <S, A2> .
Pour illustrer les opérations, nous utiliserons les deux graphes qui suivent :

G_1 :



G_2 :



$$M_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} \qquad M_2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Union

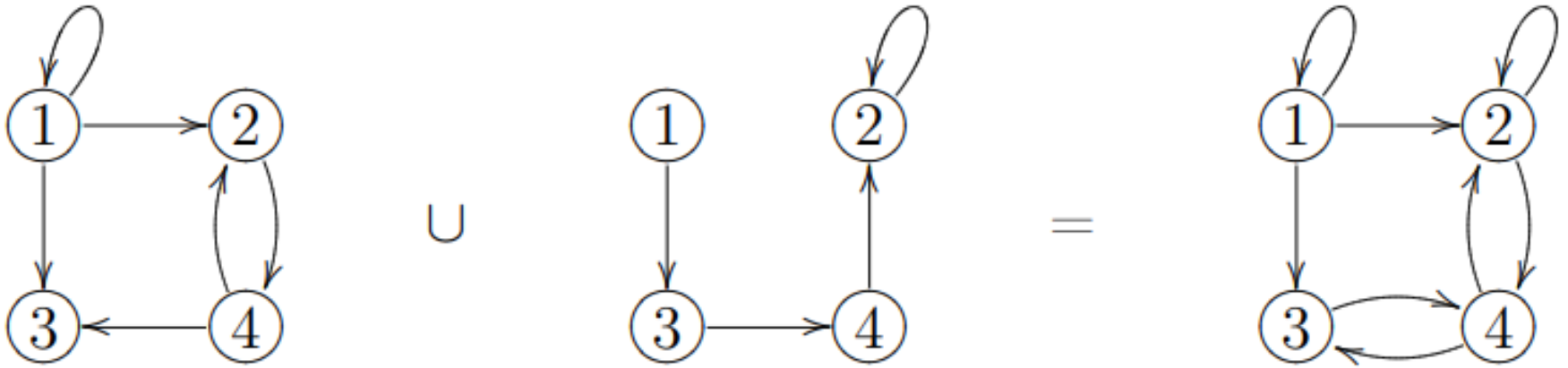
$G1 \cup G2 = \langle S, A1 \rangle \cup \langle S, A2 \rangle = \langle S, A1 \cup A2 \rangle$

$M1 \cup M2$ est définie par:

$(M1 \cup M2)[i, j] = M1[i, j] \vee M2[i, j]$

C’est-à-dire que l’union des matrices est obtenue en faisant la disjonction (somme logique ou opération logique « ou ») des entrées correspondantes.

Union



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

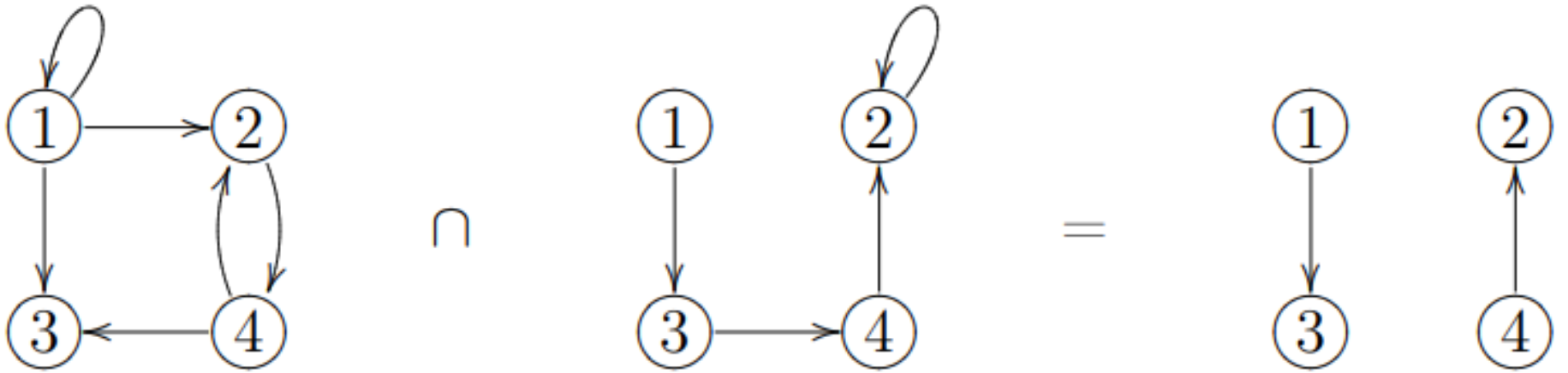
Intersection

$G1 \cap G2 = \langle S, A1 \rangle \cap \langle S, A2 \rangle = \langle S, A1 \cap A2 \rangle$

$M1 \cap M2$ est définie par $(M1 \cap M2)[i, j] = M1[i, j] \wedge M2[i, j]$

C’est-à-dire que l’union des matrices est obtenue en faisant la conjonction (produit logique ou opération logique « et ») des entrées correspondantes.

Intersection



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

Complément

$\sim G = \sim\langle S, A\rangle = \langle S, \sim A\rangle$

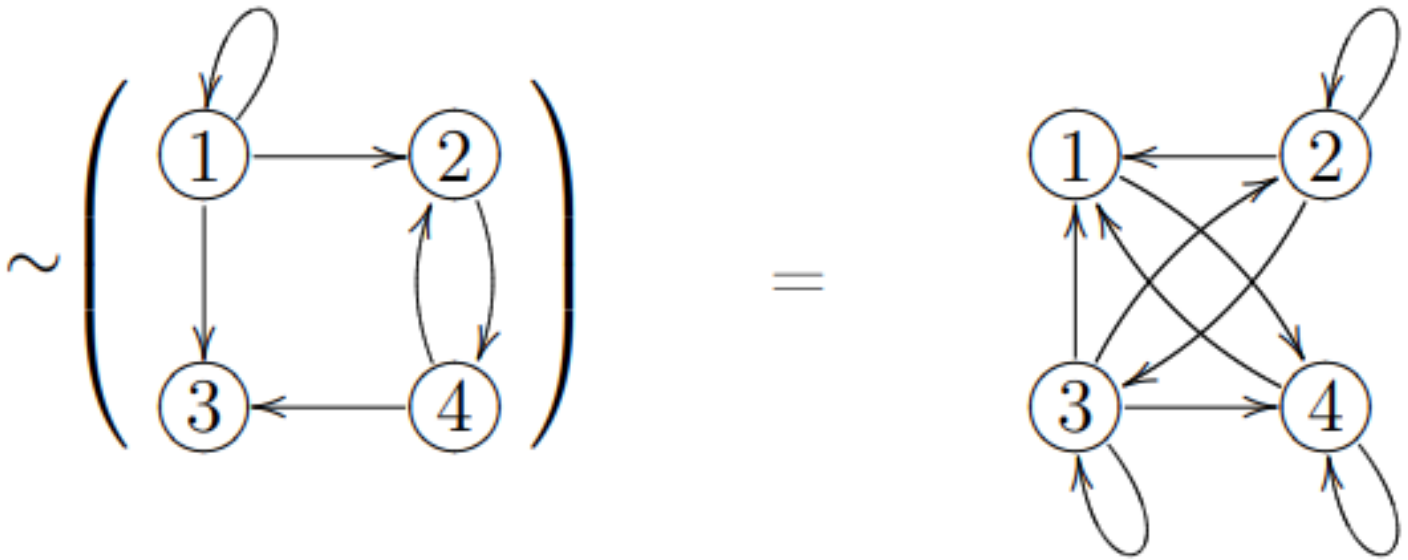
$\sim M$ est défini par:

$(\sim M)[i, j] = \neg(M[i, j])$

C’est-à-dire que le complément d’une matrice est obtenu en faisant la négation (opération logique « non » ou inversion) des entrées.

Complément

« On supprime ce qui existait et on rajoute ce qui n'existait pas! »



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

Inverse

$G^{-1} = \langle S, A \rangle^{-1} = \langle S, A^{-1} \rangle$

M^{-1} est défini par:

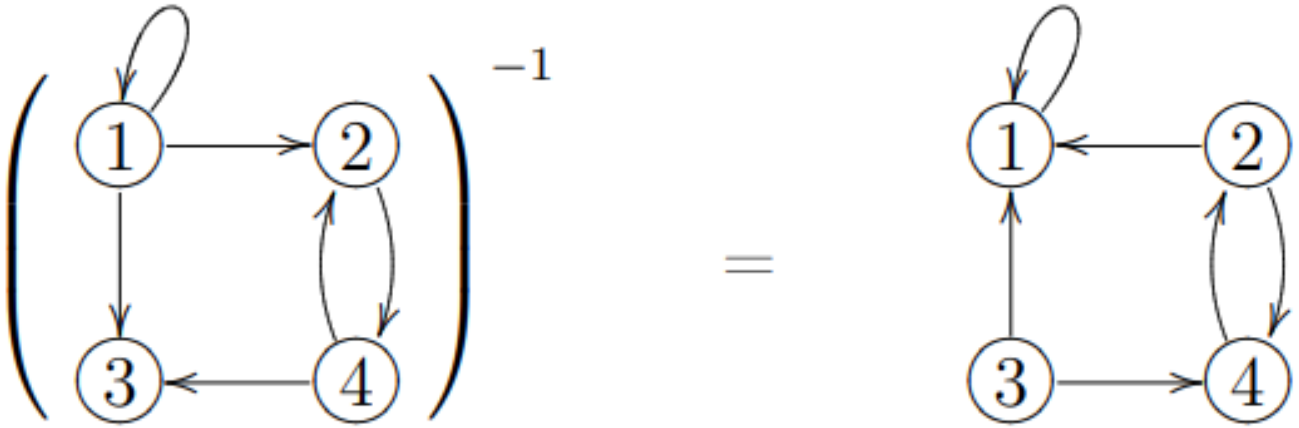
$(M^{-1})[i, j] = M[j,i]$

C'est-à-dire que l'inverse d'une matrice d'adjacence est obtenu en **transposant** la matrice.

(Il s'agit bien d'une transposition de la matrice même si le signe utilisé est $^{-1}$!!!)

Inverse

On inverse le sens des flèches!



(T)

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Attention, c'est en fait une transposition matricielle!

