Algorithmique des graphes

David Pichardie

15 Mars 2018

Bilan du CM1

- Organisation du module
- Définitions (orienté, non-orienté, degrés, chemin, cycle, connexité, composantes connexes)
- Les graphes sont partout
- Représentation des graphes
- Parcours en profondeur récursif

Parcours de graphe

Parcours en profondeur (procédure recursive)

Version avec dates début/fin

```
VISITE(i) =
  date <- date + 1
  DEBUT[i] <- date
  VU[i] <- vraie
  pour tout j ∈ Adj[i]
     si non VU[j] alors VISITE(j)
  date <- date + 1
  FIN[i] <- date
```

Efficacité

- On marque (VU) une seule fois chaque sommet
  - donc `VISITE(i)` termine toujours
  - donc `VISITE` n'est appelé qu'une seule fois par sommet

```
VISITE(i) =
  date <- date + 1
  DEBUT[i] <- date
  VU[i] <- vraie
  pour tout j ∊ Adj[i]
     si non VU[j] alors VISITE(j)
  date <- date + 1
  FIN[i] <- date

VU <- [faux, …, faux]
date <- 0
pour tout i ∊ S
   si non VU[i] alors VISITE(i)
```

Efficacité

- Le temps de calcul est dominé par les itérations de boucles
  - la boucle principale va s'exécuter |S| fois
  - la boucle secondaire va s'exécuter globalement

$$\sum_i |\text{Adj[i]}| = |A|$$

```
VISITE(i) =
  date <- date + 1
  DEBUT[i] <- date
  VU[i] <- vraie
  pour tout j ∊ Adj[i]
     si non VU[j] alors VISITE(j)
  date <- date + 1
  FIN[i] <- date

VU <- [faux, …, faux]
date <- 0
pour tout i ∊ S
   si non VU[i] alors VISITE(i)
```

Efficacité

La complexité pire cas
(en nombre d'appel de VISITE)
est $|S|+|A|$

Complexité linéaire

```
VISITE(i) =
  date <- date + 1
  DEBUT[i] <- date
  VU[i] <- vraie
  pour tout j ∈ Adj[i]
     si non VU[j] alors VISITE(j)
  date <- date + 1
  FIN[i] <- date

VU <- [faux, …, faux]
date <- 0
pour tout i ∈ S
   si non VU[i] alors VISITE(i)
```

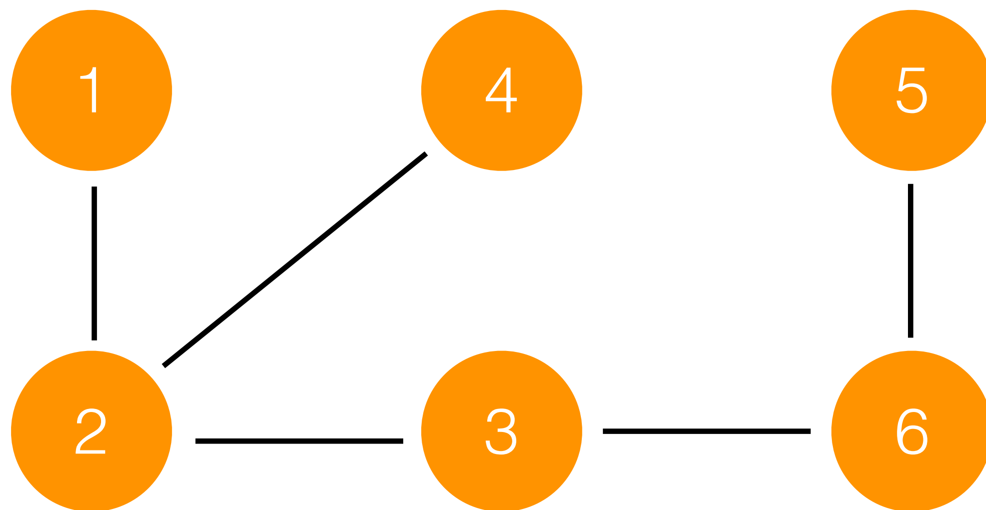
Vocabulaire

Vocabulaire

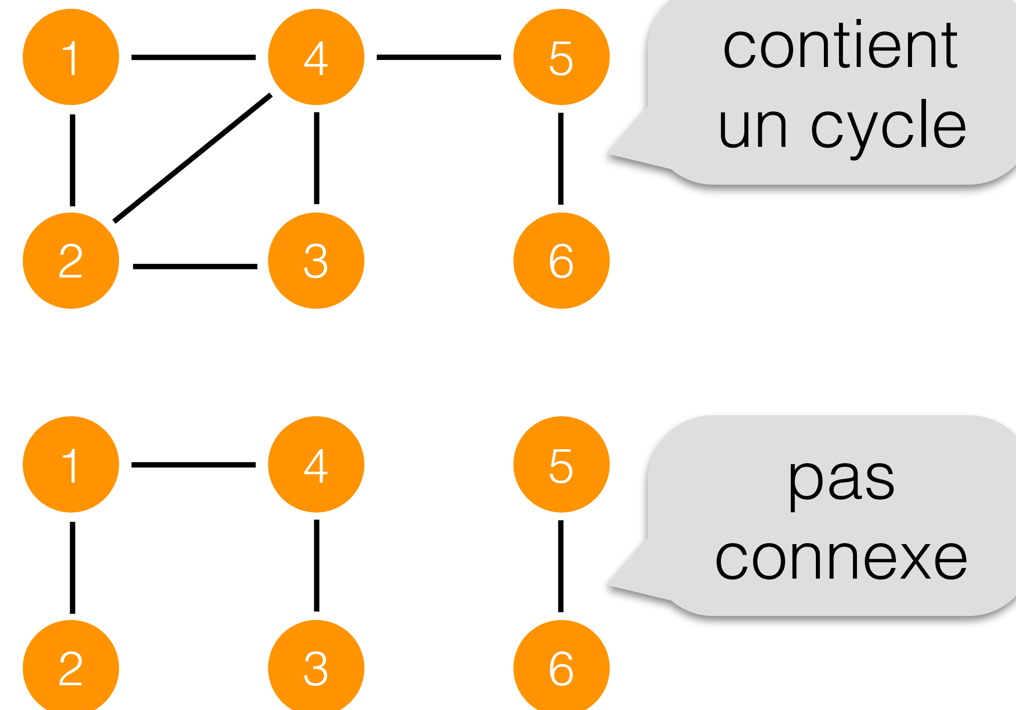
Définition

Un *arbre* est un graphe non-orienté connexe et acyclique (ne contient pas de cycle).

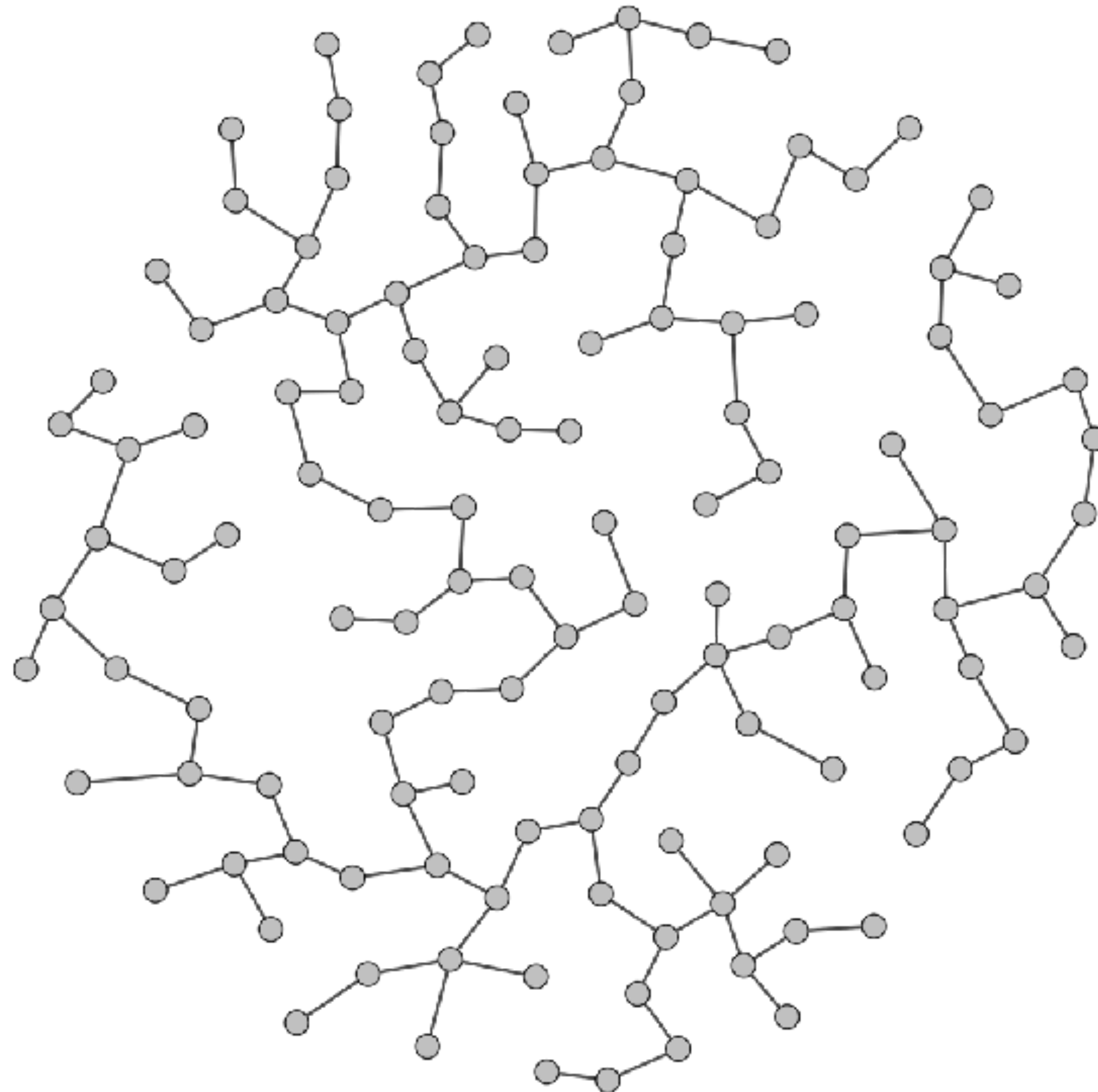
Exemple



Contre-exemple



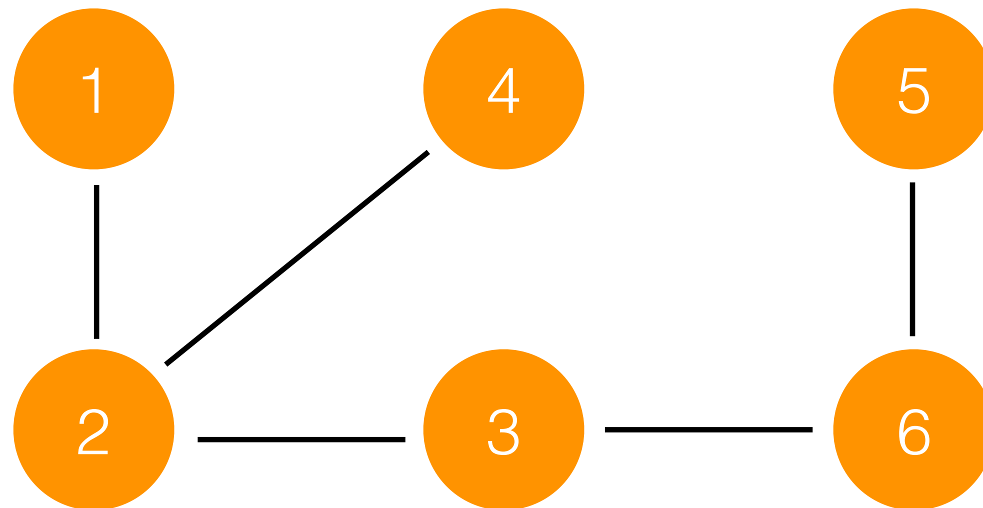
Les arbres



Les arbres

- Les arbres ont des propriétés fortes qui permettent de simplifier et accélérer certains algorithmes. Nous verrons des exemples.

- Dans un arbre *(S,A)*, $|S|=|A|+1$



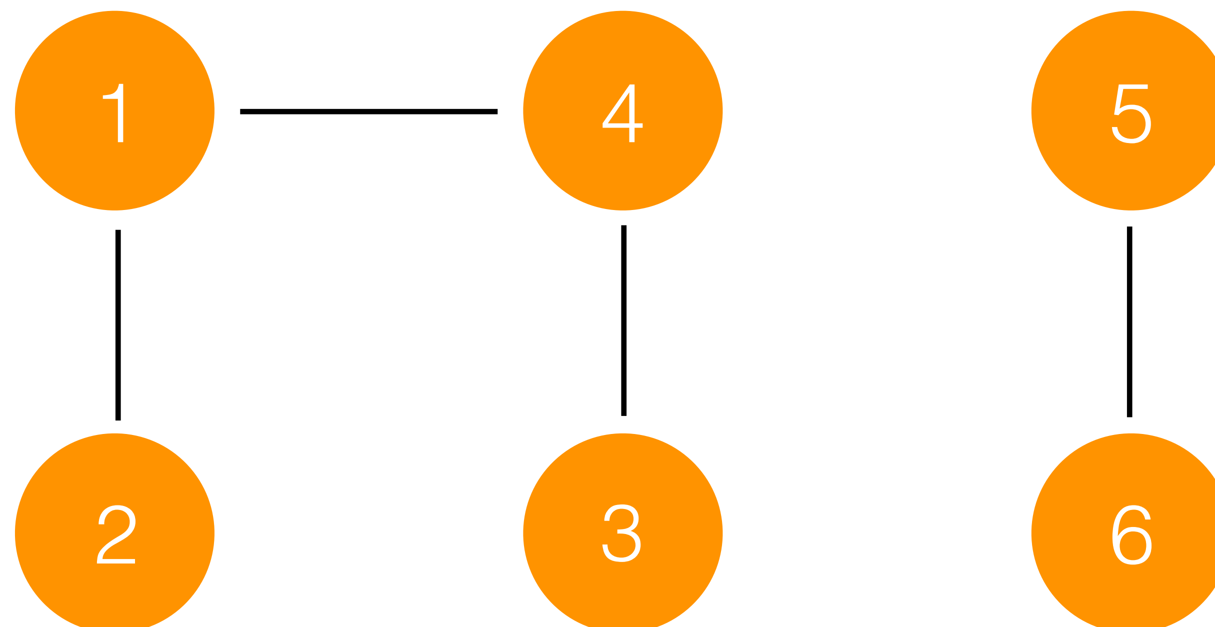
Vocabulaire

Définition

Une *forêt* est un graphe non-orienté acyclique.

Exemple

ses composantes connexes sont donc des arbres



Forêt de parcours

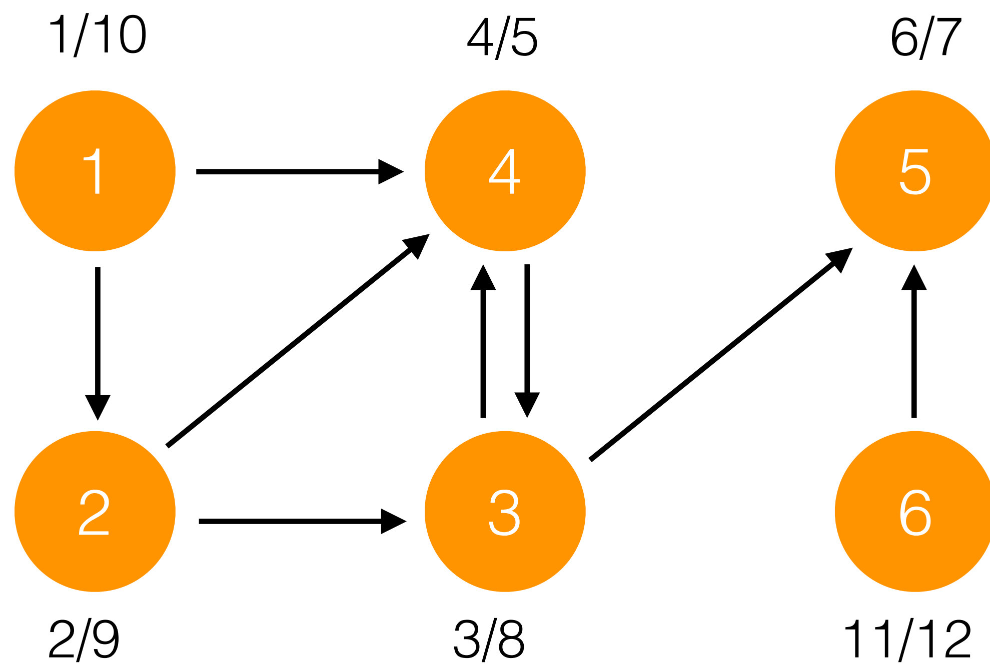
Pendant un parcours en profondeur récursif, on peut créer une relation *parent* qui mémorise pour chaque sommet j

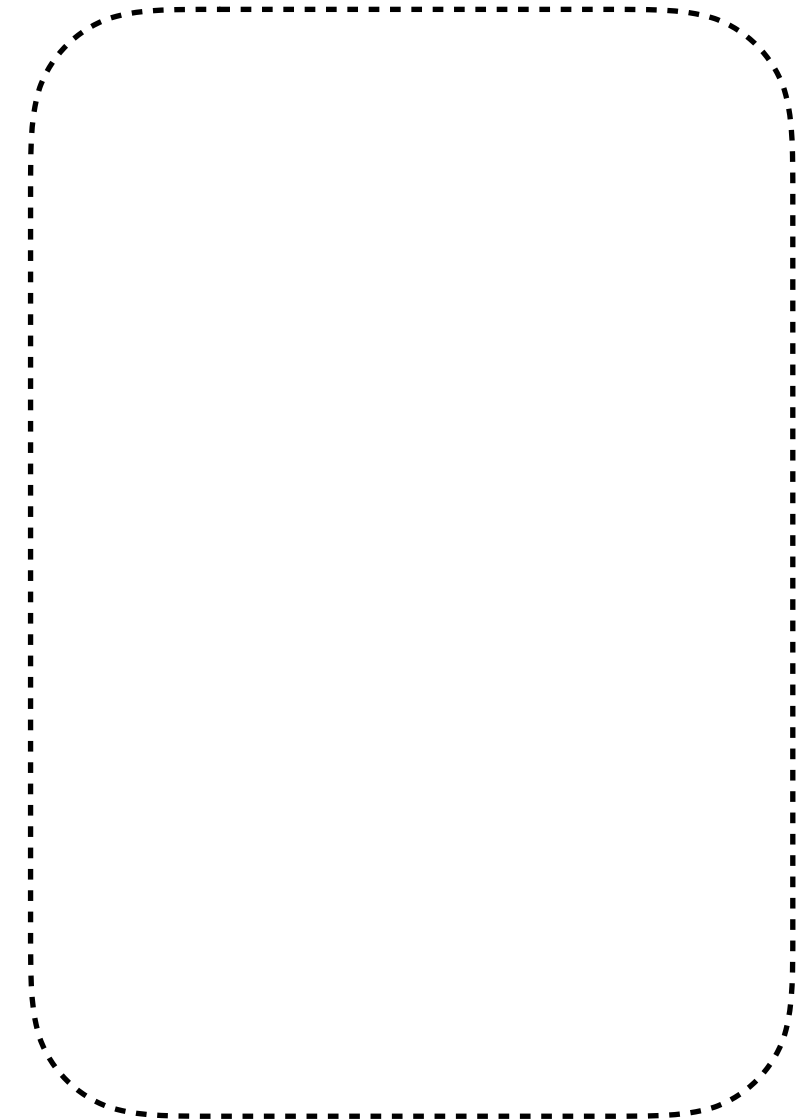
- le sommet *i* responsable de sa visite,
- -1 si le sommet *j* est visité par la boucle principale du parcours

Le graphe non-orienté associé à *parent* (une arête entre *i* et *j* si parent[j]=i), forme une *forêt de parcours*.

