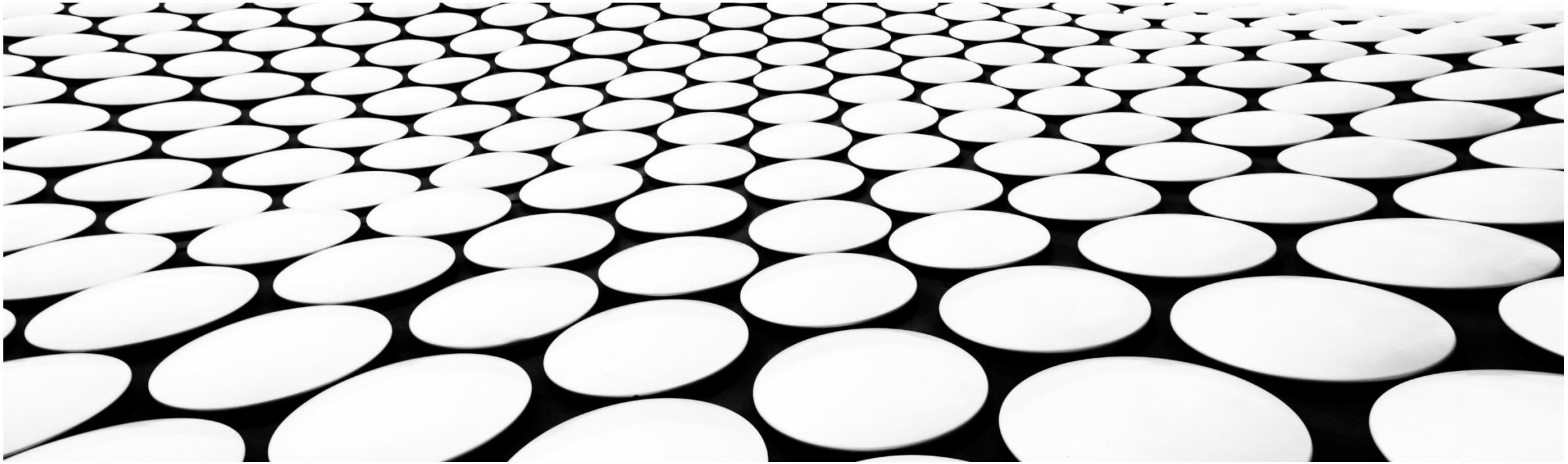
R5.C.07: *DONNEES MASSIVES*

Partie 2: les graphes

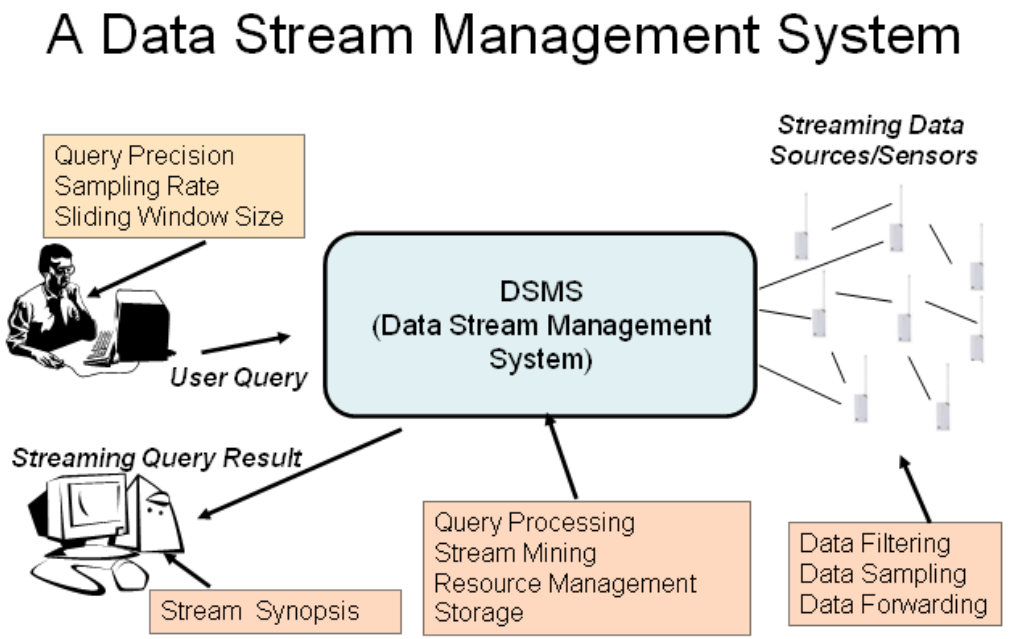


DATA STREAM MANAGEMENT SYSTEM

Un système de gestion de données doit savoir manipuler et gérer un flux de données continu.

Une BD, par contre, gère des données statiques (ou rarement modifiées) .

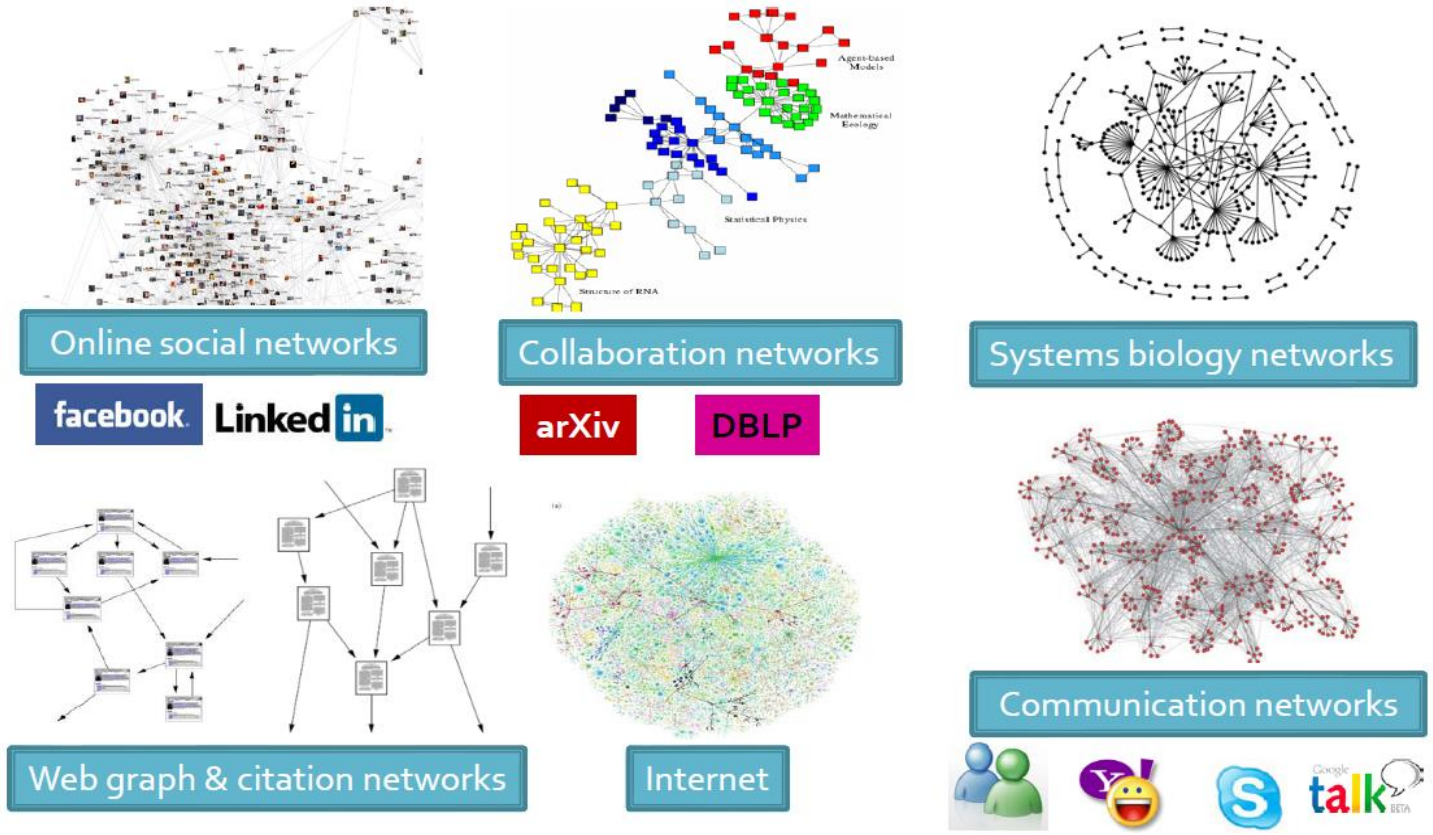
L'analyse en temps réel comprend, par exemple, les réseaux sociaux ou le web.



REPRÉSENTATION STRUCTURÉE DES DONNÉES

Les données massives, telles que les réseaux sociaux, les systèmes de recommandation et les interactions en ligne, peuvent être naturellement modélisées sous forme de **graphes**.

La théorie des graphes offre des outils pour représenter ces relations complexes de manière **structurée**, ce qui facilite leur compréhension et leur analyse.

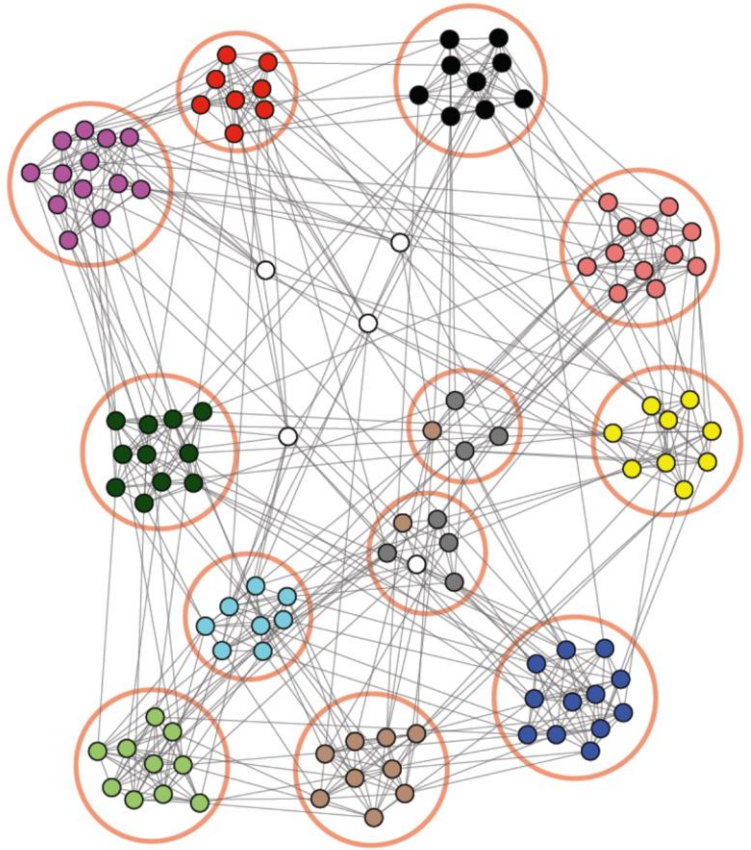


REPRÉSENTATION STRUCTURÉE DES DONNÉES

Dans le domaine des données massives, les réseaux sociaux jouent un rôle central.

Les graphes permettent de représenter les connexions entre **individus**, **groupes** et **entités**.

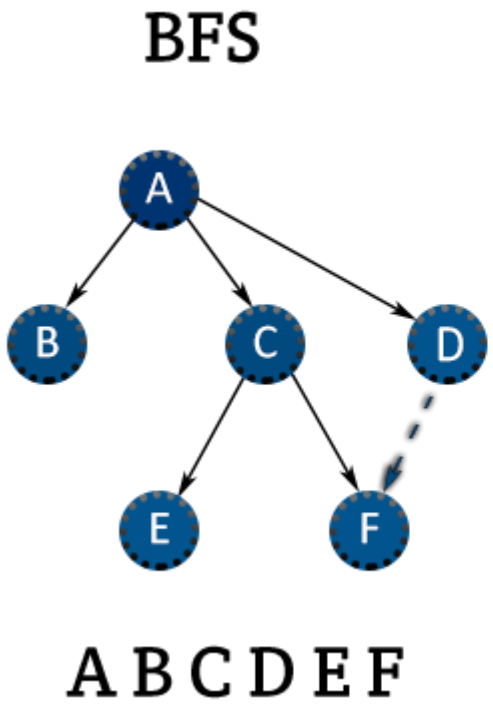
L'analyse des graphes sociaux peut révéler des *tendances*, des communautés, des influenceurs et des modèles de propagation d'information.

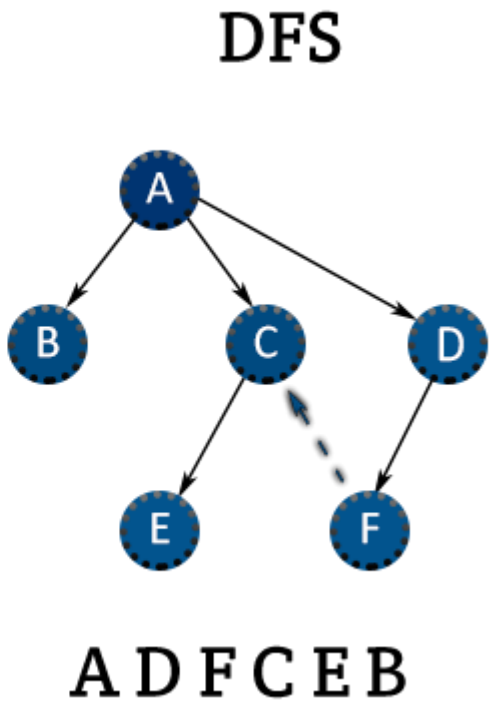


ALGORITHMES DE PARCOURS ET DE RECHERCHE

Les graphes massifs nécessitent des algorithmes efficaces pour parcourir, rechercher et extraire des informations pertinentes.

Des algorithmes comme la recherche en profondeur (**DFS**) et la recherche en largeur (**BFS**) sont utilisés pour explorer ces données rapidement et efficacement.

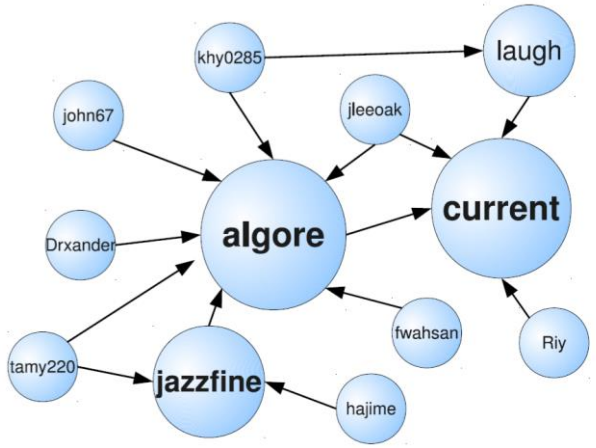


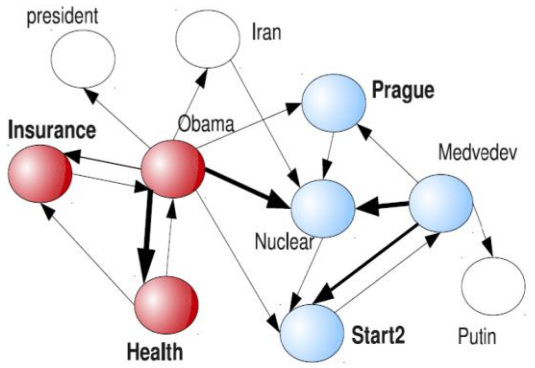


DÉCOUVERTE DE MOTIFS ET DE COMMUNAUTÉS

La détection de **motifs** et de **communautés** au sein de graphes massifs peut révéler des informations cachées.

Cela peut conduire à une meilleure compréhension des comportements, des interactions et des structures au sein de ces données.