Produit

$G1 \circ G2 = \langle S, A1 \rangle \circ \langle S, A2 \rangle = \langle S, A1 \circ A2 \rangle$

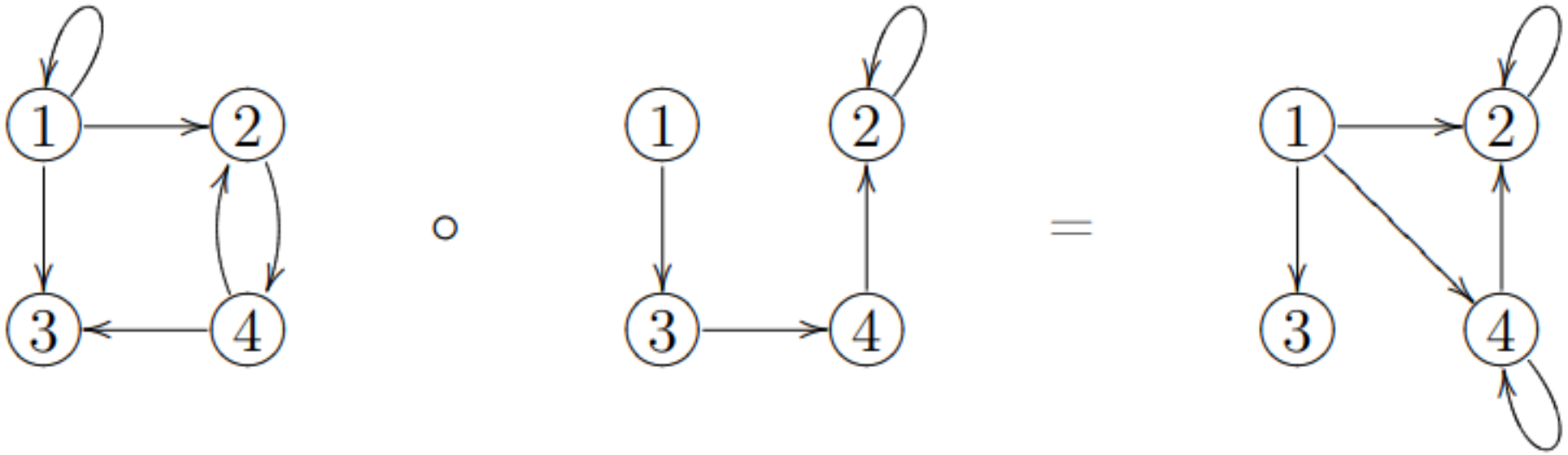
$M1 \circ M2$ est défini par:

$(M1 \circ M2)[i, j] = (\forall k \mid : M1[i, k] \wedge M2[k, j])$

Le produit des matrices est obtenu en faisant une opération similaire au produit matriciel en algèbre linéaire, en remplaçant + par $\vee$ (ou logique) et $\cdot$ par $\wedge$ (et logique).

Produit

$$(M_1 M_2)[i,j] \;=\; (\sum k \mid: M_1[i,k] \cdot M_2[k,j]) \;.$$

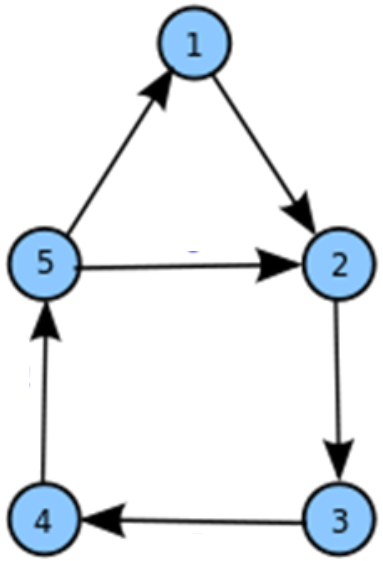


$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} \circ \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

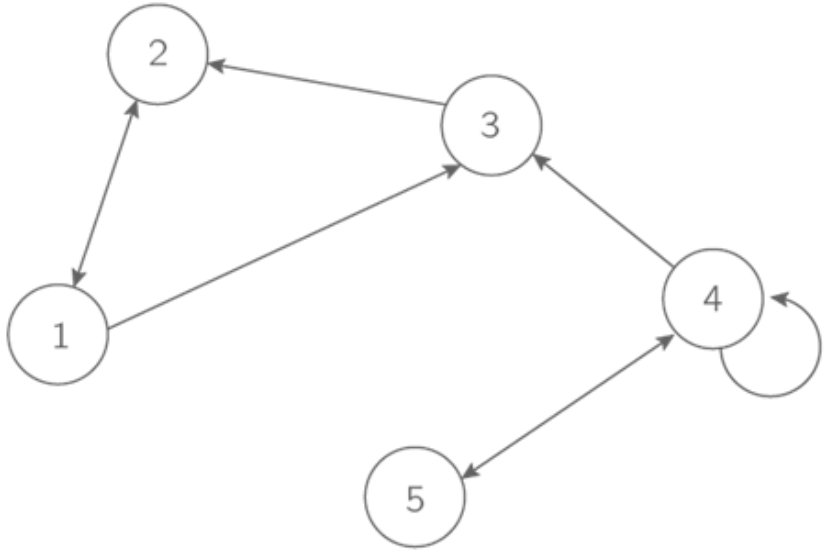
Par exemple, l'entrée $[1,2]$ vient de $(1 \wedge 0) \vee (1 \wedge 1) \vee (1 \wedge 0) \vee (0 \wedge 1) = 1$.

Exercices:

Calculez l'union, l'intersection et le produit de G1 et G2
Calculez l'inverse et le complément de G1 et de G2



G1



G2

Matrices d’adjacence:

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

G1

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

G2

Parcours de graphes

Beaucoup de problèmes sur les graphes nécessitent que l'on parcoure l'ensemble des sommets et des arcs/arêtes du graphe. Nous allons étudier dans ce chapitre les deux principales stratégies d'exploration :

- le **parcours en largeur** (ou BFS, Breath First Search) consiste à explorer les sommets du graphe niveau par niveau, à partir d'un sommet donné ;

- le **parcours en profondeur** consiste, à partir d'un sommet donné, à suivre un chemin le plus loin possible, puis à faire des retours en arrière pour reprendre tous les chemins ignorés précédemment.

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

- Initialement, tous les sommets sont « coloriés » en blanc (traduisant le fait qu'ils n'ont pas encore été "découverts").

- Lorsqu'un sommet est "découvert" (autrement dit, quand on arrive pour la première fois sur ce sommet), il est « colorié » en gris. Le sommet reste gris tant qu'il reste des successeurs de ce sommet qui sont blancs (autrement dit, qui n'ont pas encore été découverts).

- Un sommet est « colorié » en noir lorsque tous ses successeurs sont gris ou noirs (autrement dit, lorsqu'ils ont tous été découverts).

De façon pratique, on va utiliser une liste "d'attente au coloriage en noir" dans laquelle on va stocker tous les sommets gris : un sommet est mis dans la liste d'attente dès qu'il est colorié en gris. Un sommet gris dans la liste d'attente peut faire rentrer dans la liste ses successeurs qui sont encore blancs (en les coloriant en gris). Quand tous les successeurs d'un sommet gris de la liste d'attente sont soit gris soit noirs, il est colorié en noir et il sort de la liste d'attente.

La différence fondamentale entre le parcours en largeur et le parcours en profondeur provient de la façon de gérer cette liste d'attente au coloriage en noir :

- le **parcours en largeur** utilise une liste d'attente, où le premier sommet arrivé dans la file est aussi le premier à en sortir,

- tandis que le **parcours en profondeur** utilise une pile, où le dernier sommet arrivé dans la pile est le premier à en sortir.

Algorithme de parcours en largeur

De quoi part-on ?

- Un graphe G = (V,E) représenté par sa matrice d'incidence ou par une liste d'arêtes
- Un sommet quelconque r à partir duquel on va calculer les distances

Déroulement des opérations de parcours en largeur:

On part du sommet r.

Pour se souvenir que r a déjà été visité on le colore en vert (plutôt que gris dans l'exemple qui suit)

On sait aussi sans calcul que la distance r-r = 0 : $dist_G(r,r) = 0$

Ensuite on examine un par un les voisins de r et pour chacun d'eux, on fait 4 choses :

- Colorer u (u est le voisin choisi) en vert pour se rappeler qu'il a été déjà parcouru
- Noter que $dist_G(r,u) = 1$ (puisque u est voisin de r)
- Ajouter le nom du sommet u à droite d'une liste L, initialement vide
- Indiquer que le parent de u est r pour garder en mémoire le chemin entre u et r

L'ordre dans lequel on examine les voisins du sommet courant, ici r, n'a pas d'importance.

Lorsque tous les voisins de r ont été examiné, on colore r en rouge (plutôt que noir) pour indiquer que r et son voisinage a déjà été complètement exploité et qu'il est inutile d'y revenir

Si nous faisons le point, nous avons maintenant un sommet rouge, r, un certain nombre de sommets verts, les voisins de r et les autres sommets qui n'ont pas encore de couleur.

Nous avons également une liste L contenant les voisins de r.
L'opération suivante consiste à se positionner sur le sommet qui se trouve au début de la liste L (appelons-le u) et qui est coloré en vert et de recommencer les 4 opérations ci-dessus pour chaque voisin (appelé ici v) sauf si ce voisin est déjà de couleur verte ou rouge (i.e. a déjà été visité et traité) dans lequel cas, on ne fait plus rien sur ce sommet voisin :

- colorier en vert le sommet v,
- $dist_G(r,v) = dist_G(r,u) + 1$
- Ajouter le nom du sommet v à la fin (à droite) de la liste L
- Indiquer que le parent de v est u

Après avoir fait cela sur tous les voisins de u, u est coloré en rouge (fin de l'exploitation de u) et u est effacé de la liste L.

On choisit ensuite le sommet dont le nom est en tête de la liste L (après effacement de u) et on effectue le même traitement.

Le processus se termine quand la liste L est vide.

A ce moment, les sommets qui n'ont pas été coloriés sont les sommets qui ne sont pas joignables à partir de r. Leur distance $dist_G(r,x)$ = infini.

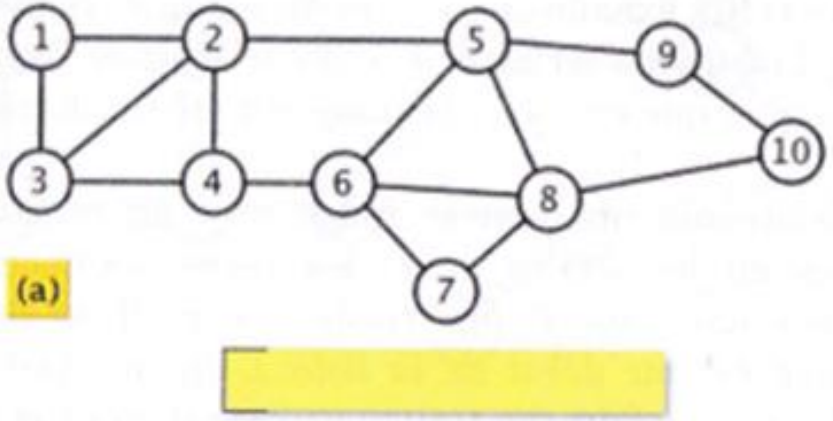
Cela peut être le cas par exemple si le graphe n'est pas connexe.

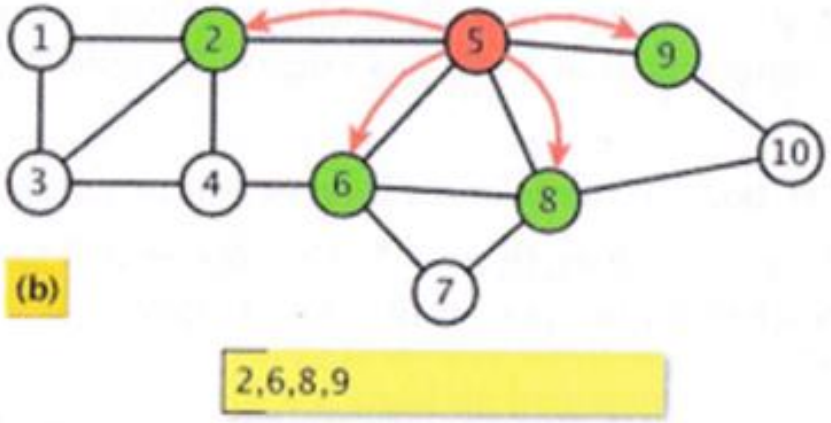

Un exemple
svp,
j’ai rien
compris!!!