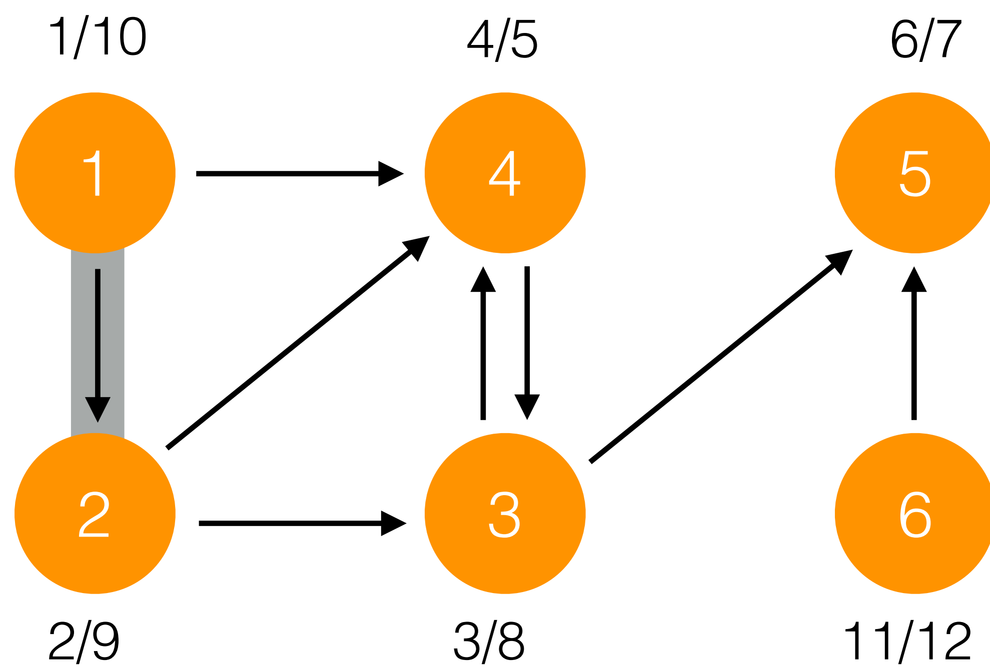
```
VISITE(i) =
  date <- date + 1
  DEBUT[i] <- date
  VU[i] <- vraie
  pour tout j ∊ Adj[i]
     si non VU[j] alors
        parent[j] <- i
        VISITE(j)
  date <- date + 1
  FIN[i] <- date

VU <- [faux, …, faux]
parent <- [-1, …, -1]
date <- 0
pour tout i ∊ S
   si non VU[i] alors VISITE(i)
```

