Graphes

Parcours

Mathilde Vernet
mathilde.vernet@univ-lehavre.fr

Master Informatique, Master Mathématiques
Université Le Havre Normandie

Automne 2020

Qu'est-ce qu'un parcours de graphe ?

- Explorer tous les sommets du graphe

Quelle est l'utilité de parcourir un graphe ?

- Répondre à certaines questions sur le graphe (notamment sur sa structure)
  - Quelles sont les composantes connexes d'un graphe
  - Un graphe est-il biparti ou acyclique
- Résoudre certains problèmes de graphe
  - Plus court chemin
  - Flot

Comment on parcourt un graphe ?

- En prenant les sommets au hasard
  - NON : inefficace, n'apporte pas d'informations
- De proche en proche
  - OUI : à partir d'un sommet initial puis en visitant ses voisins, etc

Questions

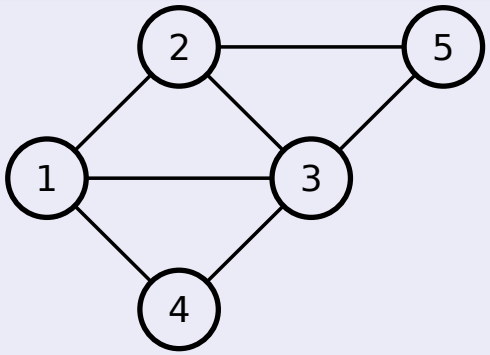
- Comment un graphe est-il représenté en mémoire ?
- Quelle place un graphe prend-il en mémoire ?
- Combien de temps faut-il pour :
  - trouver si deux sommets sont voisins ?
  - parcourir le voisinage d'un sommet ?

Matrice d'adjacence

Matrice de taille $n \times n$ où :

$$a_{uv} = \begin{cases} 1 & \text{si } (u, v) \in E \\ 0 & \text{sinon} \end{cases}$$

Exemple



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

Mémoire et parcours ?

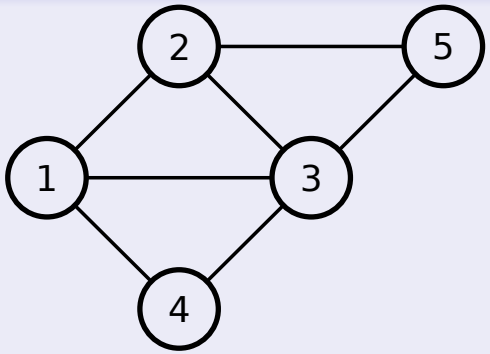
- Mémoire : $O(n^2)$
- u et v sont-ils voisins ? : $O(1)$
- Parcourir les voisins de u : $O(n)$

Listes d'adjacence

Pour chaque sommet u, on a une liste des sommets adjacents à u :

$$list(u) = [v_1, v_2, \dots]$$

Exemple



$$\begin{aligned} list(1) &= [2,3,4] \\ list(2) &= [1,3,5] \\ list(3) &= [1,2,4,5] \\ list(4) &= [1,3] \\ list(5) &= [2,3] \end{aligned}$$

Mémoire et parcours ?

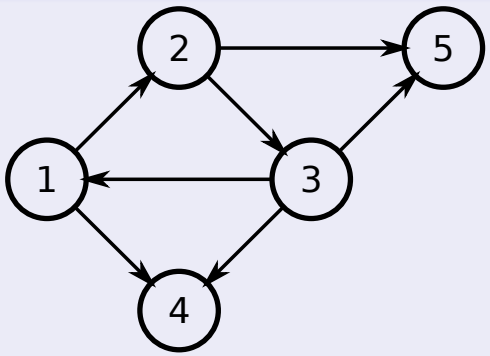
- Mémoire : $O(n+m)$
- u et v sont-ils voisins ? : $O\big(d(u)\big)$
- Parcourir les voisins de u : $O\big(d(u)\big)$
- Remarque : avec une structure supplémentaire (`HashSet` des voisins par exemple), on peut trouver si deux sommets sont voisins en temps $O(1)$ tout en restant en $O(m)$ en mémoire

Matrice d'incidence

Matrice de taille $n \times m$ où :

$$a_{ve} = \begin{cases} -1 & \text{si } v \text{ est l'origine de } e \\ 1 & \text{si } v \text{ est la destination de } e \\ 0 & \text{sinon} \end{cases}$$

Exemple



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

Mémoire et parcours ?

- Mémoire : $O(n \cdot m)$
- u et v sont-ils voisins ? : $O(m)$
- Parcourir les voisins de u : $O(m)$

Approche orientée objet

On utilise des classes pour les différents éléments du graphe

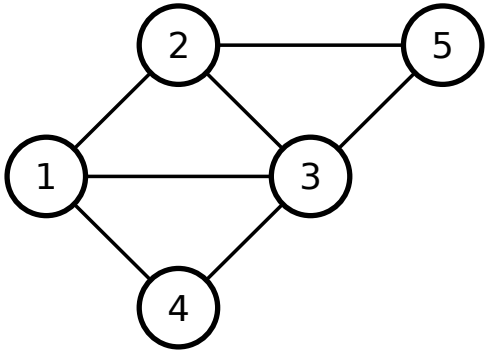
- Classe Graphe
  - Attributs : listes de sommets et d'arêtes, nom, orienté ou non, ...
  - Méthodes : ajouter/supprimer des sommets/arêtes, donner l'ensemble des sommets/arêtes, dessiner le graphe, ...
- Classe Sommet
  - Attributs : nom, indice, couleur, liste de voisins, ...
  - Méthodes : ajouter/supprimer arête incidente, donner ensemble des arêtes incidentes, donner ensemble des voisins, ...
- Classe Arête
  - Attributs : nom, indice, poids, sommets extrémités, ...
  - Méthodes : donner sommets incidents, modifier poids, ...

Mémoire et parcours ?

- Dépend des structures derrière les objets
- Remarque : en général, au plus les structures permettent des accès rapides, au plus cela prend de place en mémoire.

Idée

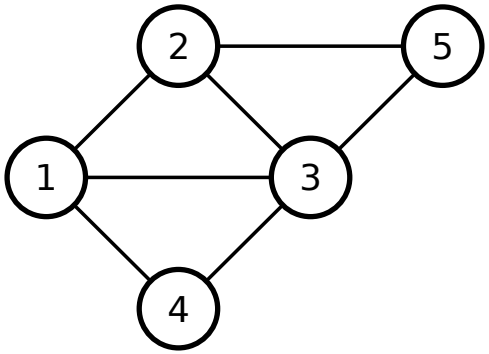
On parcourt les sommets de proche en proche



Idée

On parcourt les sommets de proche en proche

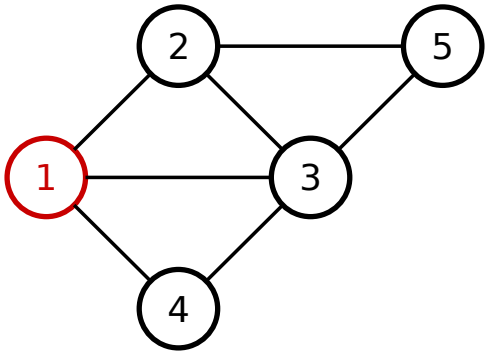
- On part d'un sommet s



Idée

On parcourt les sommets de proche en proche

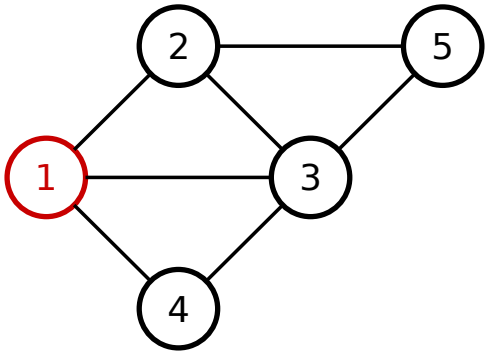
- On part d'un sommet s
- On le marque



Idée

On parcourt les sommets de proche en proche

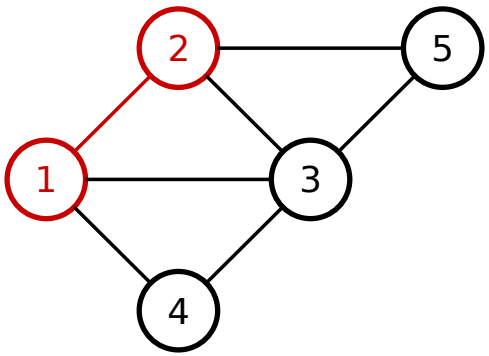
- On part d'un sommet s
- On le marque
- On passe à un voisin



Idée

On parcourt les sommets de proche en proche

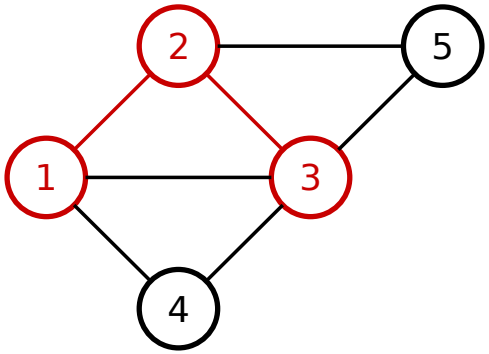
- On part d'un sommet s
- On le marque
- On passe à un voisin
- On le marque



Idée

On parcourt les sommets de proche en proche

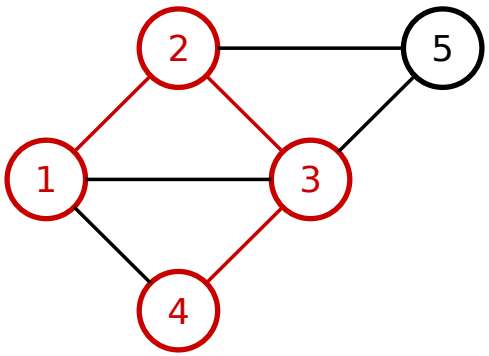
- On part d'un sommet s
- On le marque
- On passe à un voisin
- On le marque
- ...



Idée

On parcourt les sommets de proche en proche

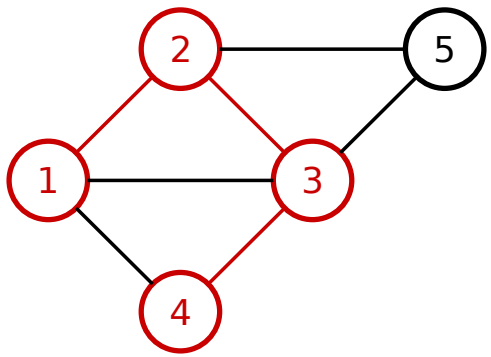
- On part d'un sommet s
- On le marque
- On passe à un voisin
- On le marque
- ...



Idée

On parcourt les sommets de proche en proche

- On part d'un sommet s
- On le marque
- On passe à un voisin
- On le marque
- ...

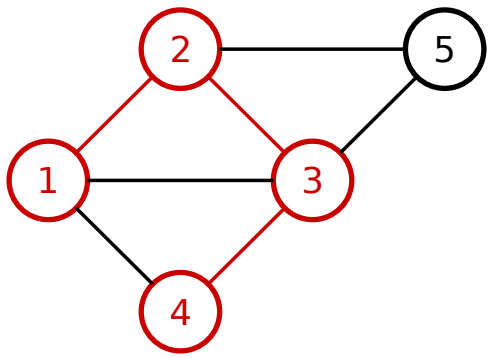


Comment faire lorsqu'il n'y a plus de voisin non marqué ?

Idée

On parcourt les sommets de proche en proche

- On part d'un sommet s
- On le marque
- On passe à un voisin
- On le marque
- ...



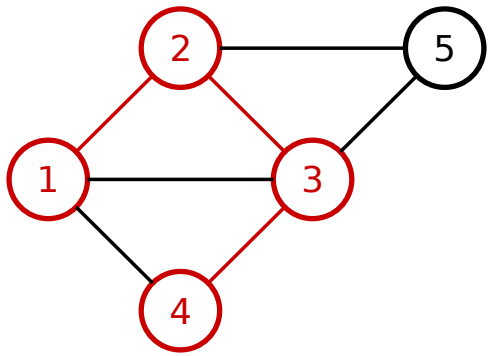
Comment faire lorsqu'il n'y a plus de voisin non marqué ?

- On ne passe pas à un voisin déjà marqué

Idée

On parcourt les sommets de proche en proche

- On part d'un sommet s
- On le marque
- On passe à un voisin
- On le marque
- ...



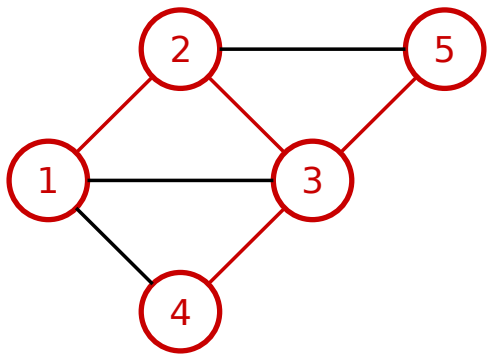
Comment faire lorsqu'il n'y a plus de voisin non marqué ?

- On ne passe pas à un voisin déjà marqué
- On doit se souvenir des sommets que l'on a visité

Idée

On parcourt les sommets de proche en proche

- On part d'un sommet s
- On le marque
- On passe à un voisin
- On le marque
- ...



Comment faire lorsqu'il n'y a plus de voisin non marqué ?

- On ne passe pas à un voisin déjà marqué
- On doit se souvenir des sommets que l'on a visité

Procedure `explore`(G,s)
  v.visited $\leftarrow$ false $\forall v \in V$
  s.visited $\leftarrow$ true
  open $\leftarrow \{s\}$
  TantQue open $\neq \emptyset$ **Faire**
    $v \leftarrow$ un élément de open
    $u \leftarrow$ un voisin de v avec u.visited $=$ false
    Si u existe **Alors**
      u.visited $\leftarrow$ true
      open $\leftarrow$ open $\cup \{u\}$
    Sinon
      open $\leftarrow$ open $\setminus \{v\}$
    FinSi
  FinTantQue
FinProcedure

Remarques

- À la fin de la procédure `explore`(G,s), on a v.visited $=$ true si et seulement si v est atteignable à partir de s.
- Si l'on veut parcourir tous les sommets du graphe, il faut recommencer la procédure à partir d'un sommet non visité tant qu'il en reste

Comment influencer sur l'ordre de parcours des sommets ?

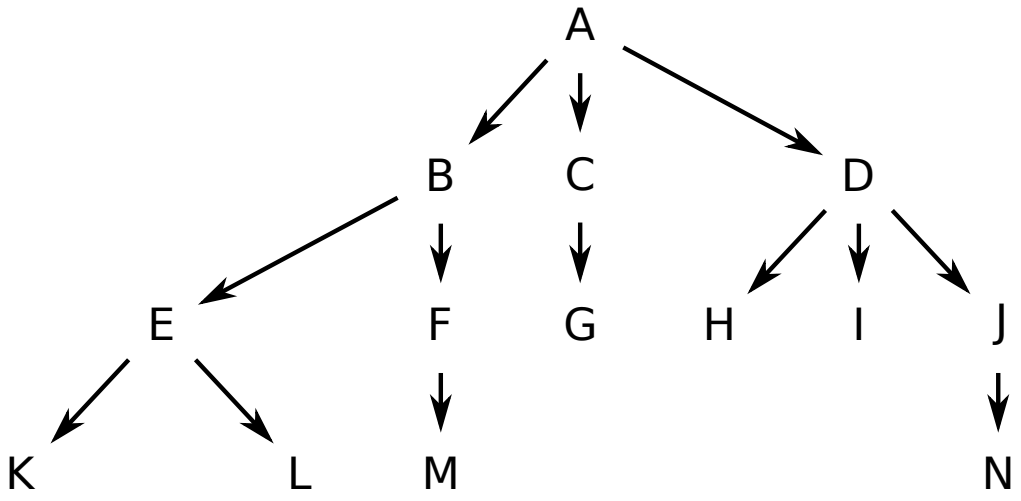
- Grâce à l'ensemble `open`

Quel ordre de parcours ?

Selon la nature de `open` :

- file : structure FIFO (*first in first out*)
  - $\rightarrow$ BFS : parcours en largeur ou *bread first search*
- pile : structure LIFO (*last in first out*)
  - $\rightarrow$ DFS : parcours en profondeur ou *depth first search*
- file à priorité : Élément ayant la plus haute priorité en premier

Principe



Rappel : parcours en largeur d'un arbre

- Tous les voisins d'abord, puis les voisins des voisins
- On découvre les éléments de l'arbre dans l'ordre suivant :
  A, B, C, D, E, F, G, H, I, J, K, L, M, N

Et dans un graphe ?

Le principe est le même !