Exemple :

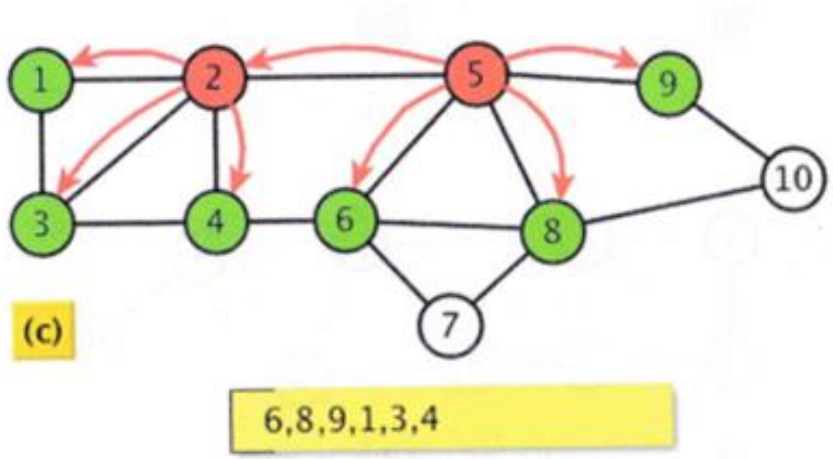
Nœud blanc = nœud non encore atteint vert= en cours de traitement rouge = terminé

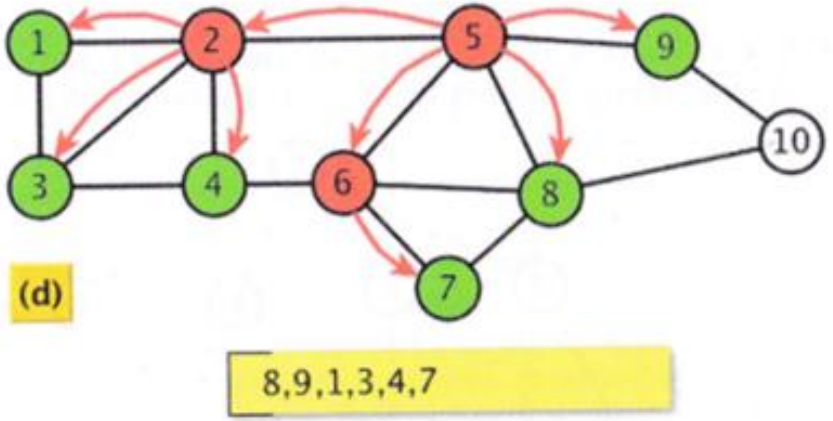
En jaune : la liste L





dist(r,2) = dist(r,6) = dist(r,8) = dist(r,9) = 1

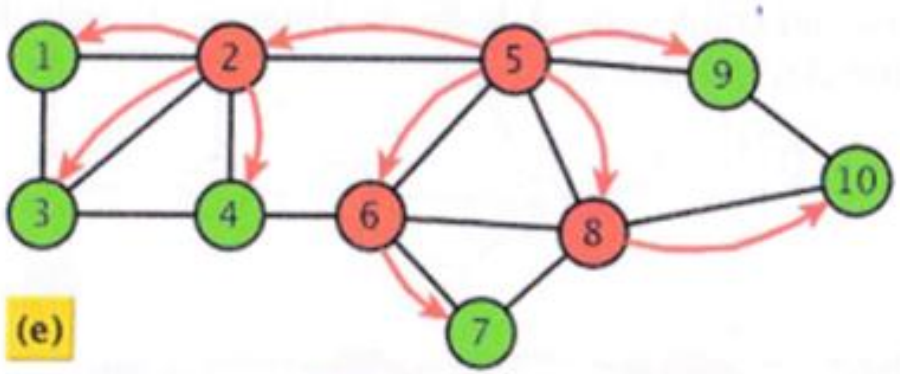




En (c) sommet 2: dist(r,1) = dist(r,3) = dist(r,4) = dist(r,2) + 1 = 2

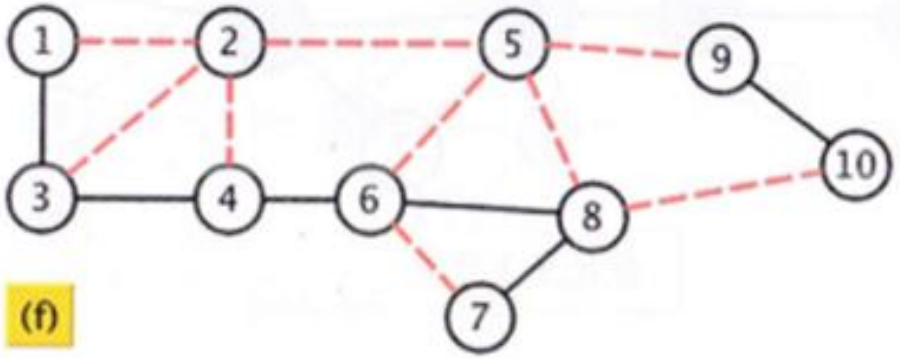
En (d) sommet 6: 5, 4, 8 déjà coloré donc rien à faire,

dist(r,7) = dist(r,6) + 1 = 2

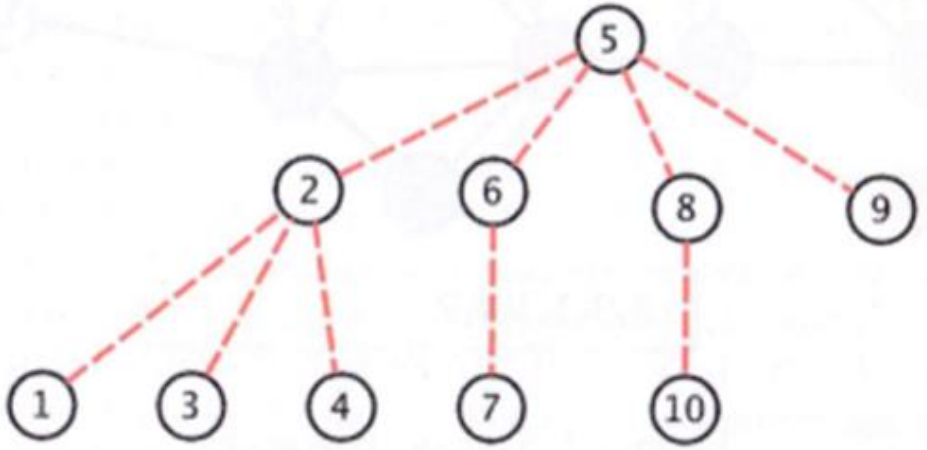


(e)

9,1,3,4,7,10



(f)



En (e) sommet 8 : seul 10 est blanc donc
dist(r,10) = dist(r,8) + 1 = 2

Ensuite, plus aucun sommet à traiter n'a de voisin blanc, ils sont donc retirés de L un à un et mis en rouge.
Le programme s'arrête quand L est vide.

La figure (f) représente en pointillés les arêtes utilisées qui forment un arbre couvrant du graphe représenté ci-dessous.


Merci!!!

L’algorithme de parcours en largeur vous donne donc toutes les distances à partir d’un nœud et un arbre des chemins les plus courts.

Cet algorithme peut aussi servir de test de connexité : si à la fin du parcours, tous les nœuds sont colorés en rouge, le graphe est connexe sinon pas.

L’ordre d’examen des voisins d’un nœud changera éventuellement l’arbre final (ce seront d’autres plus courts chemins qui y seront repris), mais pas les distances les plus courtes calculées.

Si le graphe est un graphe pondéré, il faut prendre comme distance d’un chemin la somme des poids des arêtes parcourues. Le chemin le plus court ne correspond dès lors plus au chemin comportant le minimum d’arêtes ! Un chemin entre deux sommets a et b de **deux** arêtes de poids respectif 4 et 5 à une distance de 9 alors qu’un autre chemin entre a et b comportant **trois** arêtes de poids 2, 1 et 4 possède une distance de 7 et est donc plus court !

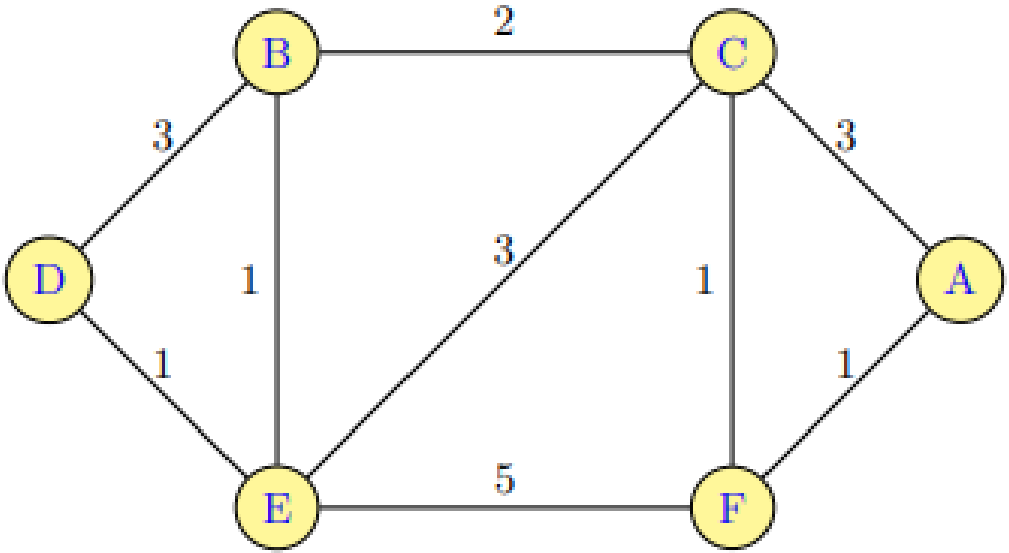
Algorithme de Dijkstra : trouver un chemin de longueur minimal allant de A à B

L'algorithme de Dijkstra permet de résoudre le problème suivant :

étant donné un graphe orienté pondéré, un nœud de départ et un nœud d'arrivée, trouver un chemin de longueur minimal allant du nœud de départ au nœud d'arrivée.

L’algorithme de Dijkstra est une généralisation du parcours en largeur au cas d’un graphe pondéré où tous les poids sont positifs.

Considérons le graphe suivant:



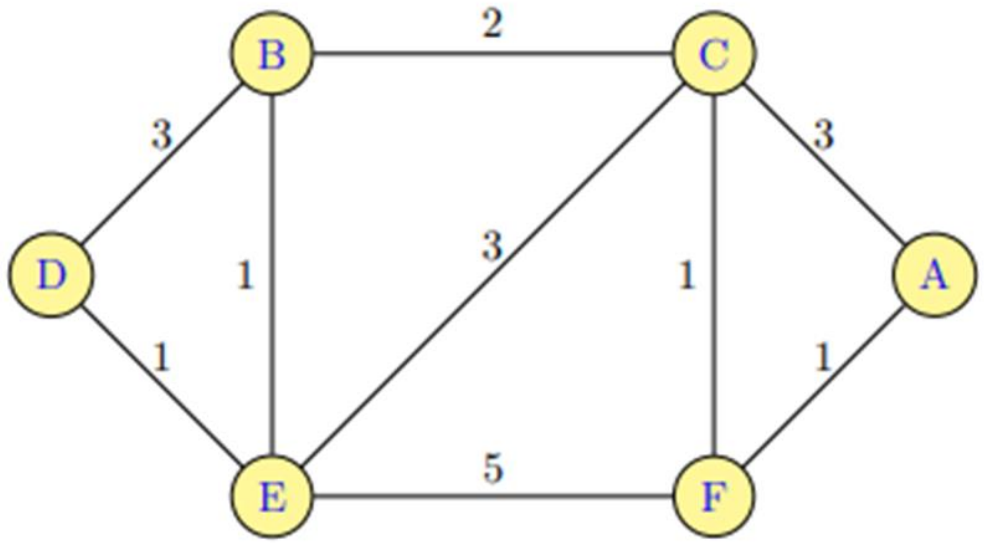
De quel type de graphe s’agit-il?