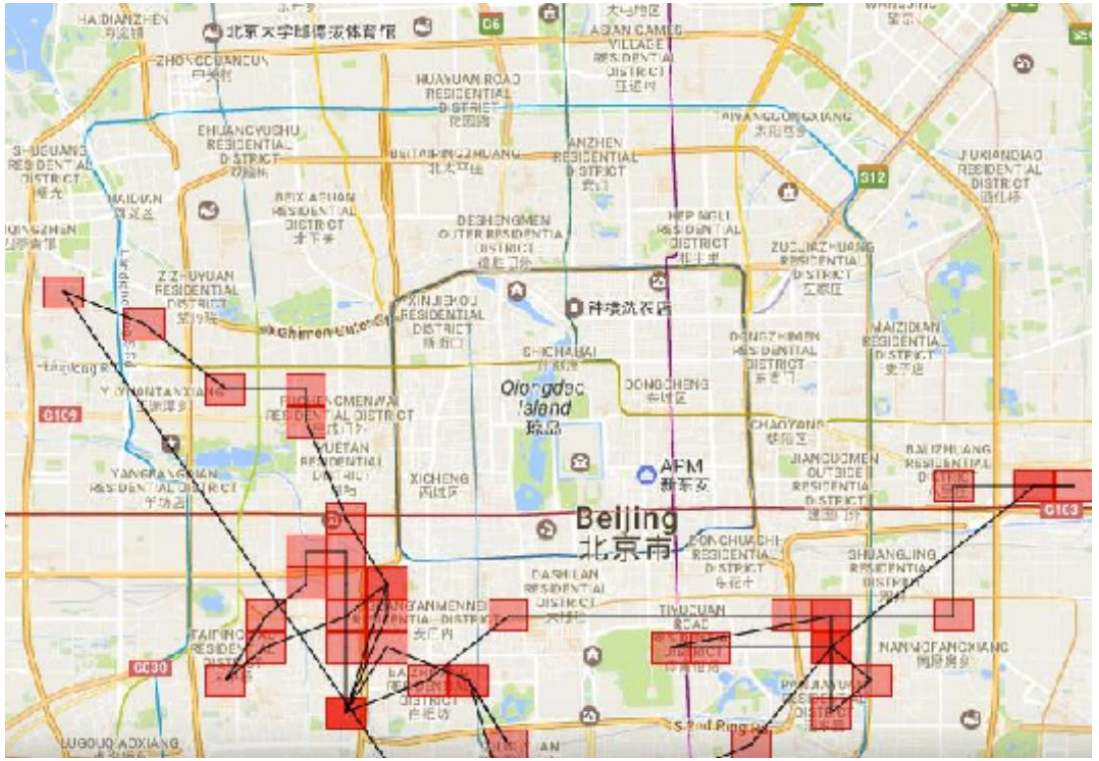




ANALYSE DE CHEMINS ET DE DISTANCES

La théorie des graphes offre des outils pour calculer les **distances** entre nœuds, trouver les chemins les plus courts et analyser les chemins critiques.

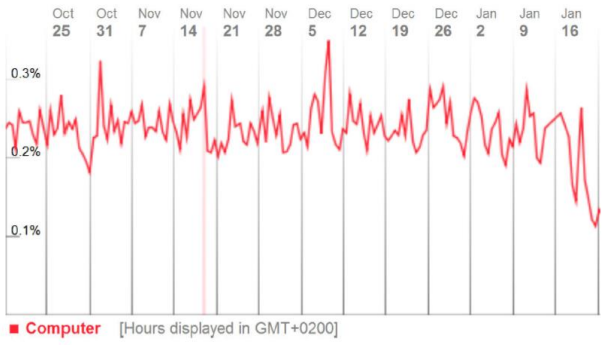
Cela est crucial pour des applications telles que la *navigation*, la *logistique* et la *recherche* d'itinéraires optimaux.

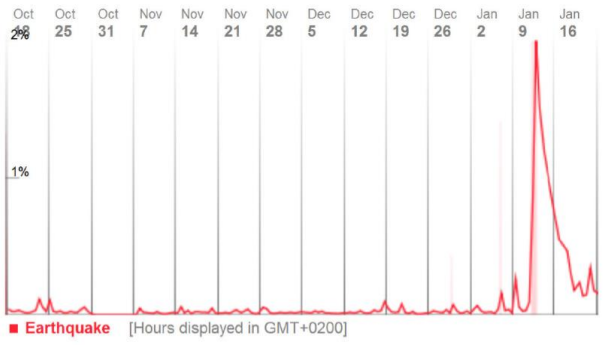


DÉTECTION D'ANOMALIES ET DE FRAUDES

Dans les données massives, il peut être difficile de repérer les **anomalies** ou les **activités frauduleuses**.

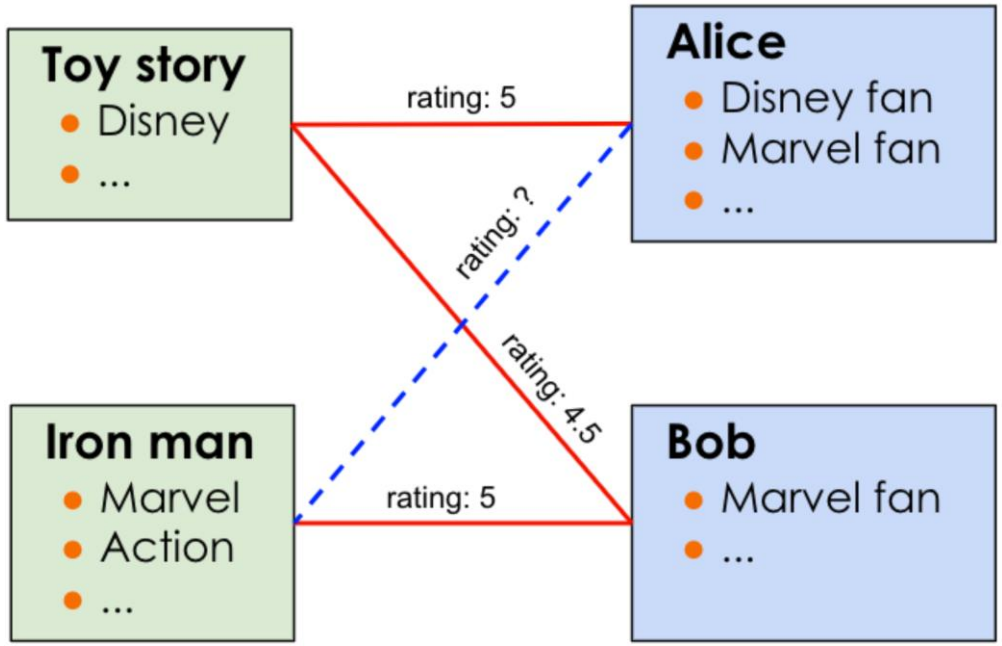
Les graphes peuvent aider à modéliser les schémas de comportement normaux et à identifier les déviations, facilitant ainsi la détection précoce de comportements suspects.





SYSTÈMES DE RECOMMANDATION

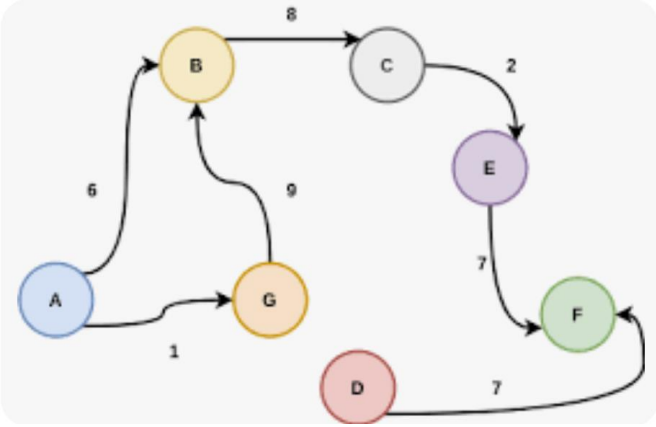
De nombreux **systèmes de recommandation** reposent sur des modèles de graphes pour suggérer des produits, des amis, des contenus ou des connexions potentielles aux utilisateurs en fonction de leurs préférences et de leurs comportements passés.

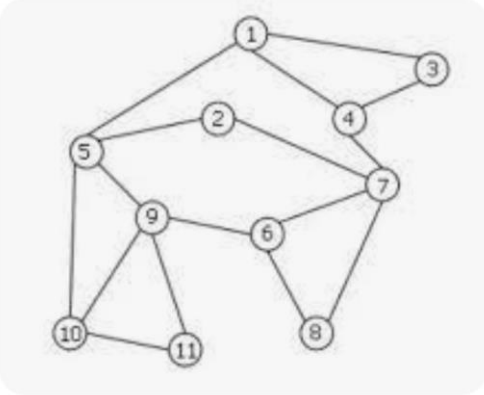


GRAPHE

Un graphe est une structure mathématique composée d'un ensemble de **sommets** (ou *nœuds*) reliés par des **arêtes**.

Les graphes sont utilisés pour modéliser des relations entre des entités ou des objets. Ils peuvent être *dirigés* (lorsque les arêtes ont une direction) ou *non dirigés* (lorsqu'elles n'ont pas de direction). Les arêtes peuvent également avoir des **poids** pour représenter des mesures telles que la distance, le coût ou la force de connexion entre les sommets.

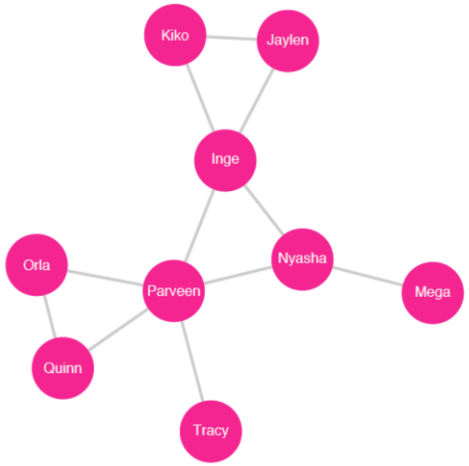


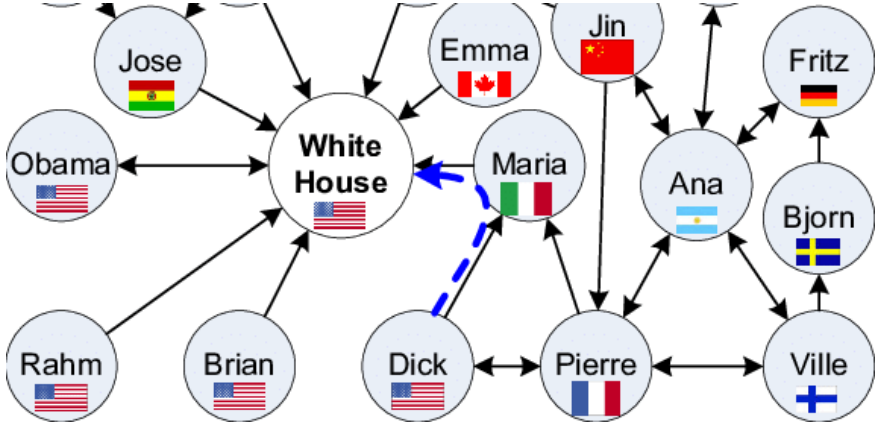


GRAPHE

Un réseau social peut être représenté sous forme de graphe en utilisant *les sommets pour représenter les individus* et *les arêtes pour indiquer les liens* entre eux.

Dans un réseau social non dirigé, les amitiés ou les connexions entre les personnes sont représentées par des arêtes non dirigées. Chaque individu est un sommet et chaque amitié est une arête reliant les sommets correspondants.
Les réseaux sociaux dirigés peuvent également être représentés, où l'amitié ou la connexion est unidirectionnelle, indiquant une asymétrie dans la relation.





GRAPHE

Un graphe

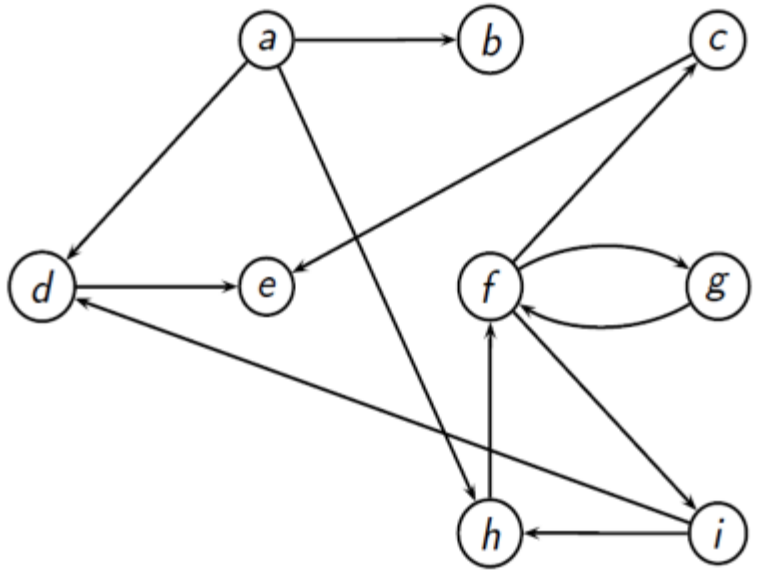
G= {V, E}

est composé par des sommets (ou *vertex*, en anglais). Etiquetés avec noms différents.

V= {a,b,c,d,e,f,g,h,i}

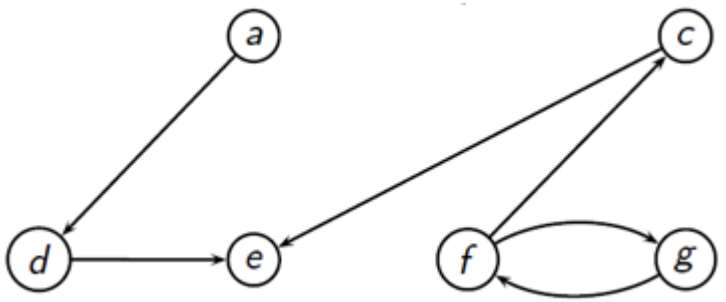
Arêtes (*edges* en aglais) normalement non étiquetés (mais il est possible), potentiellement avec poids, qui peuvent être orientés ou pas.

E={(a,b),(a,d),(a,h),(c,e),(d,e),(f,g),(f,i),(g,h),(h,f),(i,h)}



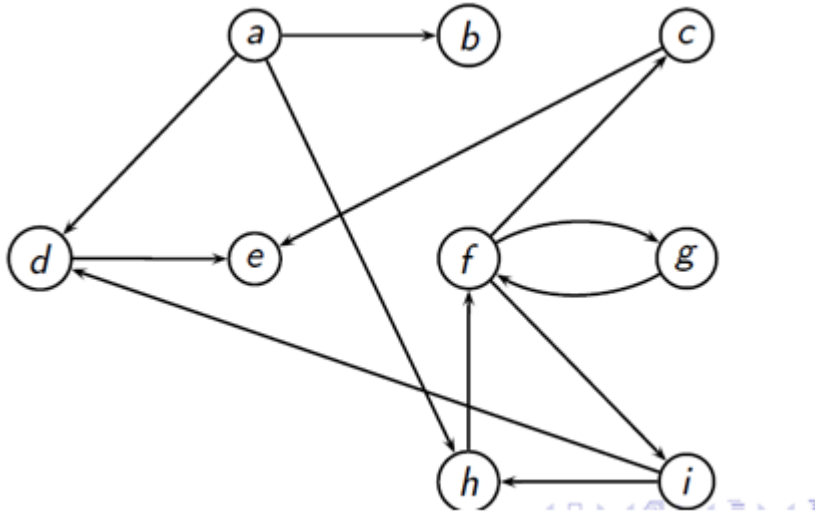
SOUS-GRAPHE

En théorie des graphes, un sous-graphe est un graphe contenu dans un autre graphe.



Formellement, un graphe $G_1=(V_1,E_1)$ est un sous-graphe de

G=(V,E) si $V_1 \subseteq V$ et $E_1 \subseteq \{ (x,y) \in E \mid x \in V \wedge y \in V\}$



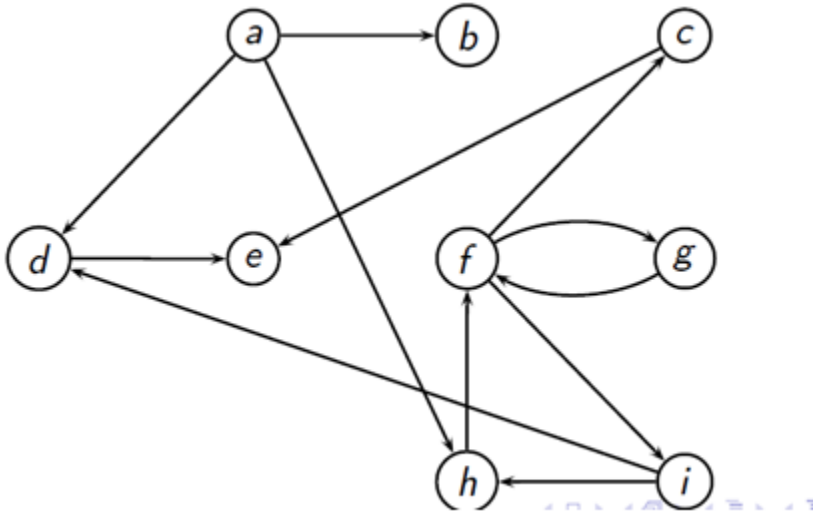
CHEMIN ET CIRCUIT

En théorie des graphes, Dans un graphe orienté, un chemin d'origine x et d'extrémité y est défini par une suite finie d'arcs consécutifs, reliant x à y

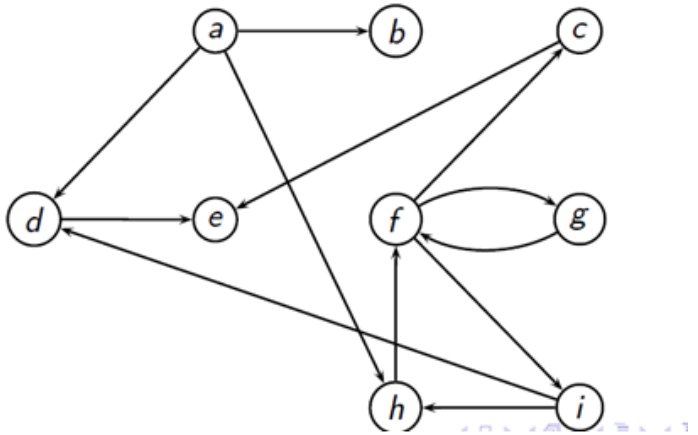
(a,h),(h,f),(f,g) est un chemin

On appelle circuit une suite d'arcs consécutifs (chemin) dont les deux sommets extrémités sont identiques.