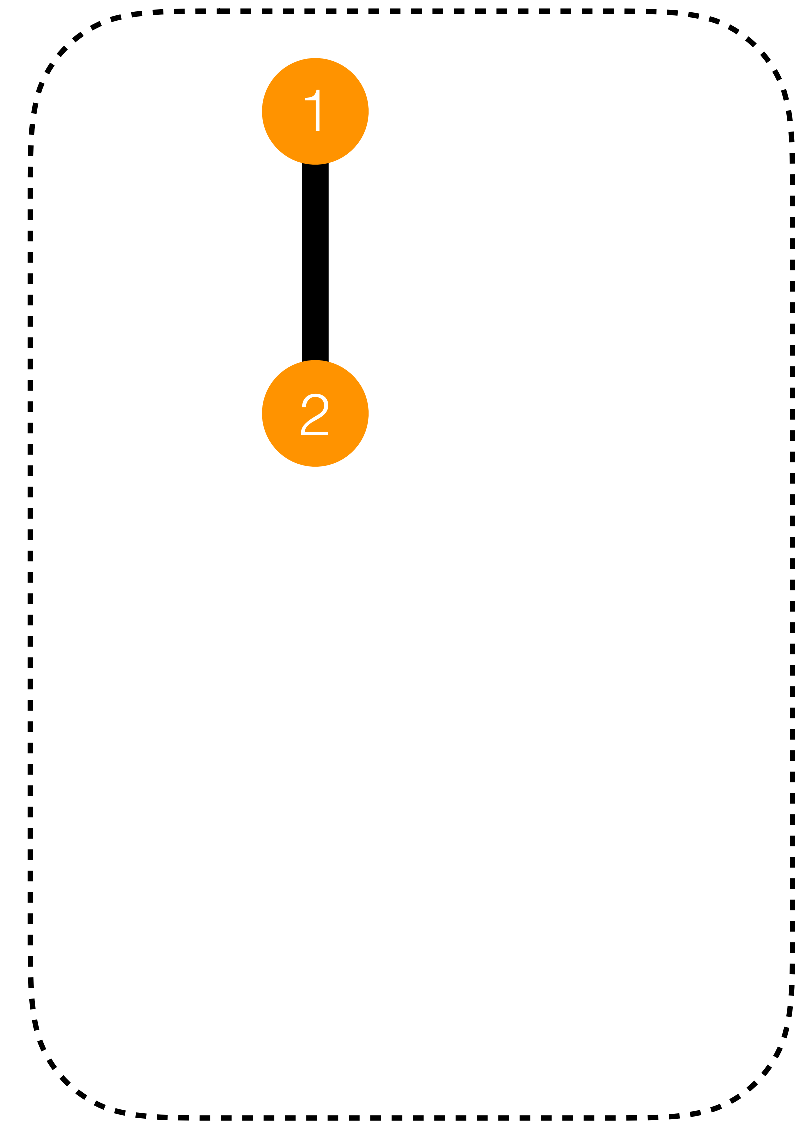
Exemple



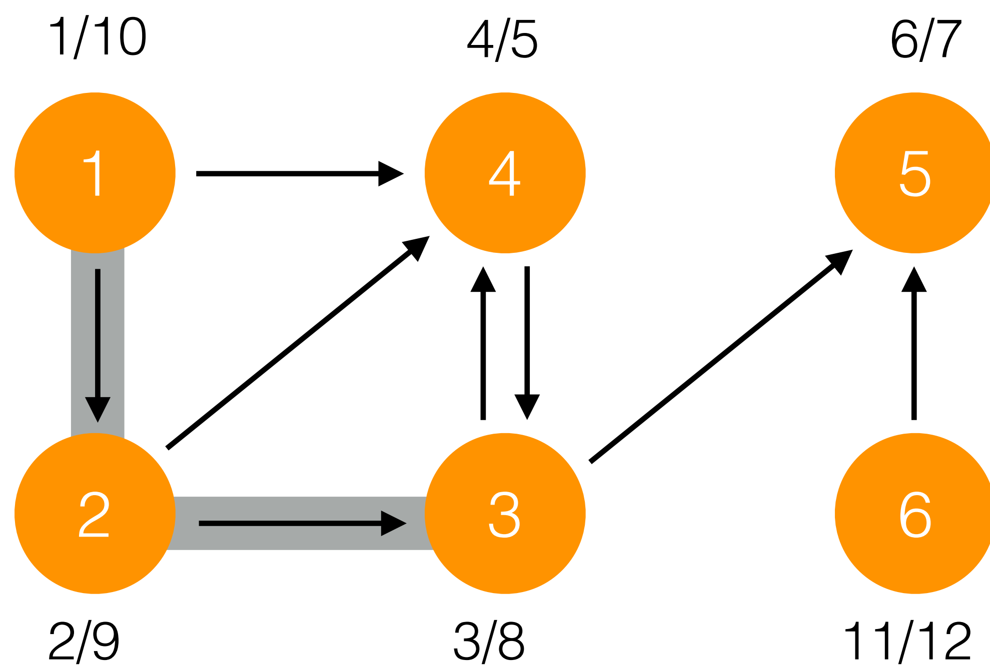


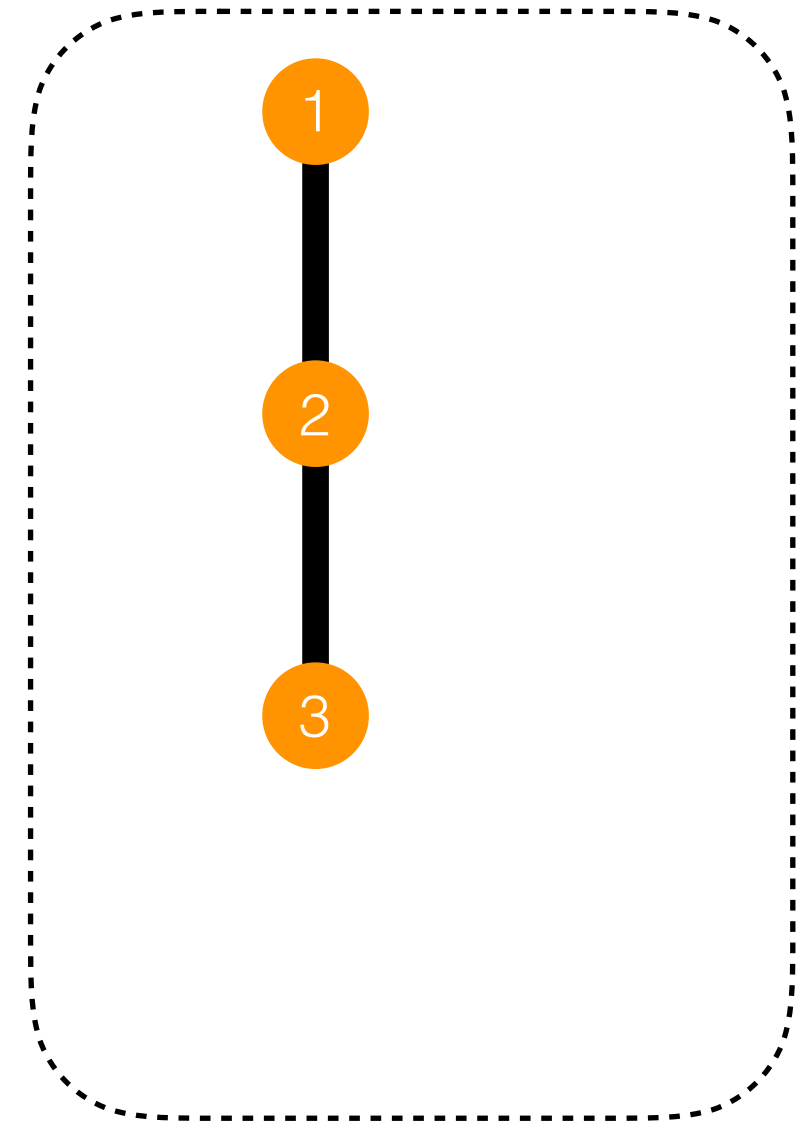
Exemple



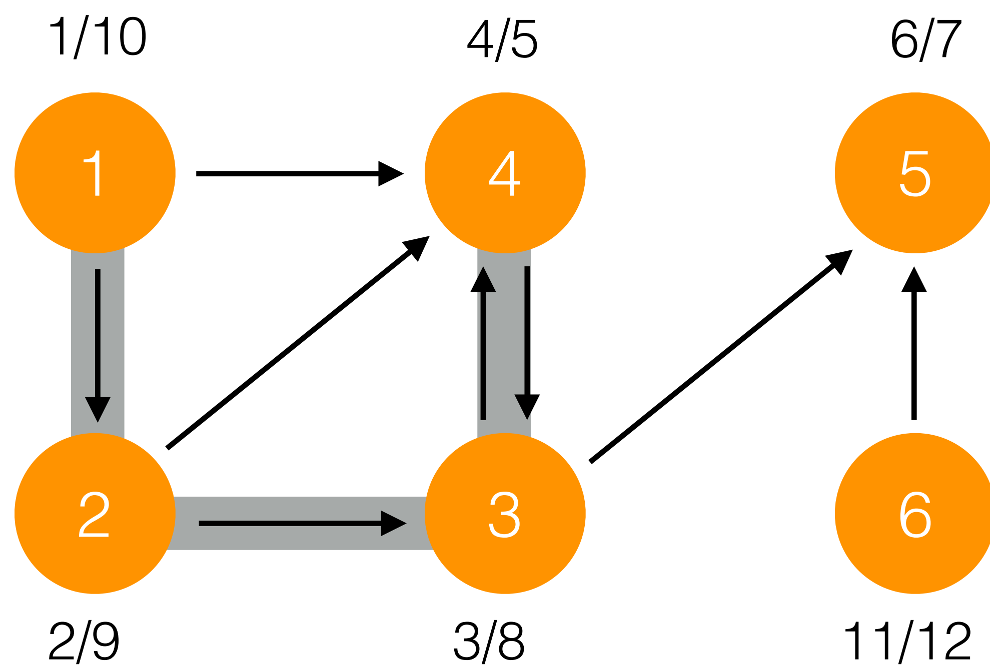


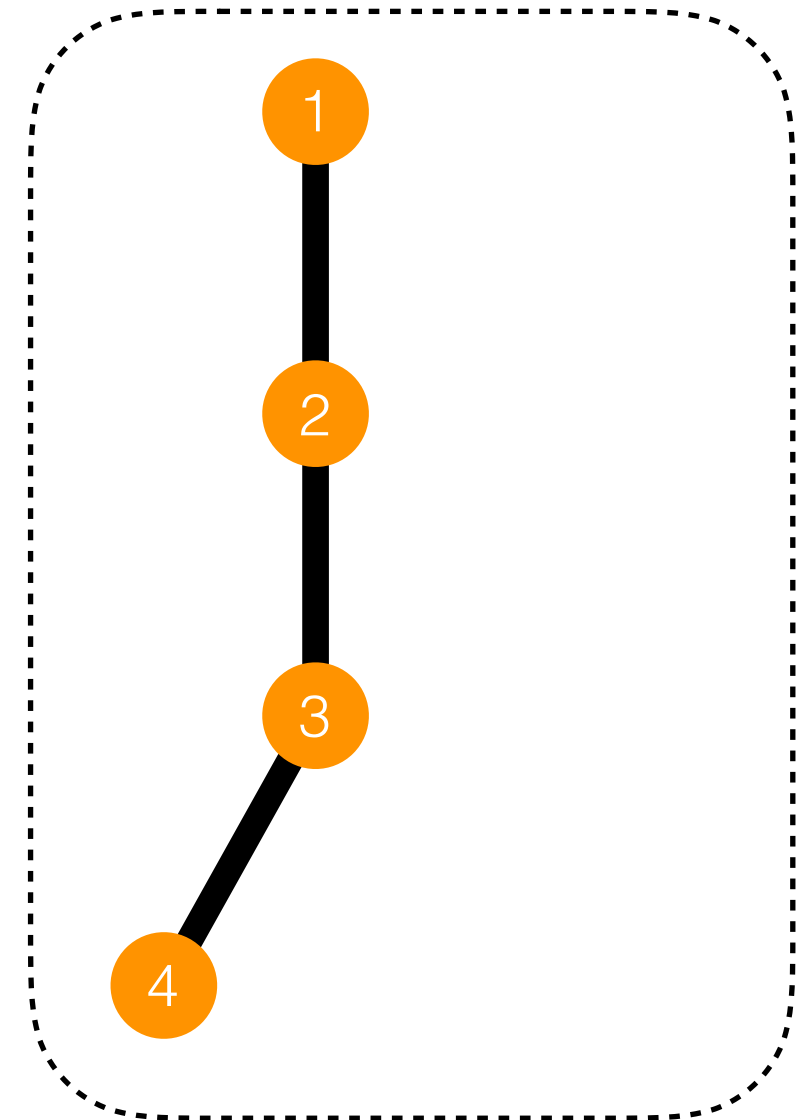
Exemple



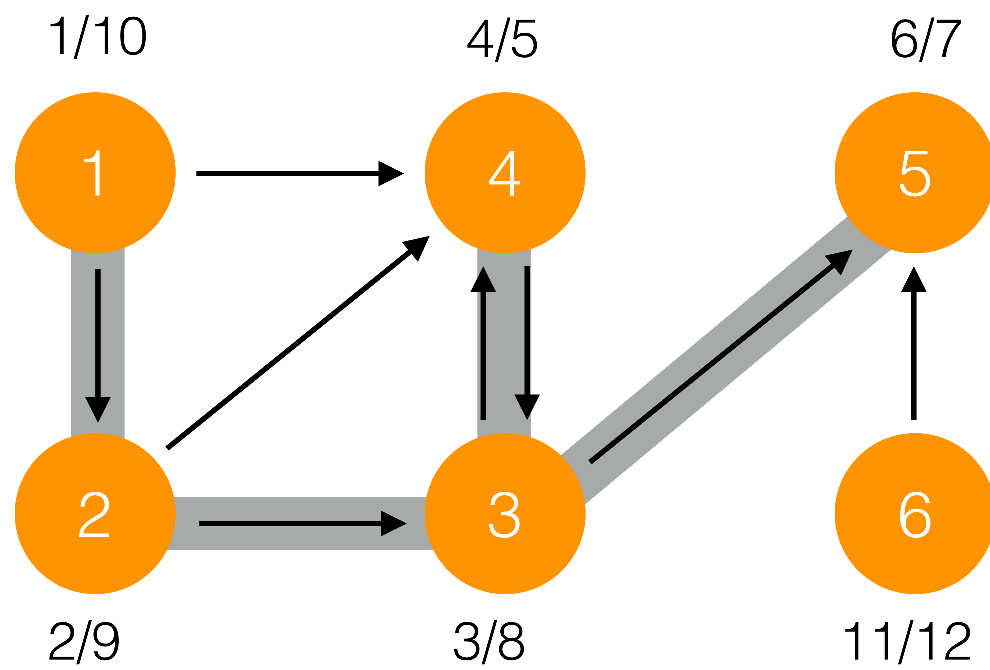


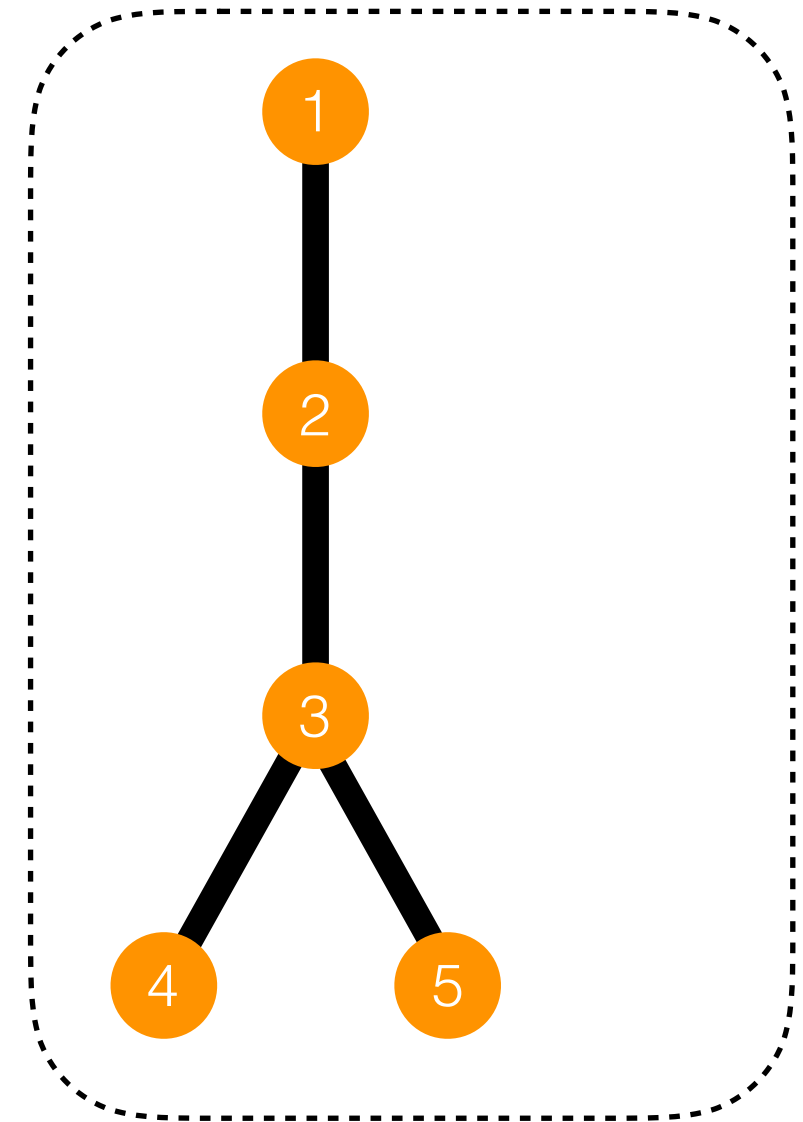
Exemple



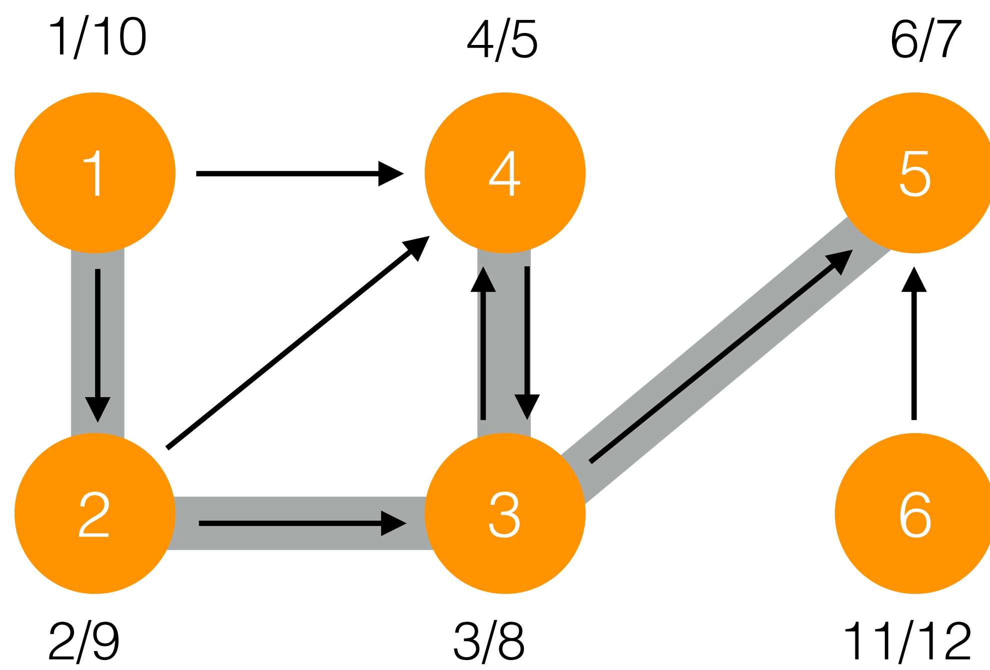


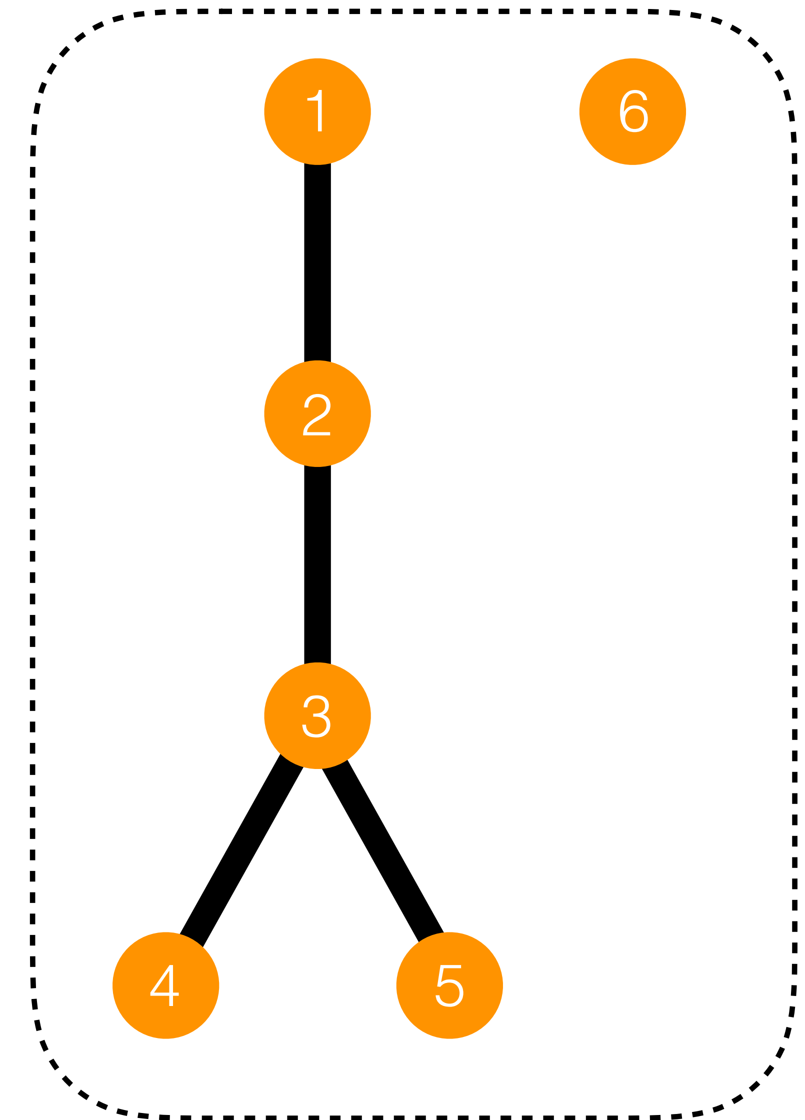
Exemple





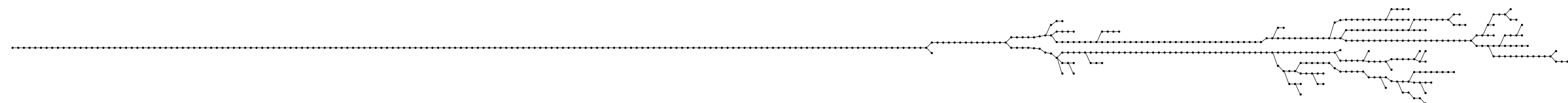
Exemple





Parcours en profondeur

N=500



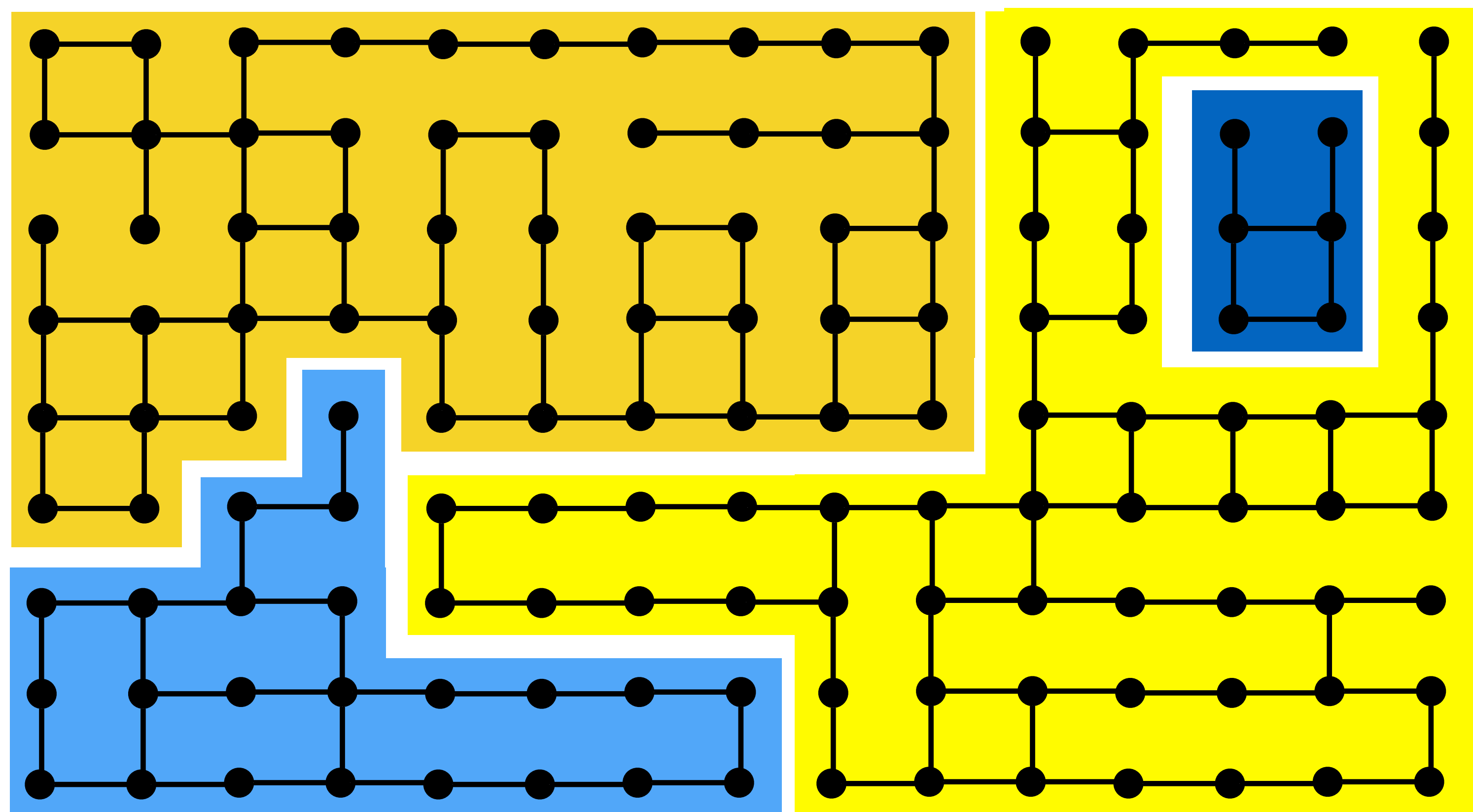
Arbre de parcours

Application #1

Compter les composantes connexes d’un graphe non-orienté

Propriétés du parcours en profondeur

- VISITE(i) ne va parcourir que des sommets accessibles depuis i.
- Il va tous les parcourir si aucun n'était marqué *VU* avant de démarrer VISITE(i). Dans un graphe non-orienté, cet ensemble forme la composante connexe de i.



Compter les composantes connexes d'un graphe non-orienté

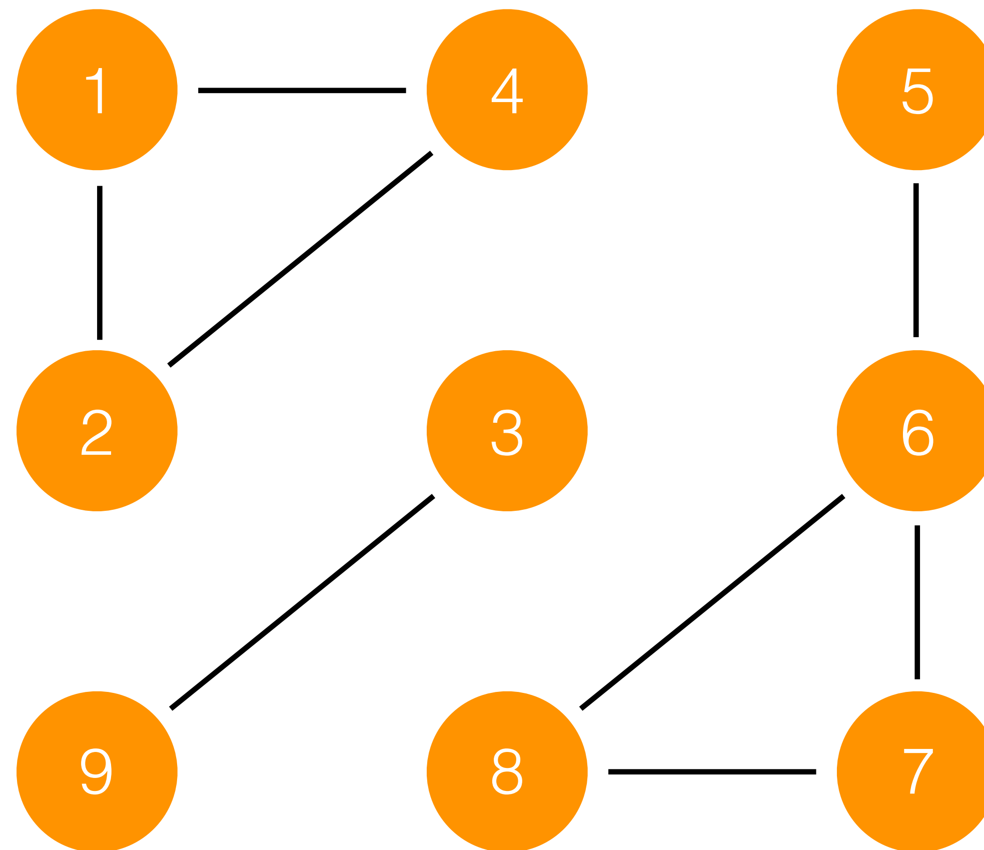
```
VISITE(i) =
 VU[i] <- vraie
  pour tout j ∊ Adj[i]
     si non VU[j] alors VISITE(j)

COMPTE_CC() =
    VU <- [faux, …, faux]
    cc <- 0
    pour tout i ∊ S
       si non VU[i] alors
           cc <- cc + 1
           VISITE(i)
    retourne cc
```

Cette variable va comptabiliser le nombre de composantes connexes dans le graphe

On découvre une nouvelle composante !