Et concrètement pour parcourir un graphe

Principe du BFS

- On utilise l'algorithme général...
- ...en gérant la structure `open` de l'algorithme général comme une file (FIFO)

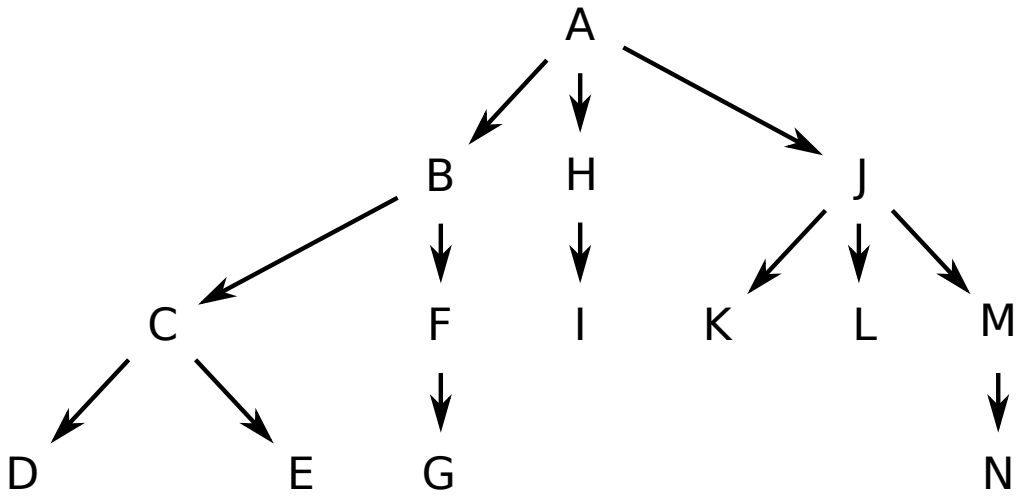
Opérations sur une file f

- f`.empty()` : la file est-elle vide ?
- f`.add(`v`)` : ajoute le sommet v à la fin de la file
- f`.peek()` : retourne le premier élément de la file
- f`.remove()` : retire et retourne le premier élément de la file

Pseudo-code de BFS

```
Procedure explore(G,s)
    v.visited ← false ∀v ∈ V
    s.visited ← true
    open.add(s)
    TantQue ¬open.empty() Faire
        v ← open.peek()
        u ← un voisin de v avec u.visited = false
        Si u existe Alors
            u.visited ← true
            open.add(u)
        Sinon
            open.remove()
        FinSi
    FinTantQue
FinProcedure
```

Principe



Rappel : parcours en profondeur d'un arbre

- Jusqu'au fond d'une branche d'abord, puis les autres branches ensuite
- On découvre les éléments de l'arbre dans l'ordre suivant :
  A, B, C, D, E, F, G, H, I, J, K, L, M, N

Et dans un graphe ?

Le principe est le même !

Et concrètement pour parcourir un graphe

Exemple

Une quête dans un labyrinthe pour trouver tous les trésors cachés. J'ai besoin de :

- marquer à la craie les endroits où je suis passé : v.visited
- Une bobine de fil à embobiner ou débobiner pour ne pas me perdre : une pile

Principe du DFS

- On utilise l'algorithme général...
- ...en gérant la structure `open` de l'algorithme général comme une pile (LIFO)

Opérations sur une pile p

- p.`empty()` : la pile est-elle vide ?
- p.`push(`v`)` : ajoute le sommet v sur le dessus de la pile
- p.`peek()` : retourne l'élément sur le dessus de la pile
- p.`pop()` : retire et retourne l'élément sur le dessus de la pile

Pseudo-code de DFS

Procedure `explore`(G,s)
  v.visited $\leftarrow$ false $\forall v \in V$
  s.visited $\leftarrow$ true
  open.`push`(s)
  TantQue $\neg$open.`empty()` **Faire**
    $v \leftarrow$ open.`peek()`
    $u \leftarrow$ un voisin de v avec u.visited $=$ false
    Si u existe **Alors**
      u.visited $\leftarrow$ true
      open.`push`(u)
    Sinon
      open.`pop()`
    FinSi
  FinTantQue
FinProcedure

Principe général du DFS

- Jusqu'au fond d'une branche d'abord, puis les autres branches ensuite

Autre vision du DFS

Un DFS à partir d'un sommet s c'est :

1. Visiter s
2. Faire un DFS à partir d'un premier voisin de s
3. Faire un DFS à partir d'un deuxième voisin de s
4. ...

Le DFS peut donc être un algorithme récursif !

Pseudo-code récursif de DFS

Procedure `explore`(G,s)
  s.visited $\leftarrow$ true
  Pour $(s,u) \in E$ **Faire**
    Si $\neg u$.visited **Alors**
      `explore`(G,u)
    FinSi
  FinPour
FinProcedure

Condition d'arrêt ?

Quand il n'y a plus d'arête (s,u)

Remarque

Avec cette procédure, on visite seulement les sommets accessibles depuis s.

Et si on veut parcourir tous les sommets du graphe ?

Il faut lancer `explore` sur des sommets non visités tant qu'il en reste

Et si on veut se servir d'un parcours pour effectuer des actions sur les sommets ?

- Effectuer une action lorsqu'on arrive sur un sommet v pour la première fois : `previsit(v)`
- Effectuer une action lorsqu'on a terminé d'explorer tous les voisins (et voisins de voisins...) d'un sommet v : `postvisit(v)`

Parcourir un graphe en entier avec DFS

Procedure DFS(G)
  v.visited $\leftarrow$ false $\forall v \in V$
  Pour $(v) \in V$ **Faire**
    Si $\neg v$.visited **Alors**
      explore(G,v)
    FinSi
  FinPour
FinProcedure

Procedure explore(G,v)
  v.visited $\leftarrow$ true
  previsit(v)
  Pour $(v, u) \in E$ **Faire**
    Si $\neg u$.visited **Alors**
      explore(G,u)
    FinSi
  FinPour
  postvisit(v)
FinProcedure

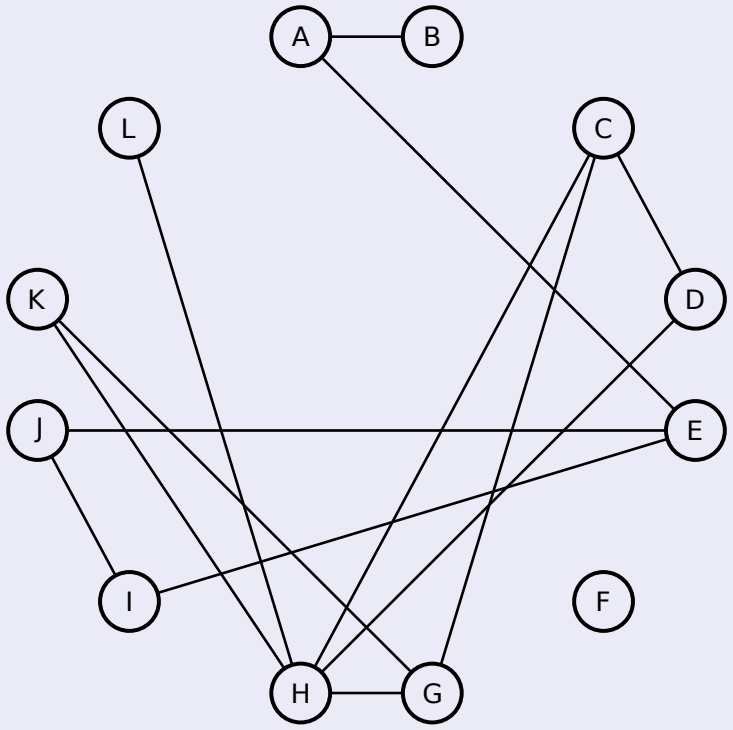
Qu'obtient-on en parcourant un graphe avec DFS ?

Une forêt couvrante de G

Complexité

$O(n+m)$

Exemple



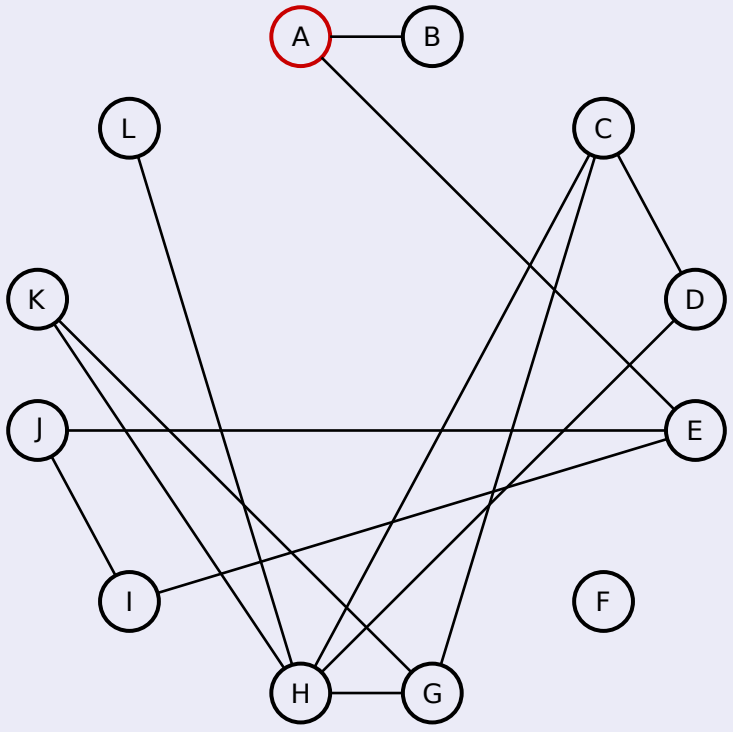
Qu'obtient-on en parcourant un graphe avec DFS ?

Une forêt couvrante de G

Complexité

$O(n+m)$

Exemple



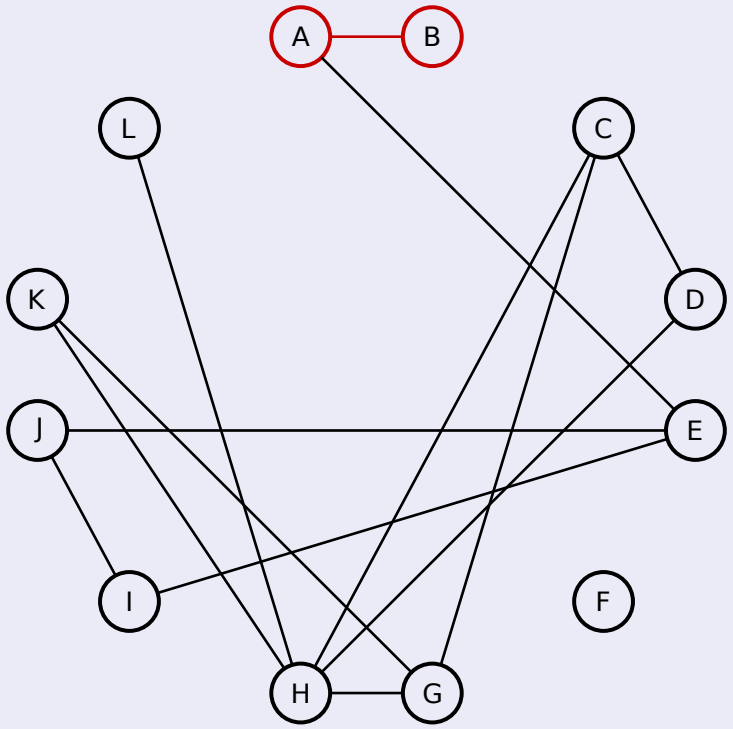
Qu'obtient-on en parcourant un graphe avec DFS ?

Une forêt couvrante de G

Complexité

$O(n+m)$

Exemple



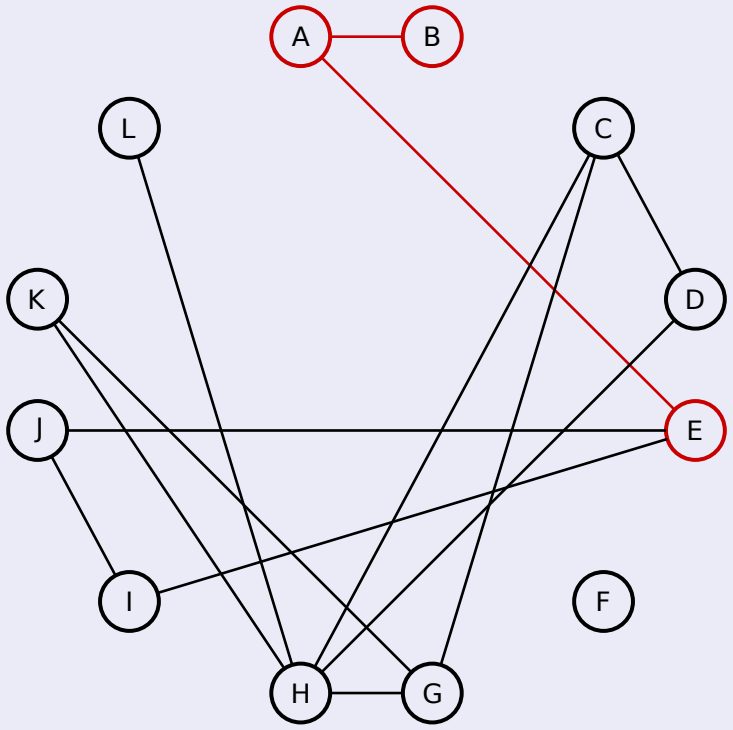
Qu'obtient-on en parcourant un graphe avec DFS ?

Une forêt couvrante de G

Complexité

$O(n+m)$

Exemple



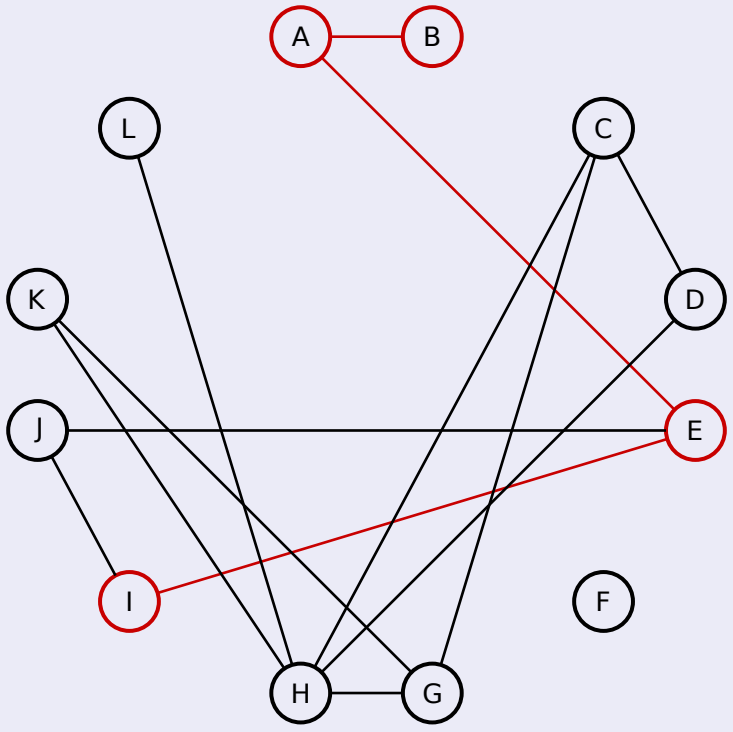
Qu'obtient-on en parcourant un graphe avec DFS ?

Une forêt couvrante de G

Complexité

$O(n+m)$

Exemple



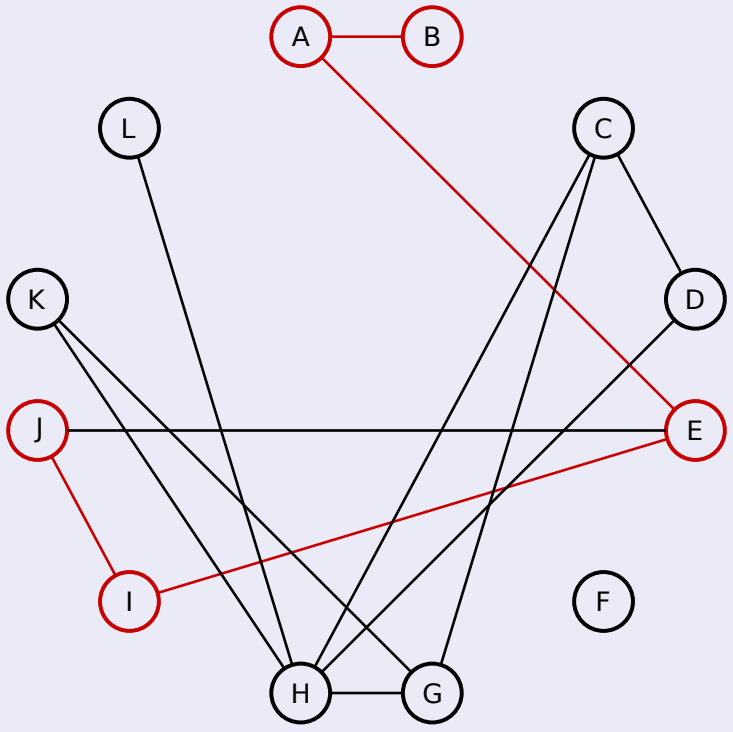
Qu'obtient-on en parcourant un graphe avec DFS ?

Une forêt couvrante de G

Complexité

$O(n+m)$

Exemple



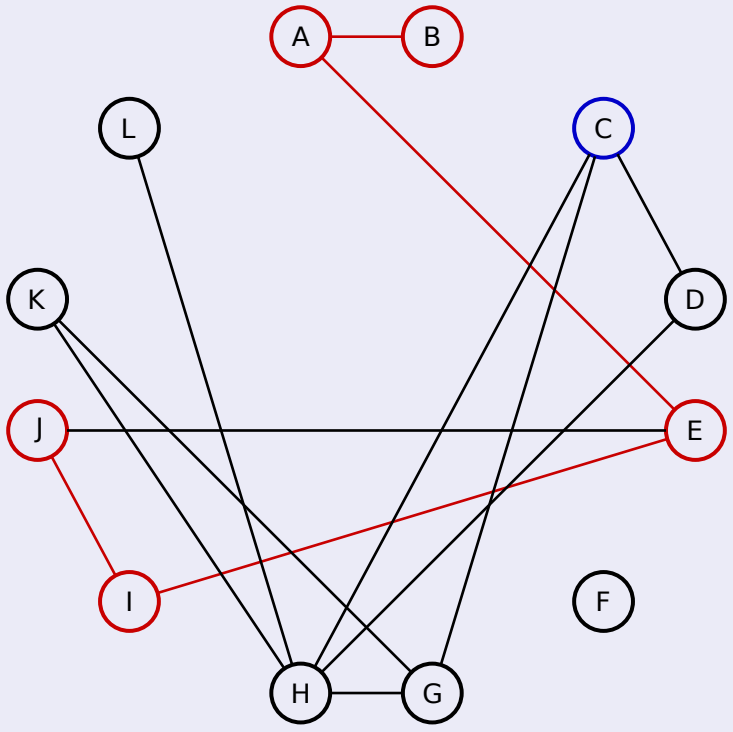
Qu'obtient-on en parcourant un graphe avec DFS ?