Définition

On appelle **matrice des poids** d'un graphe la matrice G telle que pour tout couple de sommets (i, j), A étant l'ensemble des arêtes :

⇥ si (i, j) appartient à A alors $g_{i,j}$ est égal au poids de l'arête (i, j),
⇥ si (i, j) n'appartient pas à A alors $g_{i,j}$ = infini.
⇥ si i = j alors $g_{i,i}$ = 0

Exemple:



	D	B	E	C	F	A
D	0	3	1	$+\infty$	$+\infty$	$+\infty$
B	3	0	1	2	$+\infty$	$+\infty$
E	1	1	0	3	5	$+\infty$
C	$+\infty$	2	3	0	1	3
F	$+\infty$	$+\infty$	5	1	0	1
A	$+\infty$	$+\infty$	$+\infty$	3	1	0

Que remarque-t-on?

	D	B	E	C	F	A
D	0	3	1	$+\infty$	$+\infty$	$+\infty$
B	3	0	1	2	$+\infty$	$+\infty$
E	1	1	0	3	5	$+\infty$
C	$+\infty$	2	3	0	1	3
F	$+\infty$	$+\infty$	5	1	0	1
A	$+\infty$	$+\infty$	$+\infty$	3	1	0

C’est une matrice symétrique.
Le graphe précédent n’est pas orienté.
Ainsi chaque arête (u, v) fournit une arête (v, u) de même poids.

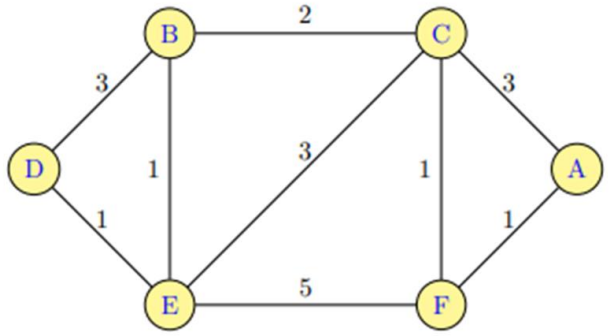
L’algorithme de Dijkstra consiste en la recherche des plus courts chemins menant d’un sommet unique s de S à chaque autre sommet d’un graphe pondéré G = (S,A).

Tous les arcs de G sont supposés de poids positif ce qui permet d’empêcher la présence de cycles de poids strictement négatifs.

Initialisation Au début de l’algorithme, on considère :
⇥ d[s] 0,
⇥ d[v] +infini pour tout sommet s de S.

Sur l’exemple précédent, et si on choisit s = D, l’étape d’initialisation s’écrit :

D	B	E	C	F	A
0	+inf	+inf	+inf	+inf	+inf



Principe de relâchement

L'opération de relâchement d'un arc (u, v) consiste en un test permettant de savoir s'il est possible, en passant par u, d'améliorer le plus court chemin jusqu'à v.
Si oui, il faut mettre à jour d[v].

Plus précisément, on effectue le test suivant :

$d[u] + g_{u,v} ? < d[v]$

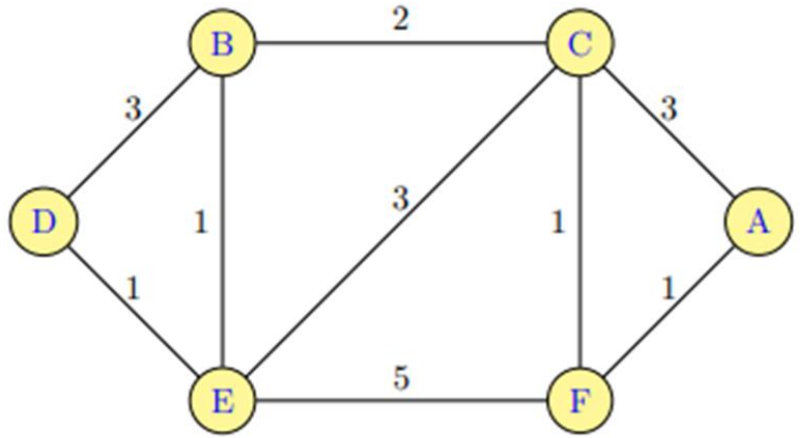
- Si le test est négatif, on n'effectue pas de mise à jour.
- Si le test est positif, cela signifie qu'un chemin de plus petite taille est trouvé pour passer du sommet s au sommet v :
  ⇥ on passe de s à u (ce chemin à le poids d[u]),
  ⇥ on termine ce chemin en utilisant l'arc (u, v) de poids $g_{u,v}$.
  Il convient alors d'effectuer la mise à jour : $d[v] = d[u] + g_{u,v}$

Exemple

1) Sur l'exemple précédent, on prend u = D

On cherche alors à déterminer, pour tout sommet v de S si on peut mettre à jours d[v] à l'aide d'un chemin dont le dernier arc est (u, v).

Pour l'étape d'initialisation, cela consiste à considérer les chemins de s à v de taille 1 :



D	B	E	C	F	A
0	3	1	+inf	+inf	+inf

2) On prend alors u = E (plus courte distance)

On cherche alors à déterminer, pour tout sommet v de S si on peut mettre à jours d[v] à l'aide d'un chemin dont le dernier arc est (u, v). Étant donnée l'étape précédente, cela consiste à considérer les chemins de s à v de taille 2 (autrement dit les chemins s ! u ! v) :

D	B	E	C	F	A
0	2	1	4	6	+inf

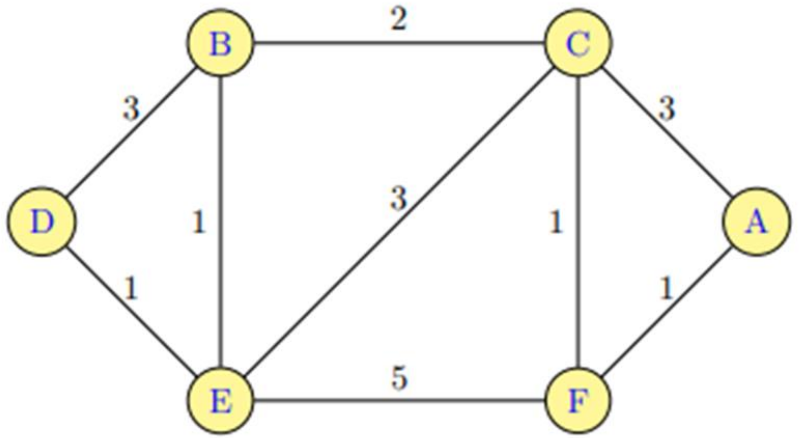
3) On prend alors u = B D et E ayant déjà été pris, on choisit B qui a la plus petite distance

On cherche alors à déterminer, pour tout sommet v de S si on peut mettre à jours d[v] à l'aide d'un chemin dont le dernier arc est (u, v).Étant donnée l'étape précédente, cela consiste à considérer les chemins de s à v de taille 3 (autrement dit les chemins s ! ... ! u ! v) :

D	B	E	C	F	A
0	2	1	4	6	+inf

Pas de changement, aucun meilleur chemin trouvé!

4) On prend alors u = C (sommet ayant la plus petite distance et non encore pris)
On cherche alors à déterminer, pour tout sommet v de S si
on peut mettre à jours d[v] à l'aide d'un chemin dont le dernier arc est (u, v).
Étant donnée l'étape précédente, cela consiste à considérer les chemins de s à v de taille 4 (autrement dit les chemins s ! ... ! ... ! u ! v) :



D	B	E	C	F	A
0	2	1	4	5	7

5) On prend alors u = F (sommet ayant la plus petite distance et non encore pris)
On cherche alors à déterminer, pour tout sommet v de S si on peut mettre à jours d[v] à l'aide d'un chemin dont le dernier arc est (u, v). Étant donnée l'étape précédente, cela consiste à considérer les chemins de s à v de taille 5 (autrement dit les chemins s ! ... ! ... ! ... ! u ! v) :

D	B	E	C	F	A
0	2	1	4	5	6

6) On prend alors u = A (sommet ayant la plus petite distance et non encore pris)
On cherche alors à déterminer, pour tout sommet v de S si on peut mettre à jours d[v] à l'aide d'un chemin dont le dernier arc est (u, v).Étant donnée l'étape précédente, cela consiste à considérer les chemins de s à v de taille 6 (autrement dit les chemins s ! ... ! ... ! ... ! ... ! u ! v) :

D	B	E	C	F	A
0	2	1	4	5	6

Pas de changement, aucun meilleur chemin trouvé!

Sélection du sommet u à chaque étape

À chaque étape de l'algorithme, on sélectionne le sommet u qui vérifie les propriétés suivantes :

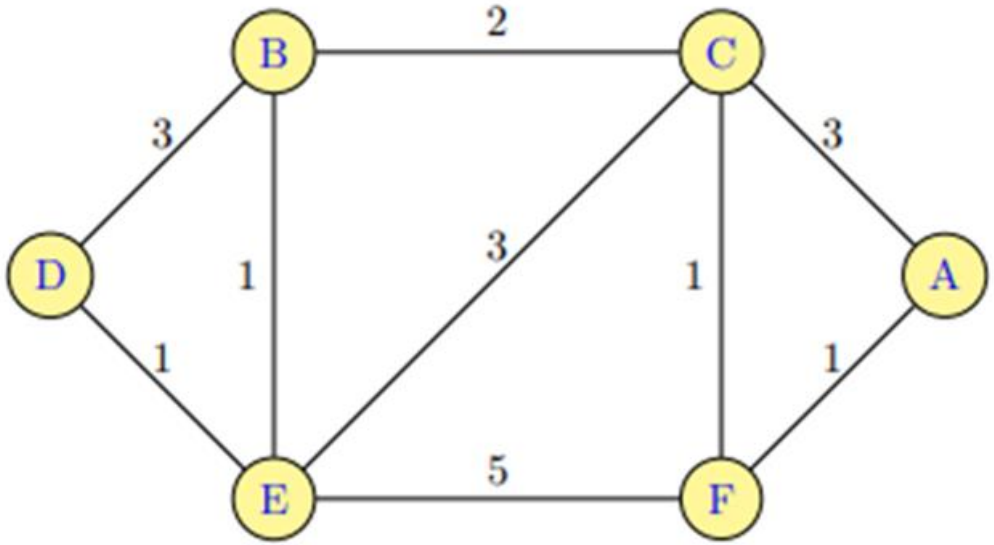
- u n'a pas encore été sélectionné,
- l'attribut d[u] est le plus faible (parmi tous les sommets non encore sélectionnés).

On parle d'algorithme glouton : on sélectionne à chaque étape la sous-solution optimale i.e. le sommet réalisant la meilleure distance.