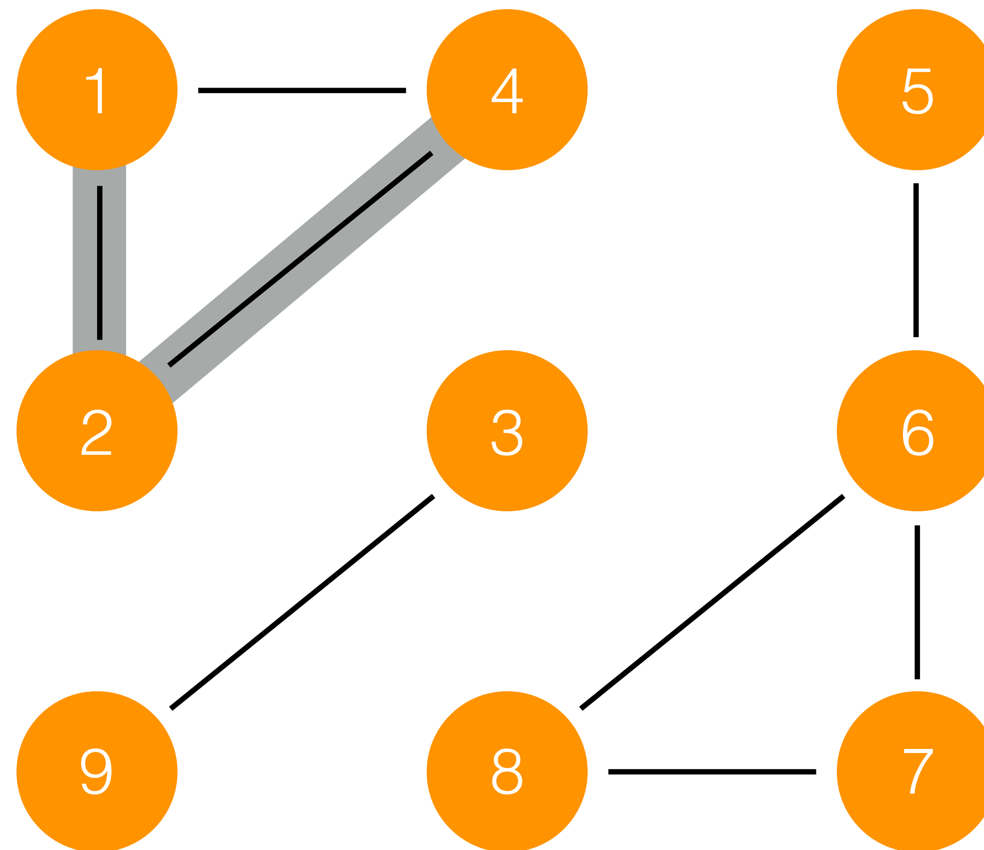
Exemple

cc = 0

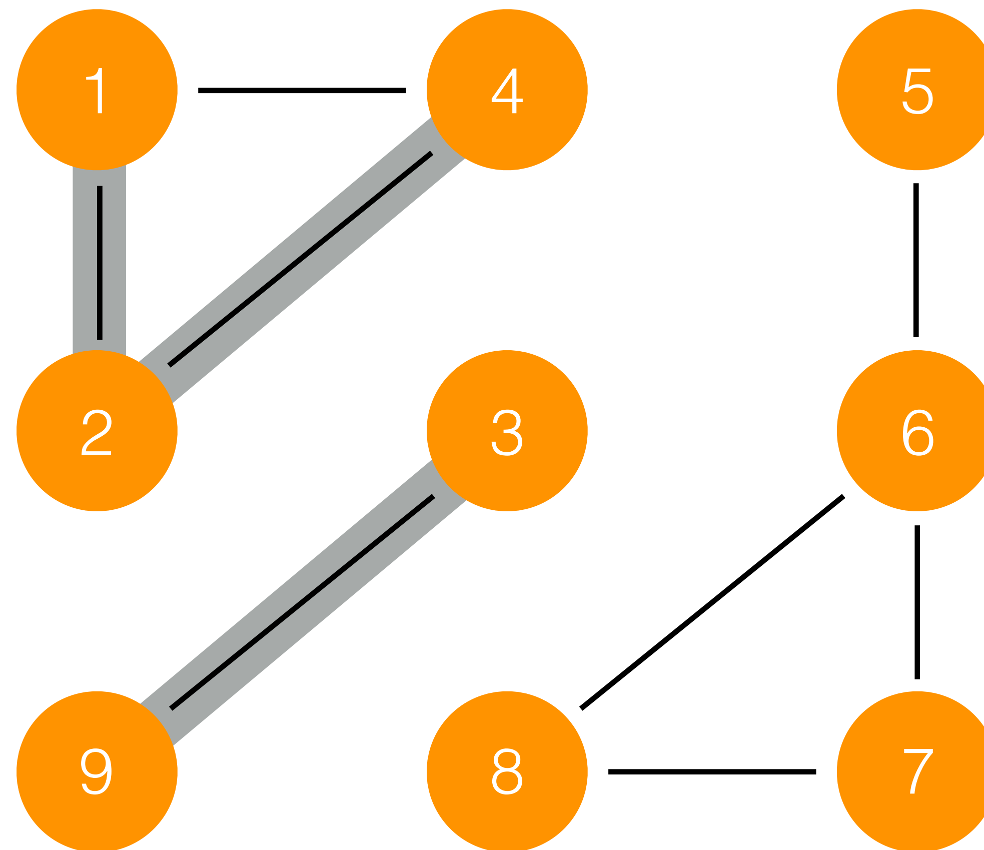


Exemple

cc = 1



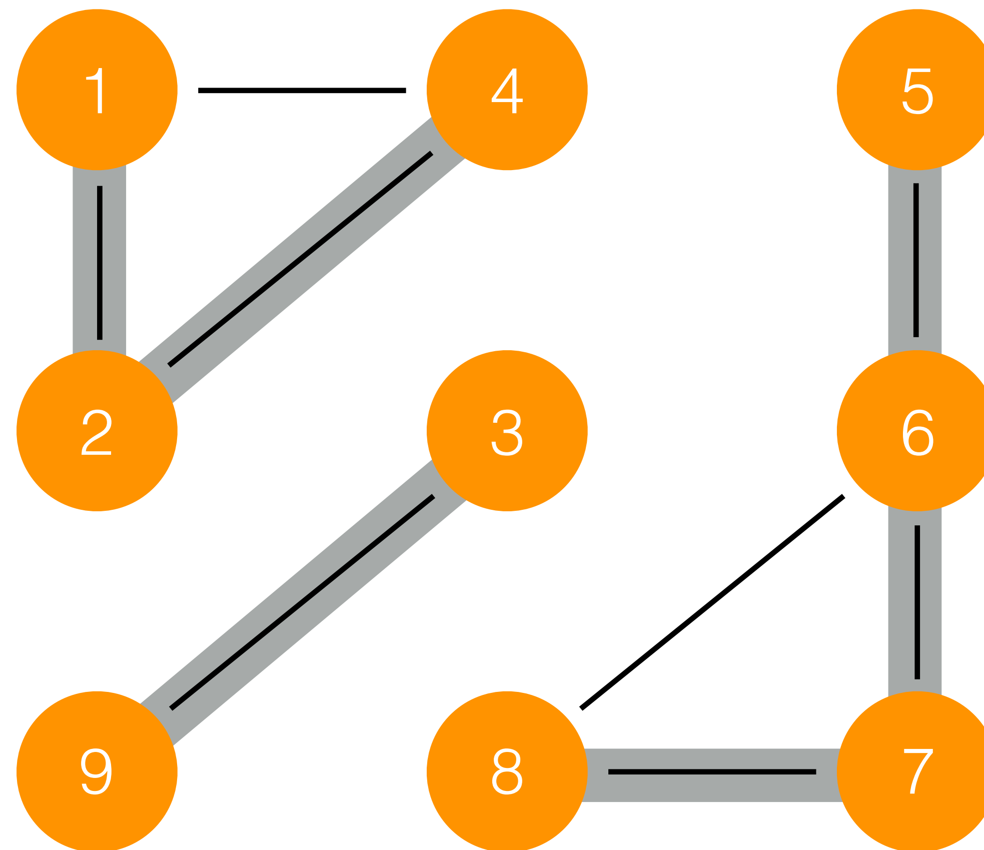
Exemple



cc = 2

Exemple

cc = 3



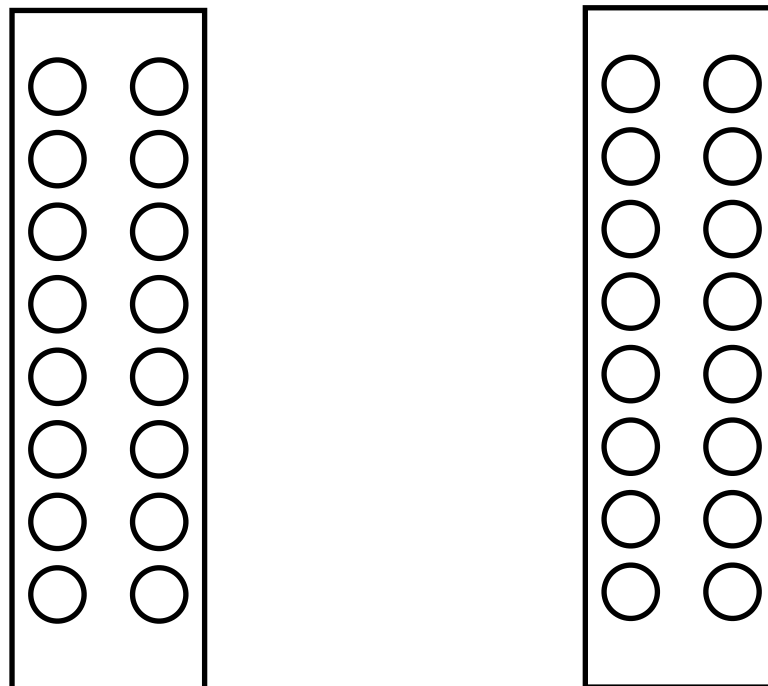
Application #2

Déterminer si un graphe est bipartie

Plan de table

Pour un repas de famille, David doit réaliser un plan de table en tenant compte des vieilles histoires de la famille. Les personnes qui ne s'aiment pas ne doivent pas se retrouver à la même table !

Avec seulement 2 tables, c'est une problème algorithmique facile.

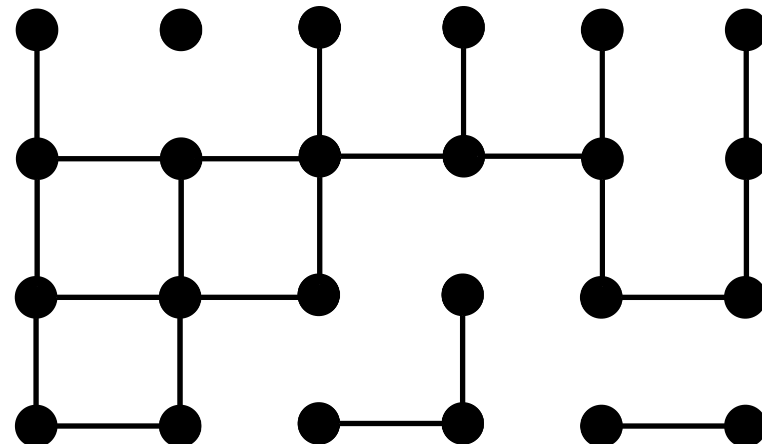


Vocabulaire

Définition

Un graphe non-orienté est *bipartie* si l'ensemble des sommets peut être séparé en deux disjoints S1 et S2, et que chaque arête du graphe relie un sommet de A1 à un sommet de A2.

Exemple

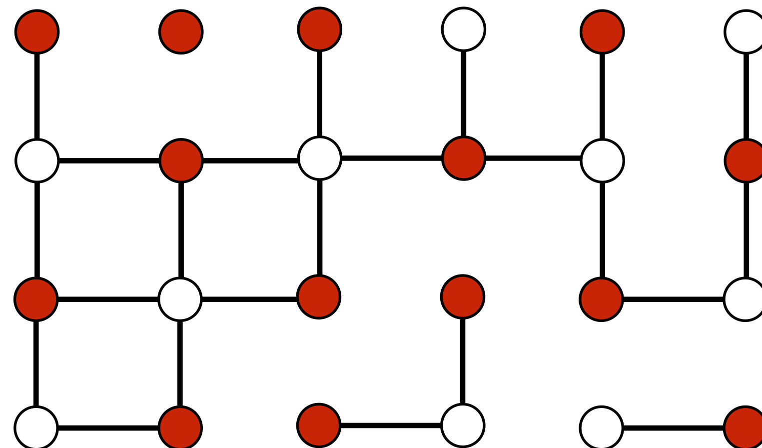


Vocabulaire

Définition

Un graphe non-orienté est *bipartie* si l'ensemble des sommets peut être séparé en deux disjoints S1 et S2, et que chaque arête du graphe relie un sommet de A1 à un sommet de A2.

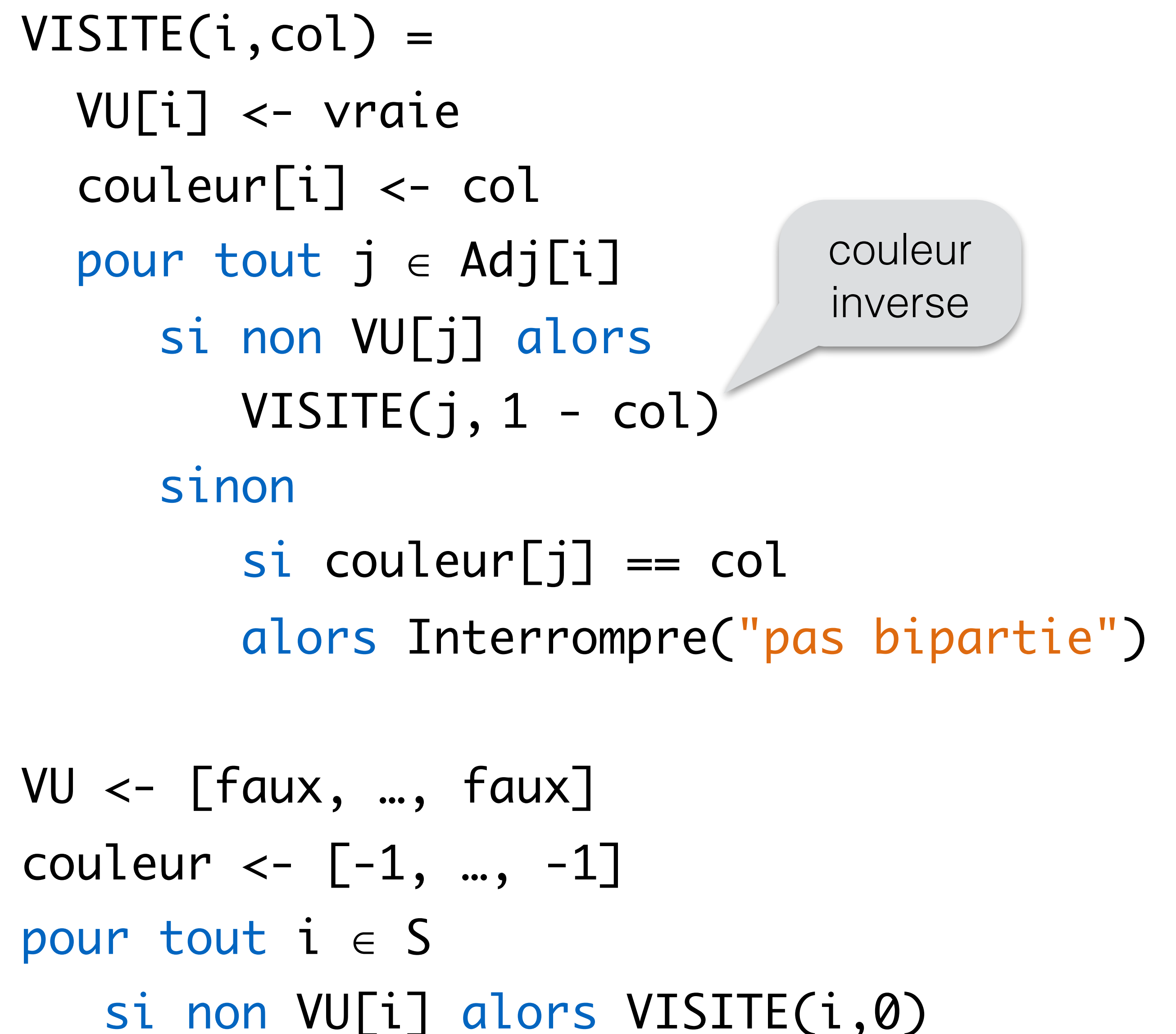
Exemple



Algorithme

Pendant un parcours en profondeur, on associe une couleur (0 ou 1) à chaque sommet, en s'assurant que deux adjacents n'ont pas la même couleur.