Une forêt couvrante de G

Complexité

$O(n+m)$

Exemple



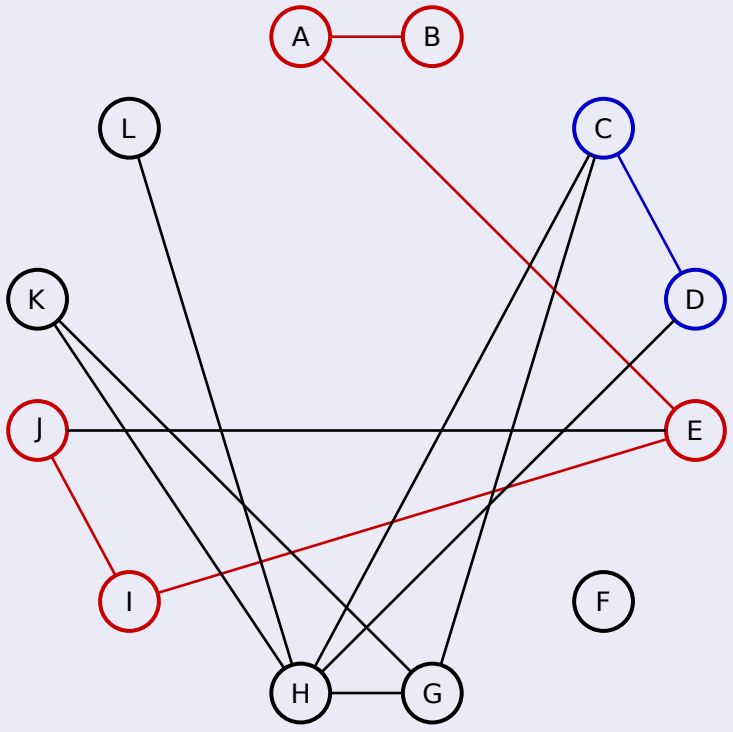
Qu'obtient-on en parcourant un graphe avec DFS ?

Une forêt couvrante de G

Complexité

$O(n+m)$

Exemple



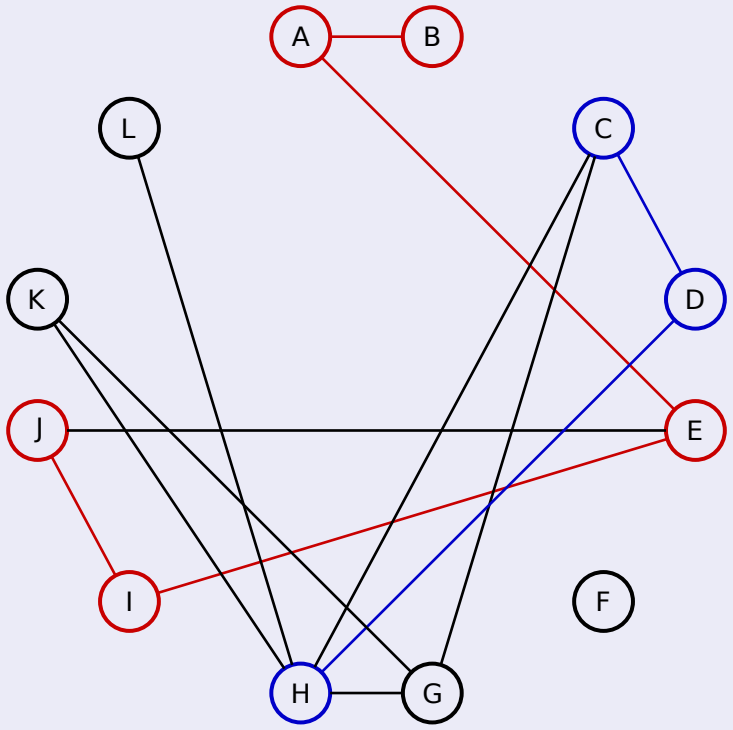
Qu'obtient-on en parcourant un graphe avec DFS ?

Une forêt couvrante de G

Complexité

$O(n+m)$

Exemple



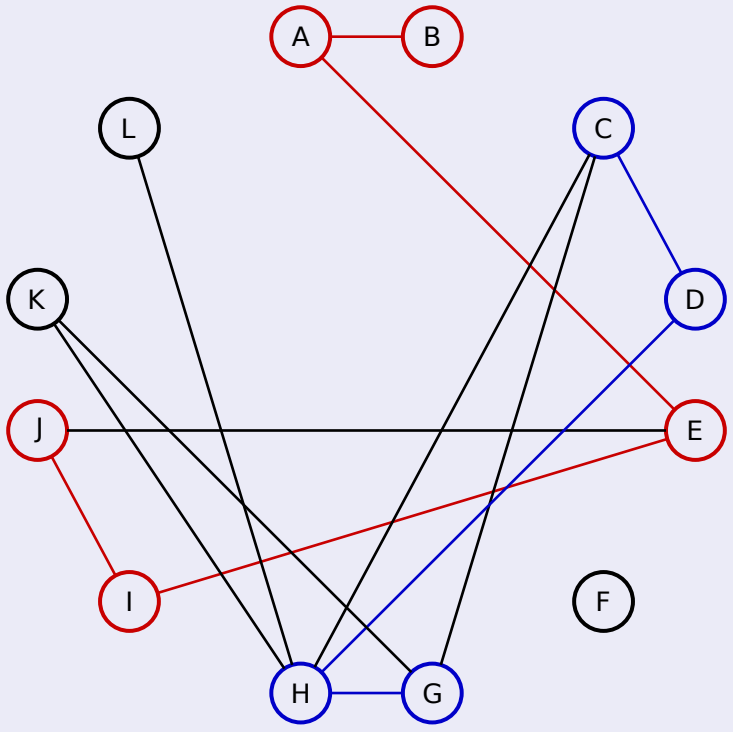
Qu'obtient-on en parcourant un graphe avec DFS ?

Une forêt couvrante de G

Complexité

$O(n+m)$

Exemple



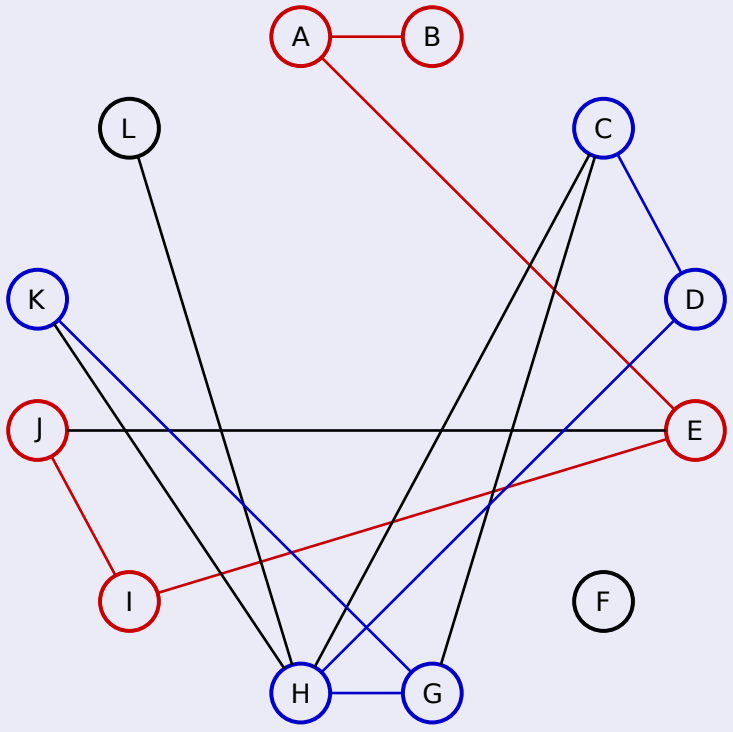
Qu'obtient-on en parcourant un graphe avec DFS ?

Une forêt couvrante de G

Complexité

$O(n+m)$

Exemple



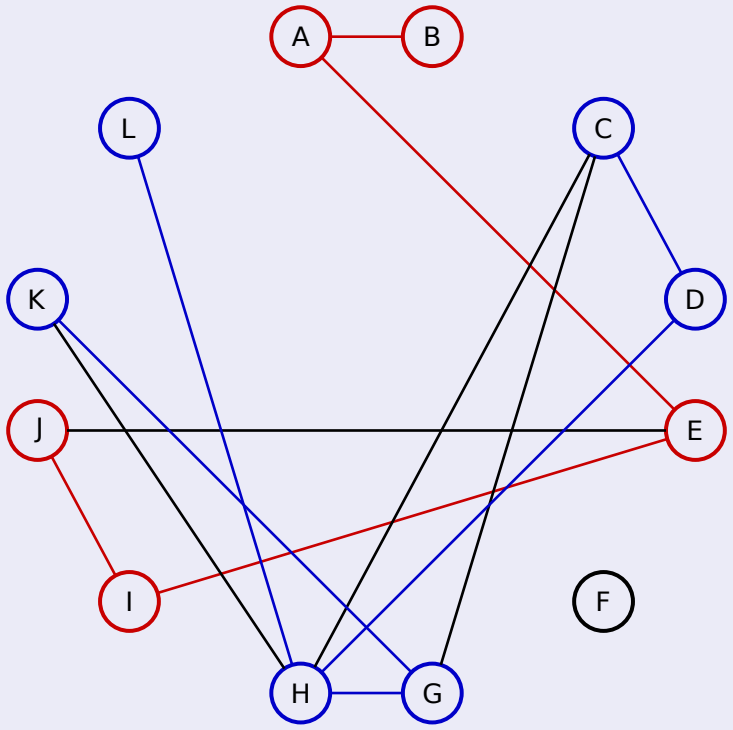
Qu'obtient-on en parcourant un graphe avec DFS ?

Une forêt couvrante de G

Complexité

$O(n+m)$

Exemple



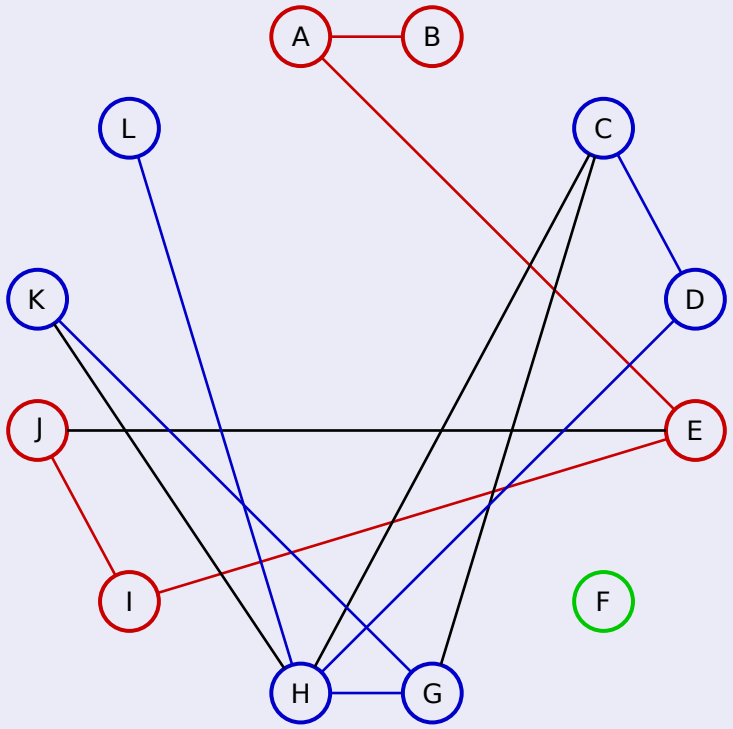
Qu'obtient-on en parcourant un graphe avec DFS ?

Une forêt couvrante de G

Complexité

$O(n+m)$

Exemple



Ni BFS, ni DFS

Principe

- On utilise l'algorithme général...
- ...en gérant la structure `open` comme une file de priorité

C'est à dire ?

Quel sommet choisit-on dans `open` ?

- On ne choisit ni le dernier arrivé ni le premier arrivé mais celui qui a la plus haute valeur de priorité

Dans quel cas ?

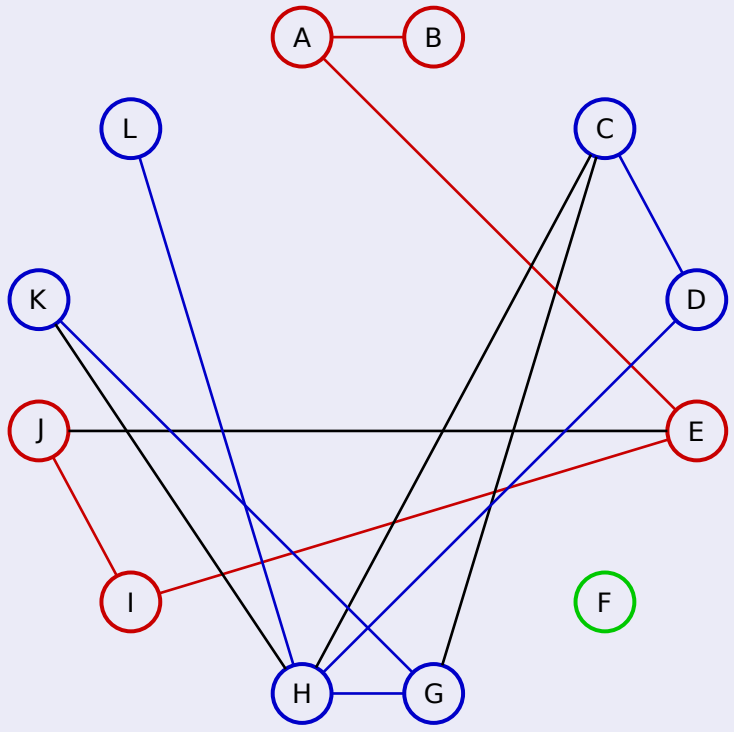
- Avec un parcours en largeur, on risquerait d'avoir une file de taille trop importante
- Avec un parcours en profondeur, on risquerait d'aller trop loin dans des « zones inintéressantes » du graphe

Exemple : pour parcourir le web

Comment identifier les composantes connexes d'un graphe ?

- On attribue à chaque composante connexe un numéro
- On donne à chaque sommet le numéro de la composante connexe à laquelle il appartient
- On fait cela lors d'un parcours du graphe

Exemple



- Composante connexe 1 : en rouge
- Composante connexe 2 : en bleu
- Composante connexe 3 : en vert

DFS pour trouver les composantes connexes

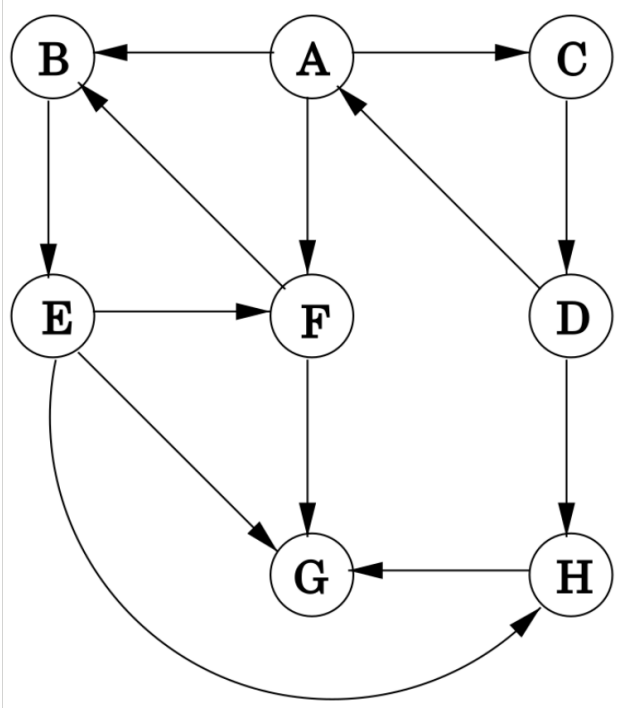
Procedure DFS(G)
  v.visited $\leftarrow$ false $\forall v \in V$
  cc$\leftarrow$0
  Pour $(v) \in V$ **Faire**
    Si $\neg v$.visited **Alors**
      cc$\leftarrow$cc+1
      explore(G,v)
    FinSi
  FinPour
FinProcedure

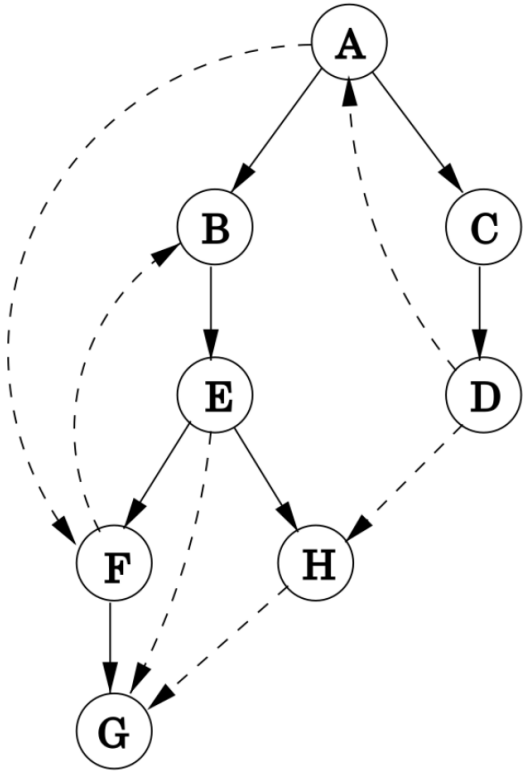
Procedure previsit(v)
  v.cc $\leftarrow$ cc
FinProcedure

Procedure explore(G,v)
  v.visited $\leftarrow$ true
  previsit(v)
  Pour $(v, u) \in E$ **Faire**
    Si $\neg u$.visited **Alors**
      explore(G,u)
    FinSi
  FinPour
FinProcedure

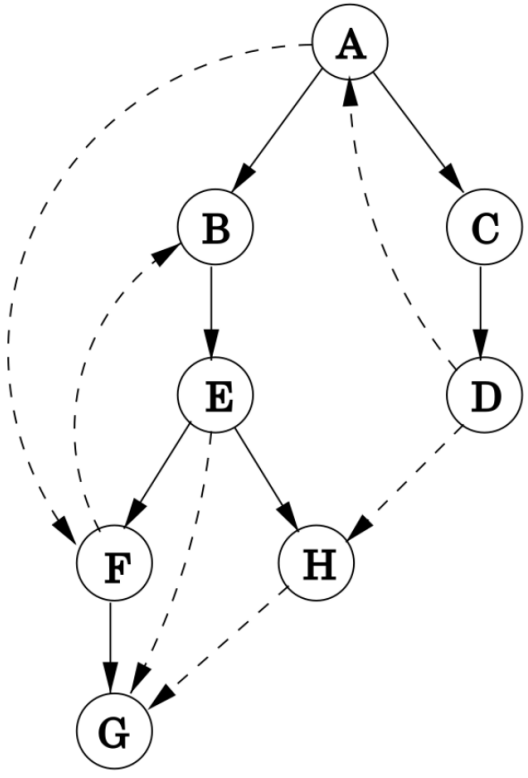
Comment parcourir un graphe orienté ?