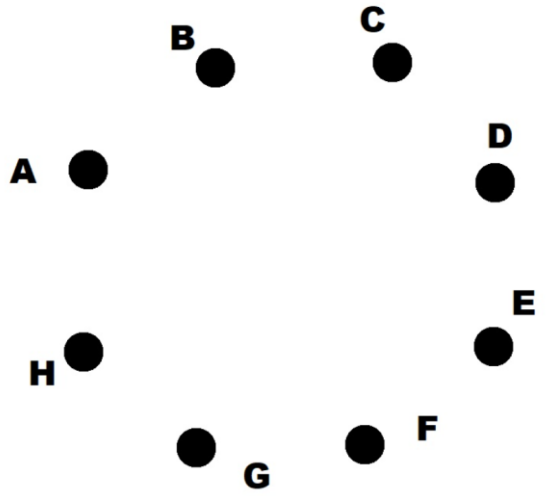
(f,i),(i,h),(h,f) est un cycle



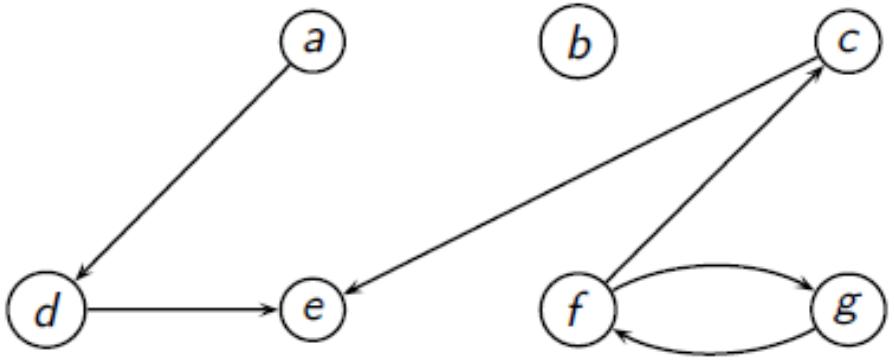
GRAPHE SIMPLE



Un graphe simple G = (V,E) est formé par un ensemble non vide de nœuds V et un ensemble (même vide) d'arêtes, où chaque arête est forme par un sous-ensemble de nœuds de cardinalité 2.



MATRICE D'ADJACENCE

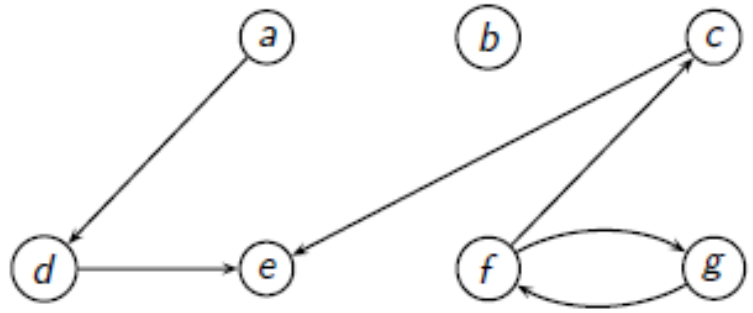


En théorie des graphes une matrice d'adjacence pour un graphe à ***n*** sommets est une matrice de dimension $n_x n$ dont l'élément non diagonal a_{ij} représente l'éventuel arête liant le sommet ***i*** au sommet ***j***.

	a	b	c	d	e	f	g
a				×			
b							
c					×		
d					×		
e							
f			×				×
g						×	

IMPLEMENTATION D'UN GRAPHE

En informatique, le graphe peut être représenté à travers plusieurs solutions (*listes, hashsets, hashmaps*, etc.)



Mathematical presentation

$$\{ \quad (a,\{d\}) \\ (b,\{\}) \\ (c,\{e\}) \\ (d,\{e\}) \\ (e,\{\}) \\ (f,\{c,g\}) \\ (g,\{f\}) \ \}$$

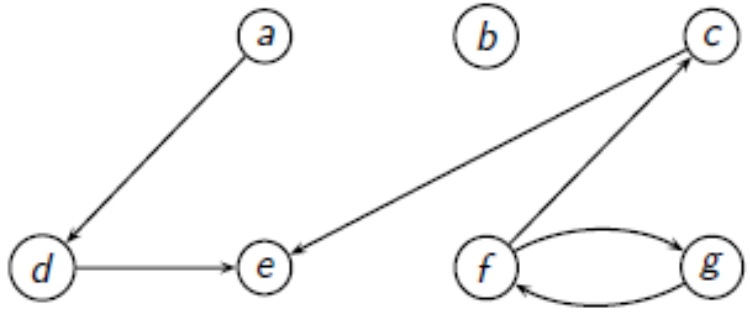
A Python presentation

```
{  'a' : [ 'd' ] ,
   'b' : [  ] ,
   'c' : [ 'e' ] ,
   'd' : [ 'e' ] ,
   'e' : [  ],
   'f' : [ 'c' , 'g'
   'g' : [ 'f' ] }
```

PARCOURS DE GRAPHE

Un parcours de graphe est un algorithme consistant à explorer les sommets d'un graphe de proche en proche à partir d'un sommet initial.

Il existe de nombreux algorithmes de parcours. Les plus couramment décrits sont le parcours **en profondeur** et le parcours **en largeur.**
L'algorithme de ***Dijkstra*** et l'algorithme de *Prim* font également partie de cette catégorie.



Mathematical presentation

$$\{ \ (a,\{d\}) \ (b,\{\}) \ (c,\{e\}) \ (d,\{e\}) \ (e,\{\}) \ (f,\{c,g\}) \ (g,\{f\}) \ \}$$

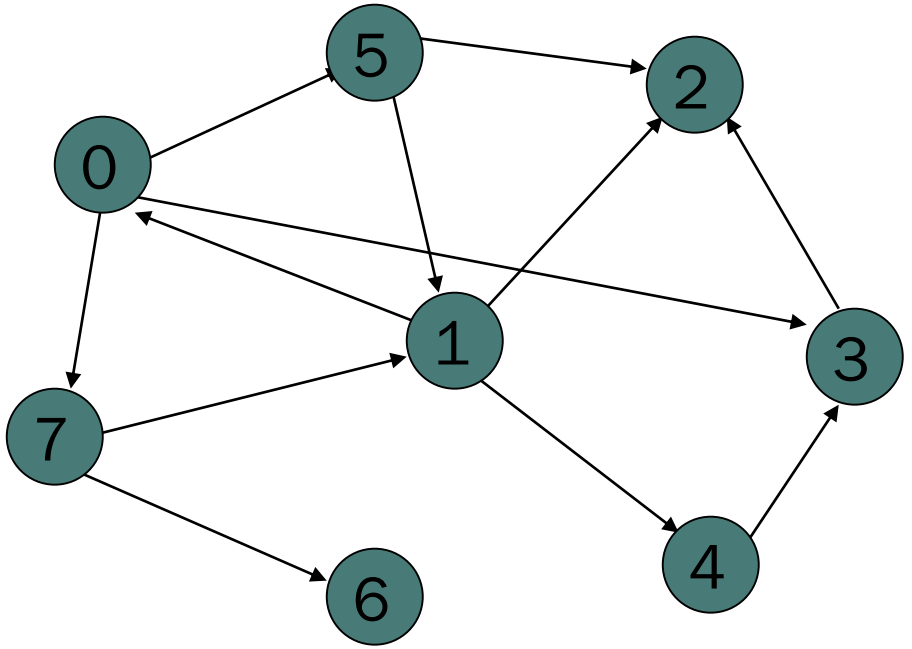
A Python presentation

```
{  'a' : [ 'd' ] ,
   'b' : [  ] ,
   'c' : [ 'e' ] ,
   'd' : [ 'e' ] ,
   'e' : [  ],
   'f' : [ 'c' , 'g'
   'g' : [ 'f' ] }
```

PARCOURS DE GRAPHE

Quel nœud choisir pour démarrer le parcours d'un graphe? A priori peu importe, mais en réalité il est préférable choisir un nœud qui n'est destination d'aucun arête.

5 *par exemple, est un bon point de départ pour une visite.*



VISITE EN PROFONDEUR

L'algorithme de parcours en profondeur (ou parcours en profondeur, ou *DFS*, pour *Depth-First Search*) est un algorithme de parcours d'arbre, et plus généralement de parcours de graphe. Il se décrit naturellement de manière récursive. S

L'exploration d'un parcours en profondeur depuis un sommet S fonctionne comme suit. Il poursuit alors un chemin dans le graphe jusqu'à un cul-de-sac ou alors jusqu'à atteindre un sommet déjà visité. Il revient alors sur le dernier sommet où on pouvait suivre un autre chemin puis explore un autre chemin.

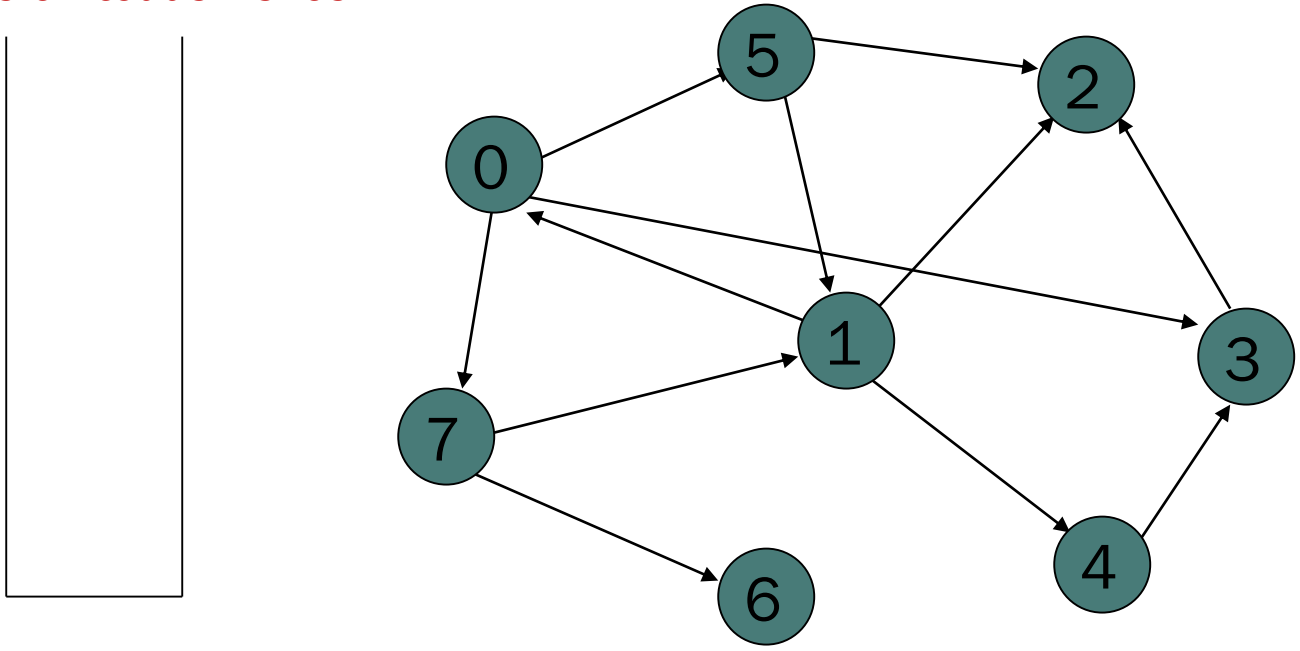
il est possible de l'implémenter itérativement à l'aide d'une pile contenant les sommets à explorer.

```
explorer(graphe G, sommet s)
      marquer le sommet s
      afficher(s)
      pour tout sommet t voisin du sommet s
            si t n'est pas marqué alors
                  explorer(G, t);
```

```
parcoursProfondeur(graphe G)
      pour tout sommet s du graphe G
            si s n'est pas marqué alors
                  explorer(G, s)
```

VISITE EN PROFONDEUR - EXEMPLE

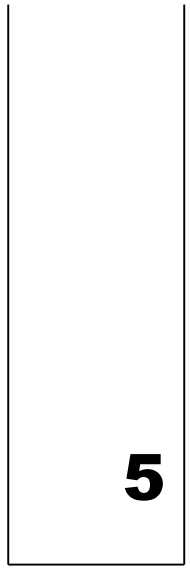
Liste nœuds visités

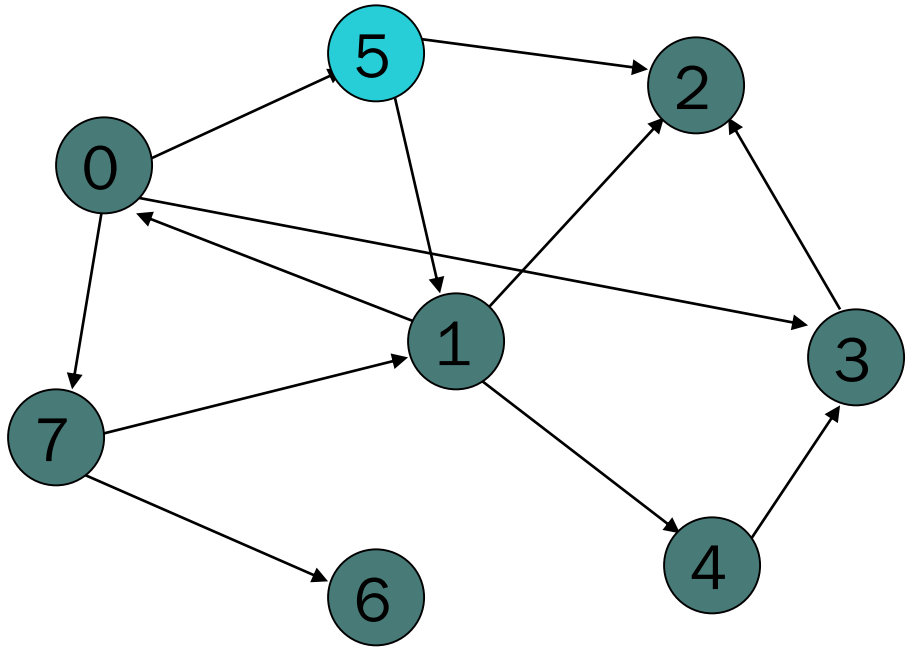


Démarrons par le nœud 5, pointé par aucun arête.