Si l'algorithme termine sans erreur, on a examiné chaque arête et attribué des couleurs cohérentes à chaque sommet. Sinon, le coloriage est impossible et l'algorithme s'interrompt avant la fin.



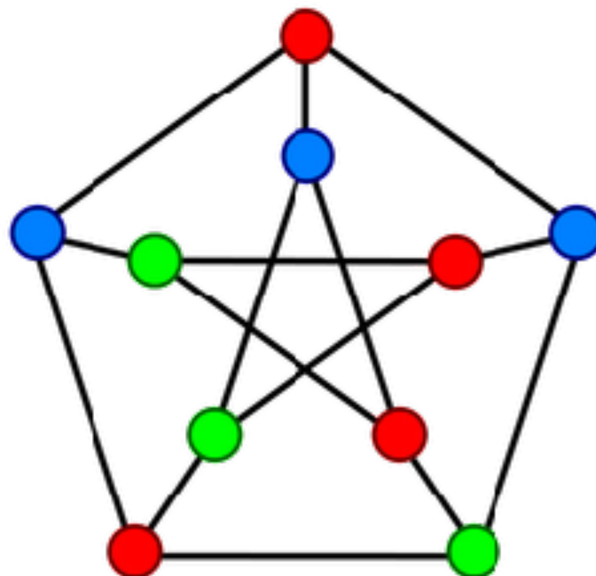
```
VISITE(i,col) =
  VU[i] <- vraie
  couleur[i] <- col
  pour tout j ∊ Adj[i]
     si non VU[j] alors
        VISITE(j, 1 - col)
     sinon
        si couleur[j] == col
        alors Interrompre("pas bipartie")

VU <- [faux, …, faux]
couleur <- [-1, …, -1]
pour tout i ∊ S
   si non VU[i] alors VISITE(i,0)
```

k-coloration

Les graphes biparties sont dits 2-coloriables. Nous venons de voir qu'il existait un algorithme efficace (linéaire). La généralisation est beaucoup plus difficile.

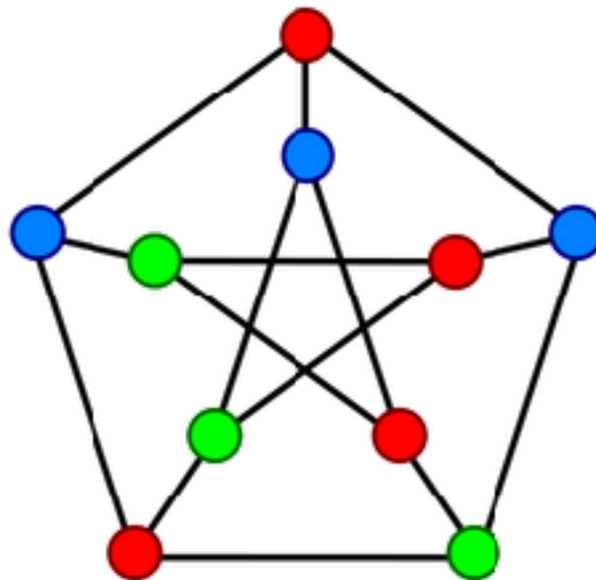
Définition

Un graphe non-orienté est *k-coloriable* si on peut colorier chaque sommet avec une couleur entre 1 et k, de telle façon que deux sommets adjacents aient des couleurs distinctes.



k-coloration

Pour k>2, le problème de déterminer si un graphe est k-coloriable est NP-complet



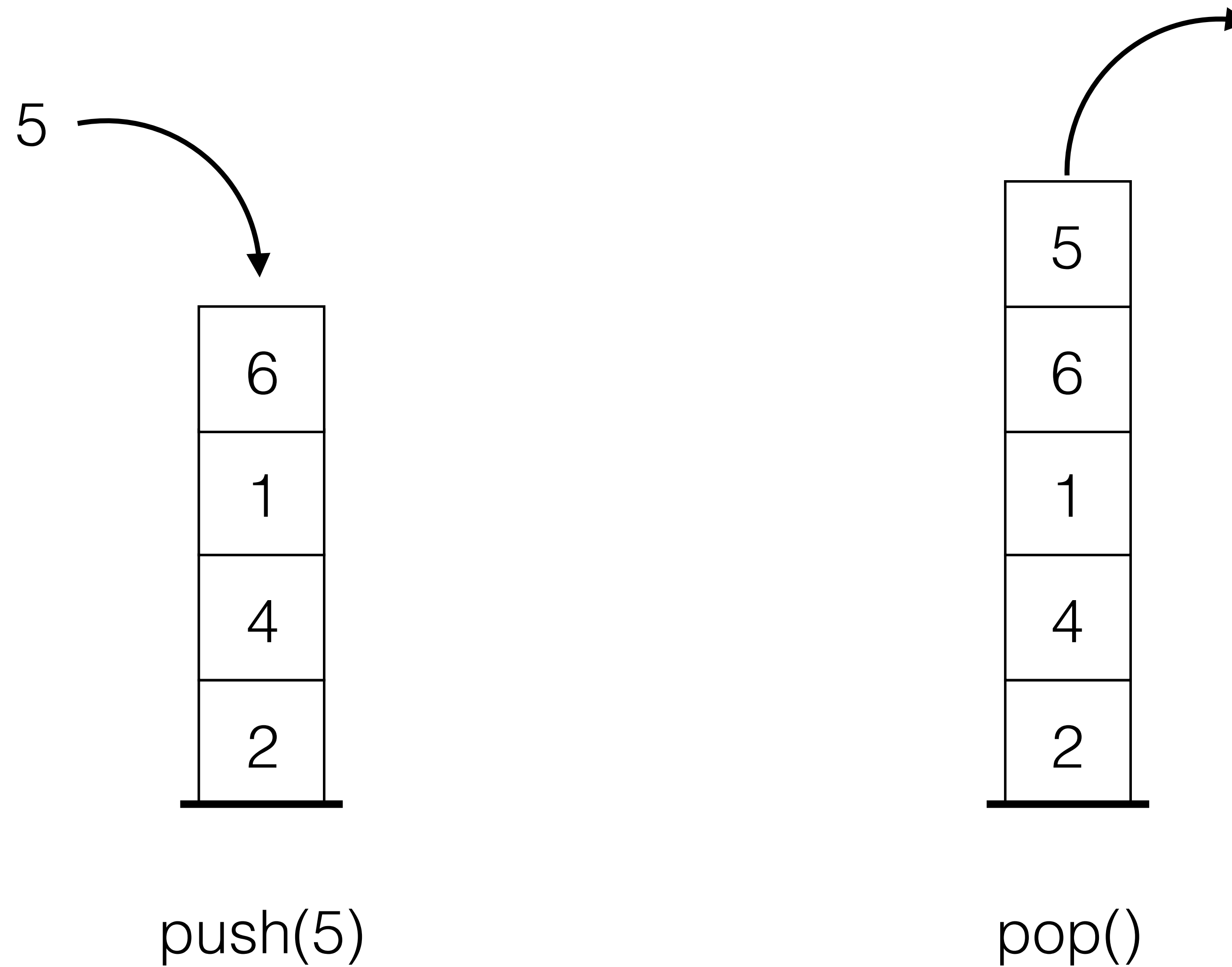
Les compilateurs doivent pourtant résoudre ce genre de problèmes pour l'allocation de registre

Parcours en profondeur (procédure itérative)

Dérécursification

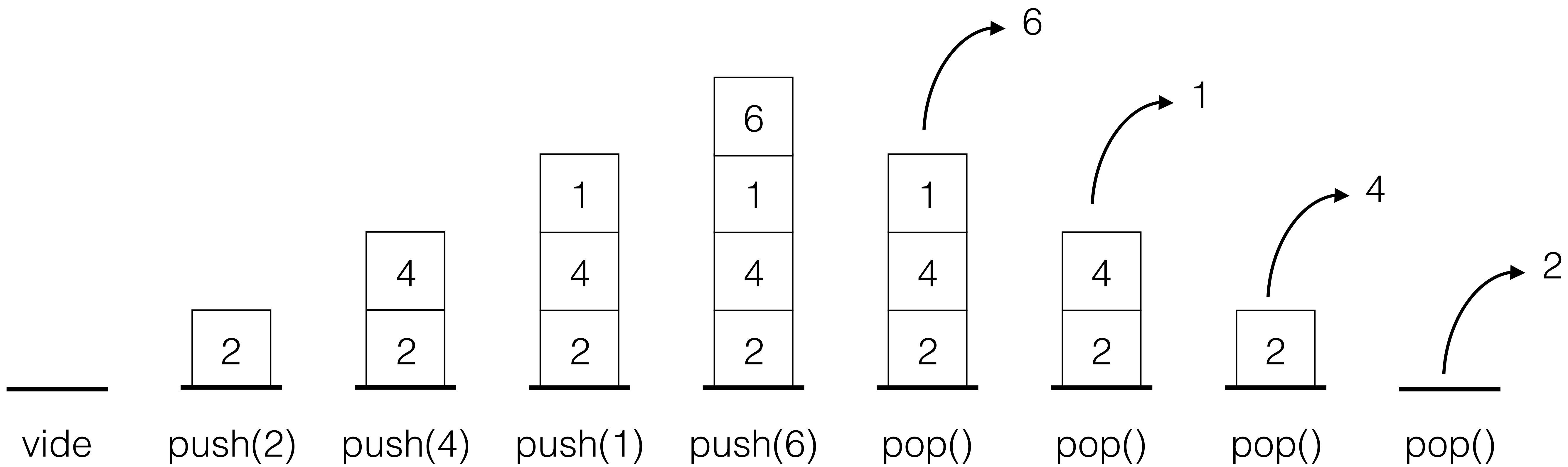
- Toute programme récursif peut être transformé en une programme équivalent non-récursif, en s'aidant d'une *pile*.
  - cela a parfois un interêt pour l'efficacité (rarement)
  - certains compilateurs savent le faire automatiquement (appels récursifs *terminaux)*
  - ici, l'interêt est surtout *pédagogique*

Pile



Last In First Out (LIFO)

Exemple



Visite non-récursive

```
VISITE(i) =
    P <- empty()
    VU[i] <- vraie
    push(P,i)
    tant que non vide?(P)
        k <- pop(P)
        DEBUT[k] <- date
        date <- date + 1
        pour tout j ∈ Adj[k]
            si non VU[j] alors
                VU[j] <- vraie
                push(P,j)
```

pile vide

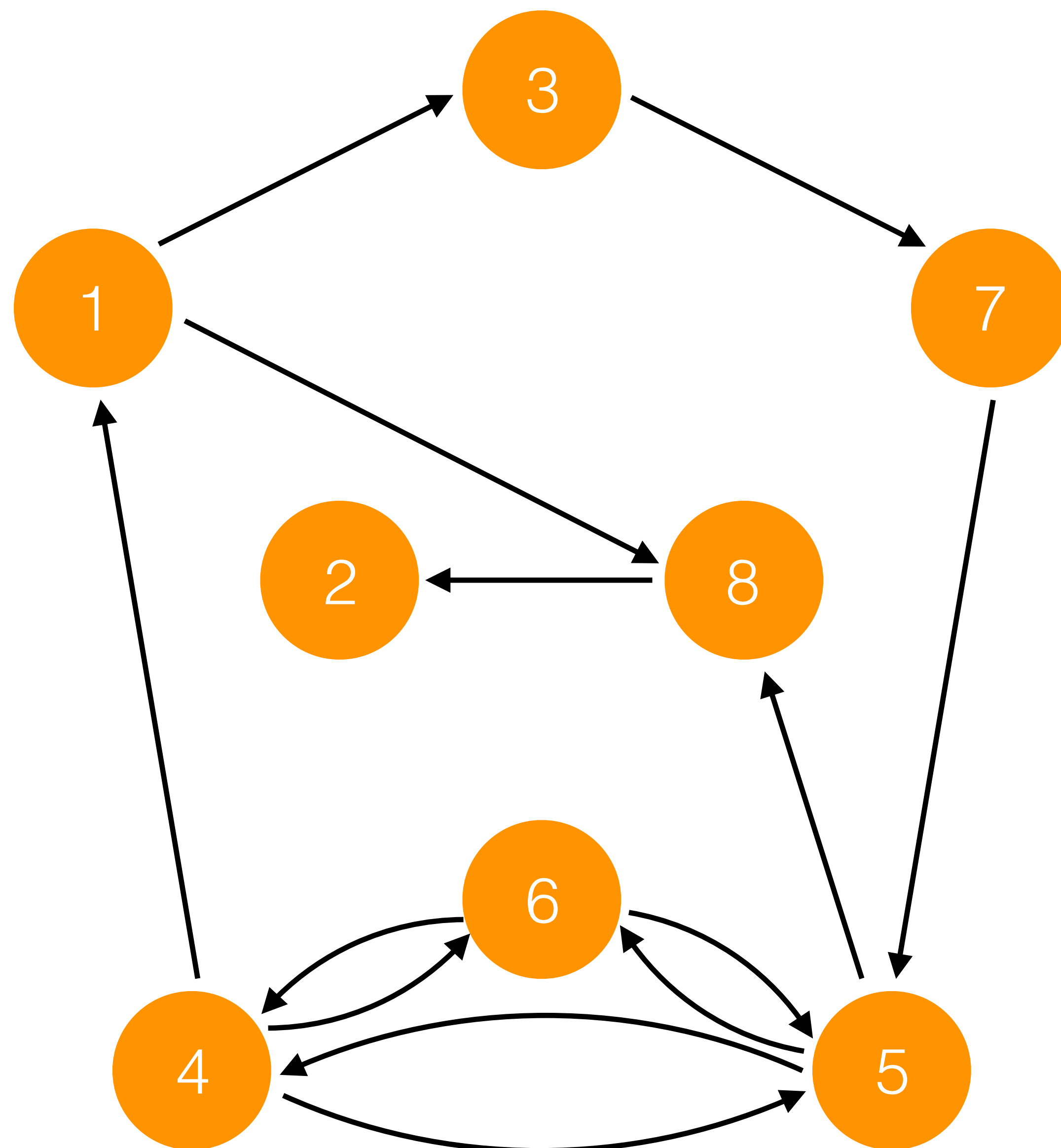
la pile P contient les éléments vus mais dont les adjacents n'ont pas encore tous été étudiés

P ne contient alors que i

on ne marque que la date d'arrivée

on empile tous les adjacents (non-vus) de k, puis on boucle

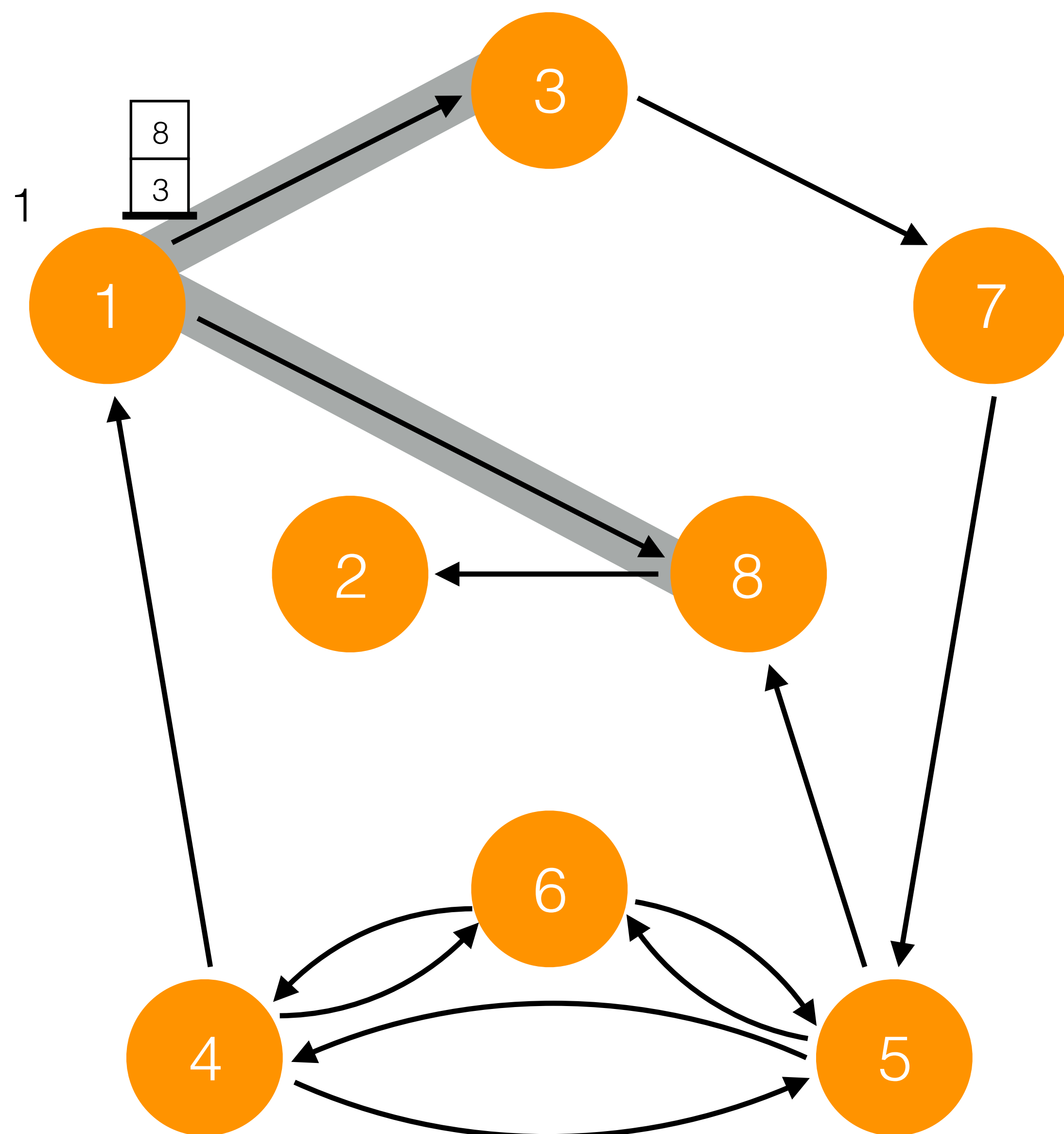
Exemple



pour chaque sommet i, on indique l'état de la pile à la fin de l'itération qui dépile i, ainsi que DEBUT[i]

```
VISITE(i) =
   P <- empty()
   VU[i] <- vraie
   push(P,i)
   tant que non vide?(P)
      k <- pop(P)
      DEBUT[k] <- date
      date <- date + 1
      pour tout j ∊ Adj[k]
         si non VU[j] alors
            VU[j] <- vraie
            push(P,j)
```