- En utilisant un algorithme vu précédemment
- En considérant uniquement les voisins sortants de sommet *u* courant





Images de *Dasgupta, Papadimitriou and Vazirani*



Que remarque-t-on ?

Il y a différents types d'arcs !

- D'un sommet (grand)*-père vers un sommet (petit)*-fils
- D'un sommet (petit)*-fils vers un sommet (grand)*-père
- Entre des sommets de « branches » différentes

Image de *Dasgupta, Papadimitriou and Vazirani*

Types d'arcs

- Arcs descendants : d'un sommet (grand)*-père vers un sommet (petit)*-fils
  - (A, B), (A, C), (A, F), (B, E), (C, D), (E, F), (E, G), (E, H), (F, G)
- Arcs montants : d'un sommet (petit)*-fils vers un sommet (grand)*-père
  - (D, A), (F, B)
- Arcs traversants : entre des sommets de « branches » différentes
  - (D, H), (H, G)

DFS avec ordre de pré et post visite

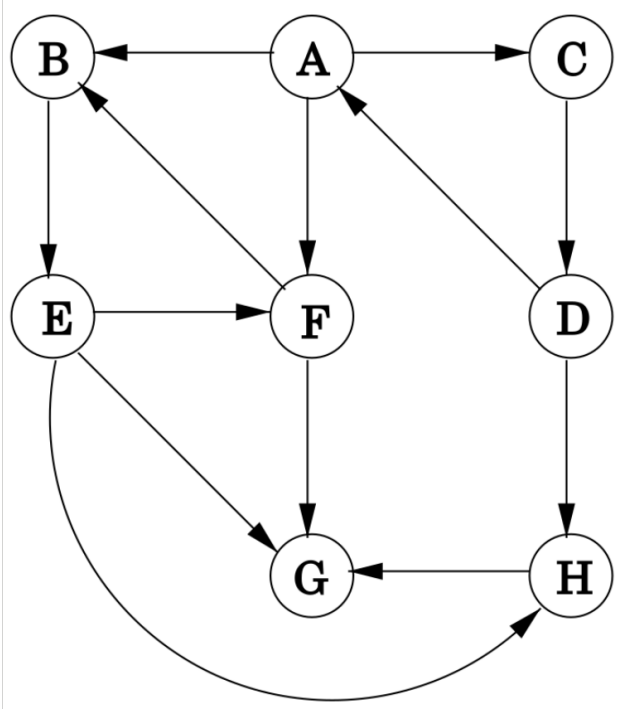
Procedure DFS(G)
  v.visited $\leftarrow$ false $\forall v \in V$
  clock$\leftarrow$0
  Pour $(v) \in V$ **Faire**
    Si $\neg v$.visited **Alors**
      explore(G,v)
    FinSi
  FinPour
FinProcedure

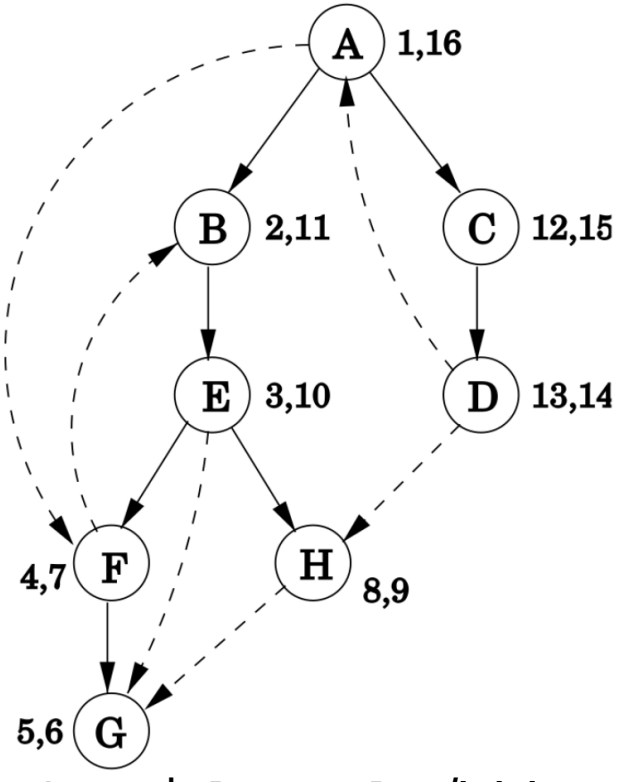
Procedure previsit(v)
  v.pre $\leftarrow$ clock
  clock$\leftarrow$clock+1
FinProcedure

Procedure explore(G,v)
  v.visited $\leftarrow$ true
  previsit(v)
  Pour $(v, u) \in E$ **Faire**
    Si $\neg u$.visited **Alors**
      explore(G,u)
    FinSi
  FinPour
  postvisit(v)
FinProcedure

Procedure postvisit(v)
  v.post $\leftarrow$ clock
  clock$\leftarrow$clock+1
FinProcedure

Ordre de pré et post visite



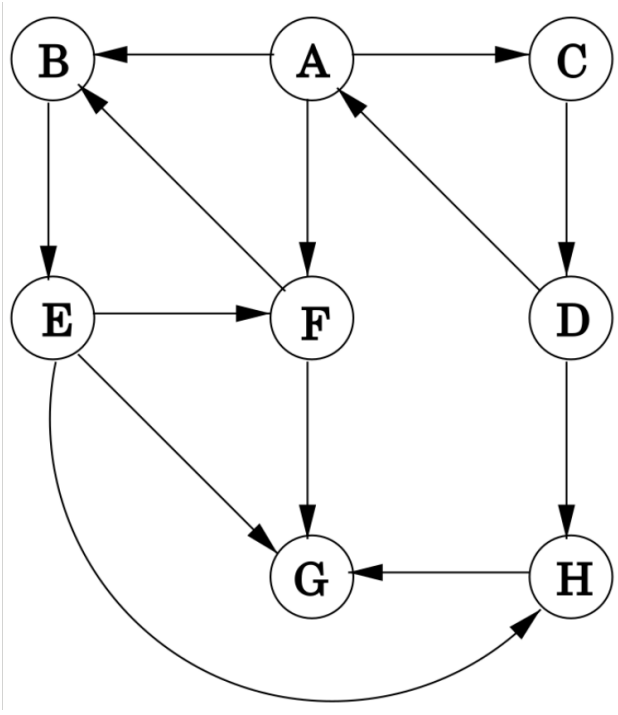


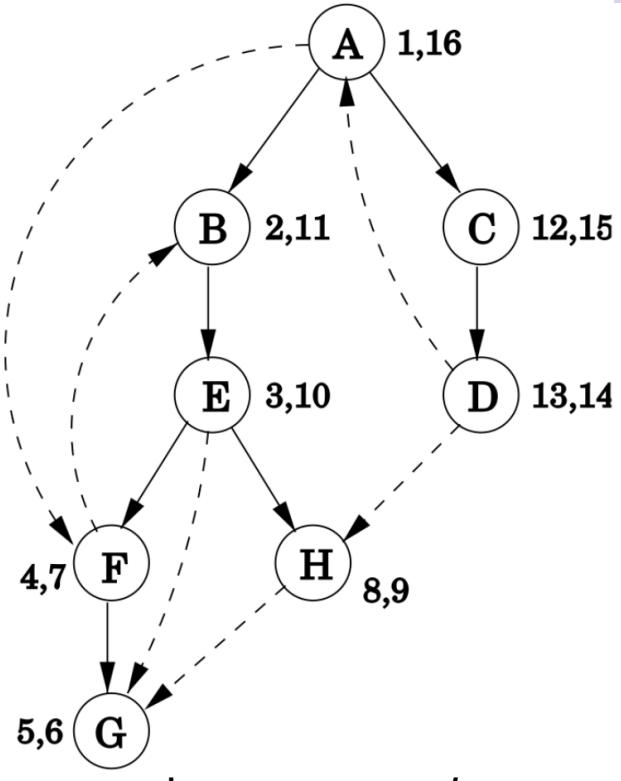
Images de *Dasgupta, Papadimitriou and Vazirani*

Propriété

Pour chaque paire de sommets u et v, les intervalles [u.pre,u.post] et [v.pre,v.post] sont soit disjoints, soit l'un est inclus dans l'autre.

Ordre de pré et post visite





Images de *Dasgupta, Papadimitriou and Vazirani*

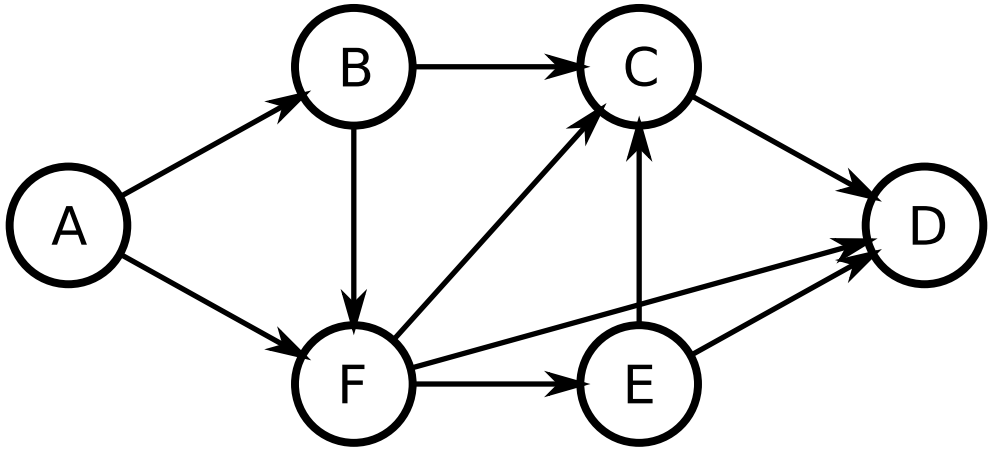
Déterminer le type d'un arc (u, v)

Arc descendant : $u.\text{pre} < v.\text{pre} < v.\text{post} < u.\text{post}$

Arc montant : $v.\text{pre} < u.\text{pre} < u.\text{post} < v.\text{post}$

Arc traversant : $v.\text{pre} < v.\text{post} < u.\text{pre} < u.\text{post}$

$u.\text{pre} < u.\text{post} < v.\text{pre} < v.\text{post}$

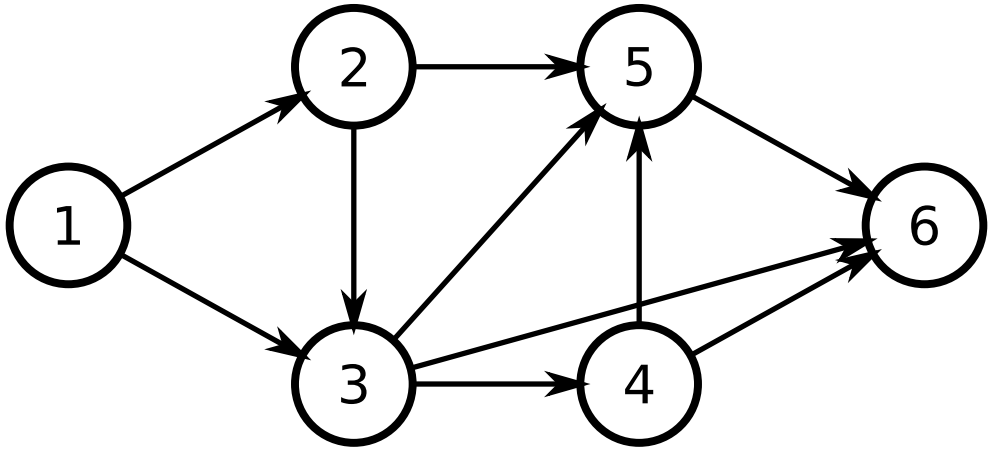


Particularité de ce graphe

- Il ne contient pas de cycle
- On peut numéroter les sommets pour tous les arcs aillent « en avant »

Vocabulaire

- Ce graphe est un DAG : un graphe orienté acyclique (*directed acyclic graph*)
- Ce graphe admet un ordre topologique : On peut numéroter les sommets pour que tous les arcs aillent d'un sommet avec un « plus petit numéro » vers un sommet avec un « plus grand numéro »



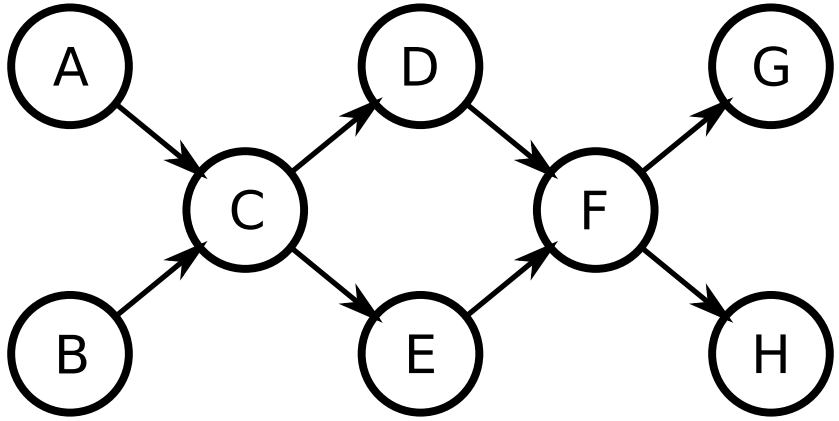
Particularité de ce graphe

- Il ne contient pas de cycle
- On peut numéroter les sommets pour tous les arcs aillent « en avant »

Vocabulaire

- Ce graphe est un DAG : un graphe orienté acyclique (*directed acyclic graph*)
- Ce graphe admet un ordre topologique : On peut numéroter les sommets pour que tous les arcs aillent d'un sommet avec un « plus petit numéro » vers un sommet avec un « plus grand numéro »

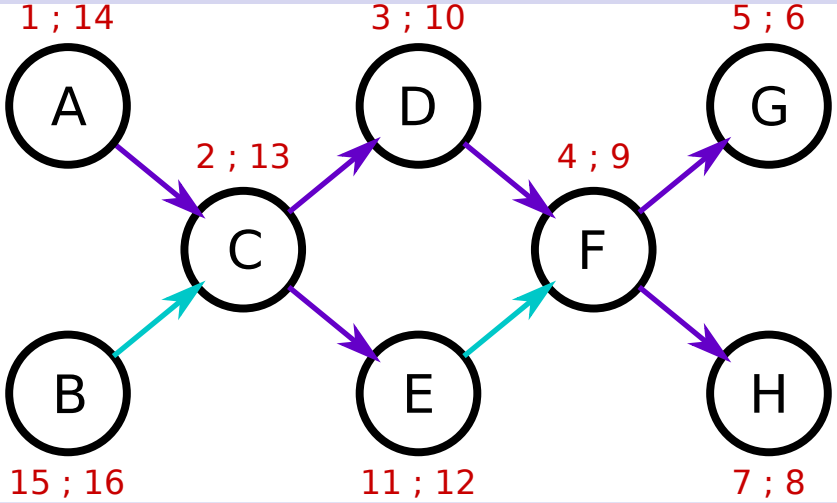
Exercice



Questions

1. Ce graphe est-il acyclique ?
2. Si non, expliquer pourquoi. Si oui, donner son ordre topologique.
3. Exécuter un parcours en profondeur de ce graphe en indiquant les intervalles de pré et post visite des sommets.
4. Qu'observe-t-on ?

Correction



Questions

1. *Ce graphe est-il acyclique ?*
   Oui, ce graphe est acyclique (il ne contient pas de cycle).
2. *Si non, expliquer pourquoi. Si oui, donner son ordre topologique.*
   {A, B, C, D, E, F, G, H} ou {B, A, C, E, D, F, H, G},...
3. *Exécuter un parcours en profondeur de ce graphe en indiquant les intervalles de pré et post visite des sommets.*
   En rouge à côté des sommets
4. *Qu'observe-t-on ?*
   Il n'y a que des arcs descendants (violet) et traversants (bleu clair)

Utilité des DAG

- Modélisation de relations de causalité
- Modélisation de relations de hiérarchie
- Modélisation de dépendances temporelles
- ...

Théorème

Soit G un graphe orienté. Les propositions suivantes sont équivalentes :

- G est acyclique
- Le parcours en profondeur de G de révèle pas d'arcs montants
- G admet un ordre topologique

Comment trouver un ordre topologique ?