Fin de l'algorithme

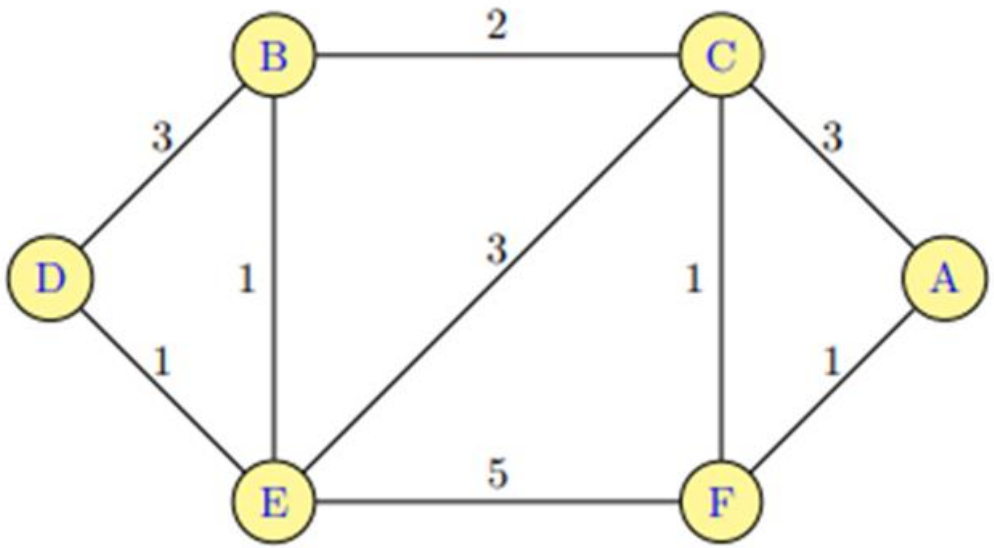
Une fois tous les sommets visités, on a trouvé tous les plus courts chemins (de s à tout v) de taille inférieure ou égale au nombre de sommets en tout. On a donc trouvé tous les plus courts de chemins (de s à tout v).

Évolution des distances en partant de D



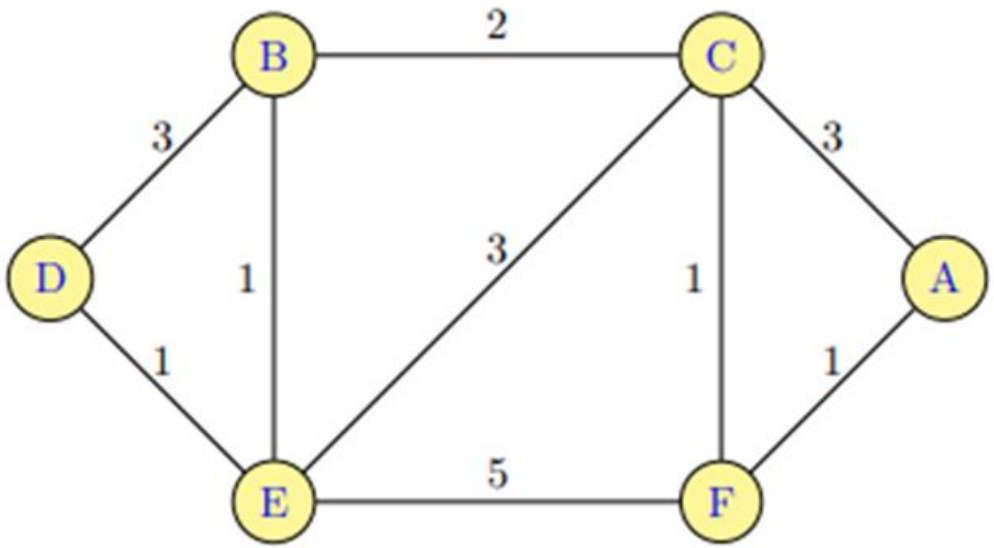
D	B	E	C	F	A
0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
0	3	1	$+\infty$	$+\infty$	$+\infty$
0	2	1	4	6	$+\infty$
0	2	1	4	6	$+\infty$
0	2	1	4	5	7
0	2	1	4	5	6
0	2	1	4	5	6

Évolution des distances en partant de E



D	B	E	C	F	A

Évolution des distances en partant de C



D	B	E	C	F	A

Afficher les plus courts chemins

• Si à une étape, d[v] a été modifié, il faut se rappeler de quel sommet u provient cette modification.

• L’idée étant alors que l’arc (u, v) sera un arc du plus court chemin.

Parcours d’un graphe en profondeur

Il s’agit de parcourir un graphe G à partir d’un sommet r en construisant un arbre couvrant T.

On obtient donc un résultat déjà obtenu avec l’algorithme précédent, mais en utilisant d’autres règles et avec un résultat possédant d’autres propriétés.

On retrouve dans l’algorithme une phase de descente à partir du sommet r dans les profondeurs du graphe suivie lorsqu’il n’est plus possible de descendre d’une remontée en revenant sur ses pas pour mieux repartir dans une exploration en profondeur.

La phase de descente

L'algorithme part du sommet de départ r vers un des sommets voisins qu'il n'a pas encore visités soit u.

Ce faisant, lorsqu'il visite pour la première fois le sommet u venant d'un sommet r, il marque l'arête (r,u) pour indiquer que u a pour parent r.

Arrivé en u, il choisit encore n'importe quel voisin de u sauf r déjà visité, soit v. Si v est visité pour la première fois, il marque l'arête (u,v) pour indiquer que v a pour parent u.

Et le processus continue jusqu'à ce que l'on soit coincé, c'est-à-dire que soit il n'y a pas d'autre voisin que le sommet dont on vient (on est alors sur une feuille) soit qu'il n'y a plus de voisin non visité du nœud où l'on se trouve. Que faire ?

La phase de remontée

Dans ce cas, l'algorithme va faire un pas en arrière et remonte vers le parent du nœud qui est donc le sommet à partir duquel il a visité le nœud pour la première fois.

Et de là, l'algorithme essaie de repartir en marche avant et de repartir en profondeur tant qu'il n'est pas calé à nouveau.

A chaque étape, les sommets sont tagués. Au départ, ils sont tous blanc. S'il a été visité mais pas encore complètement exploité, le sommet devient vert et s'il a été totalement exploité (tous les voisins visités et donc on retourne à son parent), il devient rouge.

Le parcours est terminé quand on revient au point de départ et qu'il n'y a plus de voisin de r à explorer. Tous les sommets du graphe sont rouges (en tous cas si le graphe est connexe).

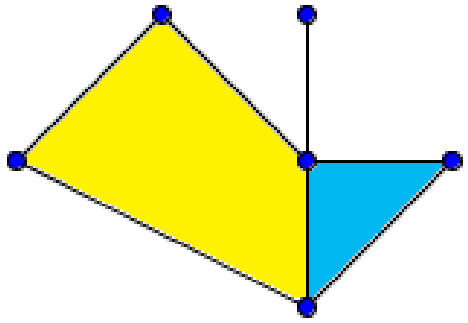
L’algorithme permet donc aussi de tester la connexité d’un graphe.

L’arbre obtenu avec les arêtes marquées est bien un arbre couvrant qui peut varier selon l’ordre choisit pour l’exploration des sommets.

Cet algorithme par contre ne procure pas d’information sur les distances. Il est surtout utilisé pour parcourir les arbres.

Graphe planaire

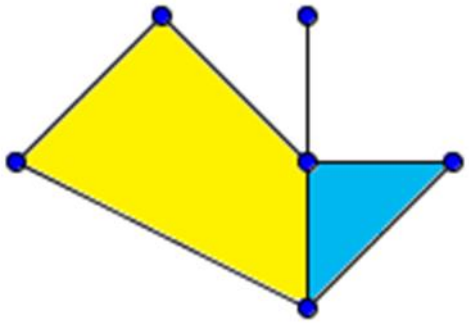
Un graphe est dit **planaire** si on peut le représenter dans le plan de telle sorte que ses arêtes ne se croisent pas .
Il est assez facile de prouver qu'un graphe est planaire : il suffit de trouver une représentation dans le plan dont les arêtes ne se coupent pas!
C'est en revanche beaucoup plus difficile de prouver qu'un graphe n'est pas planaire, car il faut pouvoir démontrer qu'aucune représentation du graphe ne convient!
Une fois représenté dans le plan, un graphe planaire nous apporte des données supplémentaires : on dispose de ses sommets, de ses arêtes, mais aussi des **faces** (ou des régions) que ces arêtes délimitent. Par exemple, dans la représentation planaire précédente, on a 3 faces (ou régions) : celle coloriée en jaune, celle coloriée en bleu, et la face extérieure au graphe (la feuille blanche sur laquelle il est imprimé).



Le résultat suivant, appelé **relation d'Euler-Descartes**, relie le nombre d'arêtes, de sommets et de faces d'un graphe planaire.

Théorème :
Soit G un graphe planaire possédant **s** sommets, **a** arêtes et **f** faces.
Alors :

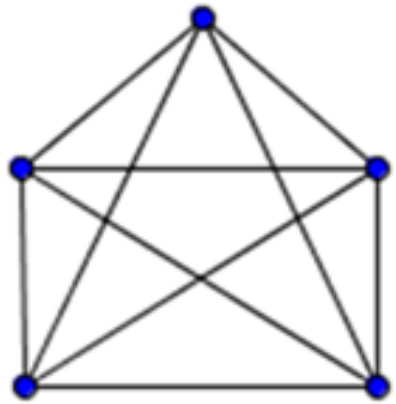
$s - a + f = 2$



6 - 7 + 3 = 2

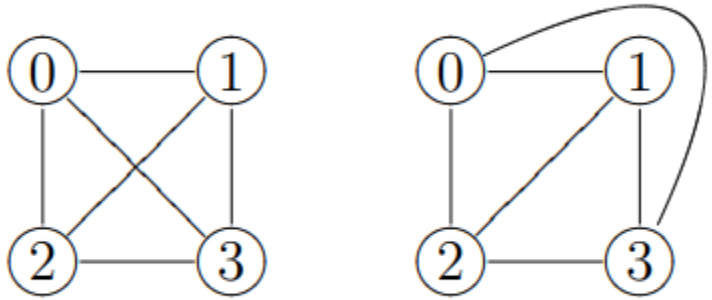
Cette relation permet de démontrer que certains graphes ne sont pas planaires.

De même un graphe planaire ayant s sommets possède au plus $3s - 6$ arêtes.

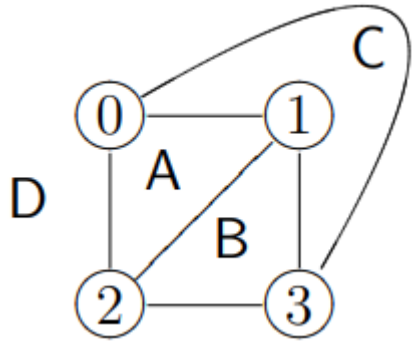


Le graphe complet à 5 sommets ci-contre possède 10 arêtes, et $10 > 9$. Il n'est pas planaire!

Autre exemple, le graphe complet K_4 peut être dessiné des deux manières données ci-contre. Comme le graphe de droite ne contient pas de croisement, K_4 est planaire.



Un graphe planaire divise le plan en régions (faces) internes et une région (face) externe. Dans le graphe ci-dessous, les régions internes sont A, B, C et la région externe est D.