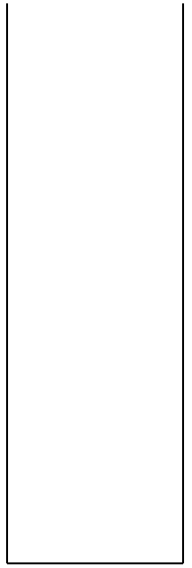
VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités





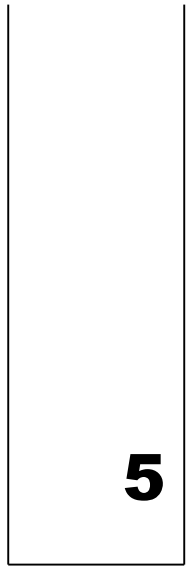
Pile

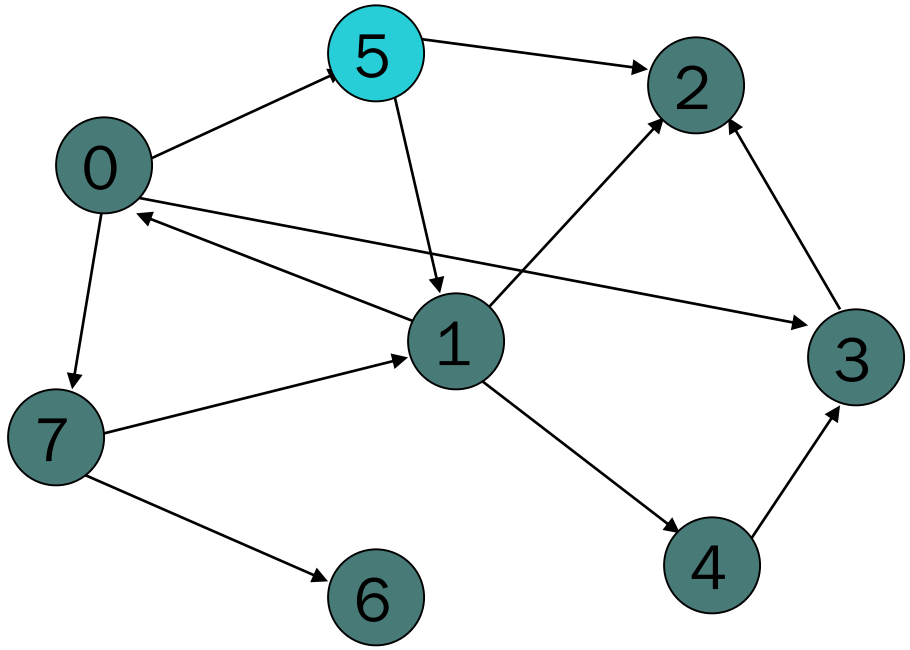


5 est visité et du coup enlevé de la pile.

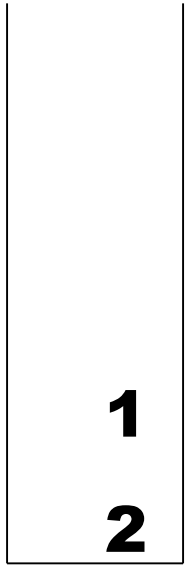
VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités





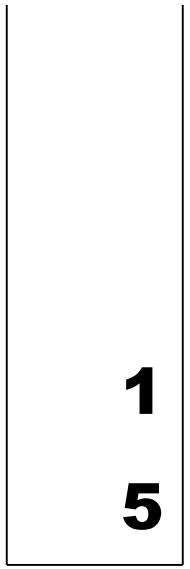
Pile

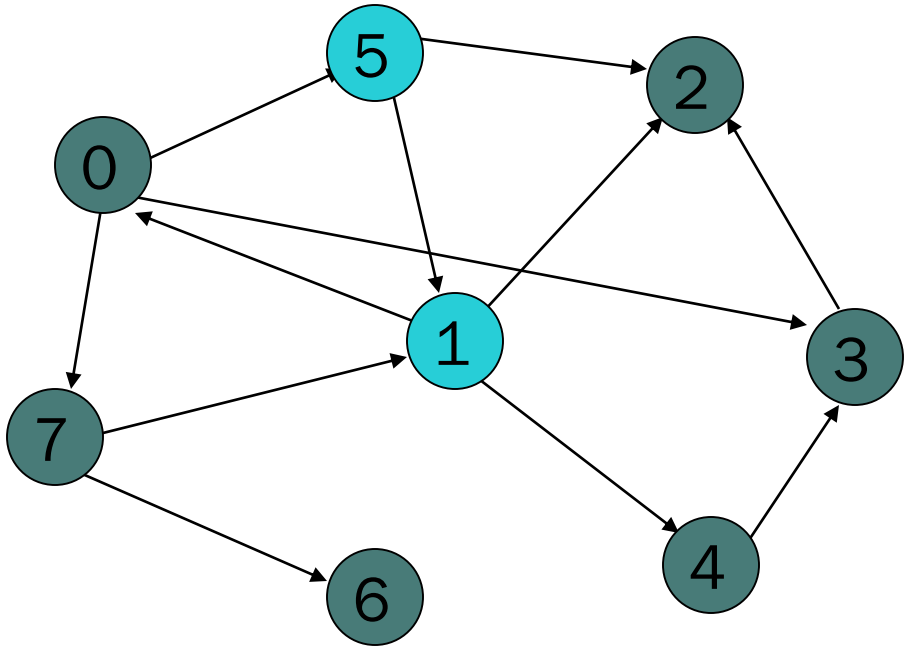


Insertion de 2 et 1, adjacents à 5

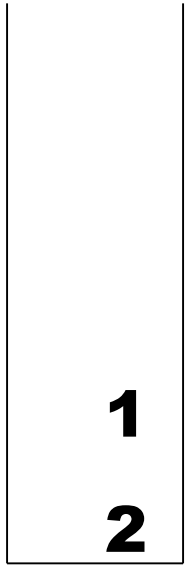
VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités





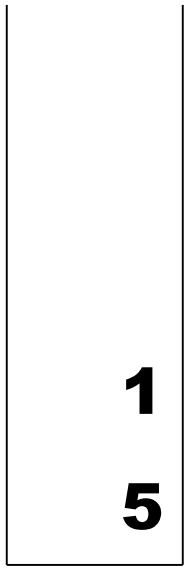
Pile

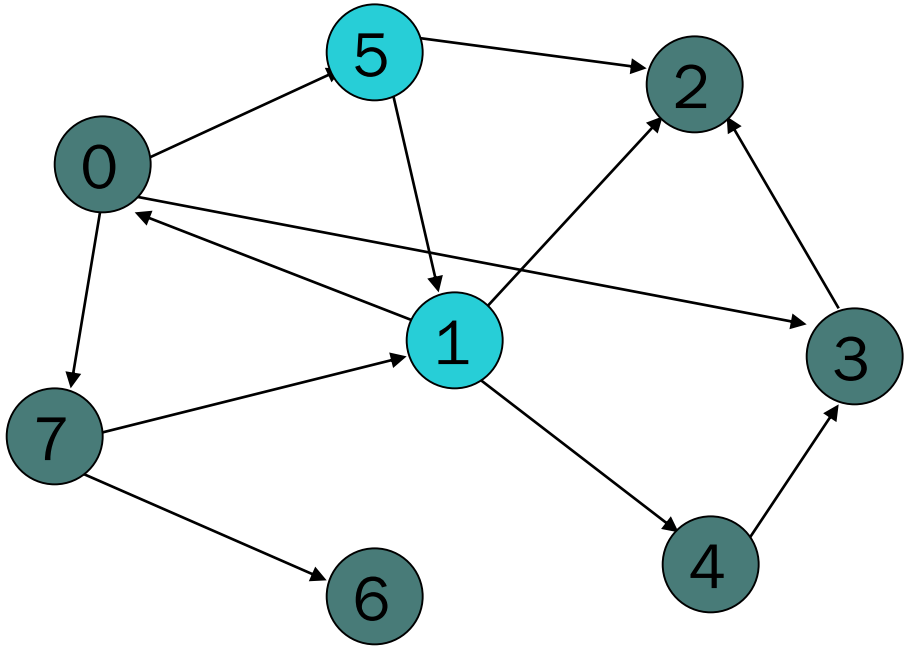


On visiter alors l'élément en tête de la pile, 1

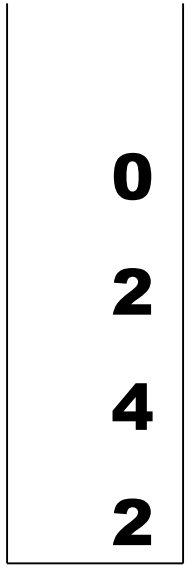
VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités





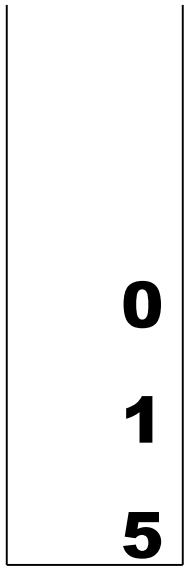
Pile

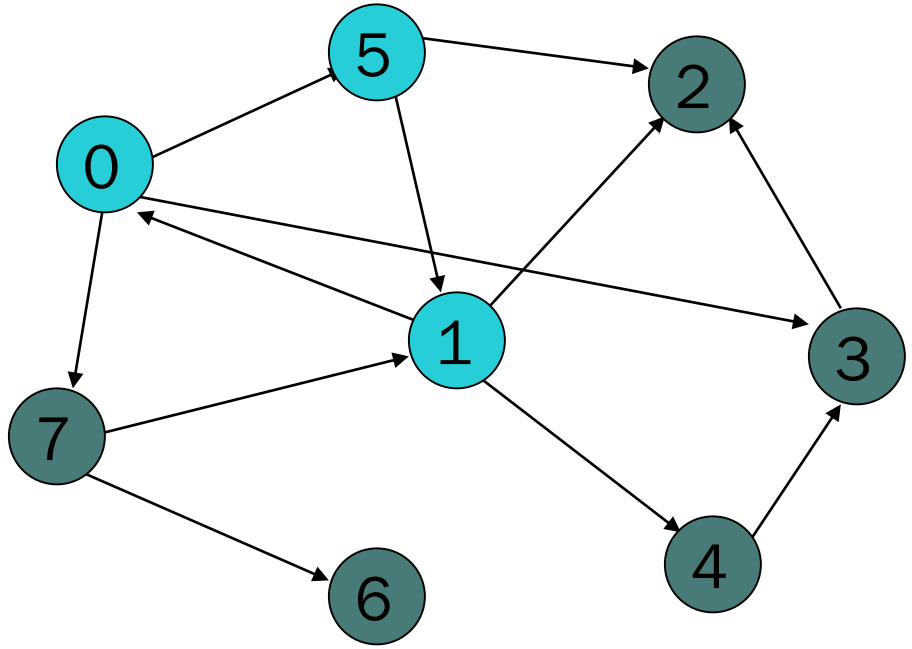


Insertion de 2, 4 et 0, adjacents à 1

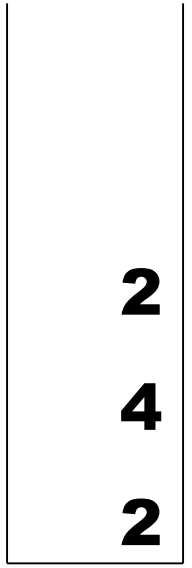
VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités





Pile

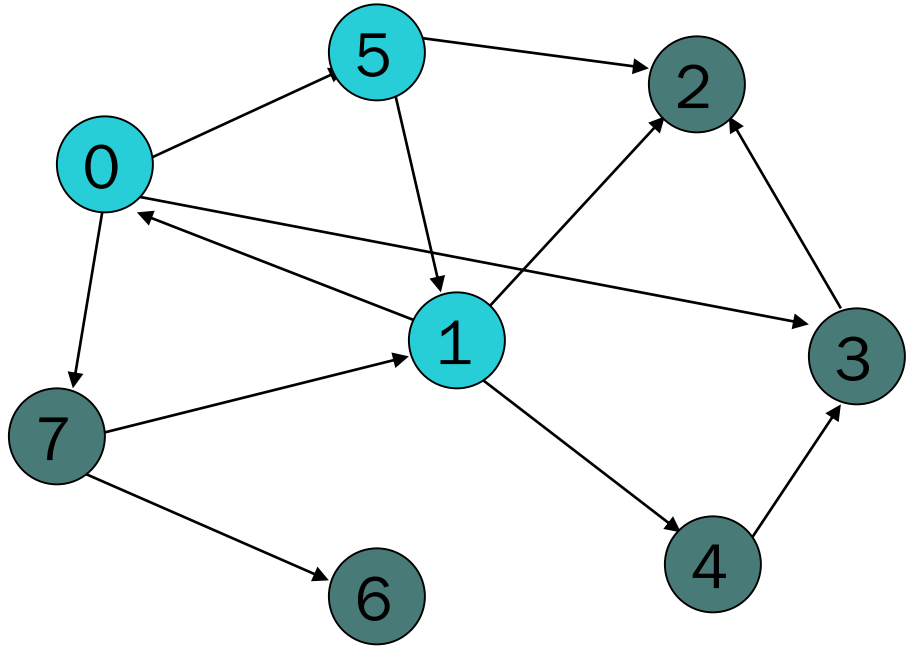


On visite alors 0

VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités

0
1
5



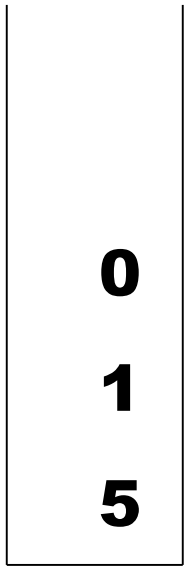
Pile

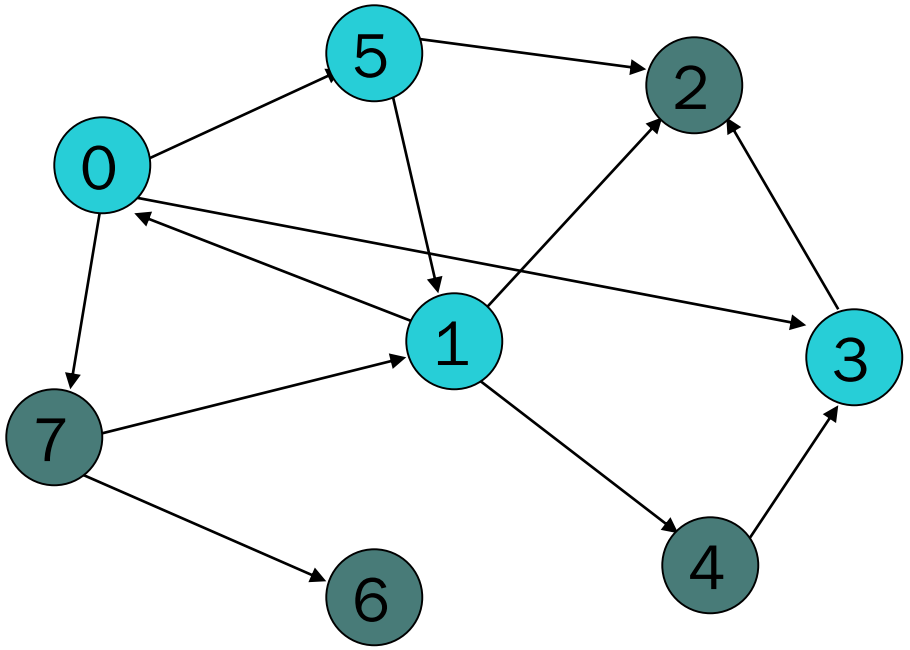
3
7
2
4
2

Insértion de 3 et 7, adjacents de 0

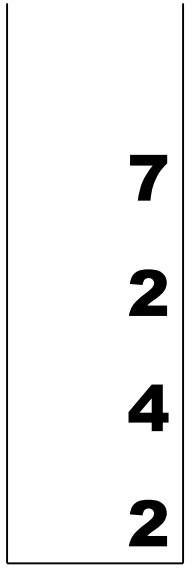
VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités





Pile

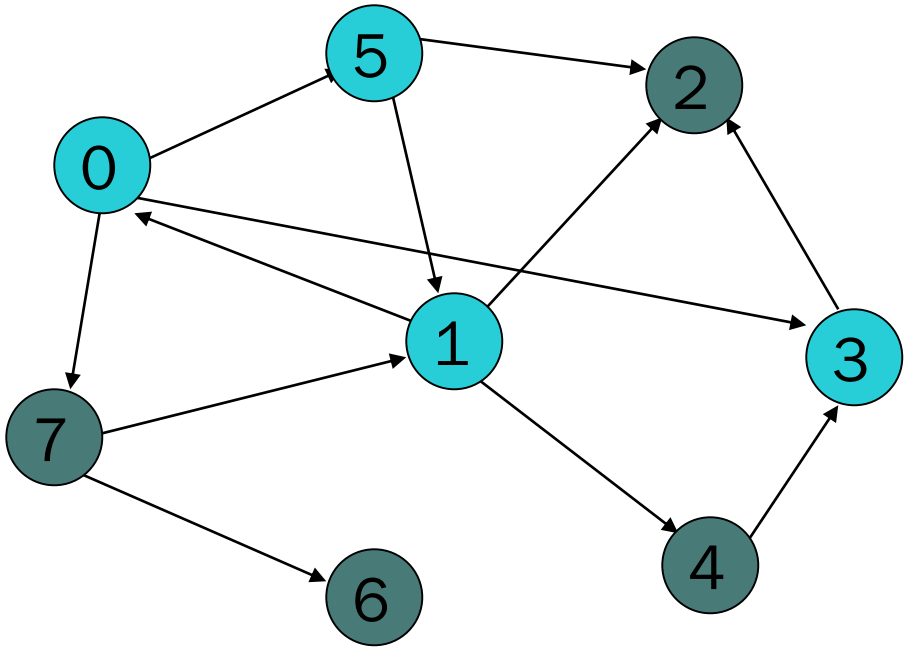


Visite de 3

VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités

3
0
1
5



Pile

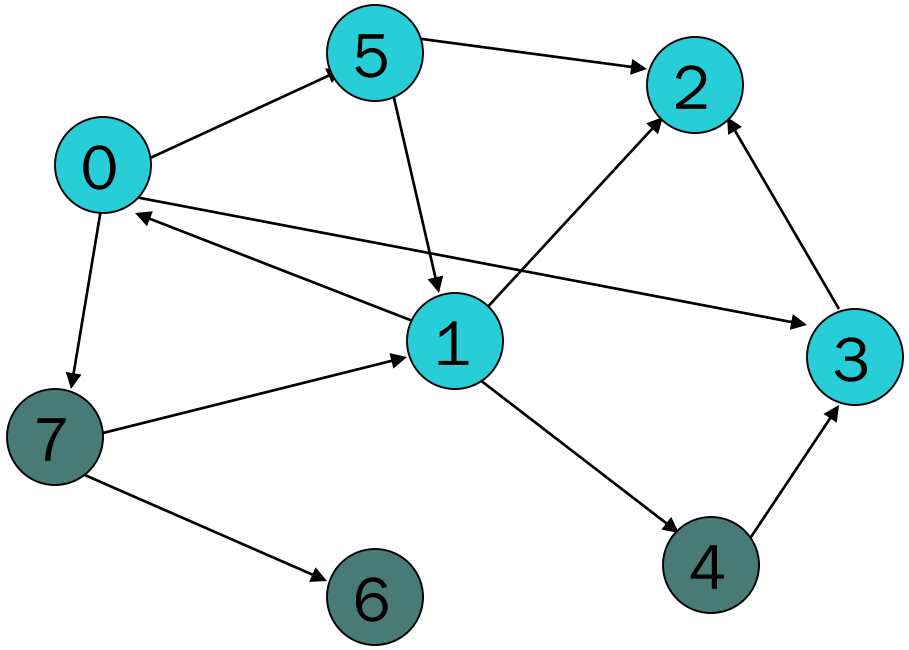
2
7
2
4
2

Insertion de 2, adjacents de 3

VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités

3
0
1
5



Pile

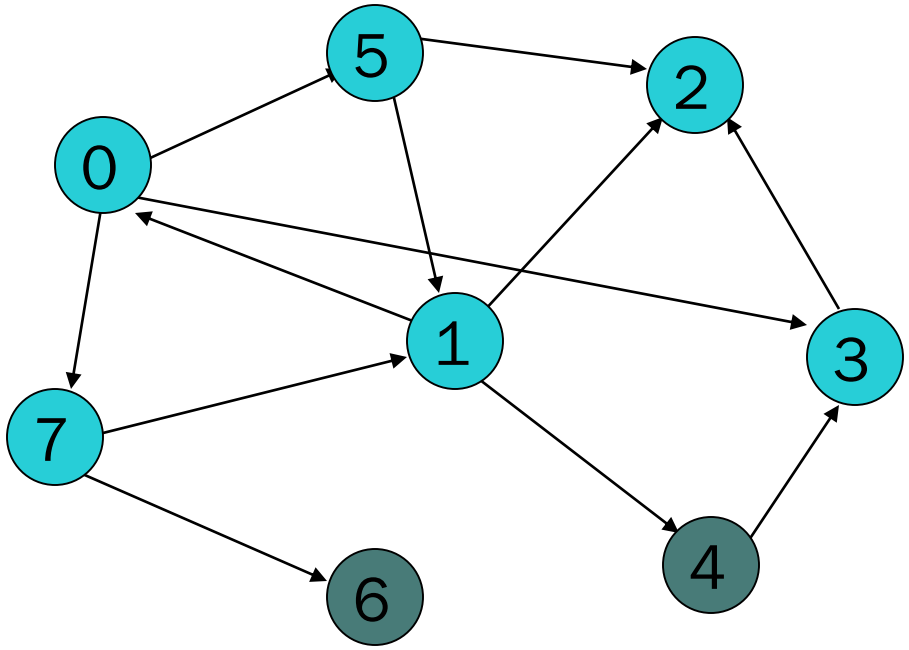
7
2
4
2

Visite du nœud 2

VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités

7
3
0
1
5



Pile

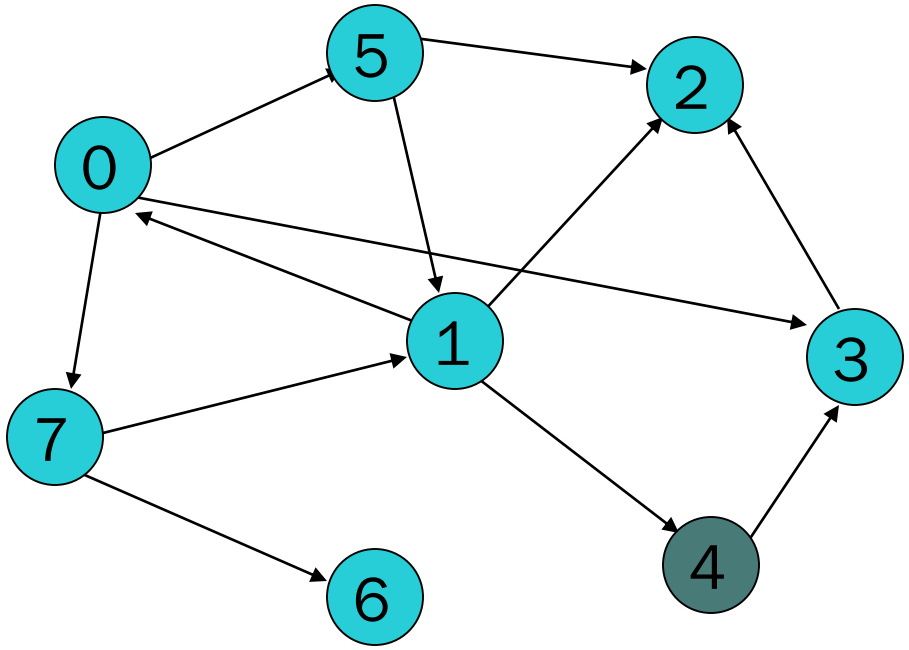
6
2
4
2

On ajoute alors 6, voisin de 7

VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités

6
7
3
0
1
5



Pile

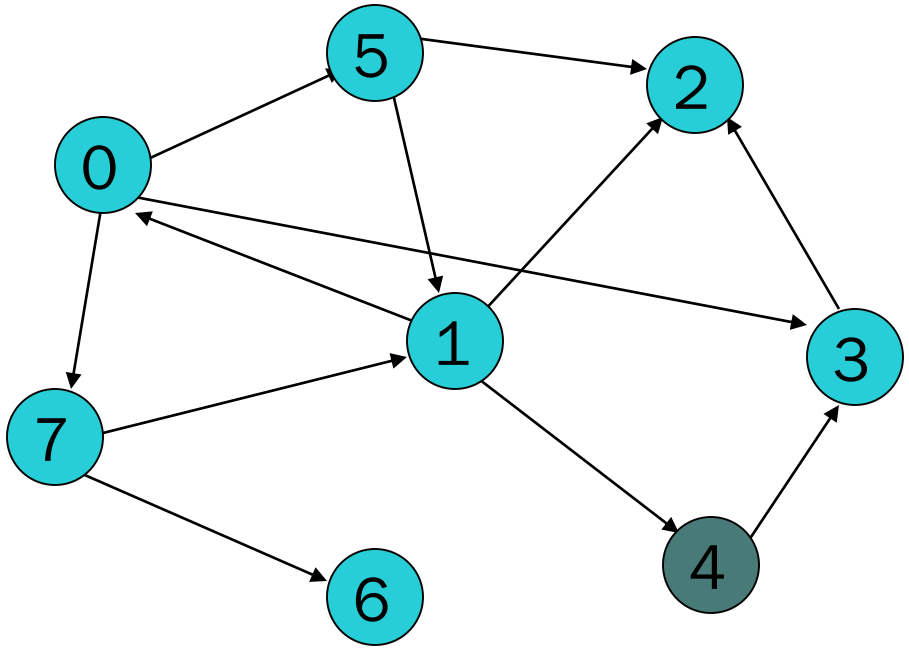
2
4
2

Aucun voisin de 6

VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités

6
7
3
0
1
5



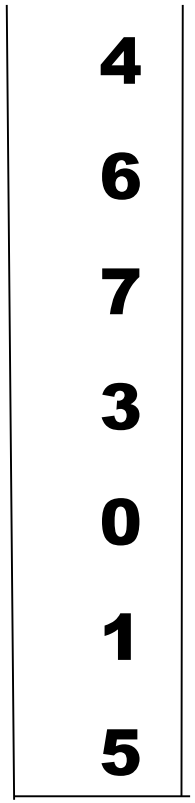
Pile

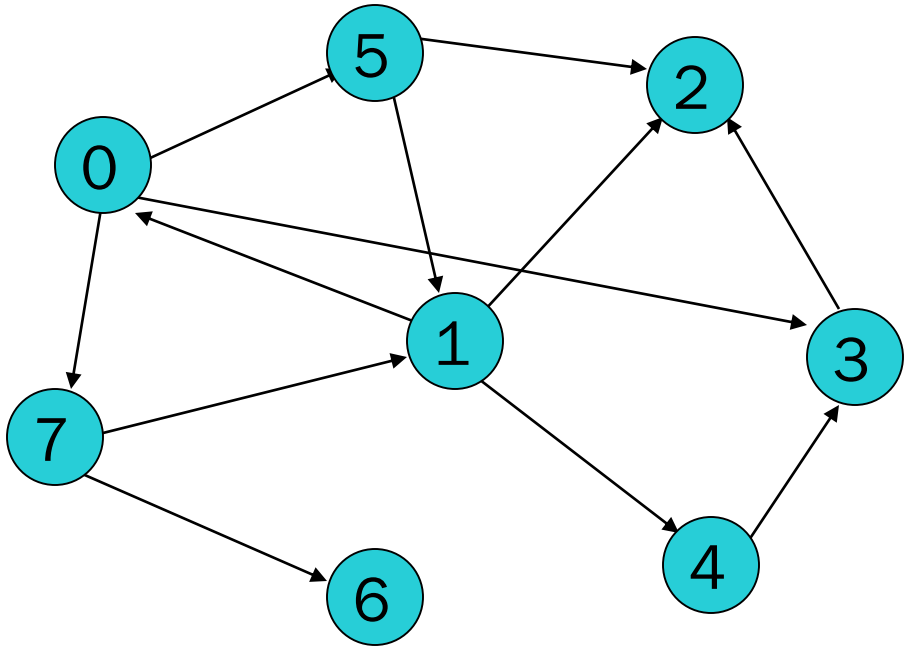
4
2

Aucun voisin de 6, mais le premier noeud à visiter a été déjà visité. Du coup on supprime

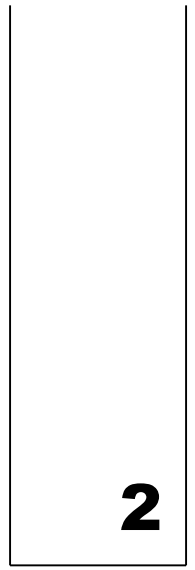
VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités





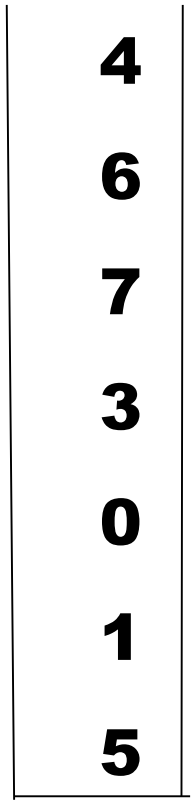
Pile

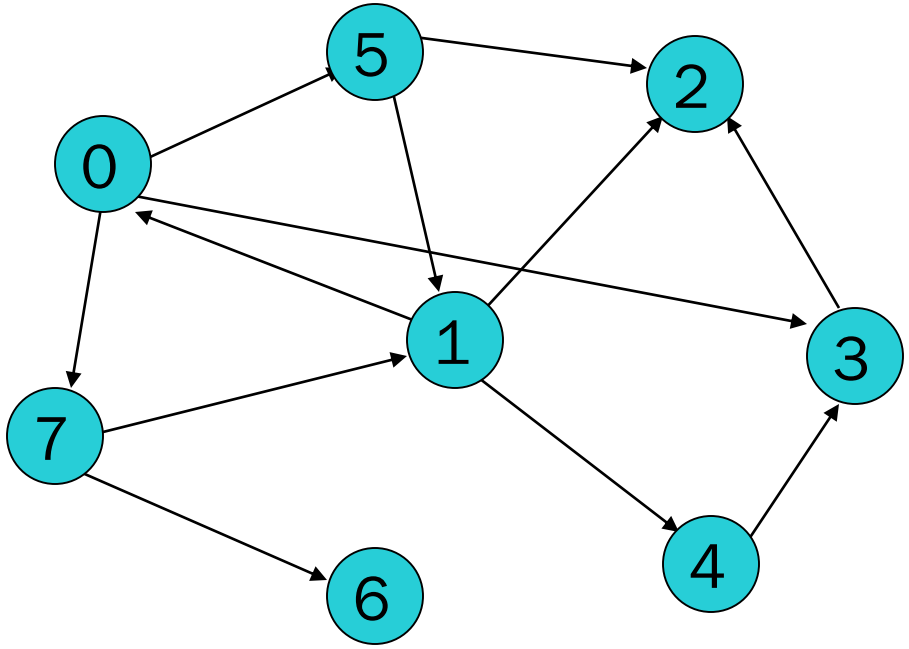


On visite 4

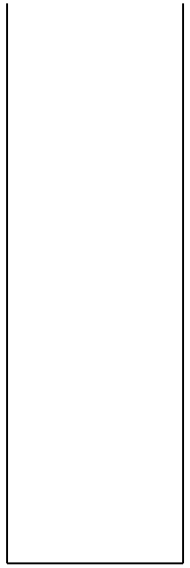
VISITE EN PROFONDEUR - EXEMPLE

Liste nœuds visités





Pile



2 est supprimé de la pile, en étant déjà visité. Visite terminée

VISITE EN LARGEUR

L'algorithme de parcours en largeur (ou **BFS**, pour *Breadth-First Search* en anglais) permet le parcours d'un graphe ou d'un arbre de la manière suivante : on commence par explorer un nœud source, puis ses successeurs, puis les successeurs non explorés des successeurs, etc.

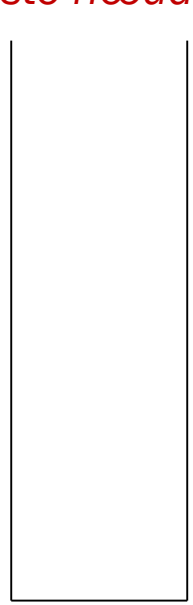
L'algorithme de parcours en largeur permet de calculer les ***distances*** de tous les nœuds depuis un nœud source dans un graphe non pondéré (orienté ou non orienté). Il peut aussi servir à déterminer si un graphe non orienté est ***connexe***.