Grâce aux dates de visite

1. Parcourir le graphe en profondeur
2. Ordonner les sommets par ordre décroissant de leur date de post-visite

Autrement

Procedure `OrdreTopologique`(G)
  $i \leftarrow 0$
  TantQue V est non vide **Faire**
    $u \leftarrow$ un sommet de V sans prédécesseur
    u.num $\leftarrow i$
    $i++$
    $V \leftarrow V \setminus \{u\}$
  FinTantQue
FinProcedure

Rappel : Composante fortement connexe

Deux sommets u et v sont dans la même composante connexe s'il existe dans le graphe :

- un chemin de u à v

ET

- un chemin de v à u

Remarque

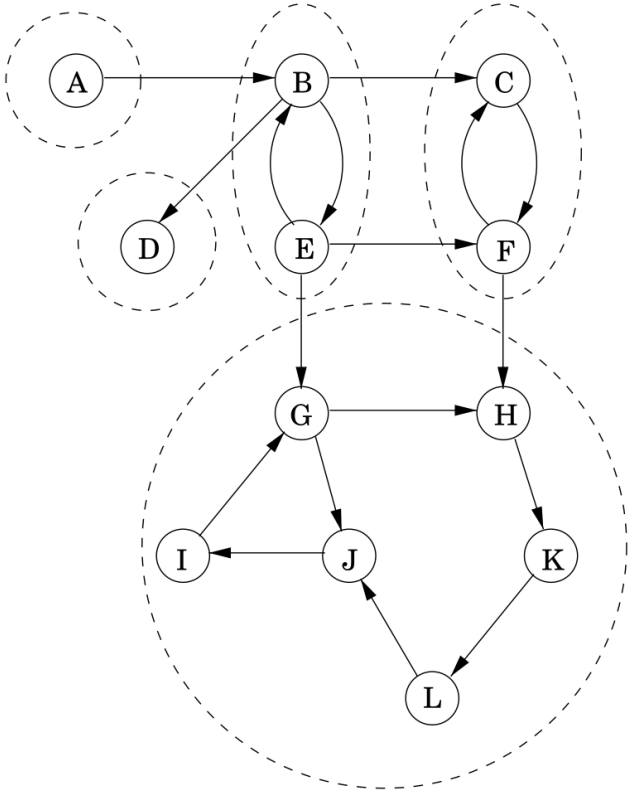
Un DAG possède n composantes fortement connexes.

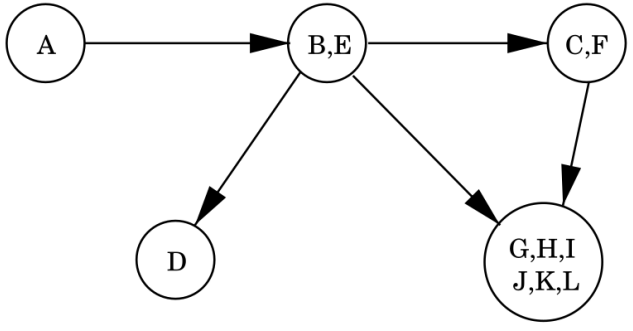
Attention

- Dans un graphe non orienté, un parcours à partir d'un sommet s me fait visiter des sommets qui sont dans la même composante connexe que s
- À l'inverse, dans un graphe orienté, si je peux atteindre le sommet u depuis le sommet s, cela ne signifie pas que je peux atteindre le sommet s depuis le sommet u

Trouvons une méthode pour identifier les composantes fortement connexes d'un graphe orienté

Méta-graphe



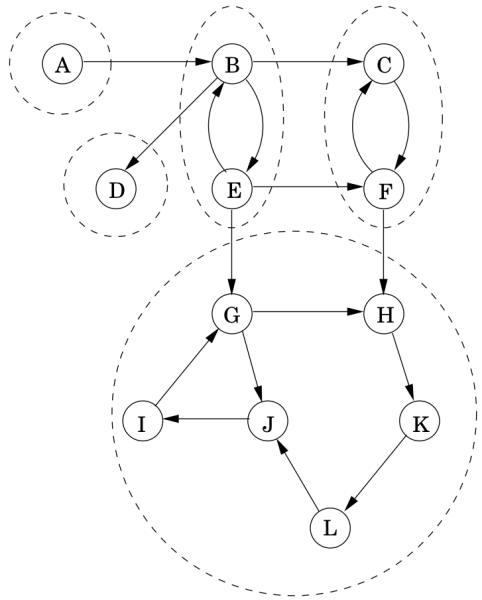


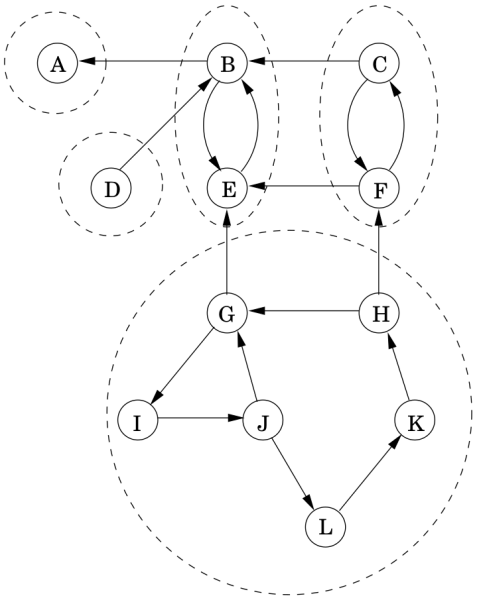
Images de *Dasgupta, Papadimitriou and Vazirani*

Définition

Le méta-graphe des composantes fortement connexes d'un graphe orienté G est un graphe orienté dans lequel chaque sommet correspond à une composante fortement connexe de G.

Graphe renversé





Images de *Dasgupta*, *Papadimitriou and Vazirani*

Définition

Le graphe renversé d'un graphe orienté $G = (V, E)$ est le graphe orienté $G^R = (V, E^R)$ tel que :

- $E^R = \{(u, v) | (v, u) \in E\}$

Propriété

Le graphe G et son graphe renversé G^R contiennent les mêmes composantes fortement connexes.

Afin de concevoir un algorithme pour identifier les composantes fortement connexes d'un graphe orienté, montrons les propriétés suivantes :

1. Le méta-graphe des composantes-fortement connexes est un DAG
2. Chaque DAG a au moins un sommet *source* (sommet avec un degré entrant nul) et au moins un sommet *puits* (sommet avec un degré sortant nul)
3. Si la procédure `explore` commence par un sommet qui se trouve dans une composante *puits*, alors elle va parcourir exactement cette composante
4. Le sommet avec la date de post-visite la plus grande se trouve forcément dans une composante *source*

Le méta-graphe des composantes-fortement connexes est un DAG

Si le méta-graphe contenait un cycle entre deux composantes, alors il existerait :

- un chemin de n'importe quel sommet de la première composante vers n'importe quel sommet de la deuxième composante

ET

- un chemin de n'importe quel sommet de la deuxième composante vers n'importe quel sommet de la première composante

Et donc tous ces sommets seraient dans la même composante fortement connexe.
Ces composantes ne formeraient alors qu'un seul sommet dans le méta-graphe.
Le méta-graphe des composantes-fortement connexes est donc un DAG.

Chaque DAG a au moins un sommet *source* et au moins un sommet *puits*

Un DAG admet un ordre topologique sur les sommets. Le premier sommet de cet ordre n'a pas de voisins entrant, et le dernier sommet de cet ordre n'a pas de voisins sortants.

Si la procédure `explore` commence par un sommet qui se trouve dans une composante *puits*, alors elle va parcourir exactement cette composante

- Une composante *puits* n'a pas d'arc sortant
- La procédure `explore` à partir d'un sommet s visite tous les sommets accessibles depuis s
- Si s se trouve dans une composante *puits* alors les seuls sommets accessibles depuis s sont ceux de sa composante fortement connexe

Donc si on lance la procédure `explore` depuis un sommet d'une composante *puits*, on va parcourir exactement les sommets de cette composante

Le sommet avec la date de post-visite la plus grande se trouve forcément dans une composante *source*

Soient C et C' deux composantes sommets du méta-graphe telles qu'il existe un arc d'un sommet de C vers un sommet de C'.

- Si on visite C avant C' alors la procédure `explore` aura visité tous les sommets de C et de C' avant de s'arrêter. Donc le sommet par lequel la procédure a commencé dans C aura une date date de post-visite plus grande que n'importe quel sommet de C'.
- Si on visite C' avant C, la procédure `explore` s'arrêtera avant d'avoir visité les sommets de C puisqu'ils ne sont pas accessible depuis C'. Donc il faudra relancer la procédure sur C et tous les sommets de C auront une date de post-visite plus grande que les sommets de C'.
- Si un sommet s' se trouve dans une composante C' qui n'est pas une composante source alors un sommet s d'une composante C telle qu'il existe un arc d'un sommet de C vers un sommet de C' aura une plus grande date de post-visite que s'.

Donc Le sommet avec la date de post-visite la plus grande se trouve forcément dans une composante *source*.

Algorithme pour les composante fortement connexes

Question

- Déduire des propriétés que l'on vient de montrer un algorithme pour identifier les composantes fortement connexes.

Idée

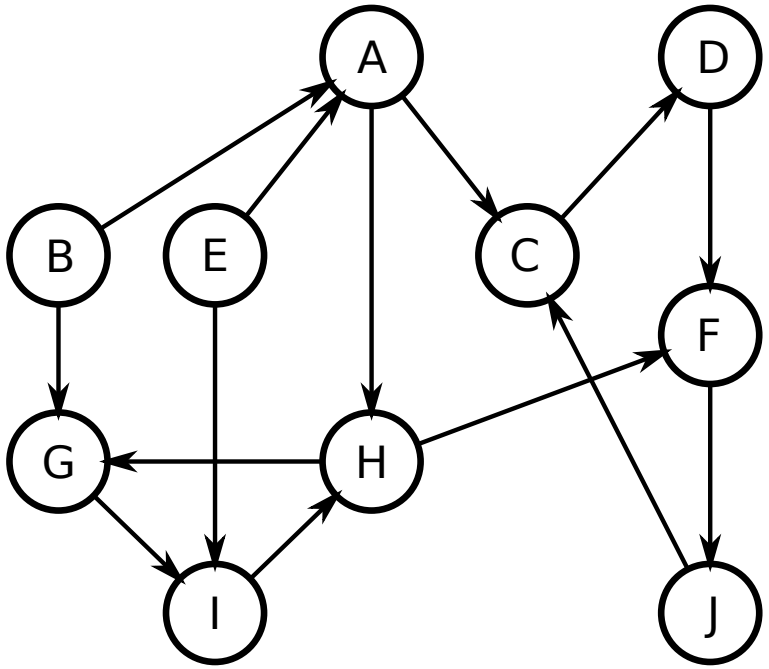
- On sait que si on lance un DFS depuis un sommet s dans une composante *puits*, on parcourra exactement les sommets de la composante fortement connexe de s
- On sait comment identifier un sommet d'une composante *source*
- Comment identifier un sommet d'une composante *puits* ?
- On sait que le graphe renversé possède les mêmes composantes fortement connexes que le graphe d'origine

Identifier les composante fortement connexes de G

Algorithme

1. Exécuter un DFS sur le graphe renversé G^R
2. Exécuter l'algorithme d'identification des composantes connexes pour les graphes non-orientés sur G en sélectionnant les sommets dans l'ordre décroissant des dates de post-visite sur G^R.

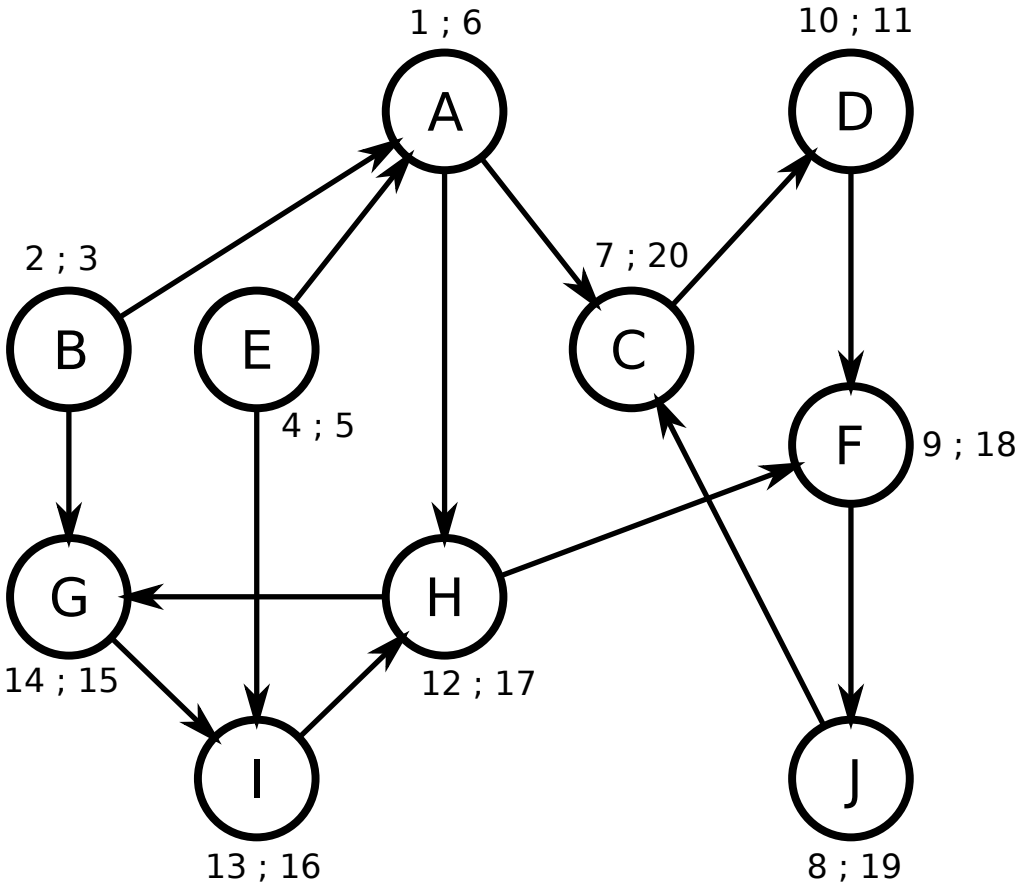
Exercice



Questions

1. Appliquer l'algorithme que l'on vient de voir sur ce graphe.
2. Quelles sont les composantes fortement connexes de ce graphe ?
3. Dessiner le méta-graphe

DFS sur le graphe renversé



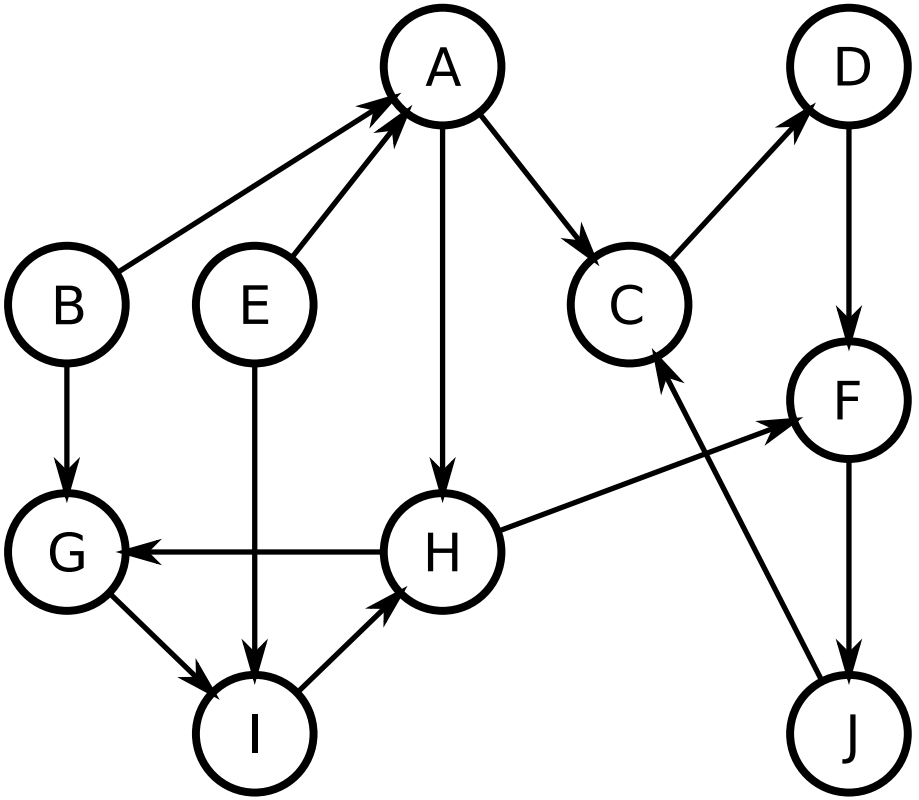
Ordre décroissant des post-visites

C ; J ; F ; H ; I ; G ; D ; A ; E ; B

DFS sur le graphe

Ordre de parcours des sommets

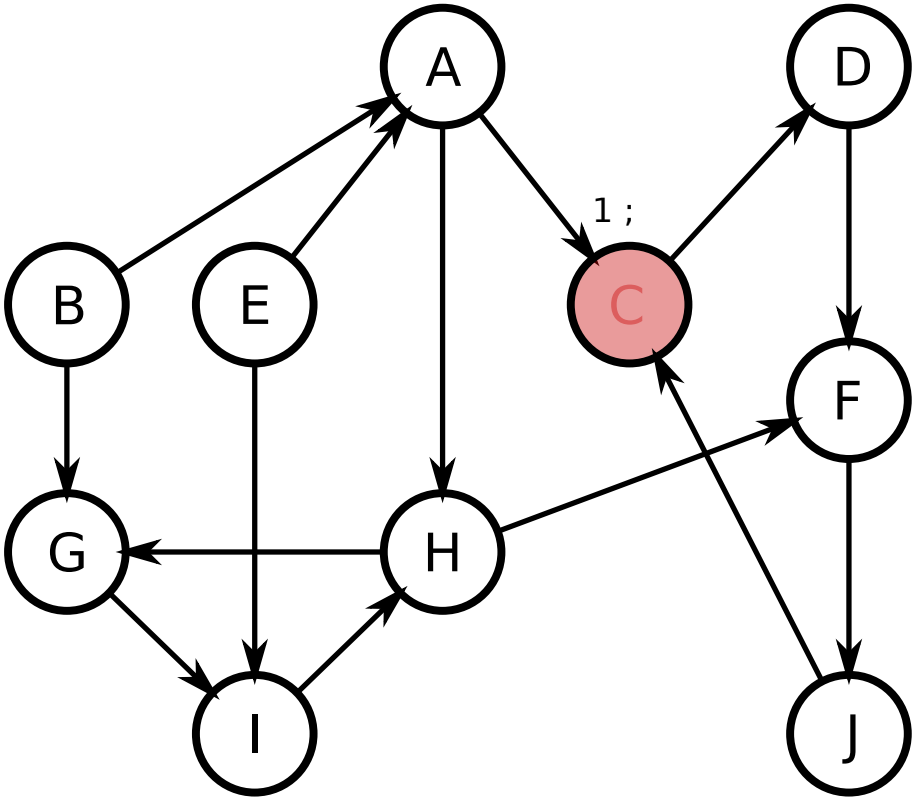
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