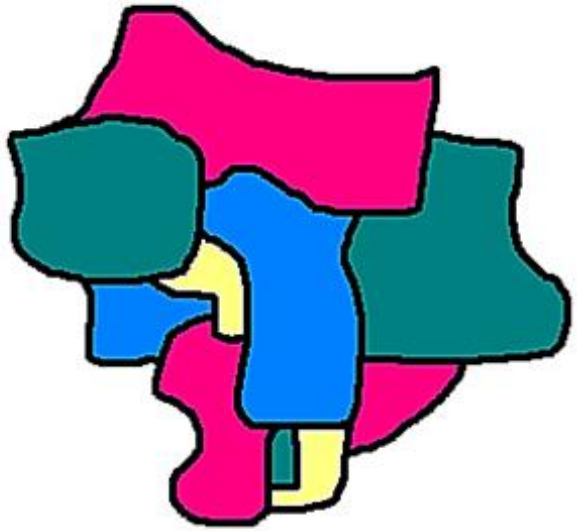


Théorème : Pour tout graphe planaire connexe avec s sommets, a arêtes et r régions,
$r = a - s + 2$. Par exemple, pour le graphe ci-dessus, $r = 4$, $a = 6$ et $s = 4$

Théorème des 4 couleurs

Prenez une carte du monde, que l'on cherche à colorier en donnant une couleur à chaque pays.

Quel nombre minimum de couleurs faut-il pour que deux pays ayant une frontière commune n'aient jamais la même couleur?



Exemple de coloration d'une carte avec 4 couleurs!

L'exemple ci-contre montre qu'on ne peut pas se contenter de 3 couleurs!

Ce résultat peut être énoncé dans le langage de la théorie des graphes de la forme suivante :

Le nombre chromatique d'un graphe planaire est inférieur ou égal à 4.

Les graphes bipartis peuvent être colorés en utilisant au plus deux couleurs.

L'histoire du théorème des quatre couleurs remonte au moins au XIXè siècle.

En 1852 Francis Guthrie, mathématicien et botaniste anglais, remarque qu'il lui suffit de quatre couleurs pour colorier la carte des cantons d'Angleterre, sans donner la même couleur à deux cantons qui ont une frontière commune.

En 1879, Kempe publie une première "preuve'' de la conjecture, qui est malheureusement fausse.

La première démonstration date en fait de… 1976, par Appel et Haken !

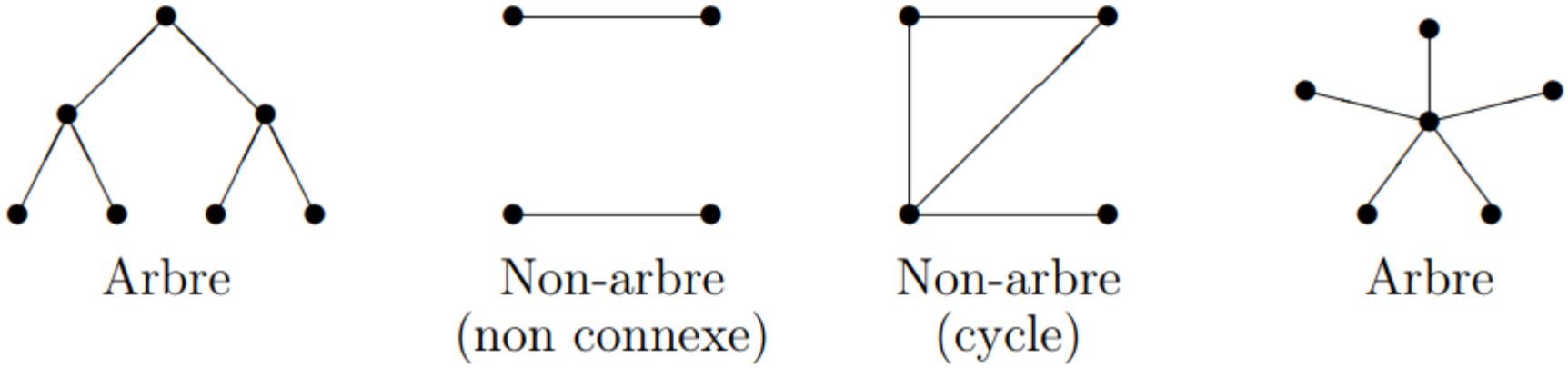
Leur preuve consiste à se ramener à un nombre fini, mais très grand, de configurations. Ensuite, pour chaque configuration, il faut vérifier que 4 couleurs suffisent.

Ces nombreuses vérifications ont été faites par ordinateur. Ceci produisit alors une tempête incroyable chez les mathématiciens. C'était la première preuve produite grâce à l'outil informatique. En particulier, on ne pouvait la vérifier humainement! Encore maintenant, on ne connait pas de preuves n'utilisant pas l'ordinateur. En revanche, de nombreuses autres conjectures célèbres ont été démontrées à l'aide de l'informatique, comme la conjecture de Képler.

Arbres et forêts en théorie des graphes

Un arbre est un graphe non orienté, connexe, et sans boucle ni cycle. Il est dénommé ainsi car, représenté dans le plan, sa forme évoque les ramifications d'une branche.

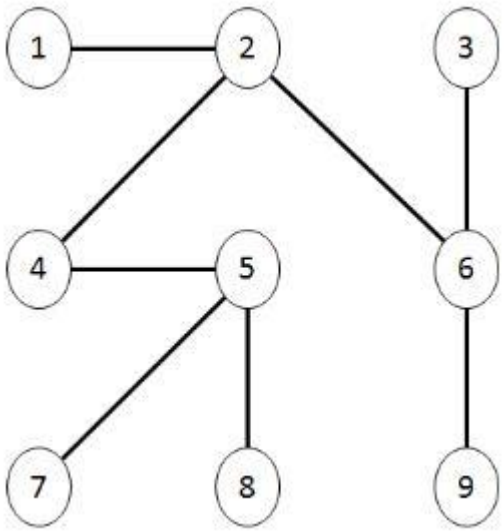
Une forêt est simplement un ensemble d'arbres (ou d'arborescences).

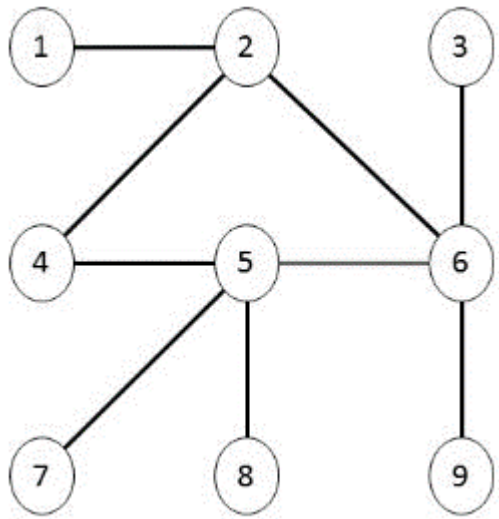


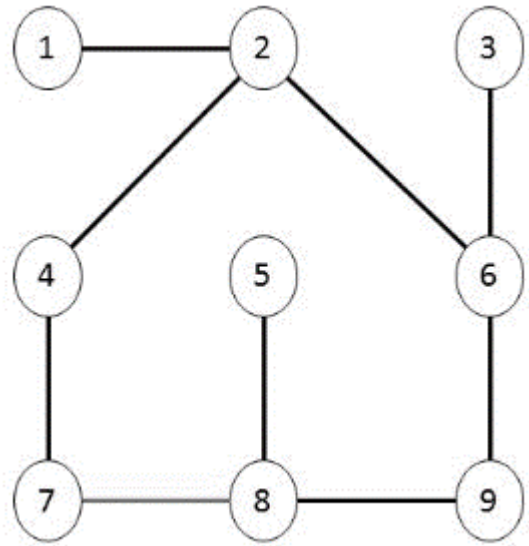
Comment savoir si un graphe est un arbre ?

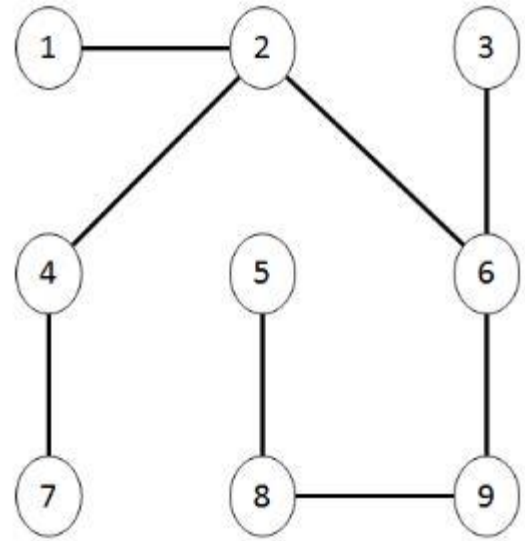
Un graphe dans lequel deux sommets quelconques sont reliés par un et un seul chemin élémentaire est connexe et n'a pas de cycle ; c'est donc un arbre.

Arbre ou pas?

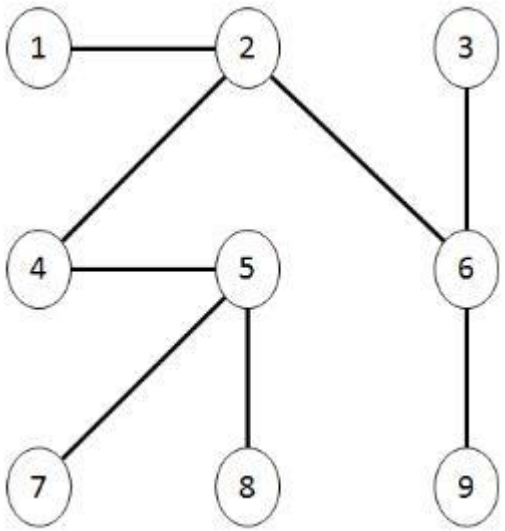




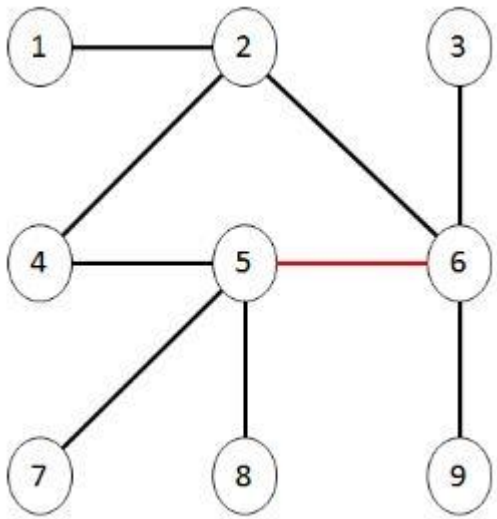




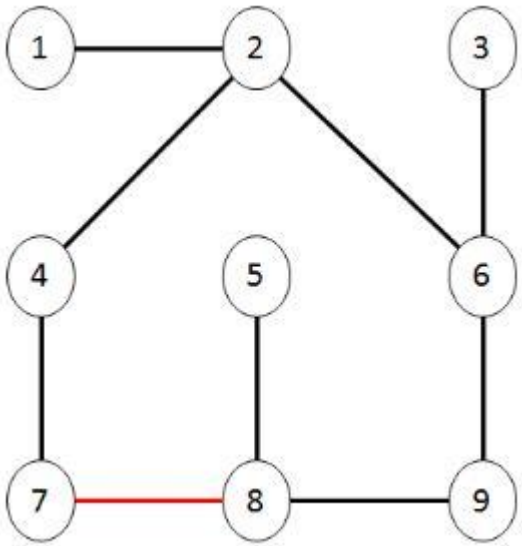
Arbre ou pas?