Exemple

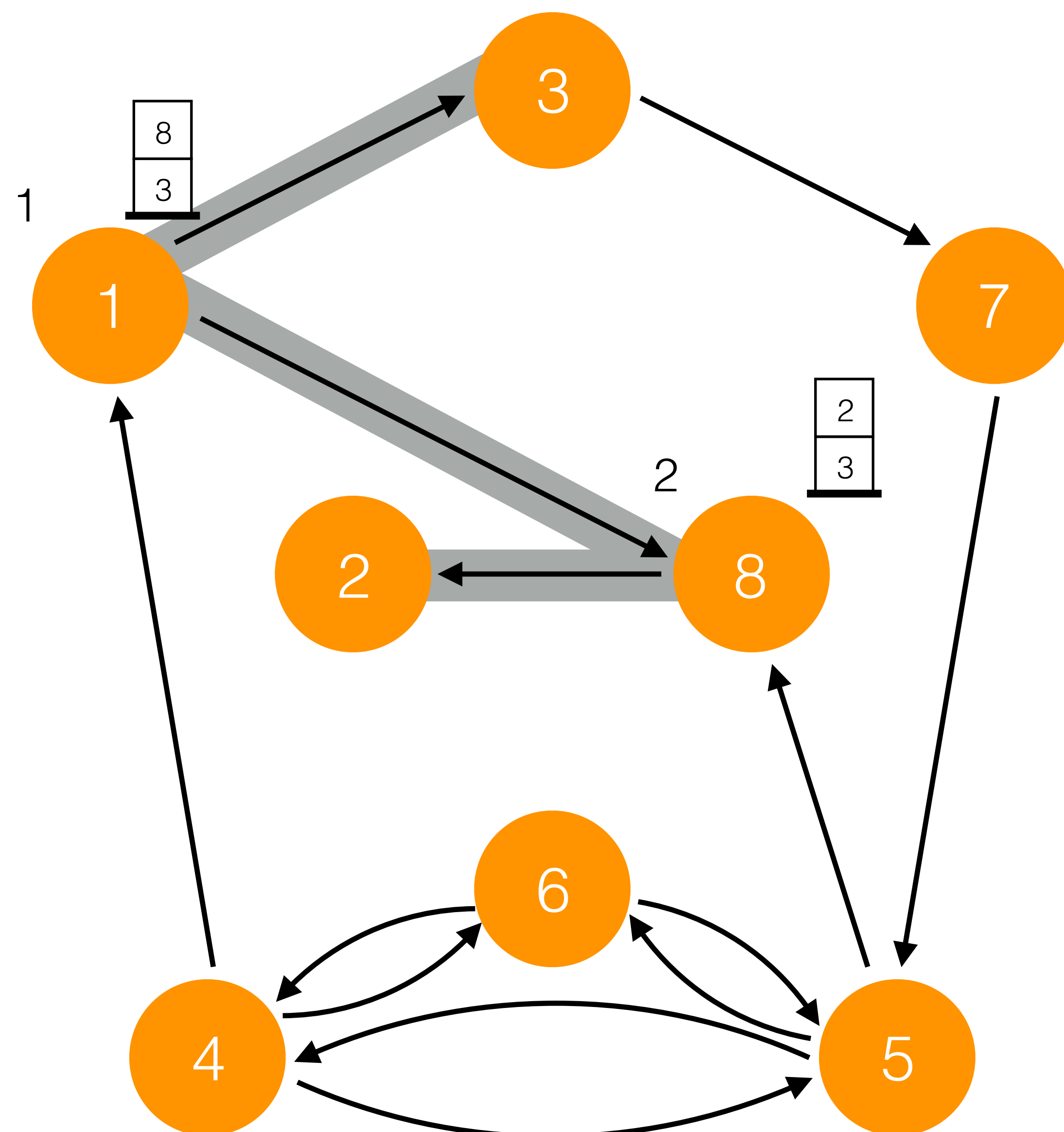


pour chaque sommet i, on indique l'état de la pile à la fin de l'itération qui dépile i, ainsi que DEBUT[i]

```
VISITE(i) =
    P <- empty()
    VU[i] <- vraie
    push(P,i)
    tant que non vide?(P)
        k <- pop(P)
        DEBUT[k] <- date
        date <- date + 1
        pour tout j ∈ Adj[k]
            si non VU[j] alors
                VU[j] <- vraie
                push(P,j)
```

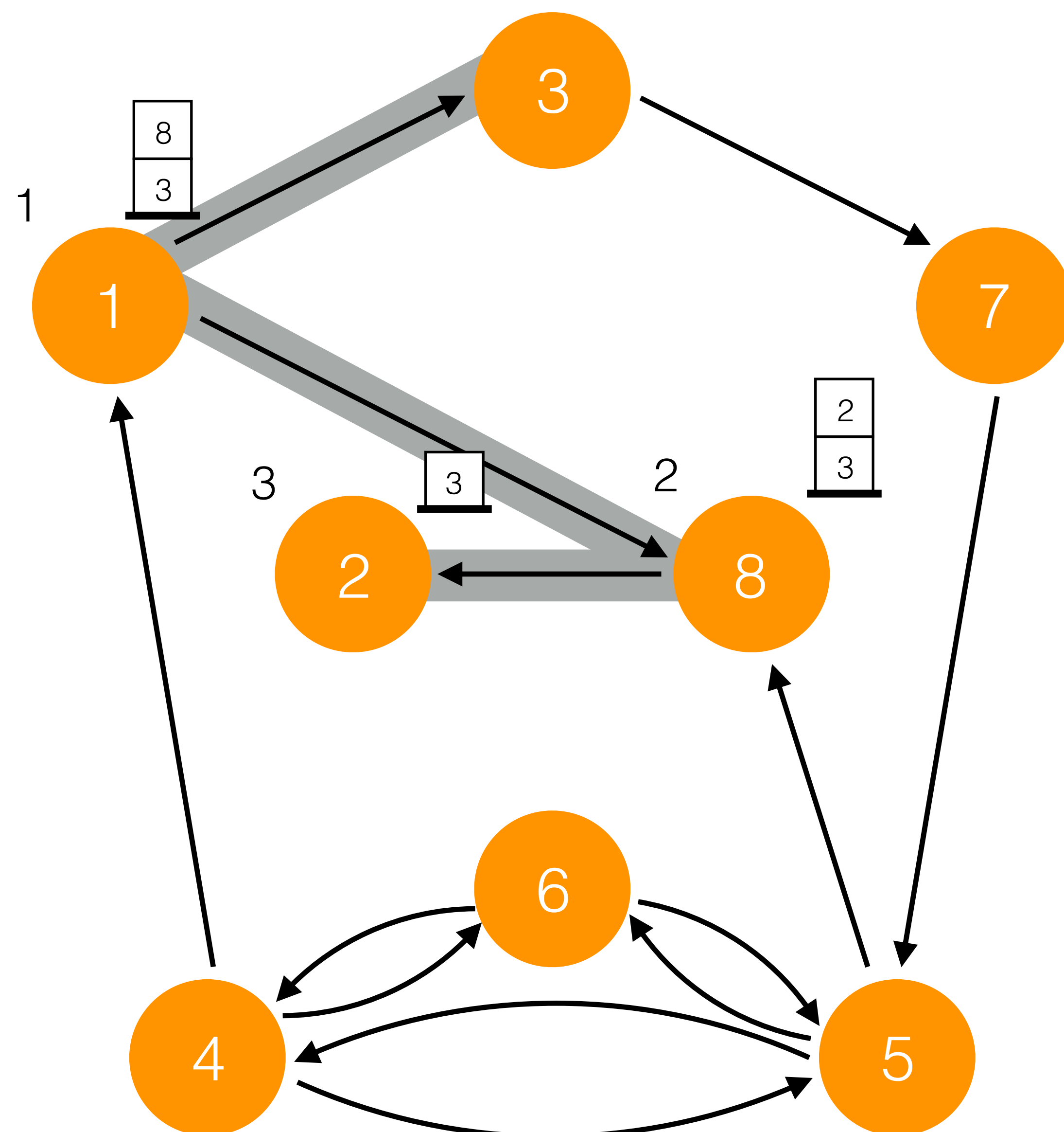
Exemple

pour chaque sommet i, on indique l'état de la pile à la fin de l'itération qui dépile i, ainsi que DEBUT[i]




```
VISITE(i) =
   P <- empty()
   VU[i] <- vraie
   push(P,i)
   tant que non vide?(P)
      k <- pop(P)
      DEBUT[k] <- date
      date <- date + 1
      pour tout j ∈ Adj[k]
         si non VU[j] alors
            VU[j] <- vraie
            push(P,j)
```

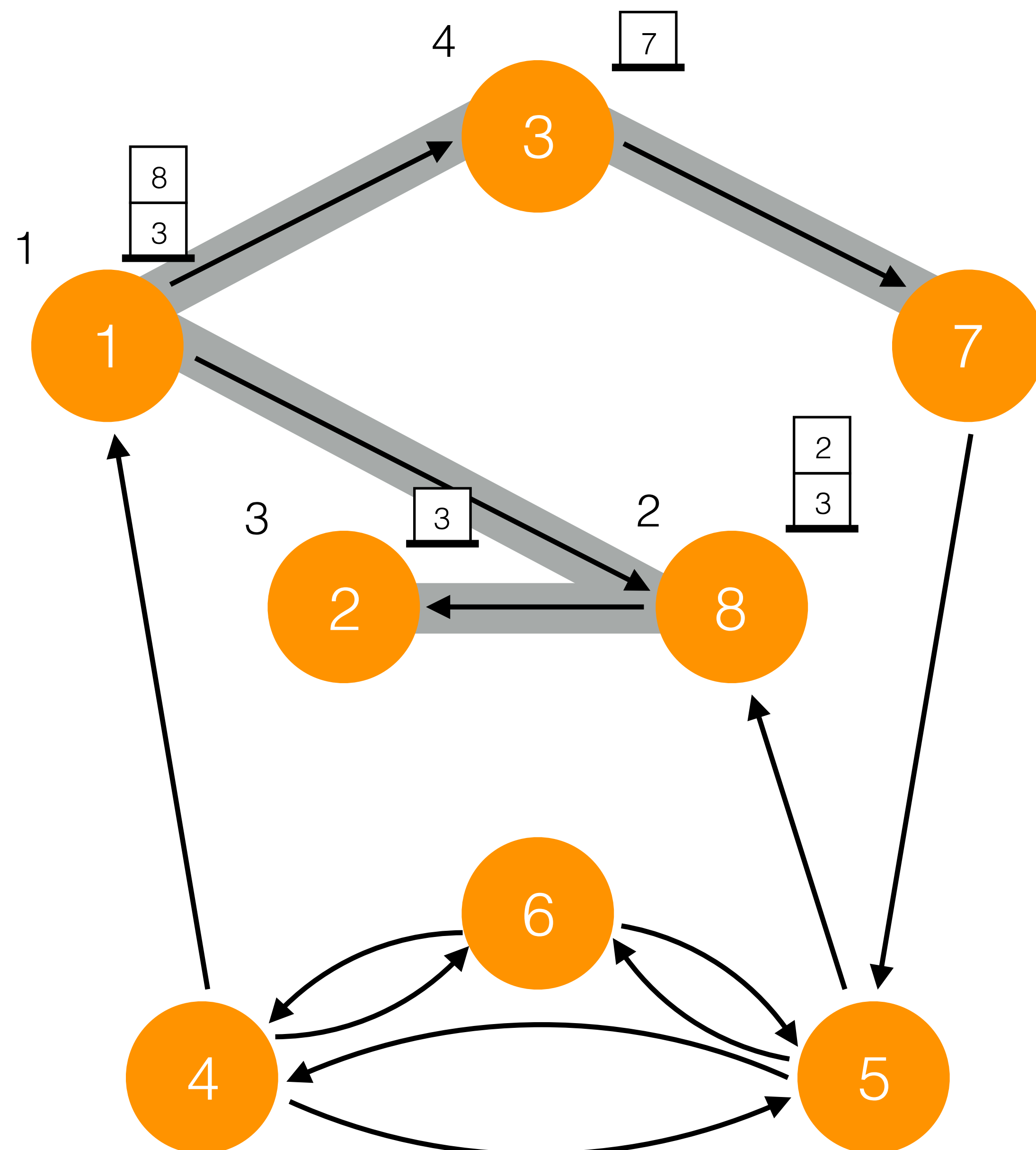
Exemple



pour chaque sommet i, on indique l'état de la pile à la fin de l'itération qui dépile i, ainsi que DEBUT[i]

```
VISITE(i) =
   P <- empty()
   VU[i] <- vraie
   push(P,i)
   tant que non vide?(P)
      k <- pop(P)
      DEBUT[k] <- date
      date <- date + 1
      pour tout j ∊ Adj[k]
         si non VU[j] alors
            VU[j] <- vraie
            push(P,j)
```

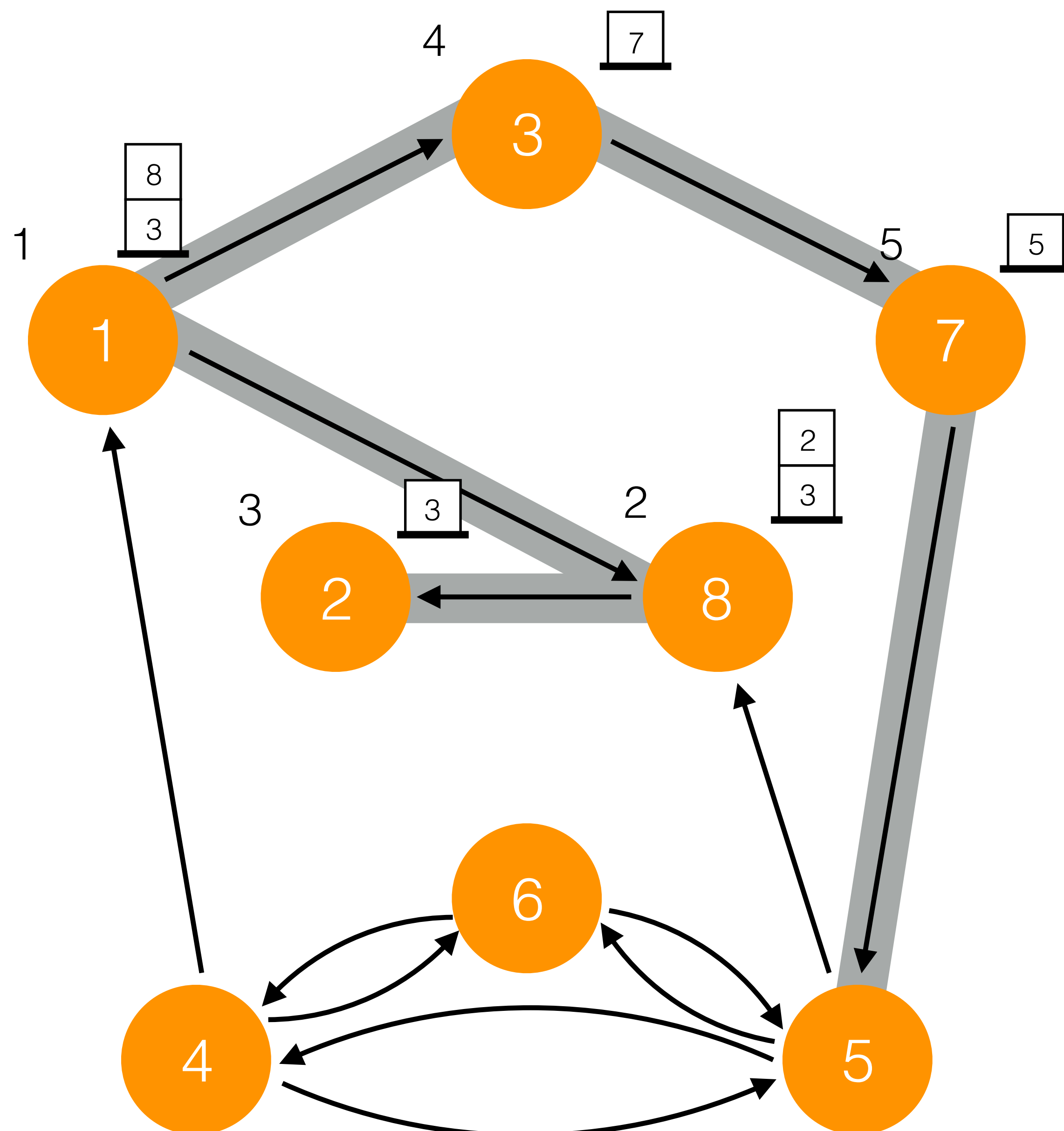
Exemple



pour chaque sommet i, on indique l'état de la pile à la fin de l'itération qui dépile i, ainsi que DEBUT[i]

```
VISITE(i) =
   P <- empty()
   VU[i] <- vraie
   push(P,i)
   tant que non vide?(P)
      k <- pop(P)
      DEBUT[k] <- date
      date <- date + 1
      pour tout j ∈ Adj[k]
         si non VU[j] alors
            VU[j] <- vraie
            push(P,j)
```