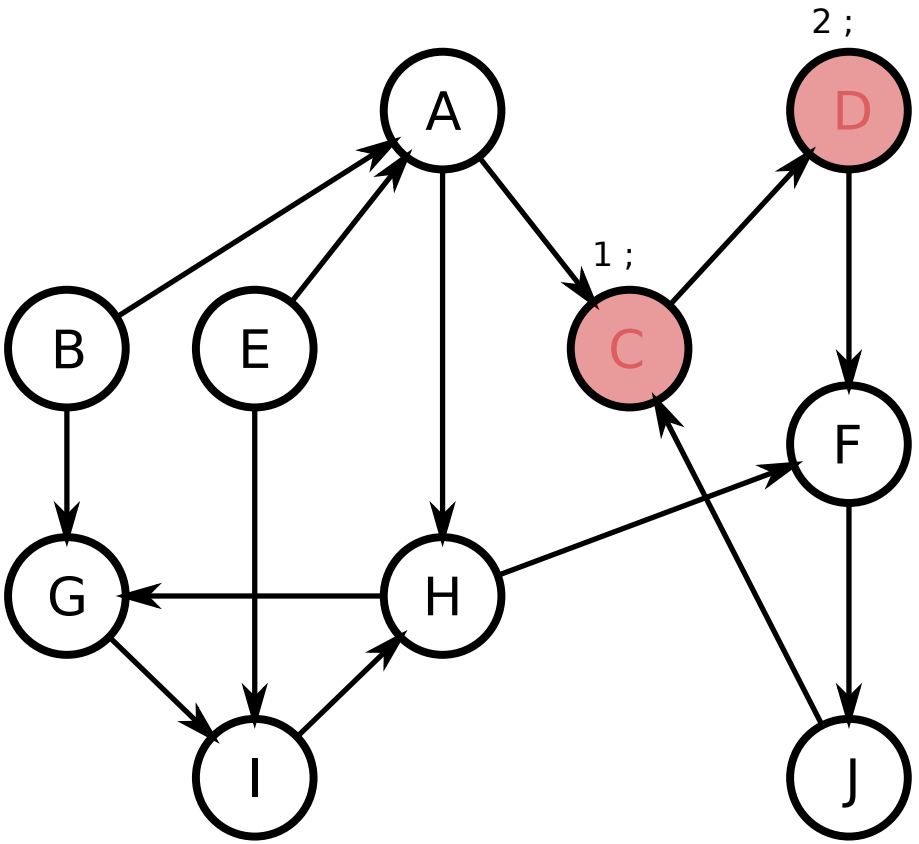
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

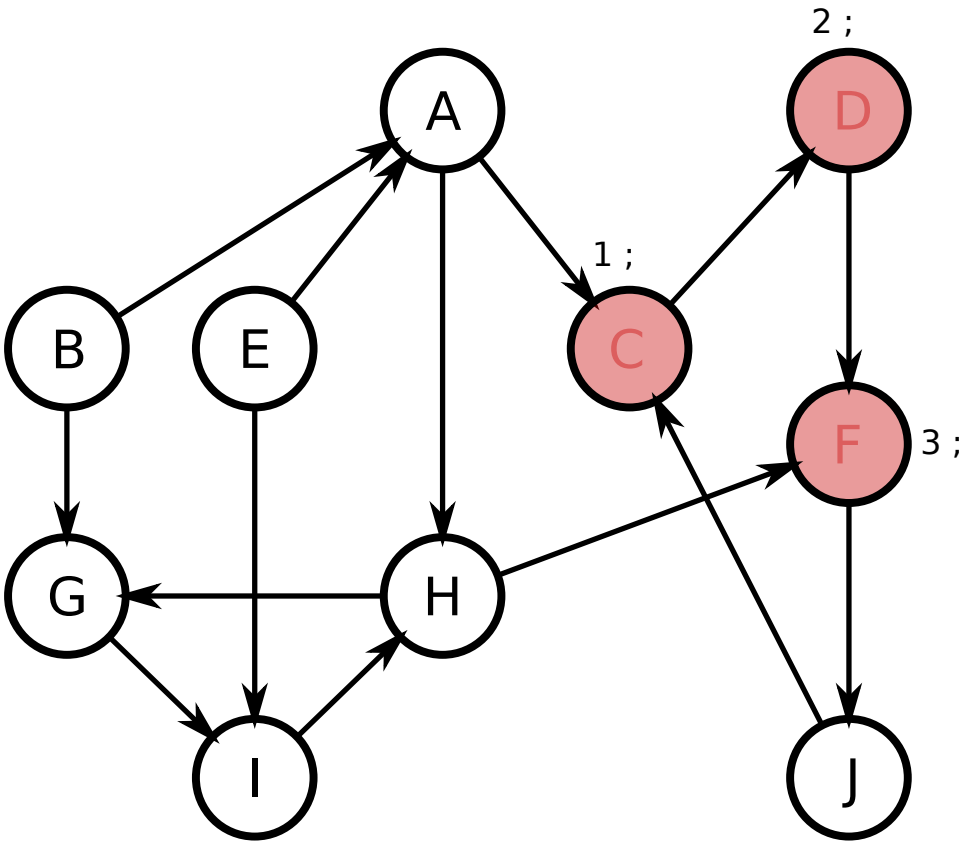
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

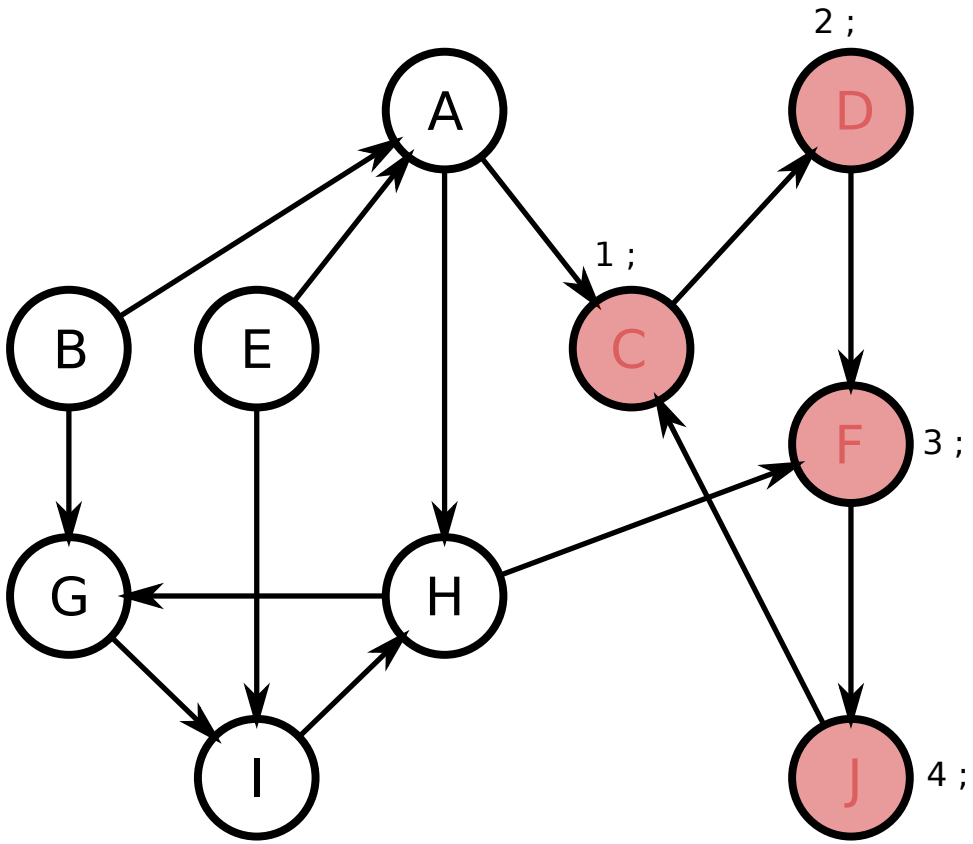
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

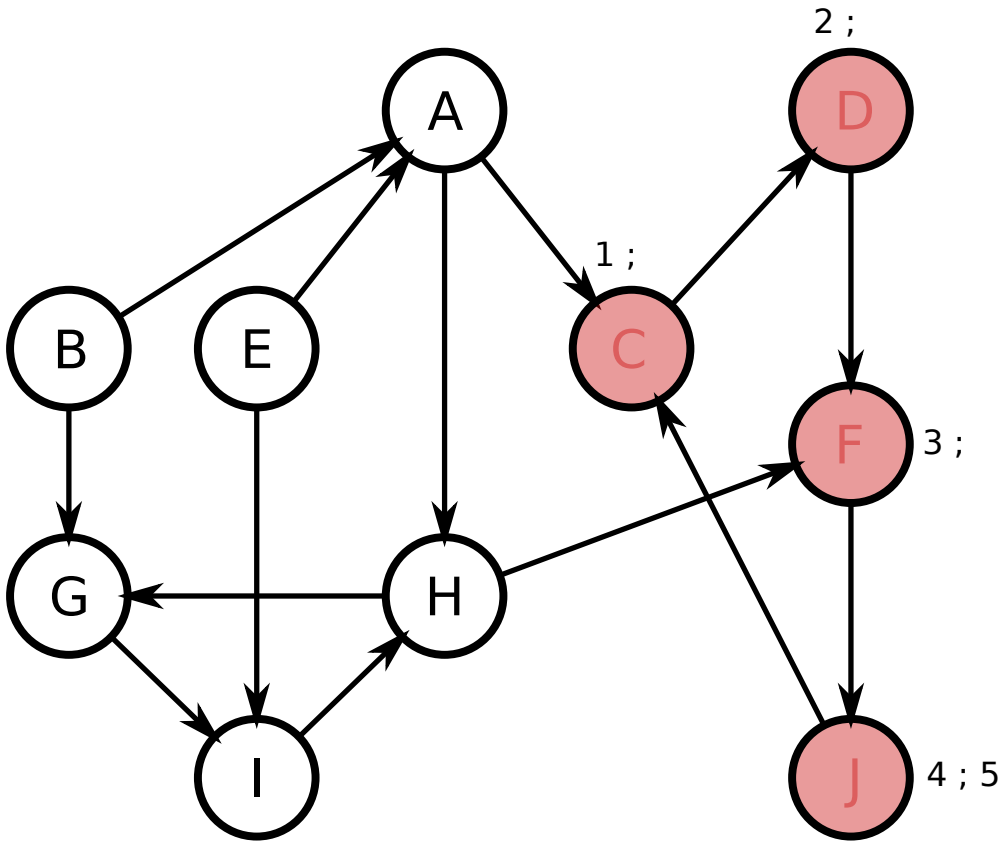
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

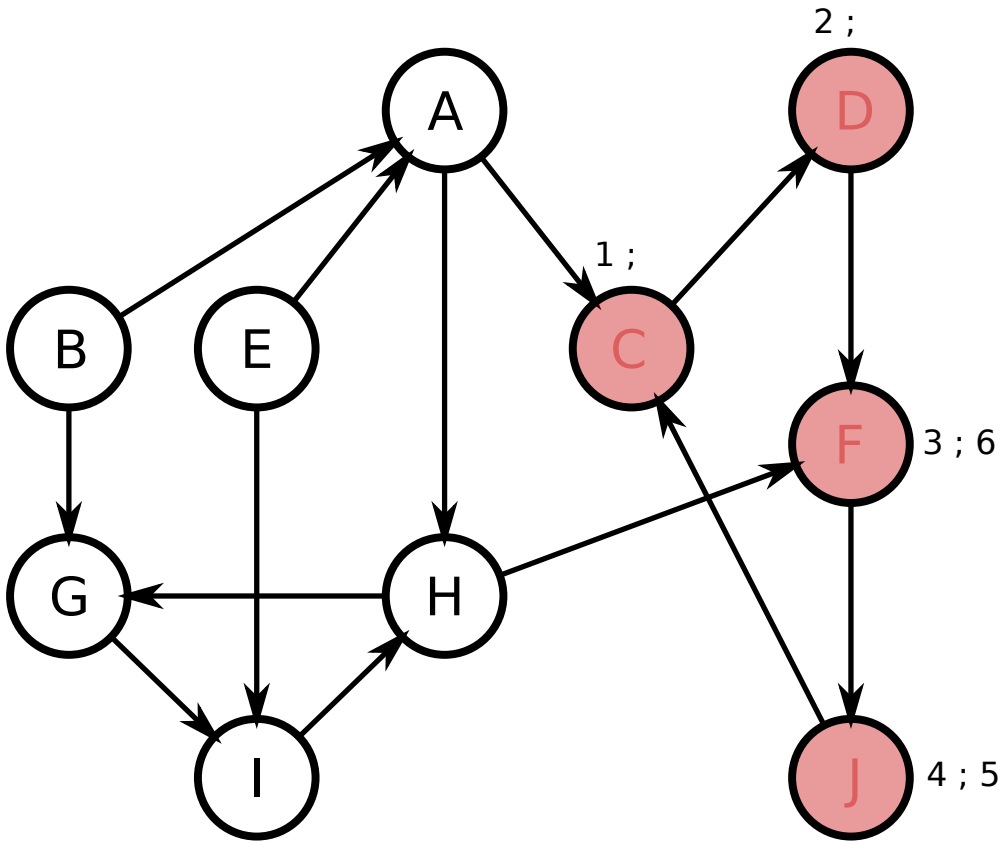
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

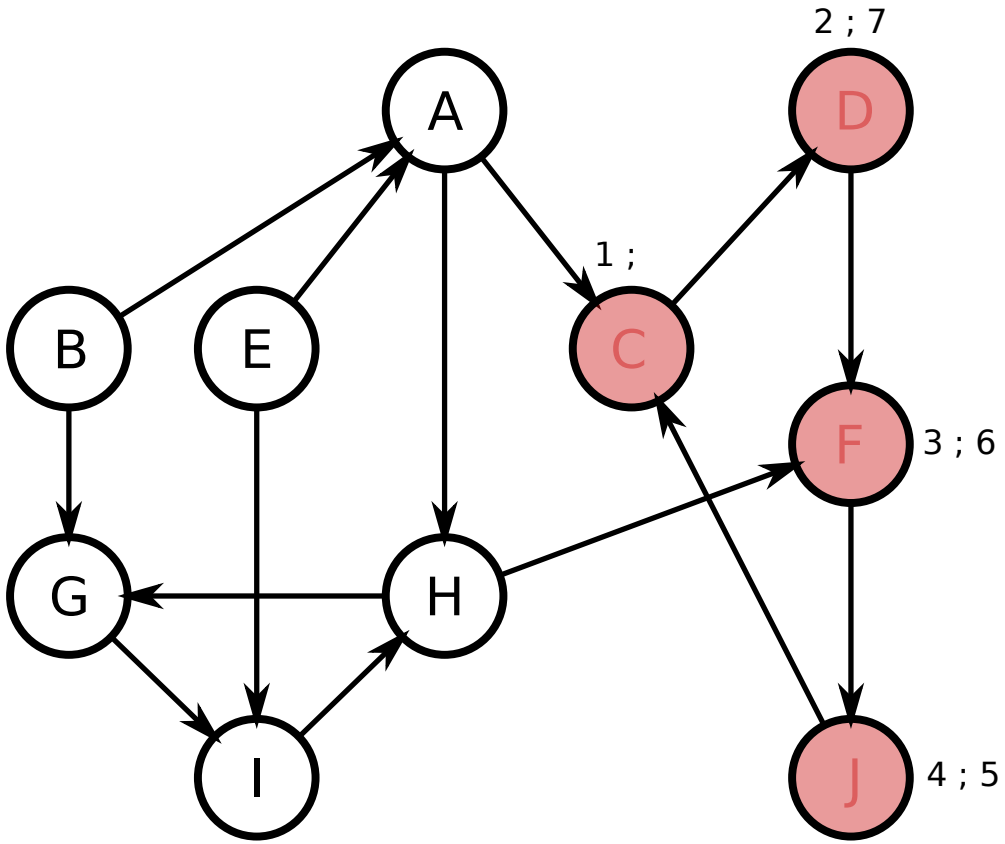
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