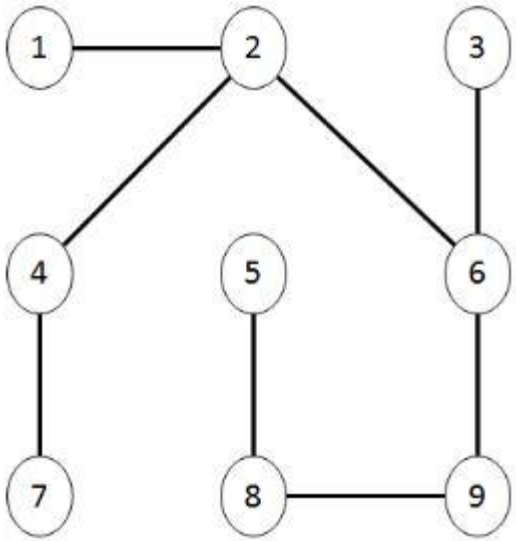
Arbre



Pas arbre



Pas arbre

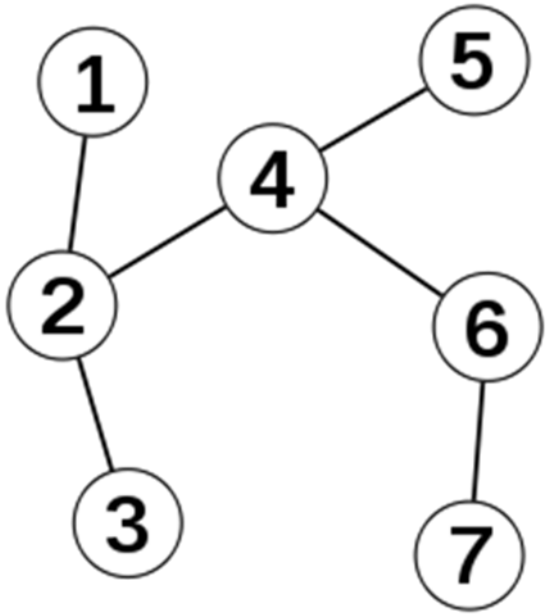


Arbre

On distingue dans les arbres deux types de sommets :

- les sommets de degré 1, on les appelle les **feuilles**. Sur l'exemple ci-dessus, il s'agit des sommets 1,3,5 et 7.

- les sommets de degré au moins 2, on les appelle les **noeuds internes**. Sur l'exemple ci-dessus, il s'agit des noeuds 2,4,6.



Les arbres possèdent de nombreuses propriétés : par exemple, ce sont des graphes bipartis.

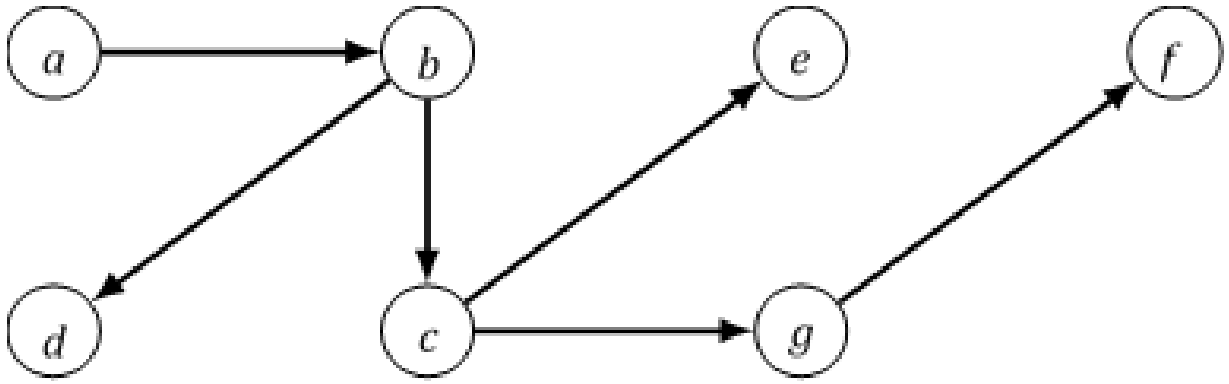
Les **graphes bipartis** sont les graphes dont le nombre chromatique est inférieur ou égal à 2

Arborescence

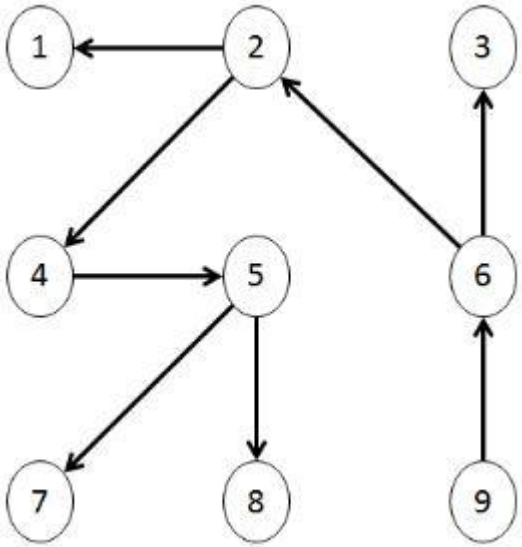
Une arborescence est un graphe orienté sans circuit admettant une racine s0 ∈ S telle que pour tout autre sommet si ∈ S, il existe un chemin unique allant de s0 vers si .

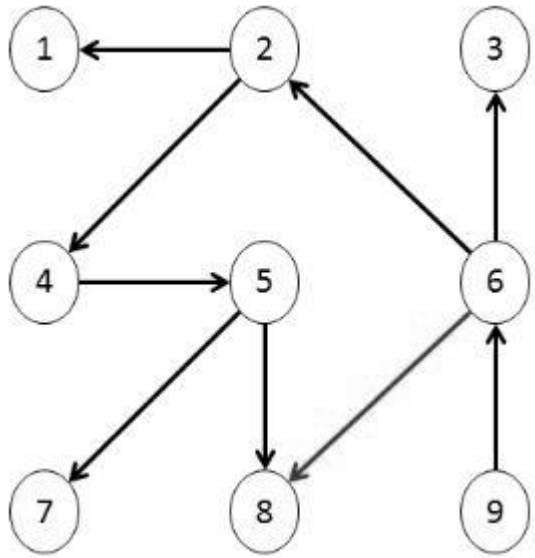
Si l'arborescence comporte n sommets, alors elle comporte exactement n − 1 arcs.

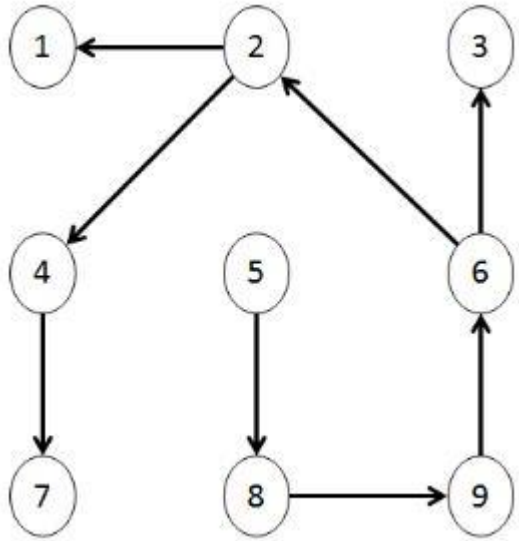
Par exemple, le graphe suivant est une arborescence de racine a :

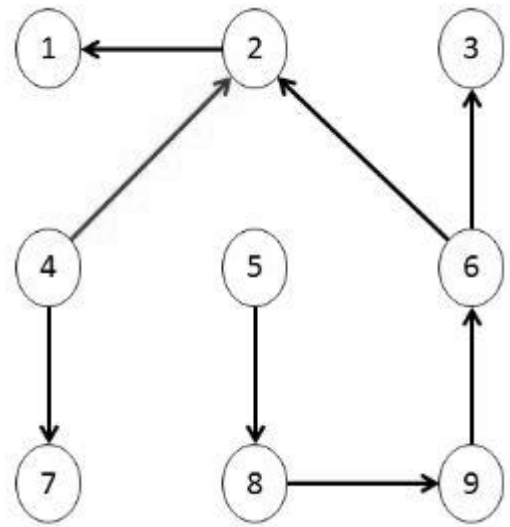


Arborescence ou pas?

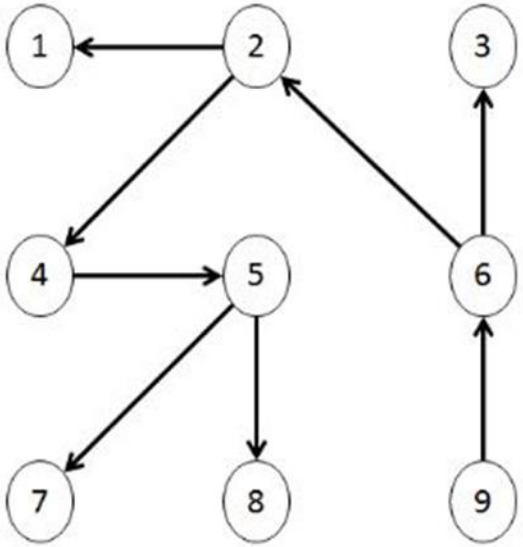




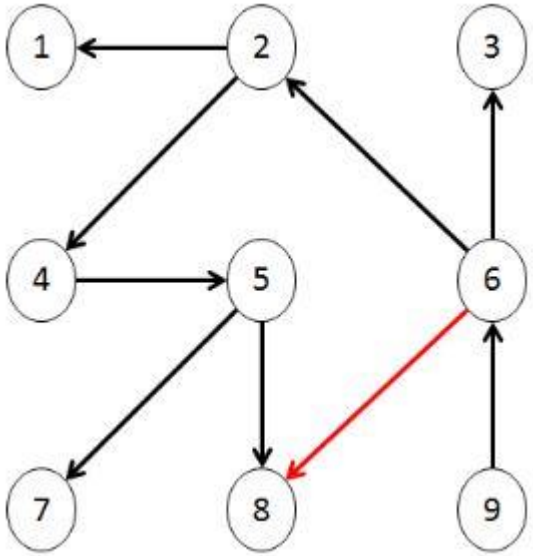




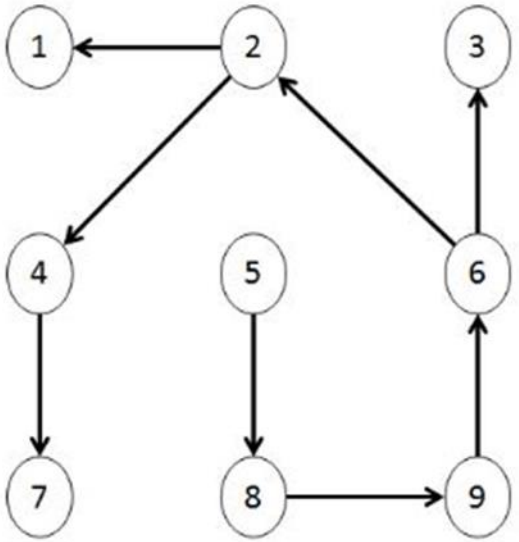
Arborescence ou pas?