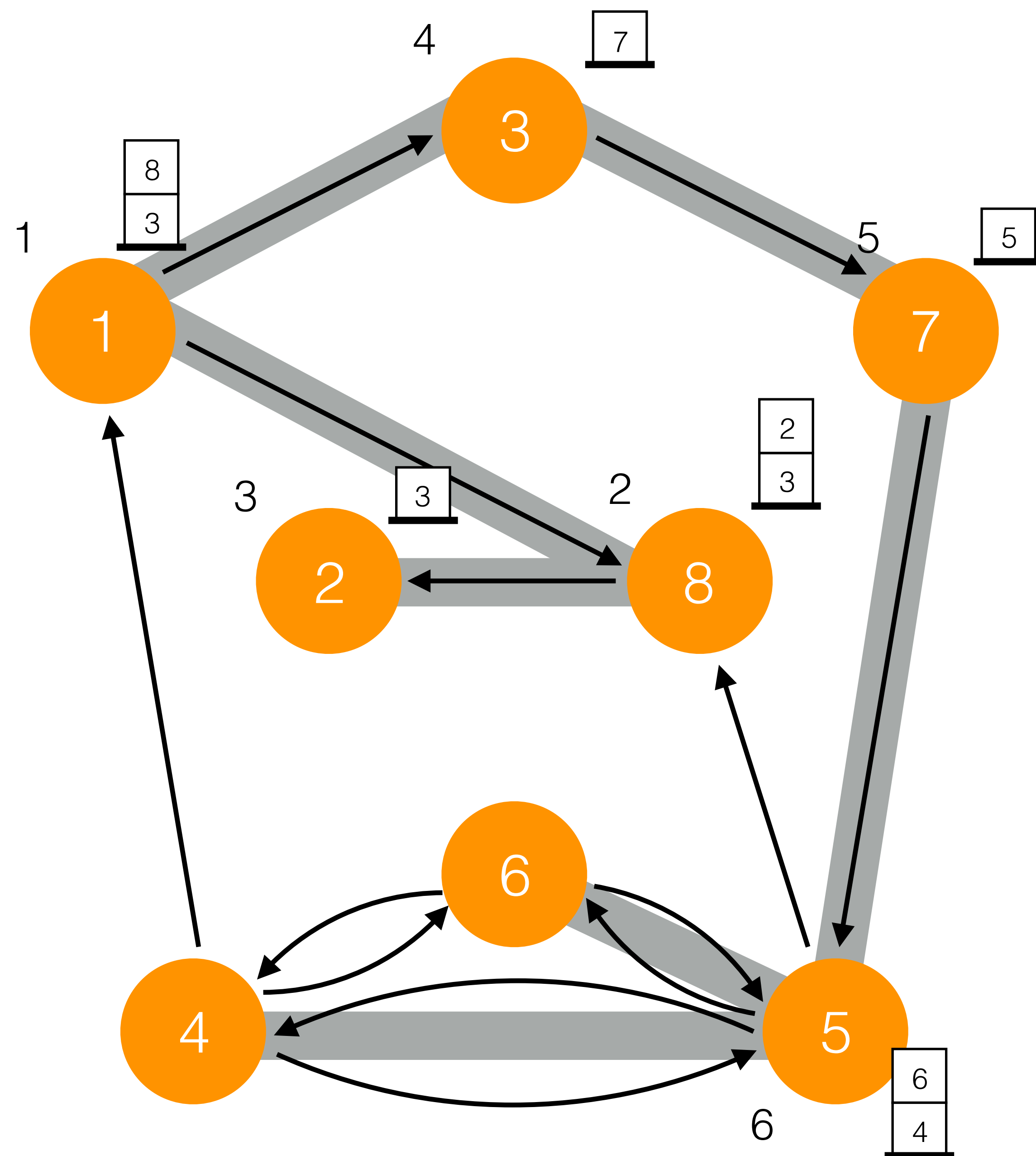
Exemple



pour chaque sommet i, on indique l'état de la pile à la fin de l'itération qui dépile i, ainsi que DEBUT[i]

```
VISITE(i) =
    P <- empty()
    VU[i] <- vraie
    push(P,i)
    tant que non vide?(P)
        k <- pop(P)
        DEBUT[k] <- date
        date <- date + 1
        pour tout j ∊ Adj[k]
            si non VU[j] alors
                VU[j] <- vraie
                push(P,j)
```

Exemple

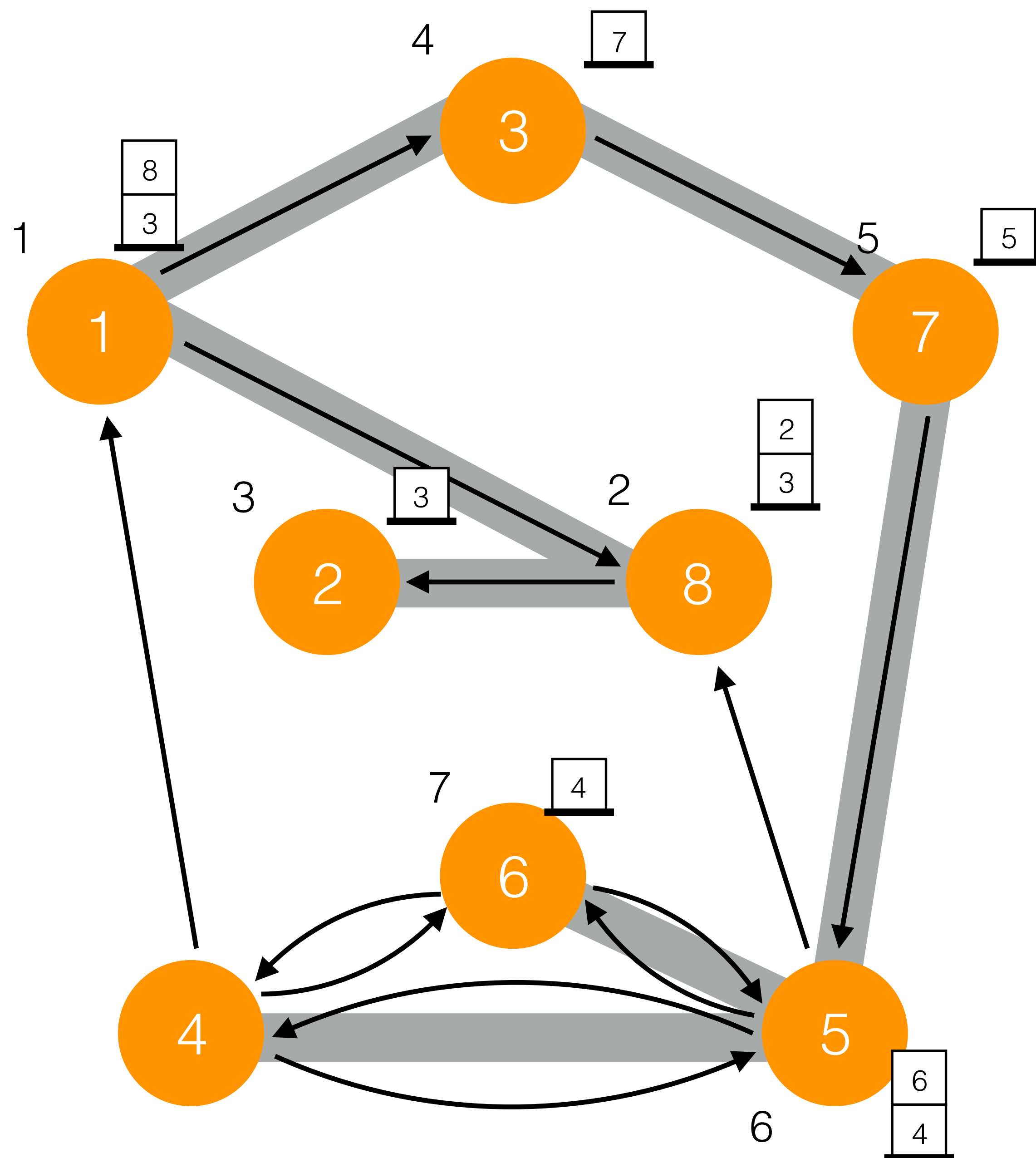


pour chaque sommet i, on indique l'état de la pile à la fin de l'itération qui dépile i, ainsi que DEBUT[i]

```
VISITE(i) =
   P <- empty()
   VU[i] <- vraie
   push(P,i)
   tant que non vide?(P)
      k <- pop(P)
      DEBUT[k] <- date
      date <- date + 1
      pour tout j ∊ Adj[k]
         si non VU[j] alors
            VU[j] <- vraie
            push(P,j)
```

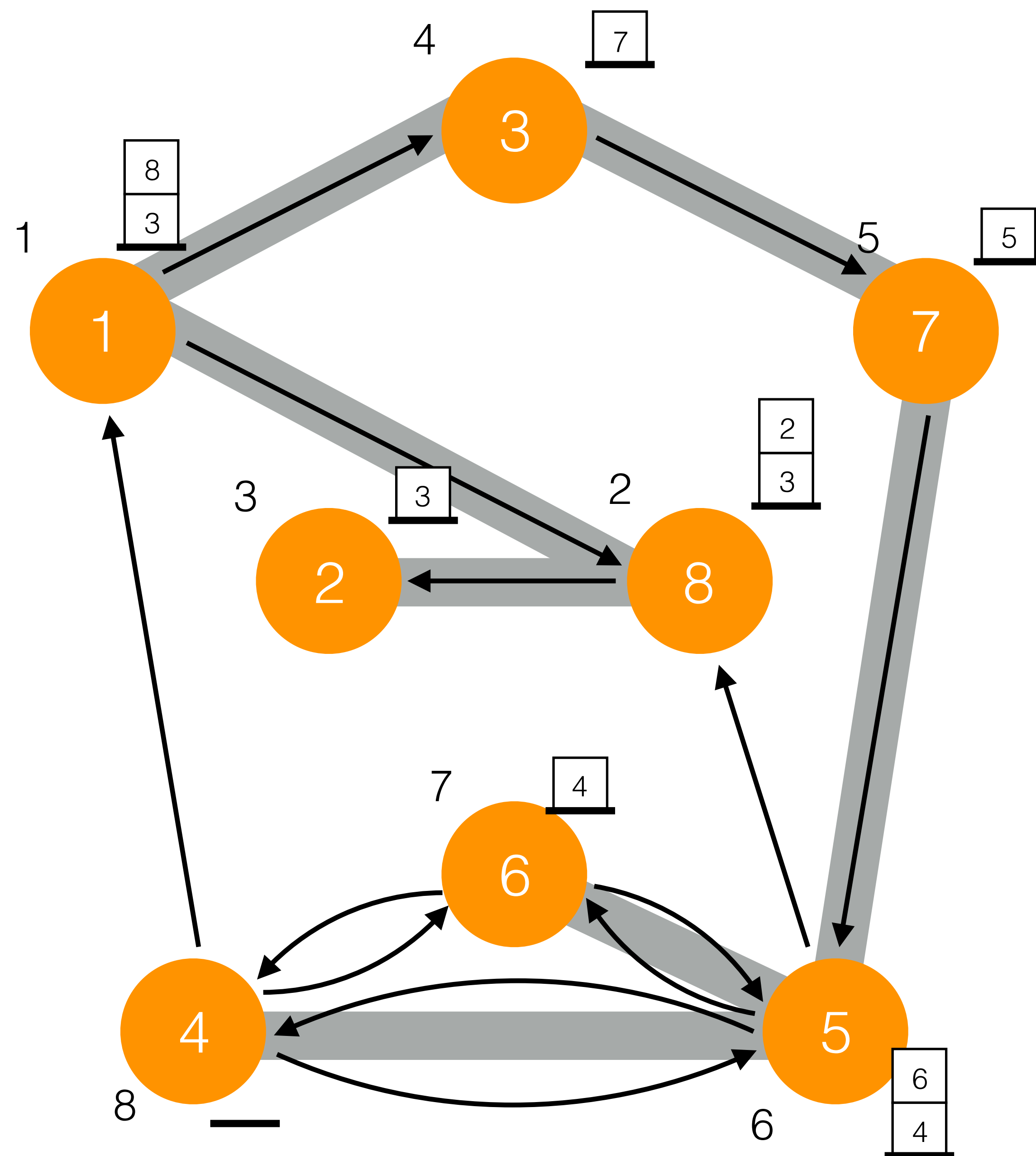
Exemple

pour chaque sommet i, on indique l'état de la pile à la fin de l'itération qui dépile i, ainsi que DEBUT[i]




```
VISITE(i) =
   P <- empty()
   VU[i] <- vraie
   push(P,i)
   tant que non vide?(P)
      k <- pop(P)
      DEBUT[k] <- date
      date <- date + 1
      pour tout j ∈ Adj[k]
         si non VU[j] alors
            VU[j] <- vraie
            push(P,j)
```

Exemple



pour chaque sommet i, on indique l'état de la pile à la fin de l'itération qui dépile i, ainsi que DEBUT[i]

```
VISITE(i) =
   P <- empty()
   VU[i] <- vraie
   push(P,i)
   tant que non vide?(P)
      k <- pop(P)
      DEBUT[k] <- date
      date <- date + 1
      pour tout j ∈ Adj[k]
         si non VU[j] alors
            VU[j] <- vraie
            push(P,j)
```

cf. fichier `maze_dfs_iter.pdf`

Boucle principale

```
VU <- [faux, …, faux]
date <- 1
pour tout i ∊ S
   si non VU[i] alors VISITE(i)
```

on initialise à 1 pour que le premier sommet soit visité à l'instant 1

cette procédure n'est plus récursive cette fois

Propriétés

- La complexité asymptotique ne change pas, mais la pile de sommet est plus petite que la pile d'appel de la procédure récursive.

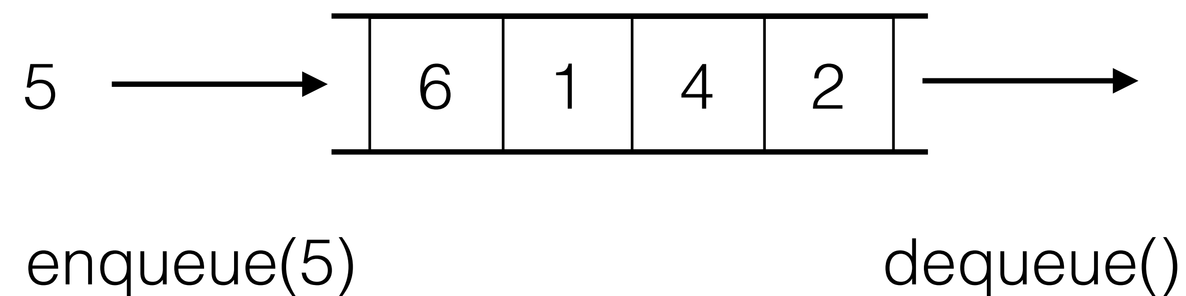
La complexité pire cas
est $|S|+|A|$

- On peut construire une forêt de parcours

- Cela reste un parcours en profondeur, mais dans un ordre un peu différent

Parcours en largeur

File d’attente



First In First Out (FIFO)

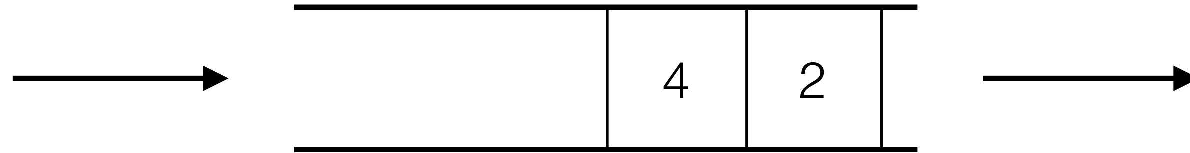
Exemple

empty()

Exemple

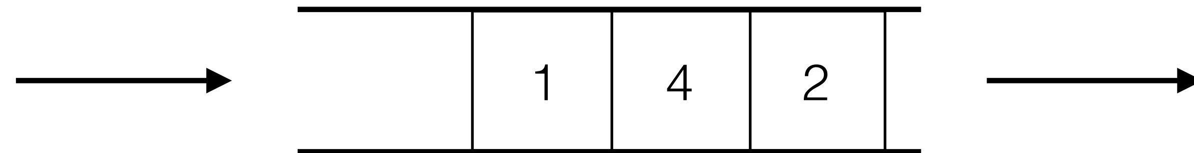
empty()
enqueue(2)