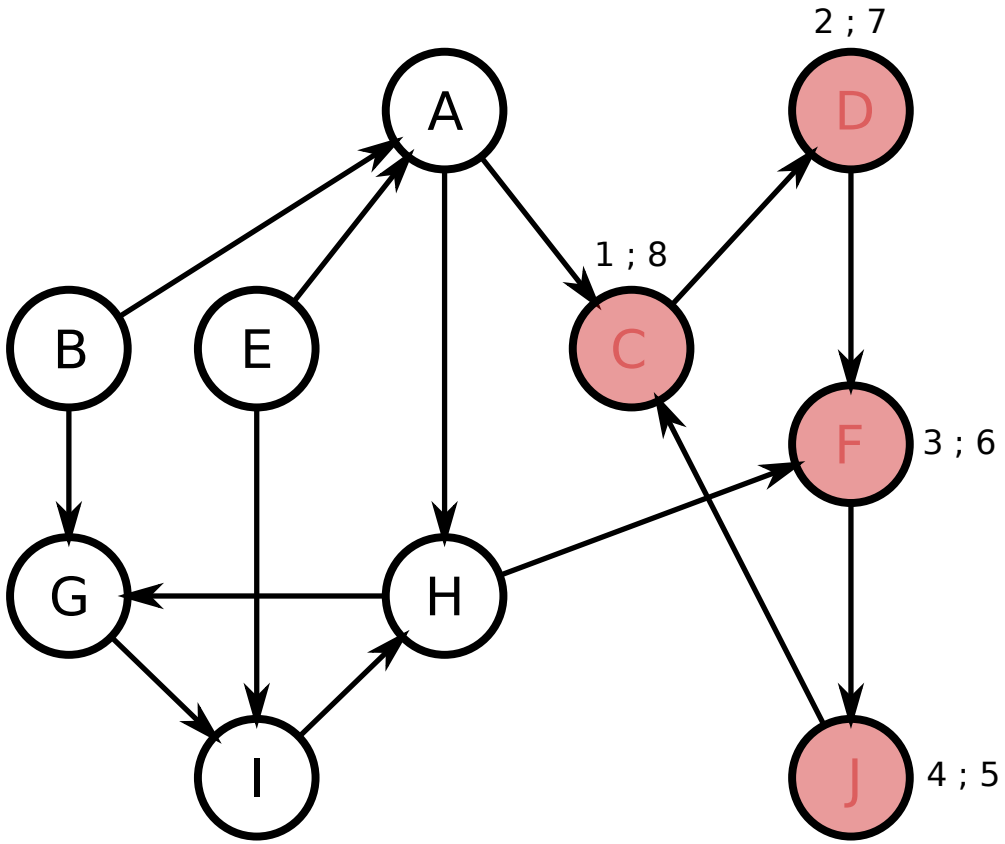
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

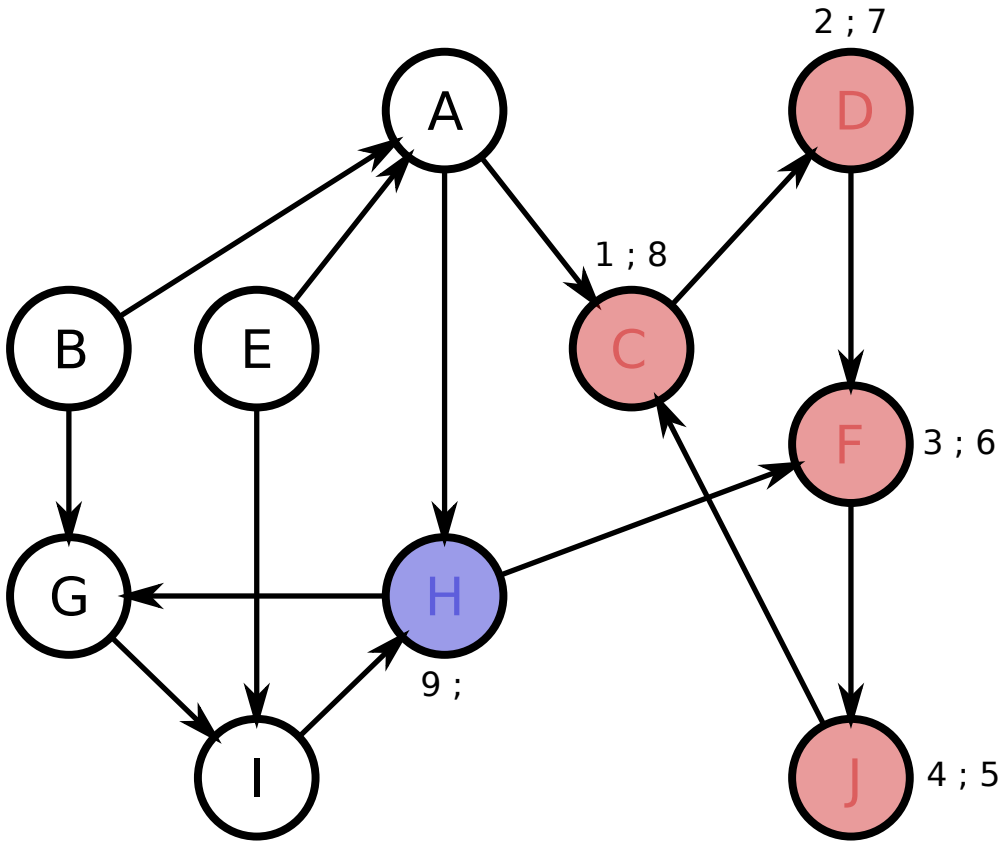
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

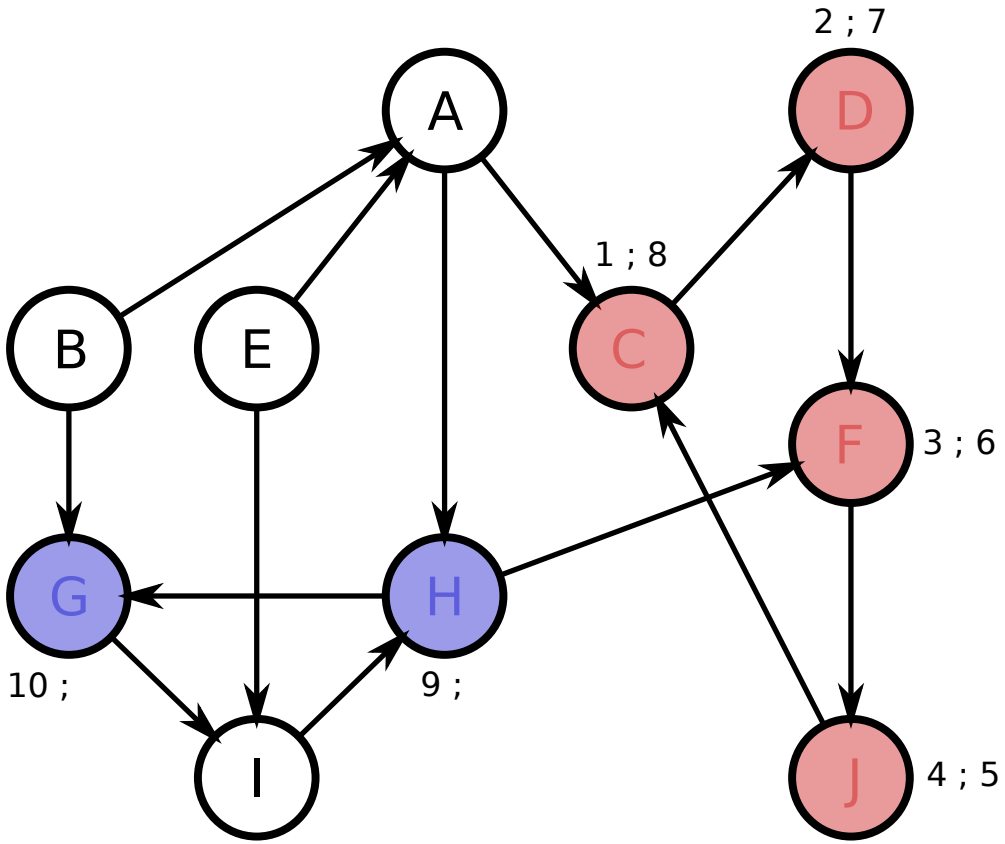
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

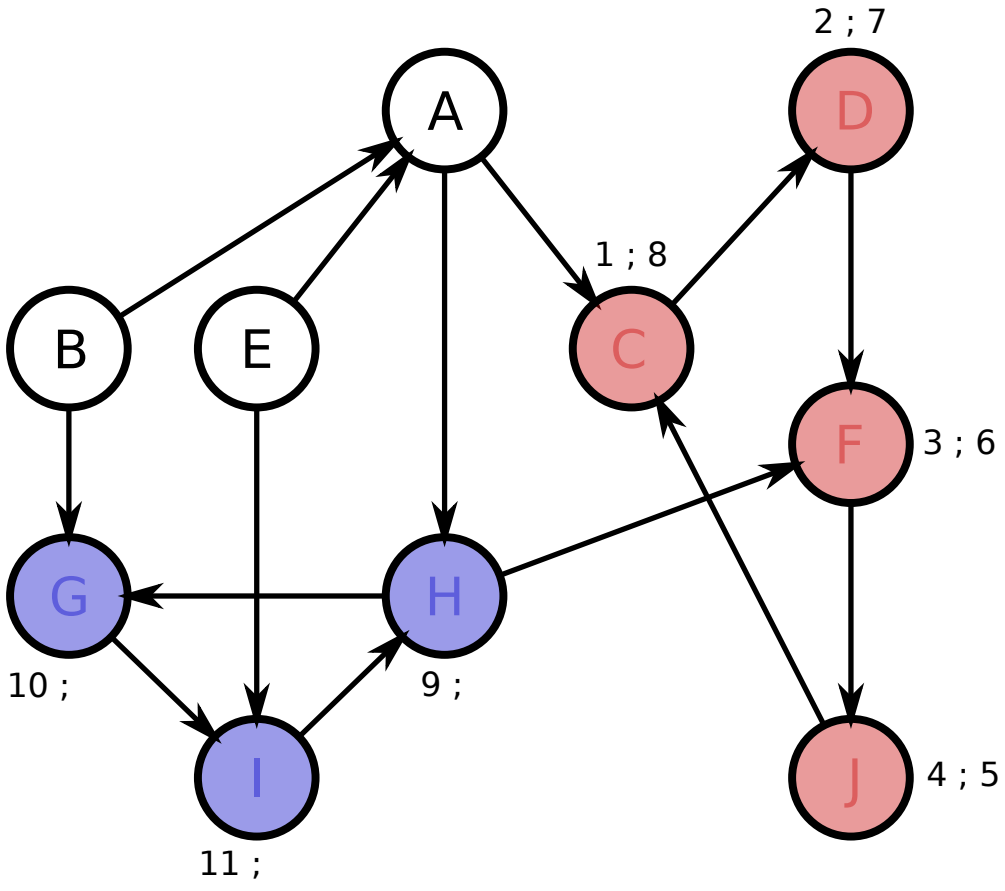
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

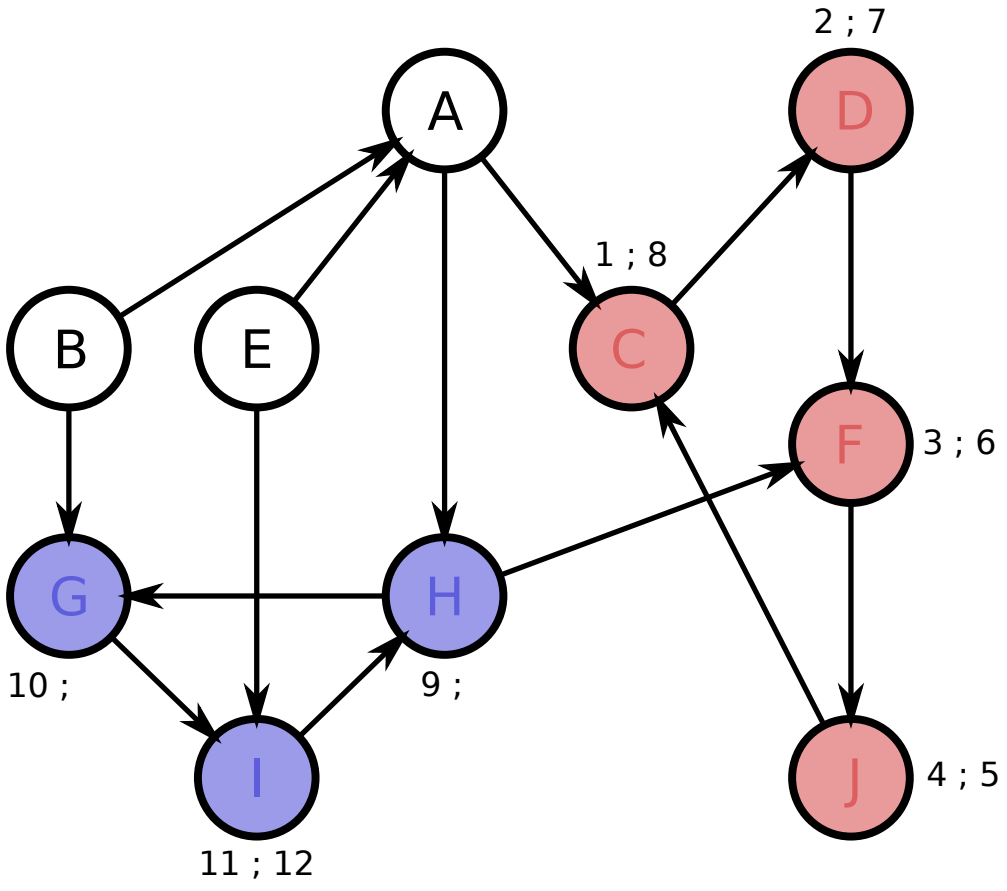
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

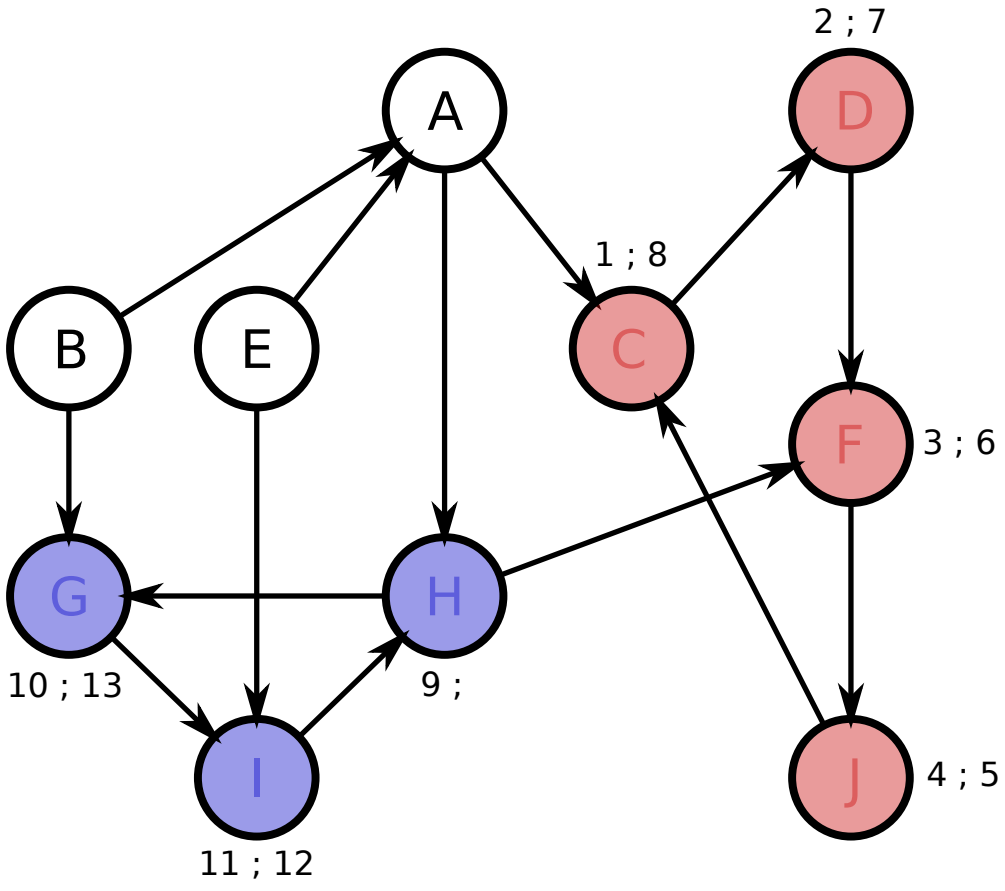
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

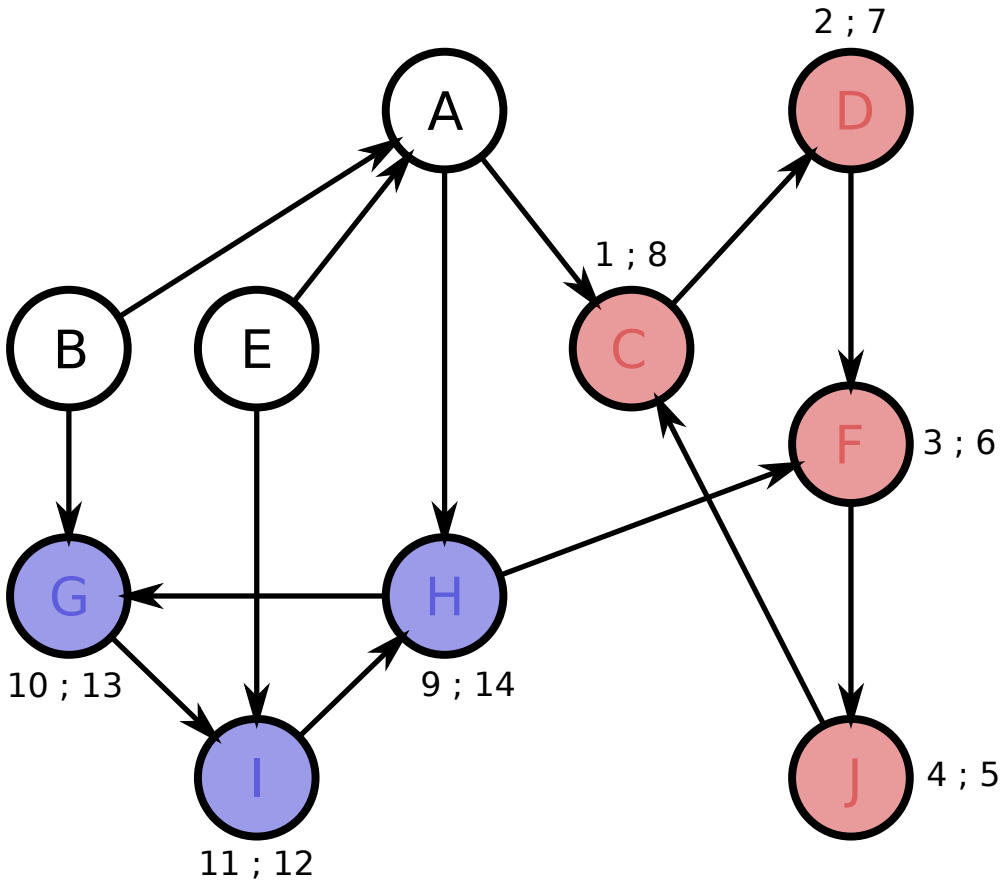
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