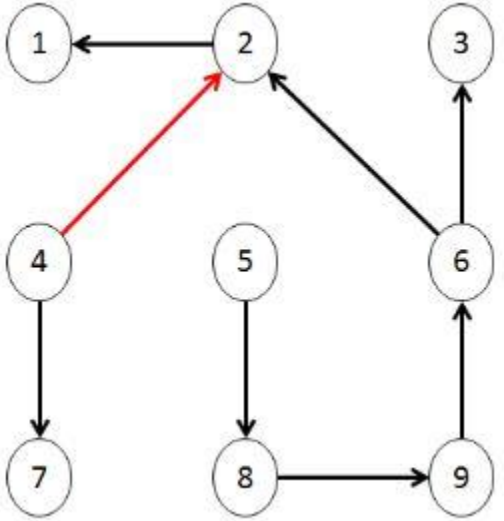
Arborescence



Pas arborescence



Arborescence



Pas arborescence

Représentation d'une arborescence ou d’un arbre

Arbres et arborescences étant des graphes, les représentations matricielles que nous avons vues s'appliquent bien entendu.

Dans le cas d'une arborescence, chaque sommet n'ayant qu'un seul prédécesseur (sauf la racine), chaque colonne de la matrice d'adjacence (sauf celle de la racine) contient un et un seul (1). La colonne correspondant à la racine ne contient que des valeurs (0).

Une représentation beaucoup plus simple peut être utilisée :

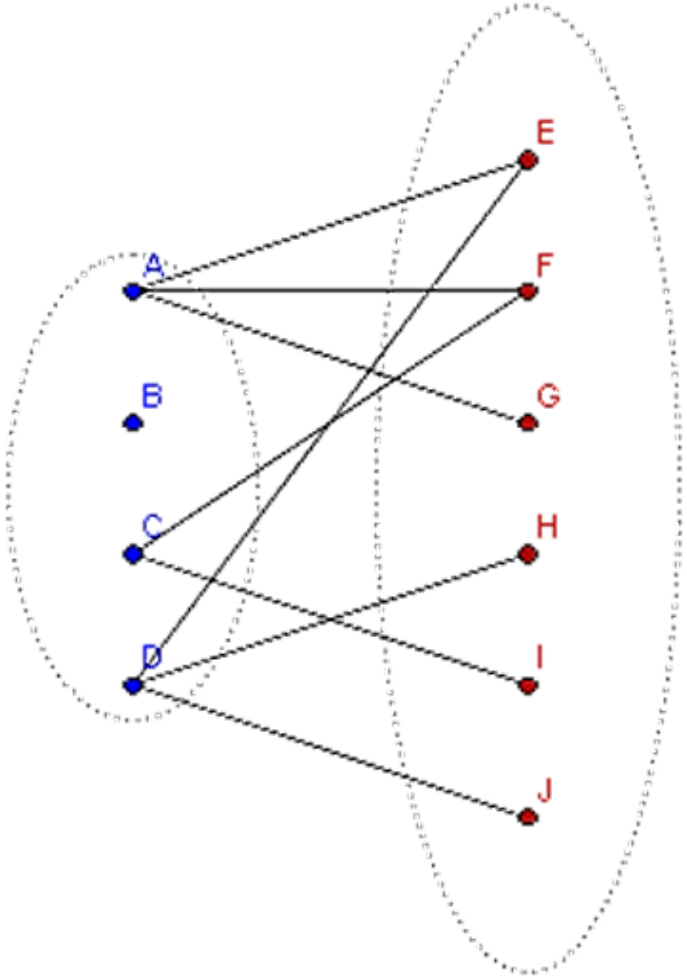
Il suffit en effet de représenter uniquement la fonction prédécesseur Γ^{-1} qui, à tout sommet, associé son unique prédécesseur ($\emptyset$ pour la racine). Un simple tableau suffit.

On peut ainsi, en partant de n'importe quel sommet x, retrouver "à rebours" le chemin qui mène de la racine à x.

Graphe biparti

Un graphe est dit **biparti** si on peut partager son ensemble de sommets en deux parties A et B tels qu'il n'y ait aucune arête entre éléments de A et aucune arête entre éléments de B.

On peut aussi dire qu’un graphe est biparti si chaque sommet peut être coloré soit en bleu, soit en orange (les couleurs sont bien sûr arbitraires) de telle manière à ce que chaque arête ait une extrémité bleue et l’autre orange.

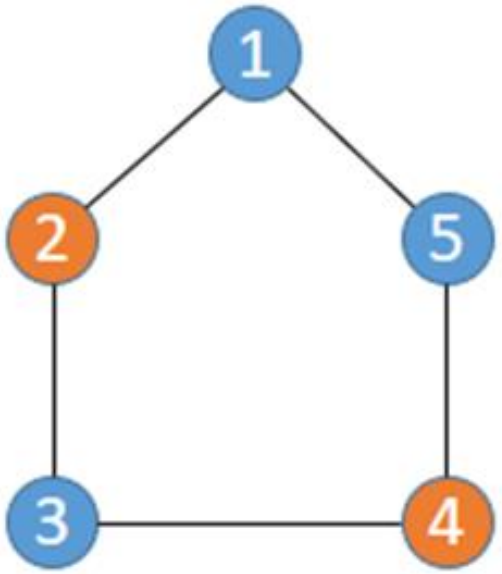


Théorème :

Un graphe est biparti si et seulement s'il ne contient pas de cycles de longueur impaire.

Rappelons que la longueur d'un cycle est égale au nombre d'arêtes qu'il contient.

En particulier, d'après le théorème précédent, les arbres sont donc des graphes bipartis.



Il est impossible d'alterner deux couleurs sur un cycle de longueur impaire !

Autres propriétés des arbres

Théorème : Entre deux sommets quelconques d'un arbre, il y a un chemin simple unique.

Théorème : Un arbre avec au moins deux sommets a au moins deux sommets de degré 1.

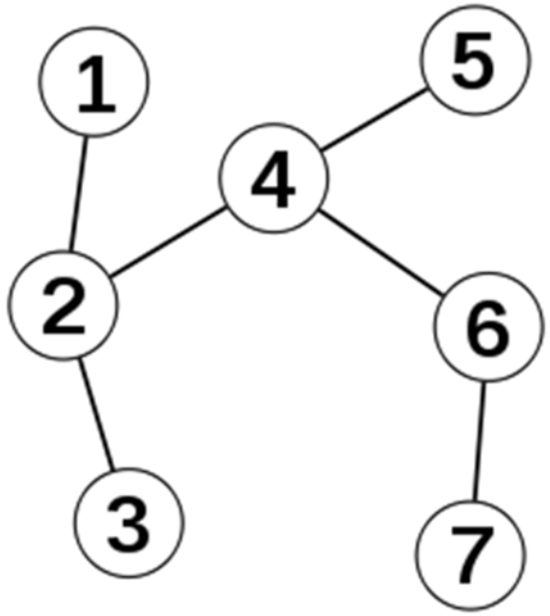
Théorème : Pour tout arbre <S, A>, #S = 1 + #A. En langage simple: le nombre de sommets excède le nombre d'arêtes d'une unité exactement.

Théorème : Soit G = <S, A> un graphe non orienté sans boucle.
Les énoncés suivants sont équivalents :

- G est un arbre.
- G est connexe et l'enlèvement d'une arête quelconque produit deux arbres.
- G n'a pas de cycle et #S = 1 + #A.
- G est connexe et #S = 1 + #A.
- G n'a pas de cycle et l'ajout d'une arête introduit exactement un cycle.

Notion de feuille

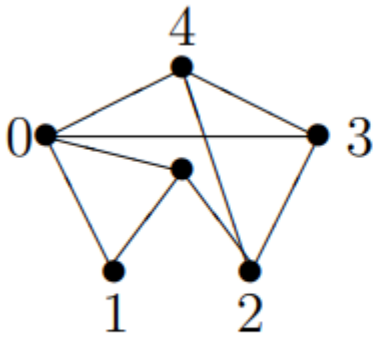
Un sommet *x* dont le degré vaut 1 est appelé feuille.



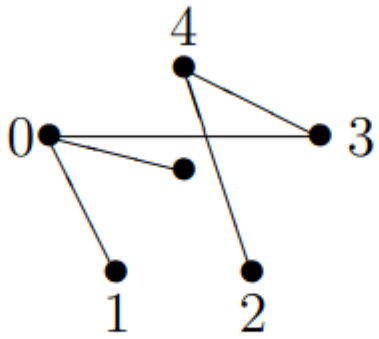
Le graphe ci-dessus est un arbre fini avec 4 feuilles (les sommets nos 1, 3, 5, 7) et 3 sommets internes (les sommets nos 2, 4 et 6).

Arbres couvrant (générateurs)

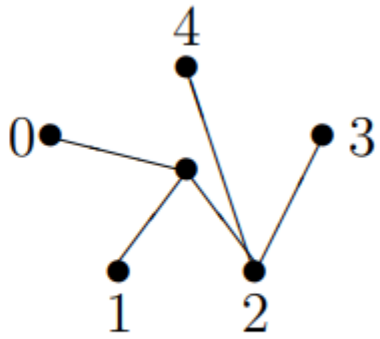
Un arbre couvrant d'un graphe non orienté et connexe est un arbre inclus dans ce graphe et qui connecte tous les sommets du graphe.



G



Arbre générateur 1



Arbre générateur 2

Il y a plusieurs autres arbres générateurs pour ce graphe.

Arbres couvrant

Un arbre couvrant T de G est un arbre partant d'un sommet r de G comprenant tous les autres sommets de G mais uniquement certaines de ses arêtes.

Une propriété intéressante de cet arbre couvrant T est que chaque chemin de T entre r et n'importe quel sommet u est aussi un plus court chemin entrez r et u à travers G.

Extraire un chemin couvrant permet donc de calculer les plus courts chemins à partir d'un sommet vers tous les autres ainsi que leur longueur.

Racine

Un **arbre enraciné** est simplement un arbre dont un sommet porte le statut particulier de **racine**. Formellement, un arbre enraciné est un triplet (S,A,r) où (S,A) est un arbre et r est un élément de S appelé racine.

Cette notion de racine permet de se munir d'un ordre de type ancêtre/descendant entre les sommets de l'arbre.

Hauteur d'un sommet x: Distance entre x et la racine r.
Enfant (ou fils) d'un sommet x: Sommet adjacent à x dont la hauteur vaut celle de x + 1
Parent d'un sommet x: Sommet tel que x est son enfant.
Descendant d'un sommet x: Sommet qui découle d'une lignée parent/enfant partant de x.
Ancêtre d'un sommet x: Sommet tel que x est un descendant.

Arbre binaire

un arbre binaire est un arbre, tel que le degré de chaque nœud soit au plus 3.

La racine d'un arbre binaire est le nœud d'un graphe de degré maximum 2.

Avec une racine ainsi choisie, chaque nœud aura un unique parent défini et deux fils ;

Un arbre binaire (ou binaire-unaire) est un arbre avec une racine dans lequel chaque nœud a au plus deux fils.

Un arbre binaire strict ou localement complet est un arbre dont tous les nœuds possèdent zéro ou deux fils.

Un arbre binaire **dégénéré** est un arbre dans lequel tous les nœuds internes n'ont qu'un seul fils. Ce type d'arbre n'a qu'une unique feuille et peut être vu comme une liste chaînée.

Un **arbre binaire parfait** est un arbre binaire strict dans lequel toutes les feuilles (nœuds n'ayant aucun fils) sont à la même distance de la racine (c'est-à-dire à la même profondeur). Il s'agit d'un arbre dont tous les niveaux sont remplis : où tous les noeuds internes ont deux fils et où tous les noeuds externes ont la même hauteur.