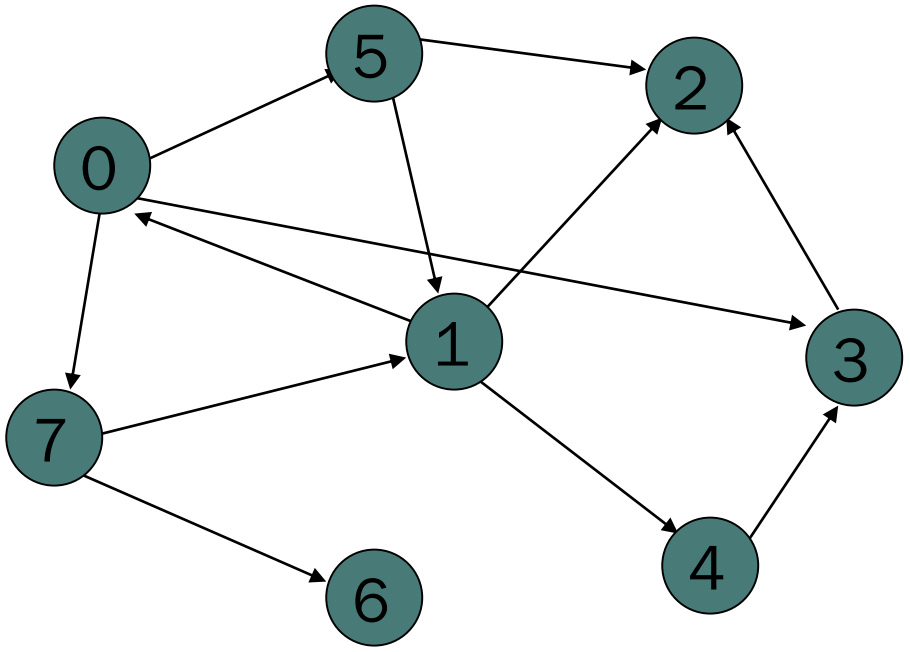
```
explorer(graphe G, sommet s)
      marquer le sommet s
      afficher(s)
      pour tout sommet t voisin du sommet s
            si t n'est pas marqué alors
                  explorer(G, t);
```

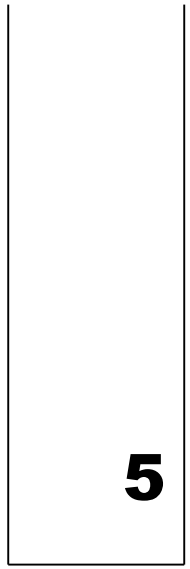
```
parcoursProfondeur(graphe G)
      pour tout sommet s du graphe G
            si s n'est pas marqué alors
                  explorer(G, s)
```

VISITE EN LARGEUR - EXEMPLE

Liste nœuds visités





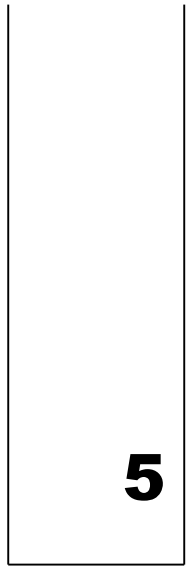


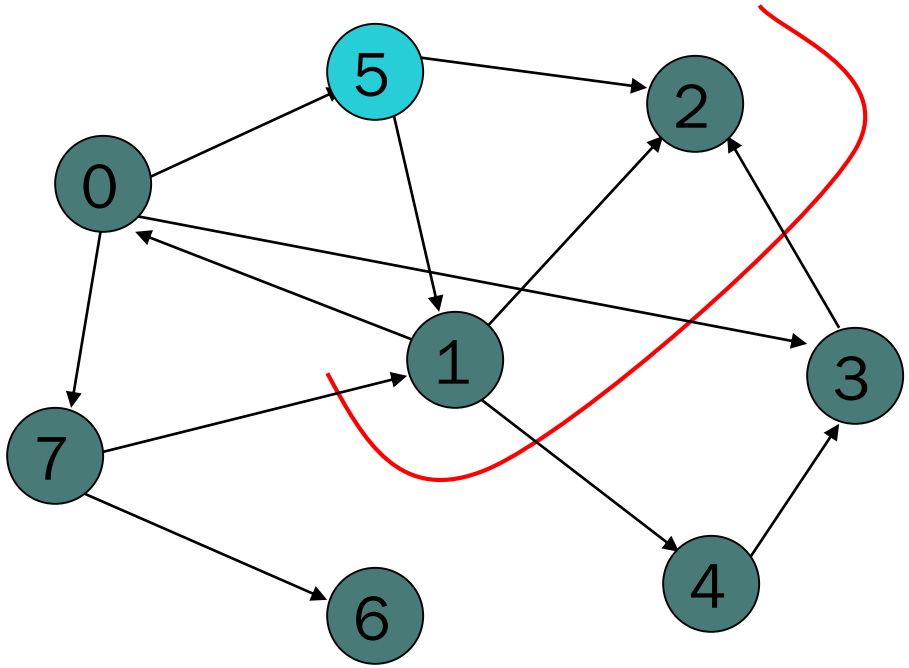
LISTE! (on récupère toujours le premier élément!)

Démarrons par le nœud 5, pointé par aucun arête.

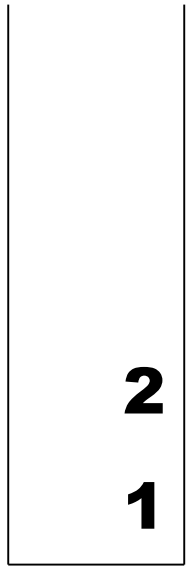
VISITE EN LARGEUR - EXEMPLE

Liste nœuds visités





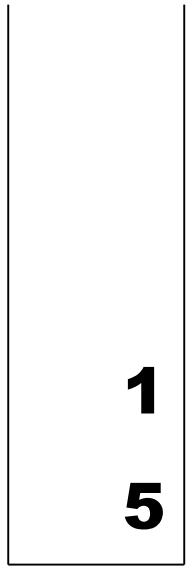
LISTE!

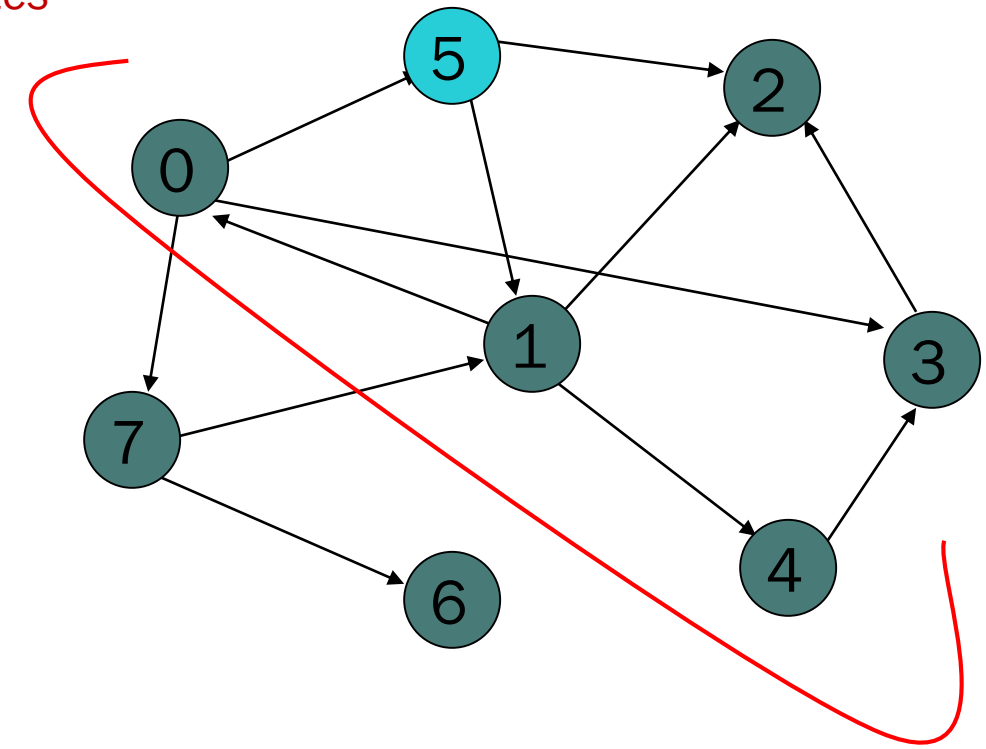


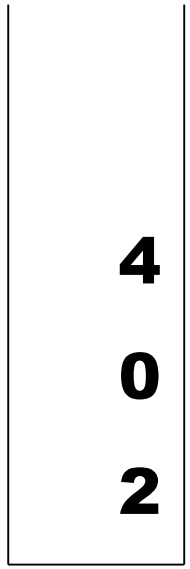
Démarrons par le nœud 5, et puis on ajoute dans la liste tous les nœuds atteignables par 5

VISITE EN LARGEUR - EXEMPLE

Liste nœuds visités



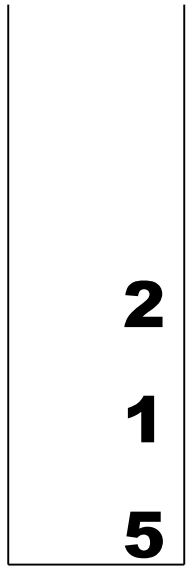


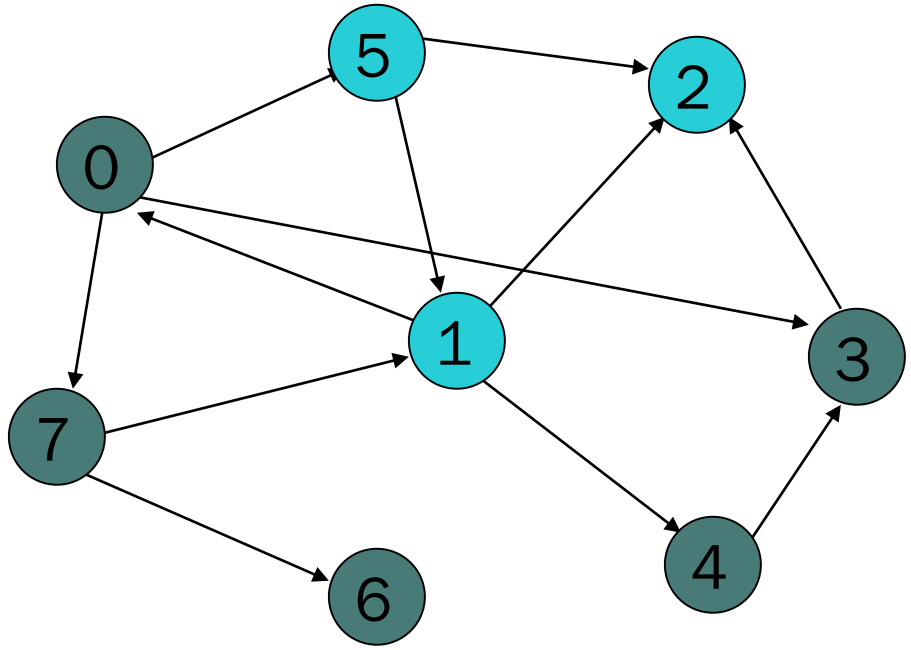


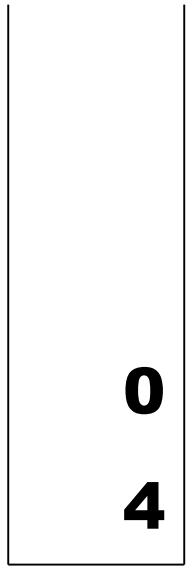
J'insère au fond de la liste les voisins de 0

VISITE EN LARGEUR - EXEMPLE

Liste nœuds visités



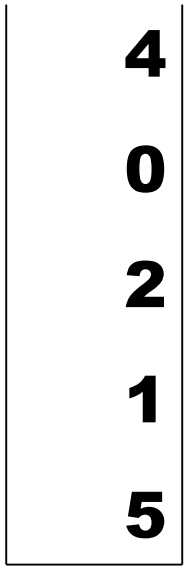


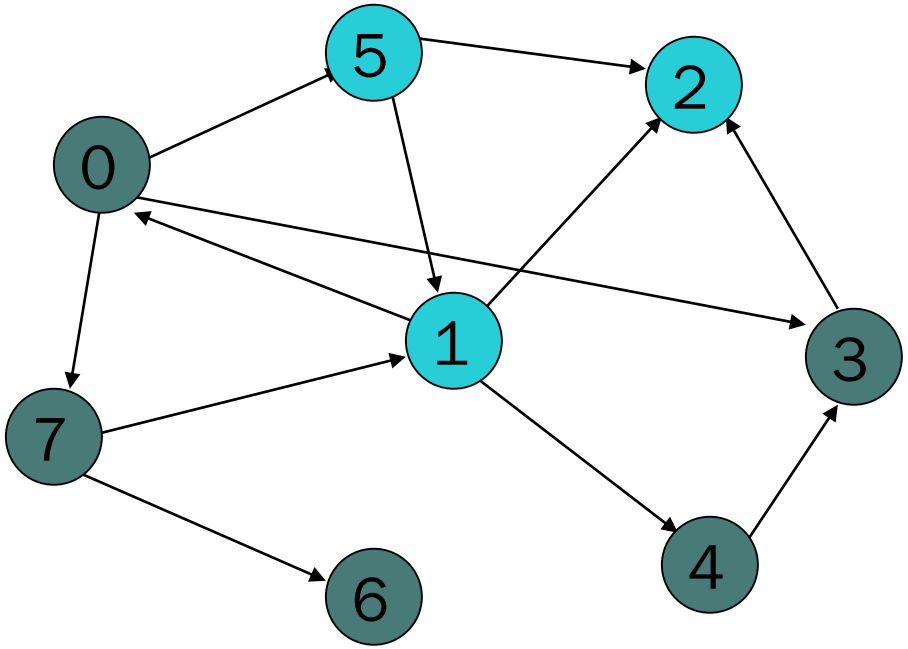


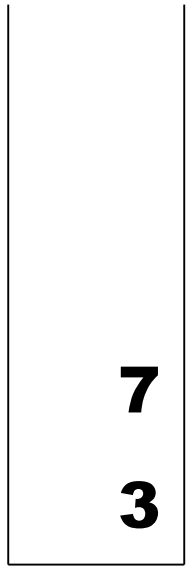
Une fois visité 1 et 2, je visiterai les voisins du 1 et, à chaque étape, j'ajoute au fond de la liste les voisins de chaque nœud visité.

VISITE EN LARGEUR - EXEMPLE

Liste nœuds visités





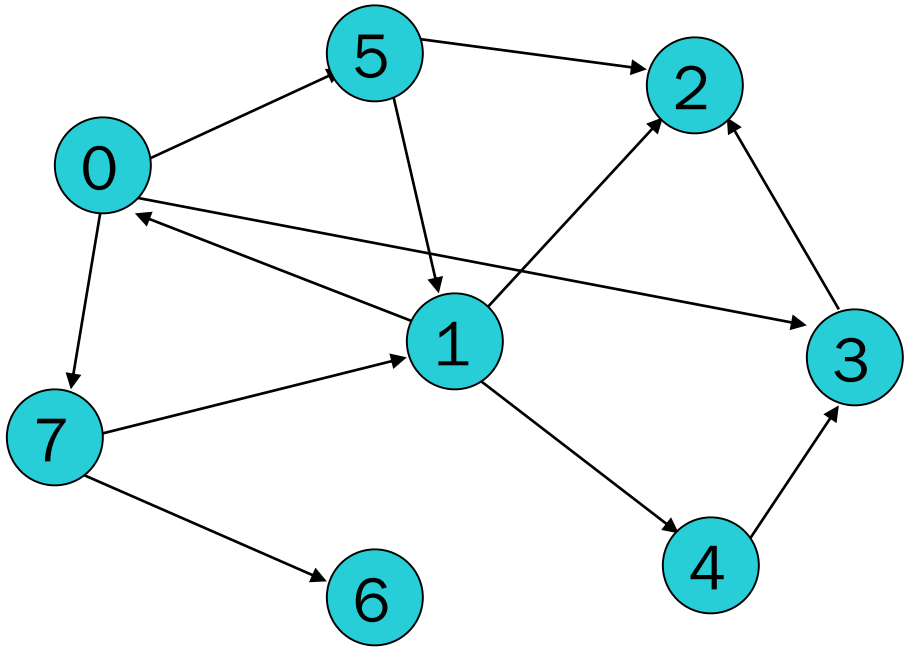


Puis on passe au niveau succèsif..etc etc.