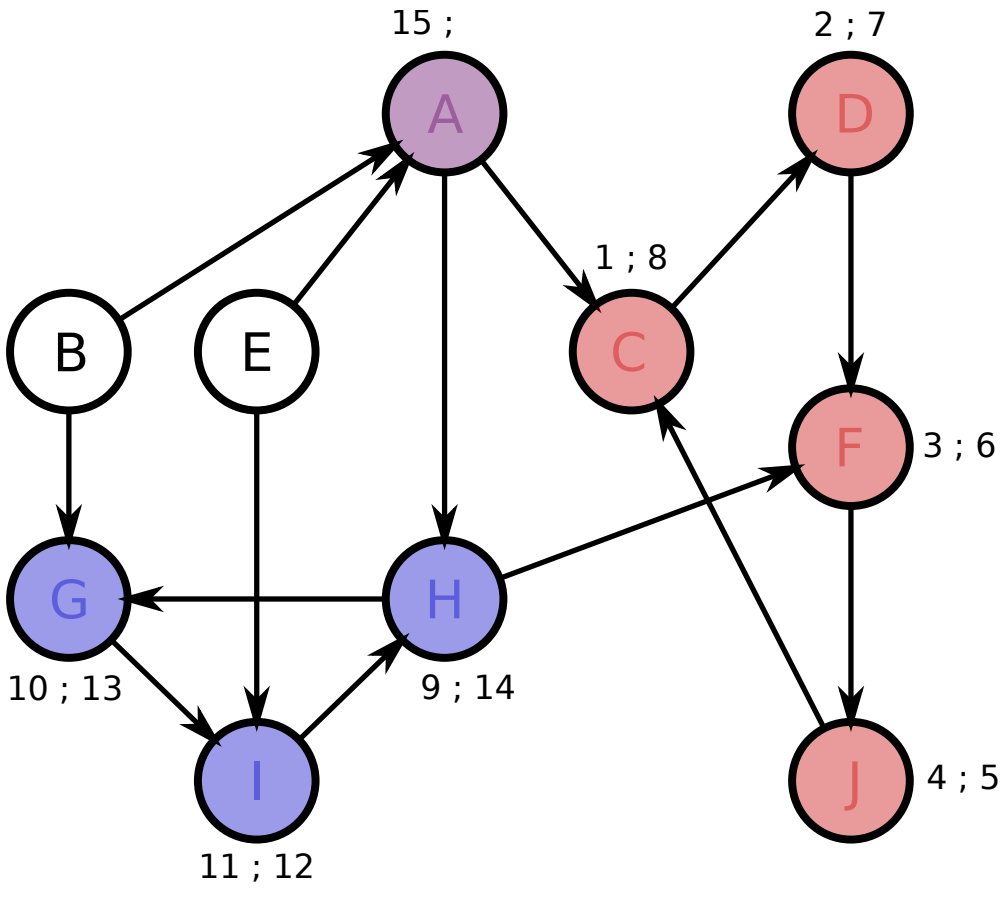
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

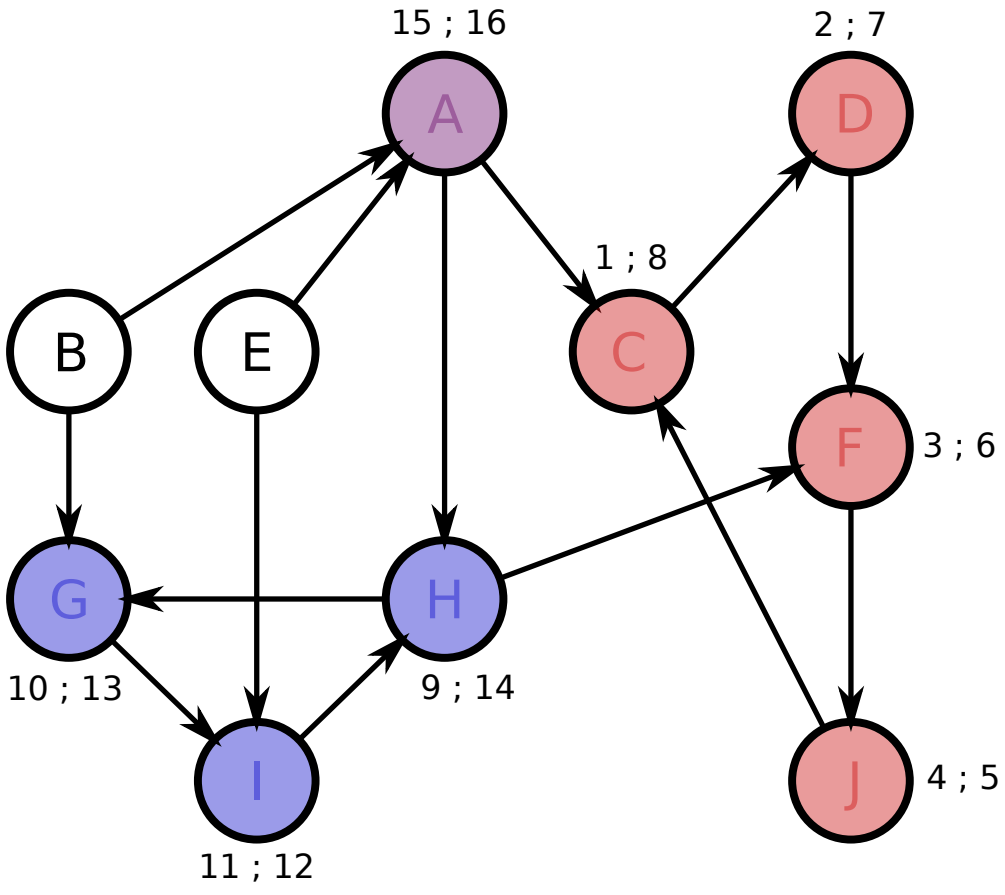
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

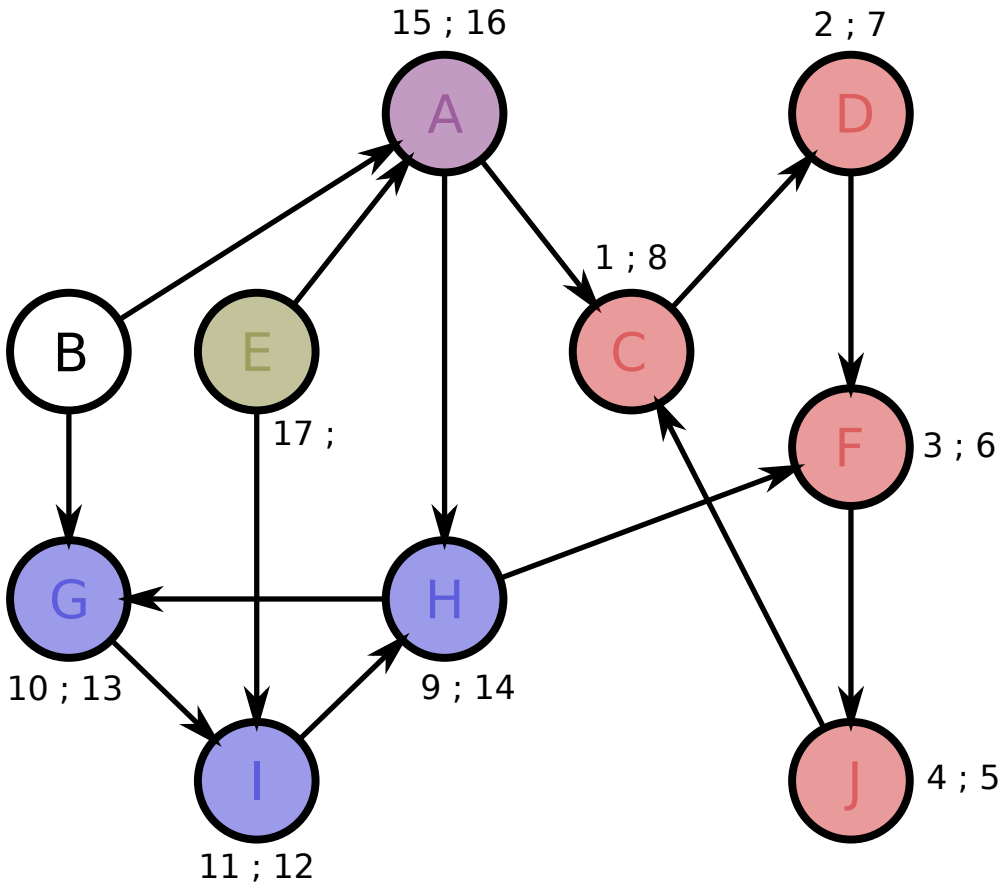
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

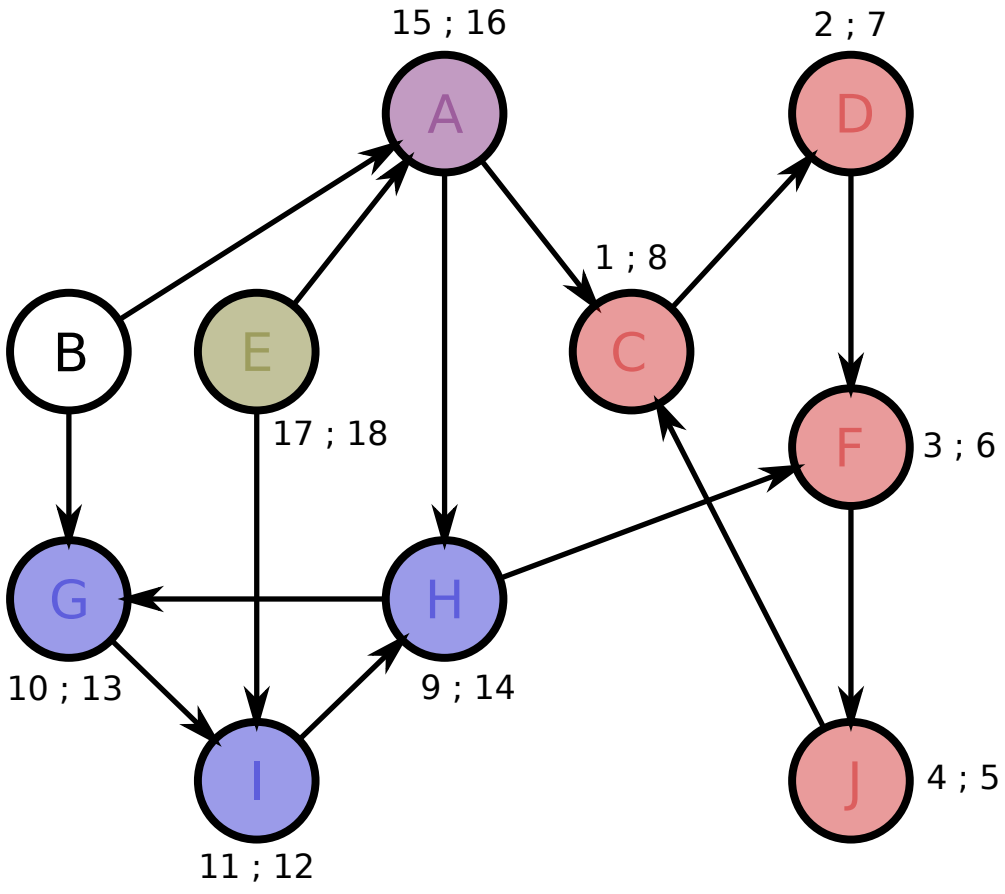
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

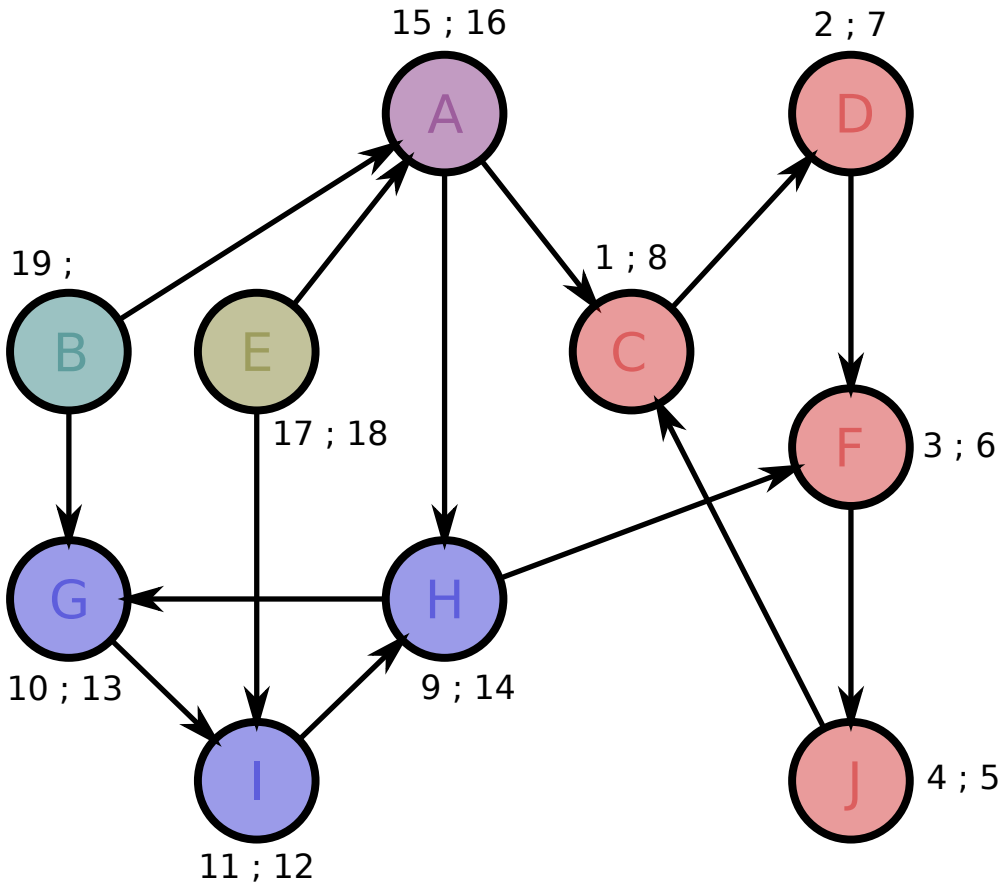
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

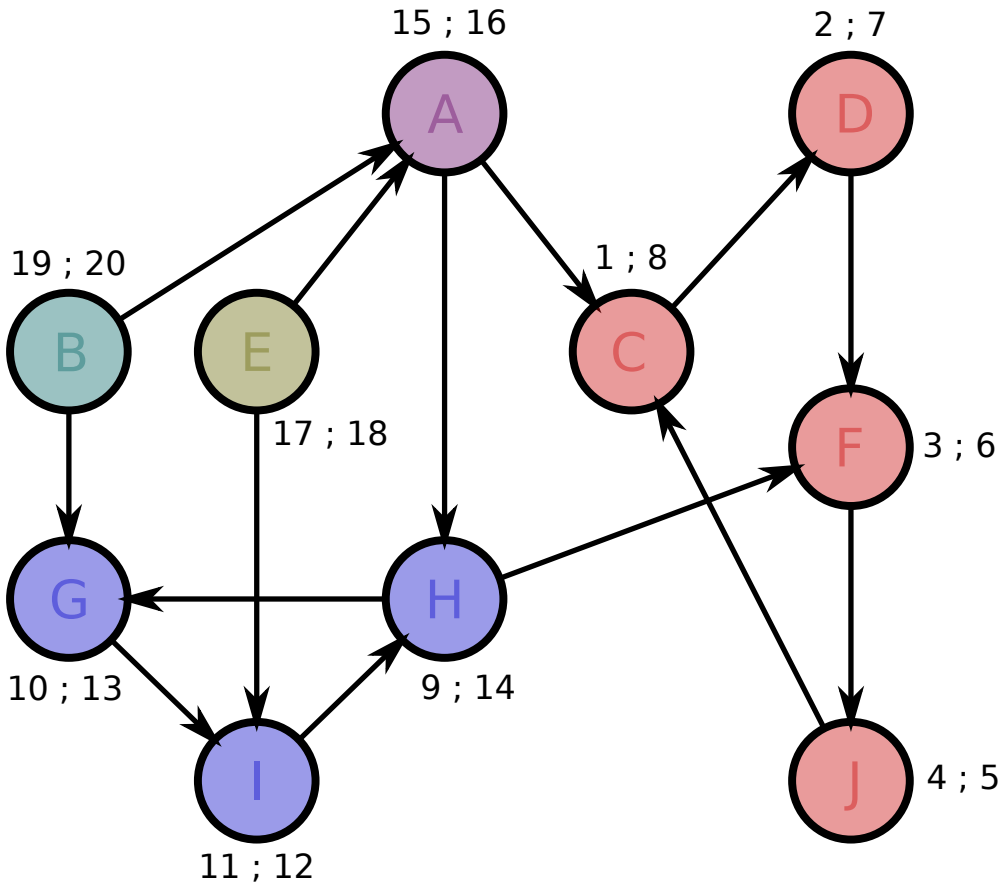
C ; J ; F ; H ; I ; G ; D ; A ; E ; B



DFS sur le graphe

Ordre de parcours des sommets

C ; J ; F ; H ; I ; G ; D ; A ; E ; B



Méta-graphe