VISITE EN LARGEUR - EXEMPLE

Liste nœuds visités

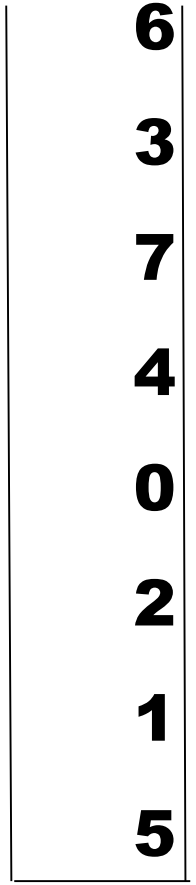
3 7 4 0 2 1 5

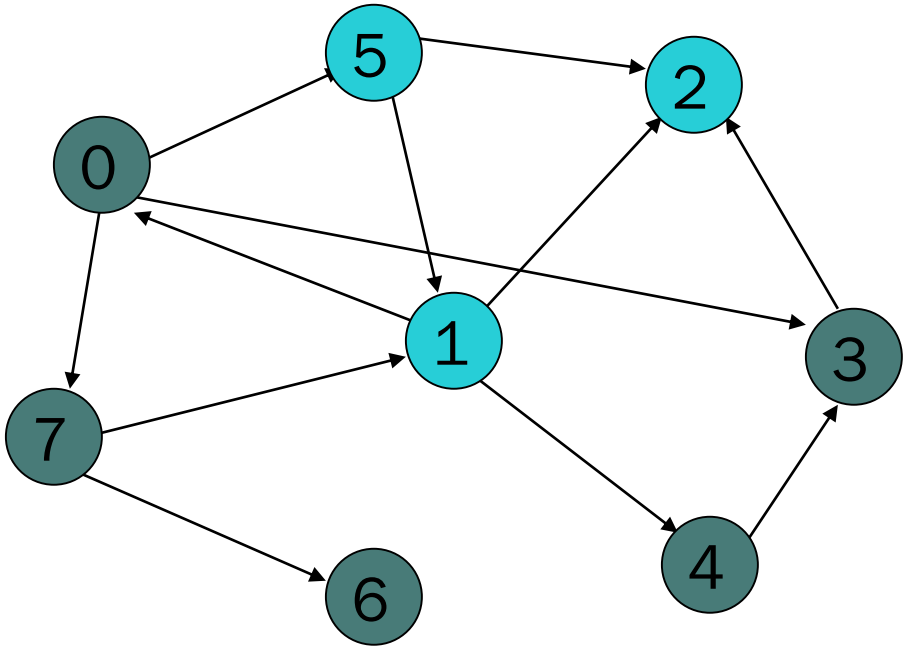


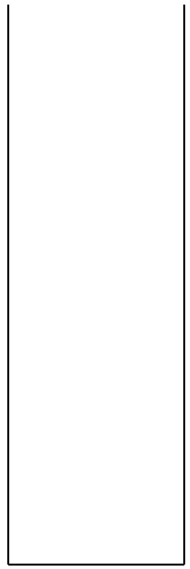
6

VISITE EN LARGEUR - EXEMPLE

Liste nœuds visités







PROBLEMES CLASSIQUES LIES AUX GRAPHES

Le problème du plus court chemin : Trouver le chemin le plus court entre deux sommets d'un graphe pondéré (en minimisant la somme des poids des arêtes parcourues). Le plus célèbre algorithme pour résoudre ce problème est l'algorithme de Dijkstra.

Le problème du cycle hamiltonien : Trouver un cycle hamiltonien dans un graphe, c'est-à-dire un cycle qui visite chaque sommet exactement une fois et revient au sommet de départ.

Le problème du voyageur de commerce (TSP) : Trouver le chemin le plus court qui visite un ensemble donné de sommets exactement une fois et revient au sommet de départ.

Le problème de la planarité : Déterminer si un graphe peut être dessiné dans le plan sans que ses arêtes ne se croisent

LE PLUS COURT CHEMIN - DIJKSTRA

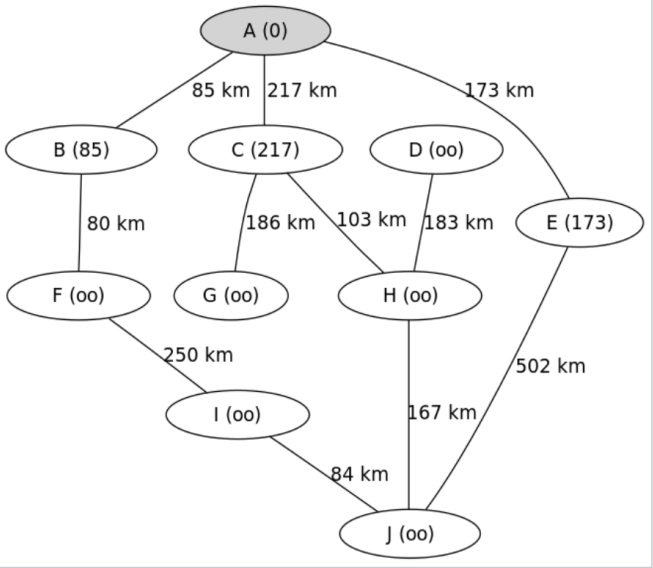
Algorithme de **Dijkstra**. **Il fonctionne sur des graphes orientés et non-orientés.**

- Il maintient une liste de nœuds non visités et une liste de distances minimales à partir du nœud source jusqu'à chaque nœud.
- À chaque étape, il sélectionne le nœud non visité avec la distance minimale connue, puis met à jour les distances des nœuds adjacents si un chemin plus court est trouvé.
- L'algorithme se termine lorsque tous les nœuds ont été visités ou lorsque le nœud de destination est atteint.

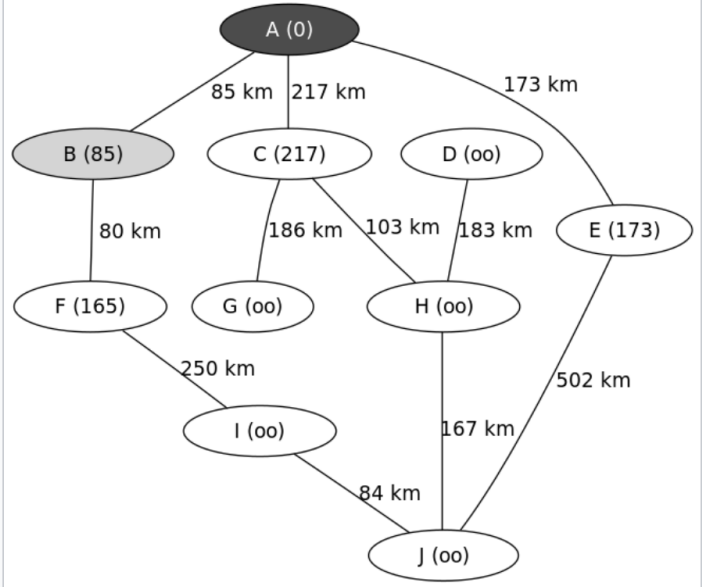
- Si le graphe n’est pas pondéré (pas de poids pour les arêtes), ça revient à effectuer une BFS.

DIJKSTRA - EXEMPLE

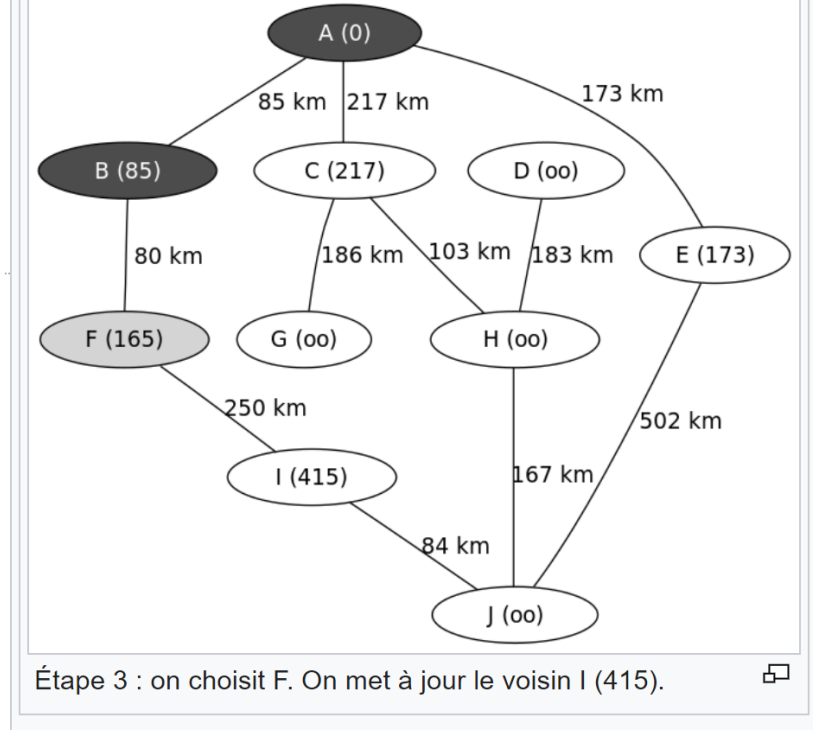
Source: wikipedia



Étape 1 : on choisit la ville A. On met à jour les villes voisines de A qui sont B, C, et E. Leurs distances deviennent respectivement 85, 217, 173, tandis que les autres villes restent à une distance infinie.



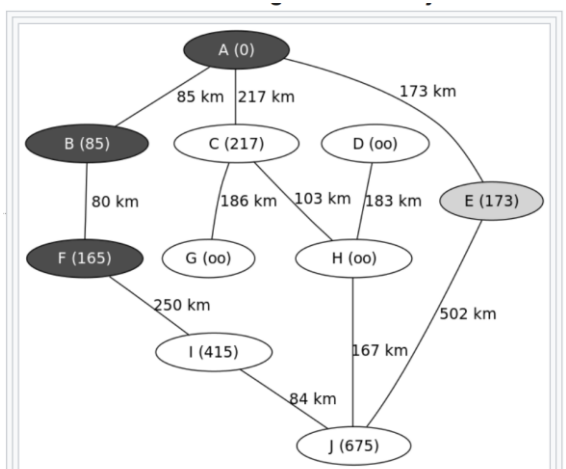
Étape 2 : on choisit la ville B. En effet, c'est la ville hors du sous-graphe qui est à la distance minimale (85). On met à jour le seul voisin (F). Sa distance devient 85+80 = 165.



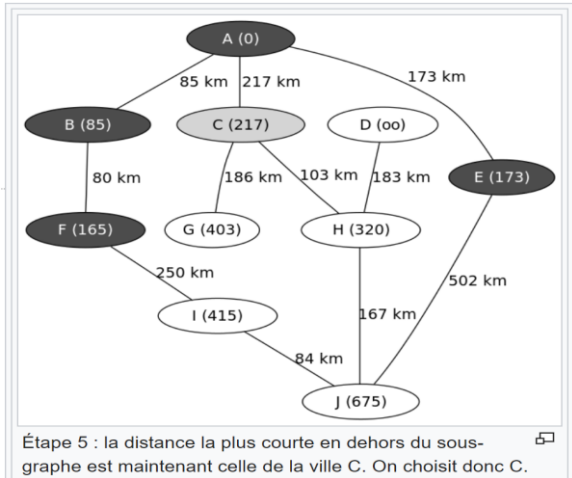
Étape 3 : on choisit F. On met à jour le voisin I (415).

DIJKSTRA - EXEMPLE

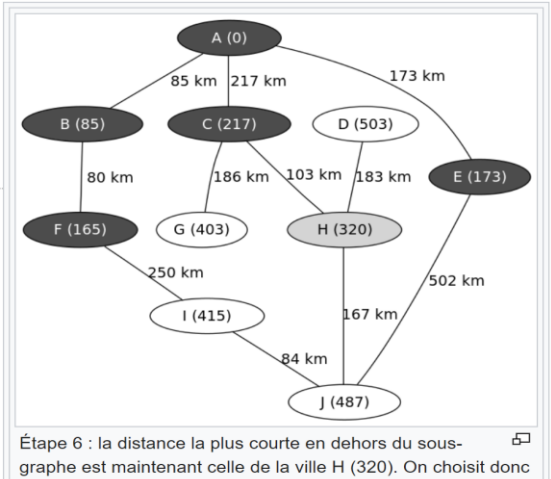
Source: wikipedia



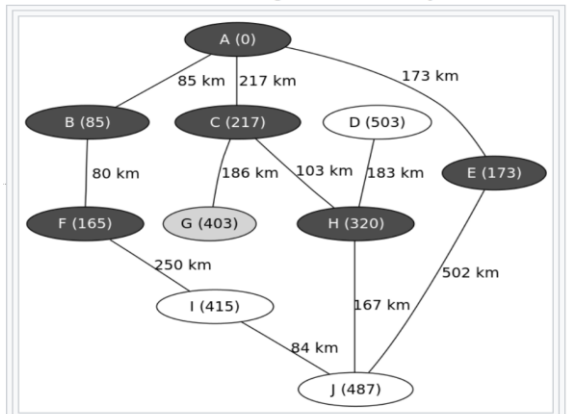
Étape 4 : on choisit E. On met à jour le voisin J (675).



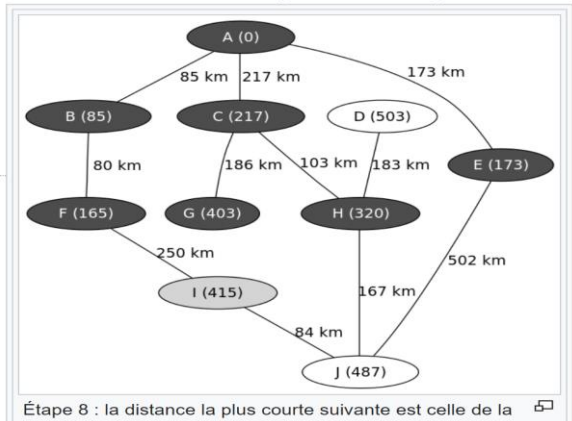
Étape 5 : la distance la plus courte en dehors du sous-graphe est maintenant celle de la ville C. On choisit donc C. On met à jour la ville G (403) et la ville H (320).



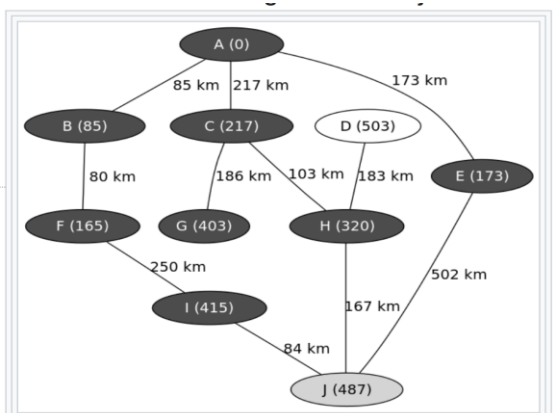
Étape 6 : la distance la plus courte en dehors du sous-graphe est maintenant celle de la ville H (320). On choisit donc H. On met à jour la ville D (503) et la ville J (487< 675).



Étape 7 : la distance la plus courte suivante est celle de la ville G. On choisit G. La mise à jour ne change aucune autre distance.



Étape 8 : la distance la plus courte suivante est celle de la ville I. La distance de la ville voisine J n'est pas modifiée car la distance existante est inférieure à celle que l'on obtiendrait en passant par I (415 + 84 > 487).

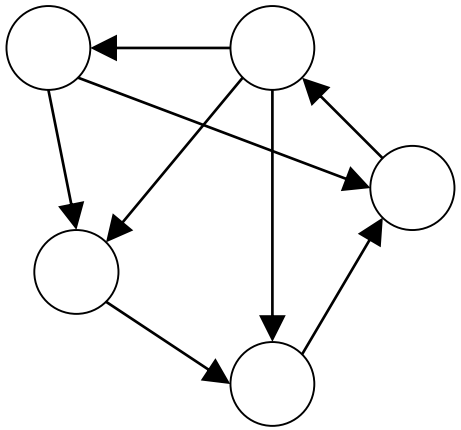


Étape 9 : la ville dont la distance est la plus courte est J (487). On choisit J et on l'ajoute au sous-graphe. On s'arrête puisque la ville d'arrivée est maintenant dans le sous-graphe.

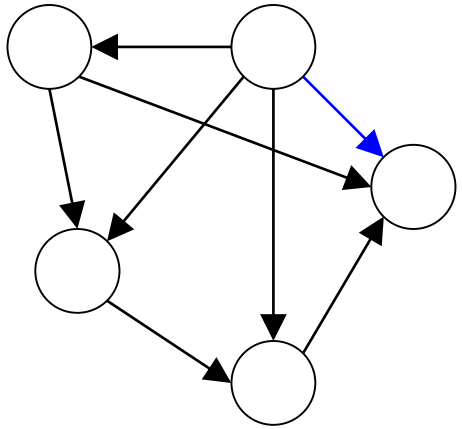
COMPOSANTE FORTEMENT CONNEXE

Une composante fortement connexe, ou SCC (*Strongly Connected Component* en anglais), est à un sous-ensemble de nœuds dans un graphe dirigé (orienté) où il existe un chemin dirigé de chaque nœud du sous-ensemble vers chaque autre nœud du même sous-ensemble

Strongly Connected



Not Strongly Connected



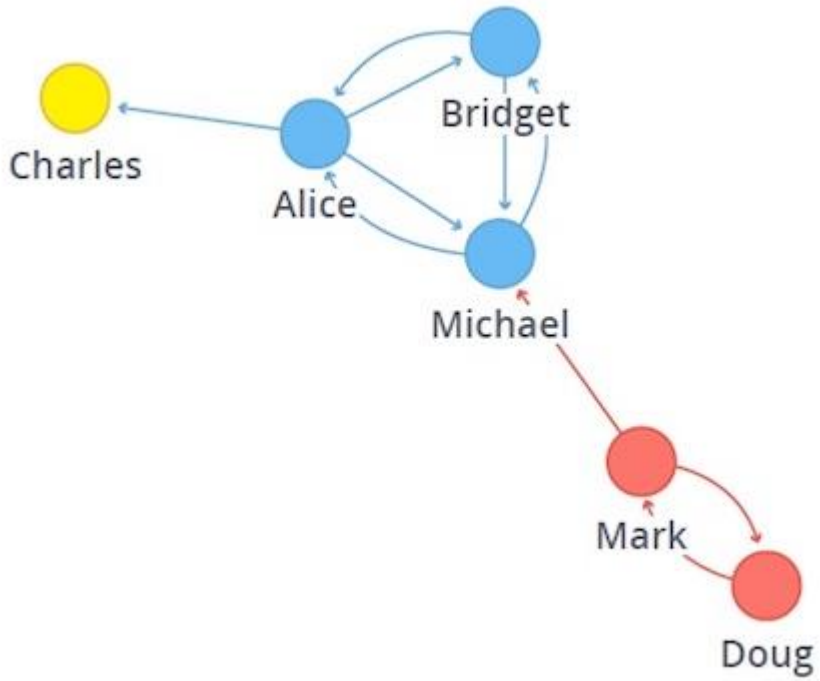
COMPOSANTE FORTEMENT CONNEXE

Pour trouver des composantes fortement connexes (SCC) dans un graphe dirigé:

1. Effectuez un parcours en profondeur (DFS) du graphe original et enregistrez l'ordre dans lequel les nœuds sont explorés.
2. Inversez toutes les arêtes du graphe original pour obtenir un graphe inversé.
3. Effectuez un deuxième DFS sur le graphe inversé en commençant par les nœuds dans l'ordre inverse obtenu à l'étape 1.
4. Chaque fois que vous terminez un DFS lors de l'étape c, les nœuds explorés forment une SCC.

COMPOSANTE FORTEMENT CONNEXE

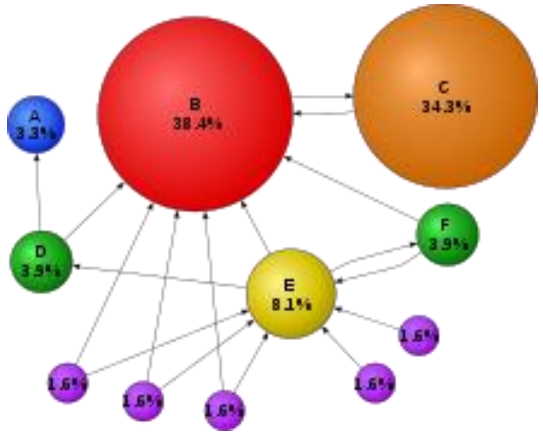
Dans les réseaux sociau, par exemple, la recherche de SCC peut aider à identifier des **groupes de personnes fortement connectées**, ce qui peut être utile pour l'analyse des communautés, la recommandation de contenu et la détection de fraudes.



Visualization of Strongly Connected Components

LINK ANALYSIS

Le principe de l'analyse des liens et de la propagation de l'importance à travers un réseau est essentiel dans les graphes.



Un exemple assez connu est l'algorithme de **PageRank**. Le PageRank est un algorithme développé par Larry Page et Sergey Brin, les fondateurs de **Google**, pour évaluer la pertinence des pages web dans les résultats de recherche. Bien qu'il soit souvent associé à l'analyse de la pertinence des pages web, le PageRank peut également s'appliquer à d'autres types de graphes, pas seulement aux pages web.

PAGE RANK

Le principe de base est d'attribuer à chaque page (nœud dans un graphe) une valeur (ou score) proportionnelle au nombre de fois que passerait par cette page un utilisateur parcourant le graphe du Web en cliquant aléatoirement, sur un des liens apparaissant sur chaque page.

Ainsi, une page a un PageRank d'autant plus important qu'est grande la somme des PageRanks des pages qui pointent vers elle (elle comprise, s'il y a des liens internes).

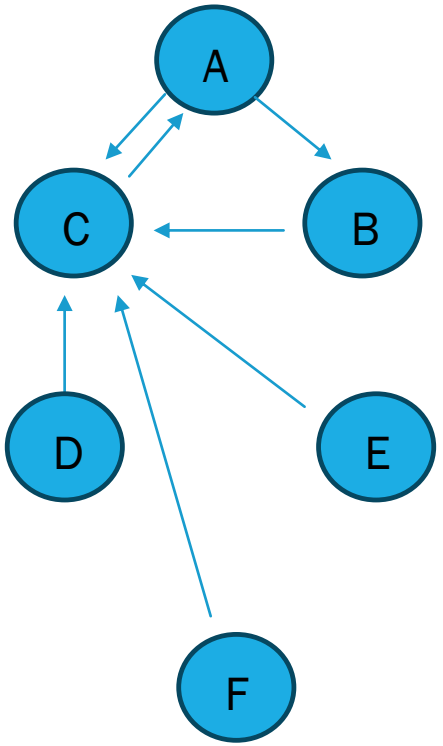
PAGE RANK

Idée de base:

1. **Définition du graphe :** Le PageRank commence par représenter le réseau sous forme de graphe dirigé et les arêtes représentent les liens ou les connexions entre ces entités.
2. **Calcul des valeurs initiales** : Chaque nœud du graphe reçoit une valeur de PageRank initiale. Dans le contexte des pages web, cela pourrait être une valeur égale pour toutes les pages. Cependant, dans d'autres domaines, vous pouvez attribuer des valeurs initiales en fonction de la pertinence ou de l'importance de chaque nœud par rapport à votre domaine d'application spécifique.
3. **Propagation des valeurs** : L'idée fondamentale derrière le PageRank est de propager les valeurs d'importance à travers les arêtes du graphe. Chaque nœud distribue une partie de sa valeur d'importance à ses nœuds voisins. Plus un nœud a de liens entrants de haute qualité (nœuds importants), plus sa valeur d'importance augmente.
4. **Itération** : Le processus de propagation et de répartition des valeurs d'importance est itératif. Il se répète un certain nombre de fois jusqu'à ce que les valeurs de PageRank convergent vers une valeur stable. Cela signifie que les nœuds les plus importants finissent par accumuler la majeure partie de l'importance.
5. **Obtention des classements** : Une fois que les valeurs de PageRank ont convergé, vous pouvez classer les nœuds en fonction de leurs valeurs de PageRank. Les nœuds avec les valeurs de PageRank les plus élevées sont considérés comme les plus importants dans le réseau.

EXEMPLE

1. Initialement, nous attribuons des valeurs de PageRank égales à chaque nœud, c'est-à-dire que chaque nœud commence avec un PageRank de 1/6 (car il y a six nœuds).

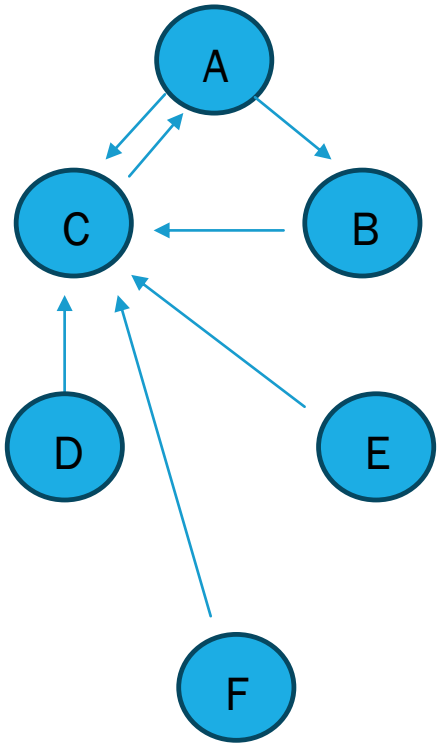


- **Nœud A** : 1/6
- **Nœud B** : 1/6
- **Nœud C** : 1/6
- **Nœuds D**: 1/6
- **Nœuds E**: 1/6
- **Nœuds F**: 1/6

EXEMPLE

2. Nous commençons à faire la propagation. Chaque nœud envoie la moitié de son poids aux voisins.

- **Nœud A** : A envoie une partie de son PageRank à B et une partie à C. B et C reçoivent chacun 1/2 de 1/6, soit 1/12.
- **Nœud B** : B envoie 1/2 de son PageRank (1/6) à C. Il reçoit 1/2 de 1/6 de A , soit 1/12.
- **Nœud C** : C reçoit des contributions de A, B, D, E, et F
- **Nœuds D, E, F** : Ces nœuds ne sont connectés qu'à C, donc ils envoient chacun 1/12 de leur importance à C,



EXEMPLE

2. A la fin de cette itération nous avons:
   - **Nœud A** : 1/6 - 1/12 – 1/12 + 1/12 = **1/12**
   - **Nœud B** : 1/6 - 1/12 + 1/12 = **1/6**

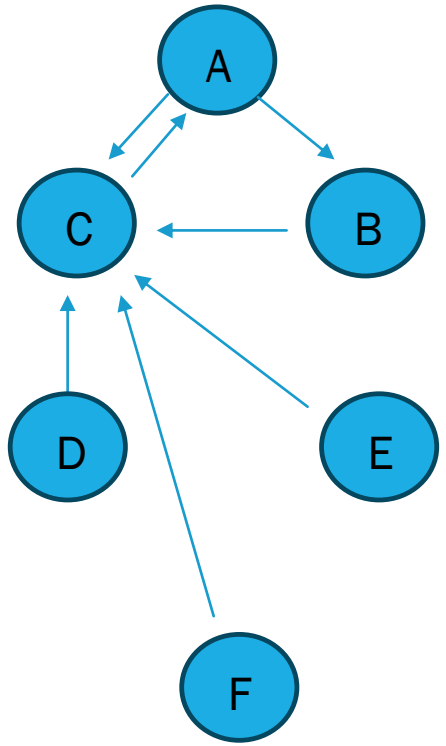
     Nœud C : 1/6 + 1/12 + 1/12 + 1/12 + 1/12 +1/12 = **7/12**

     Nœuds D, E, F : 1/6 – 1/12: **1/12**

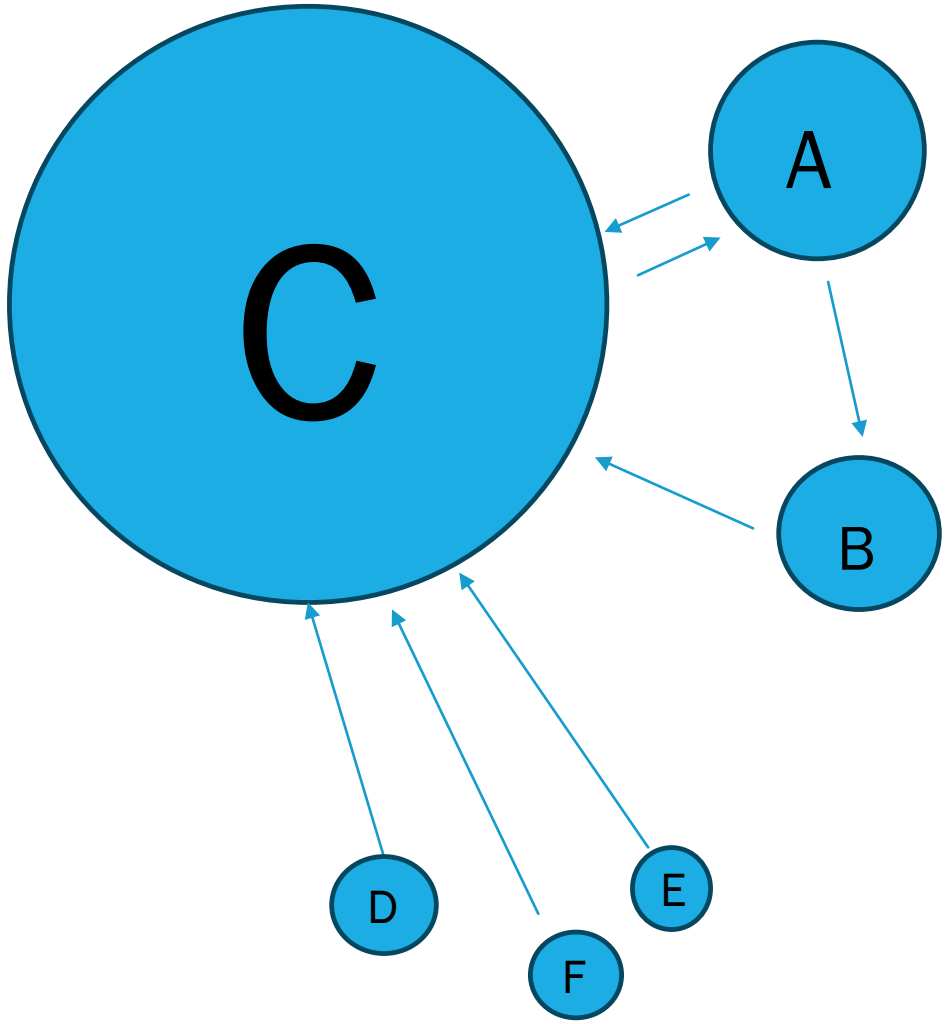
3. On réitère l'operation, mais avec les nouveaux poids:
   - **Nœud A** : 1/12 – 1/24 – 1/24 + 7/24 = **7/24**
   - **Nœud B** : 1/6 -1/12+ 1/24: **3/24**

     Nœud C : 7/12 + 1/24 + 1/12 + 1/24 + 1/24 + 1/24 = **20/24**

     Nœuds D, E, F : 1/12 – 1/24: **1/24**



EXEMPLE



Après cette deuxième phase, l'importance de C est augmenté sensiblement. Mais, encore plus important, A aussi a augmenté sa valeur de pageRank à cause de sa connexion avec B!

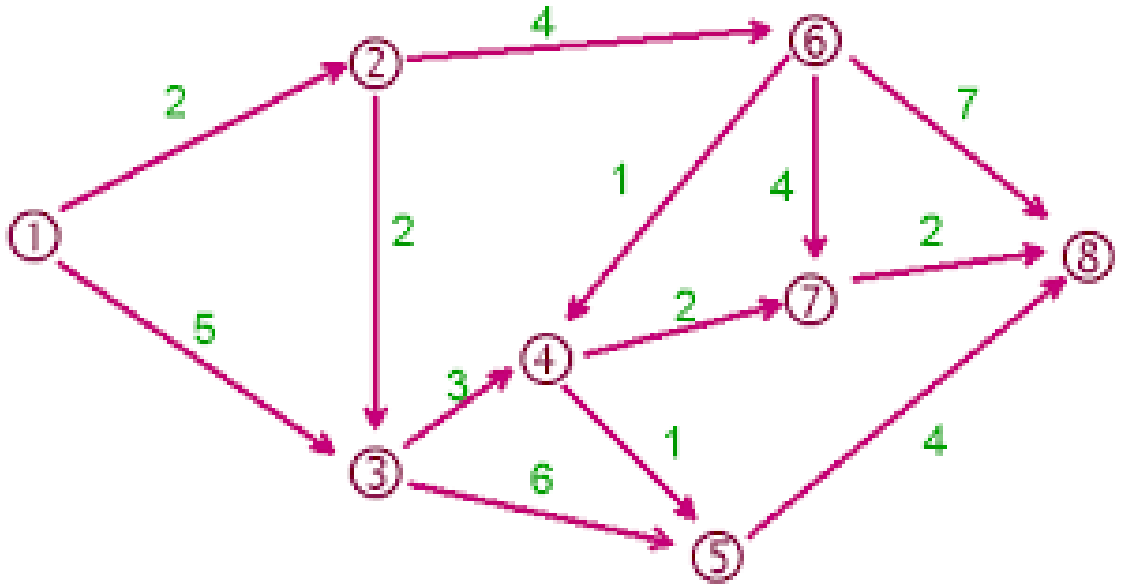
- **Nœud A** : 1/12 – 1/24 – 1/24 + 7/24 = **7/24**
- **Nœud B** : 1/6 -1/12+ 1/24: **3/24**
  Nœud C : 7/12 + 1/24 + 1/12 + 1/24 + 1/24 + 1/24 = **20/24**
  Nœuds D, E, F : 1/12 – 1/24: **1/24**

3. On continue à itérer jusqu'à une stabilisation des poids

PAGE RANK DANS UN GRAPHE PONDERE

Il est également possible d'appliquer le PageRank sur un graphe pondéré (graphe ou les arêtes ont un poids).

Dans ce cas de figure, au lieu de distribuer équitablement le poids pendant les itérations, on prendra en compte les poids des arêtes (selon les modalités choisies).



PAGE RANK

En résumé, le PageRank est un algorithme d'analyse de réseau qui peut être appliqué à divers types de graphes pour évaluer l'importance des nœuds dans le contexte de leur connectivité.