Exemple



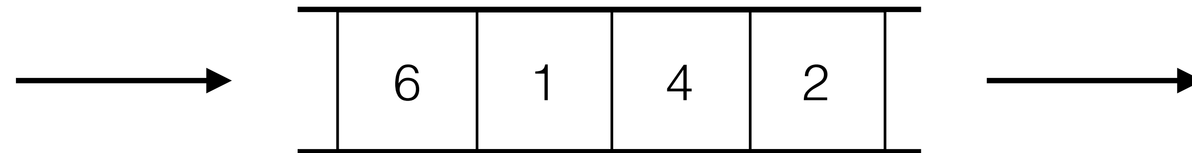
empty()
enqueue(2)
enqueue(4)

Exemple



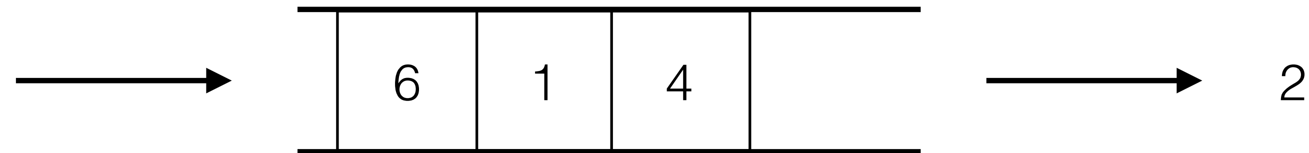
empty()
enqueue(2)
enqueue(4)
enqueue(1)

Exemple



empty()
enqueue(2)
enqueue(4)
enqueue(1)
enqueue(6)

Exemple

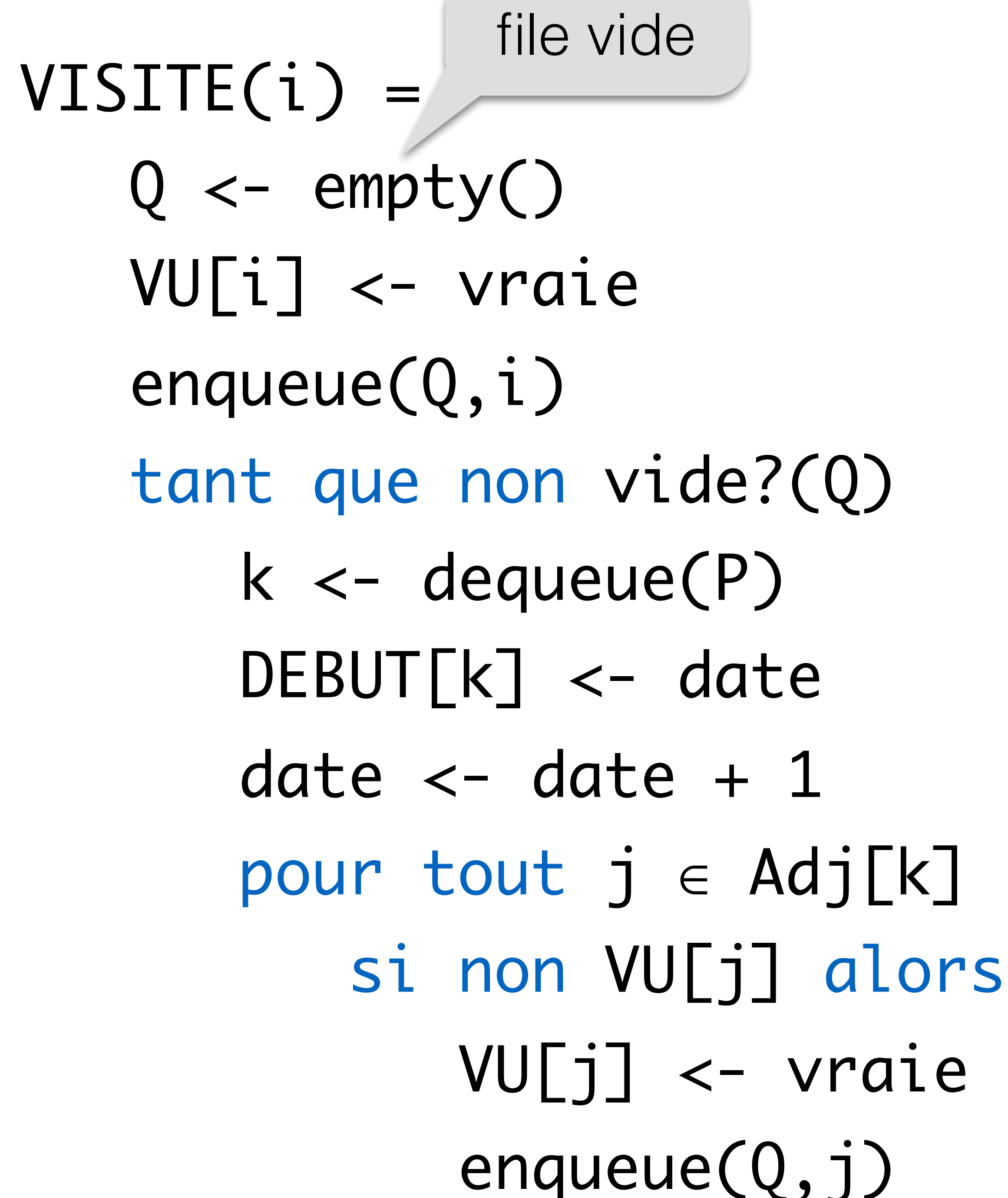


empty()
enqueue(2)
enqueue(4)
enqueue(1)
enqueue(6)
x <- dequeue()

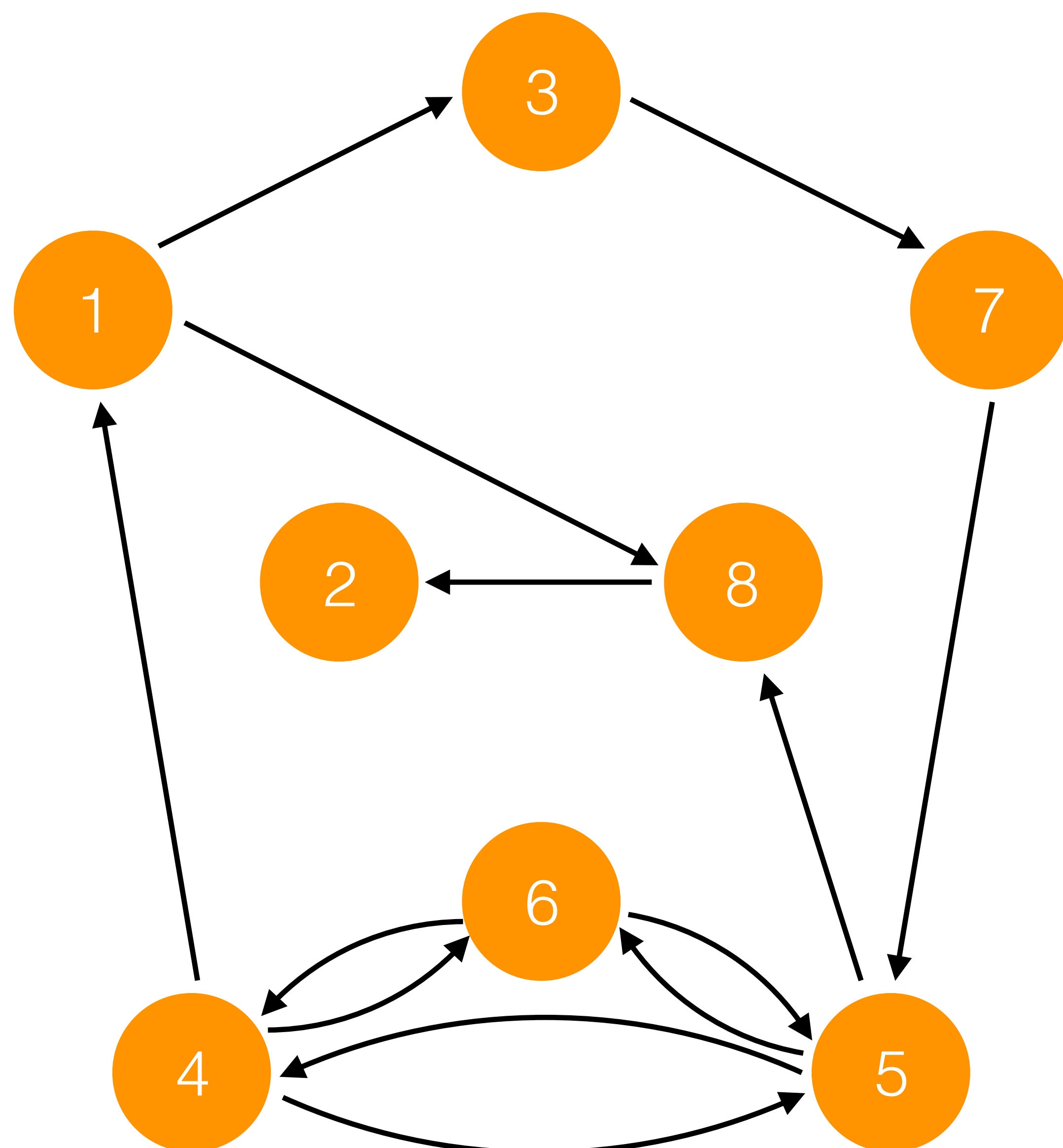
Exemple

```
empty()
enqueue(2)
enqueue(4)
enqueue(1)
enqueue(6)
x <- dequeue()
y <- dequeue()
```

Parcours en largeur



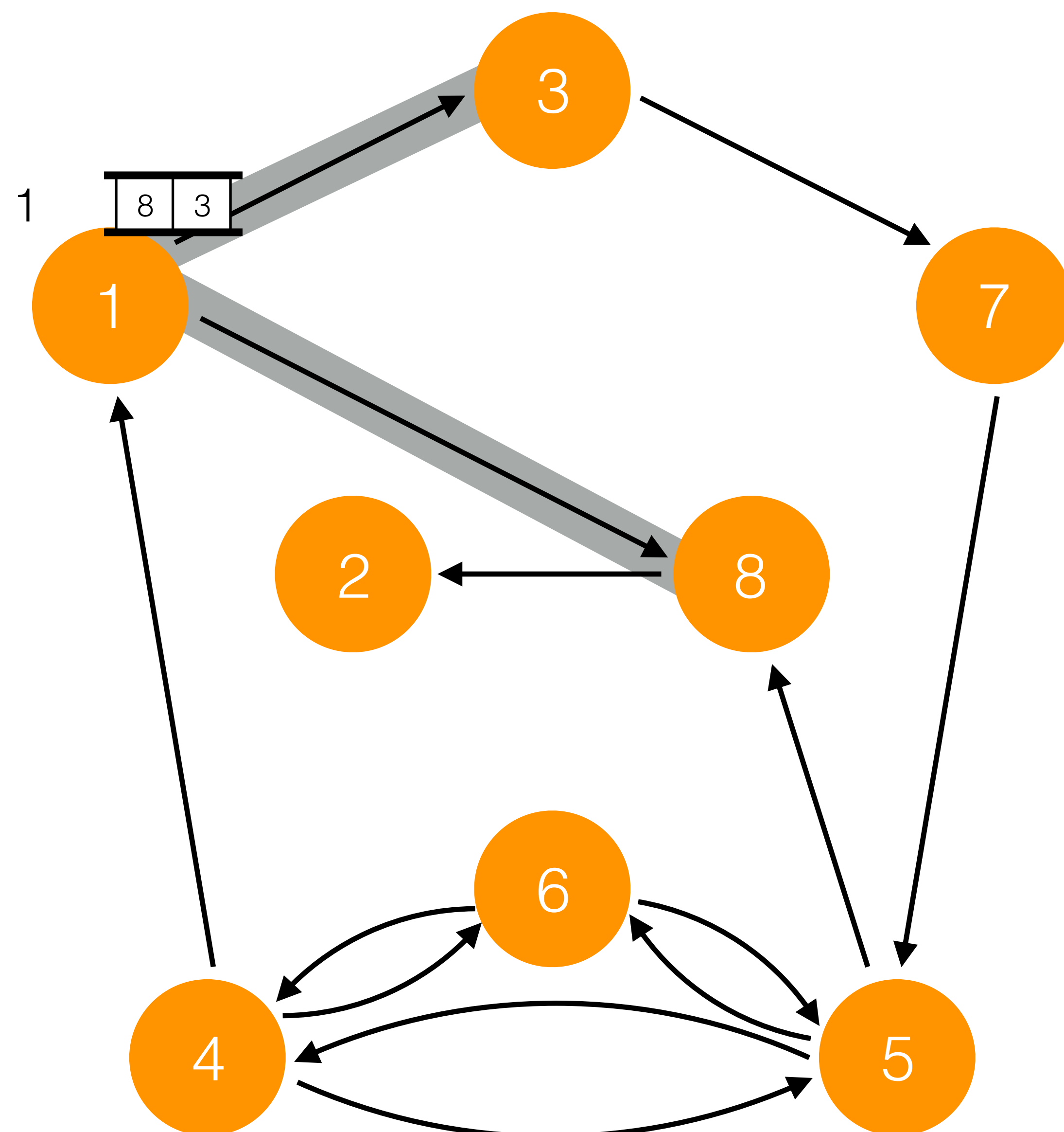
Exemple



pour chaque sommet i, on indique l'état de la file à la fin de l'itération qui dépile i, ainsi que DEBUT[i]

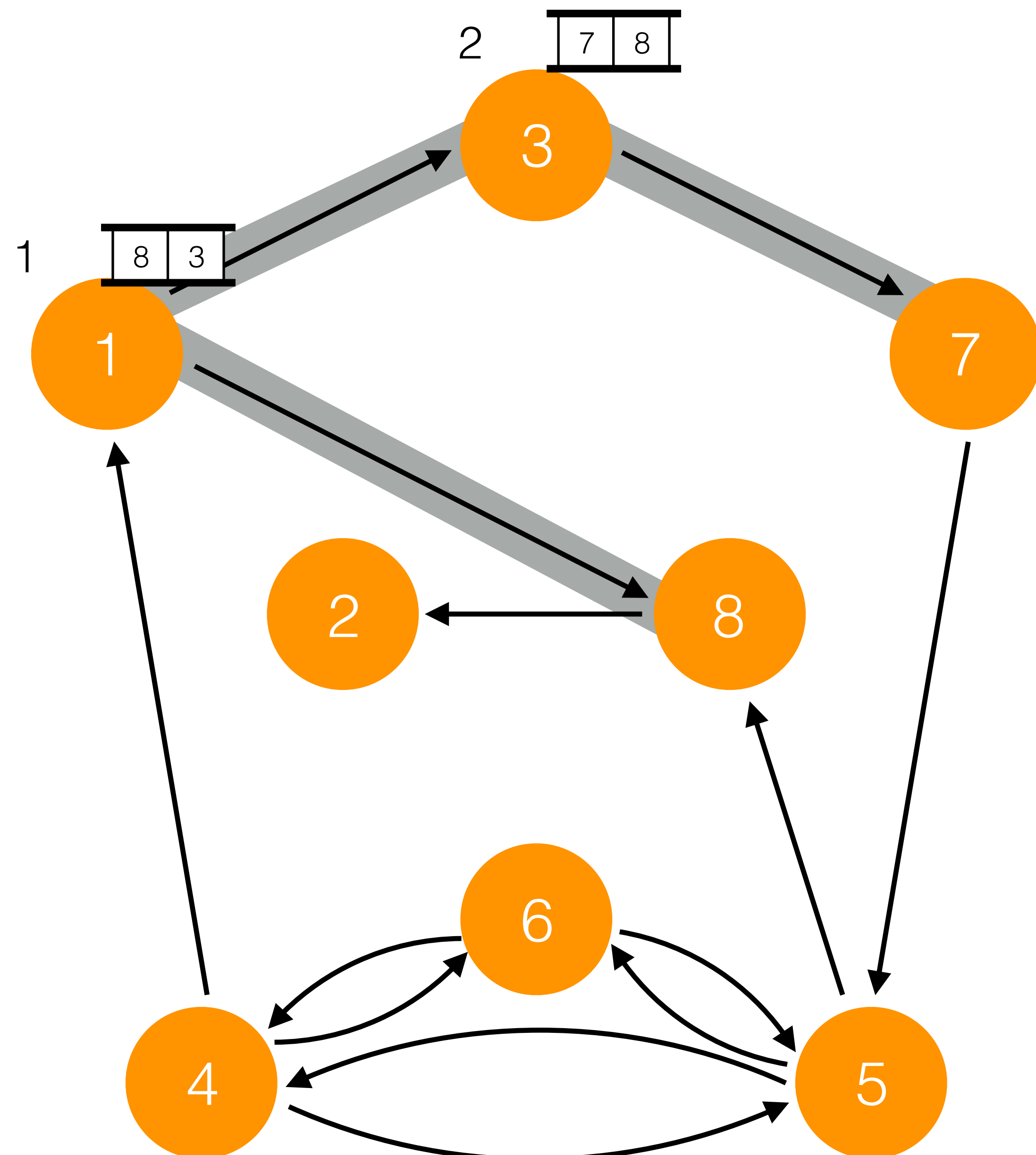
```
VISITE(i) =
    Q <- empty()
    VU[i] <- vraie
    enqueue(Q,i)
    tant que non vide?(Q)
        k <- dequeue(P)
        DEBUT[k] <- date
        date <- date + 1
        pour tout j ∊ Adj[k]
            si non VU[j] alors
                VU[j] <- vraie
                enqueue(Q,j)
```

Exemple



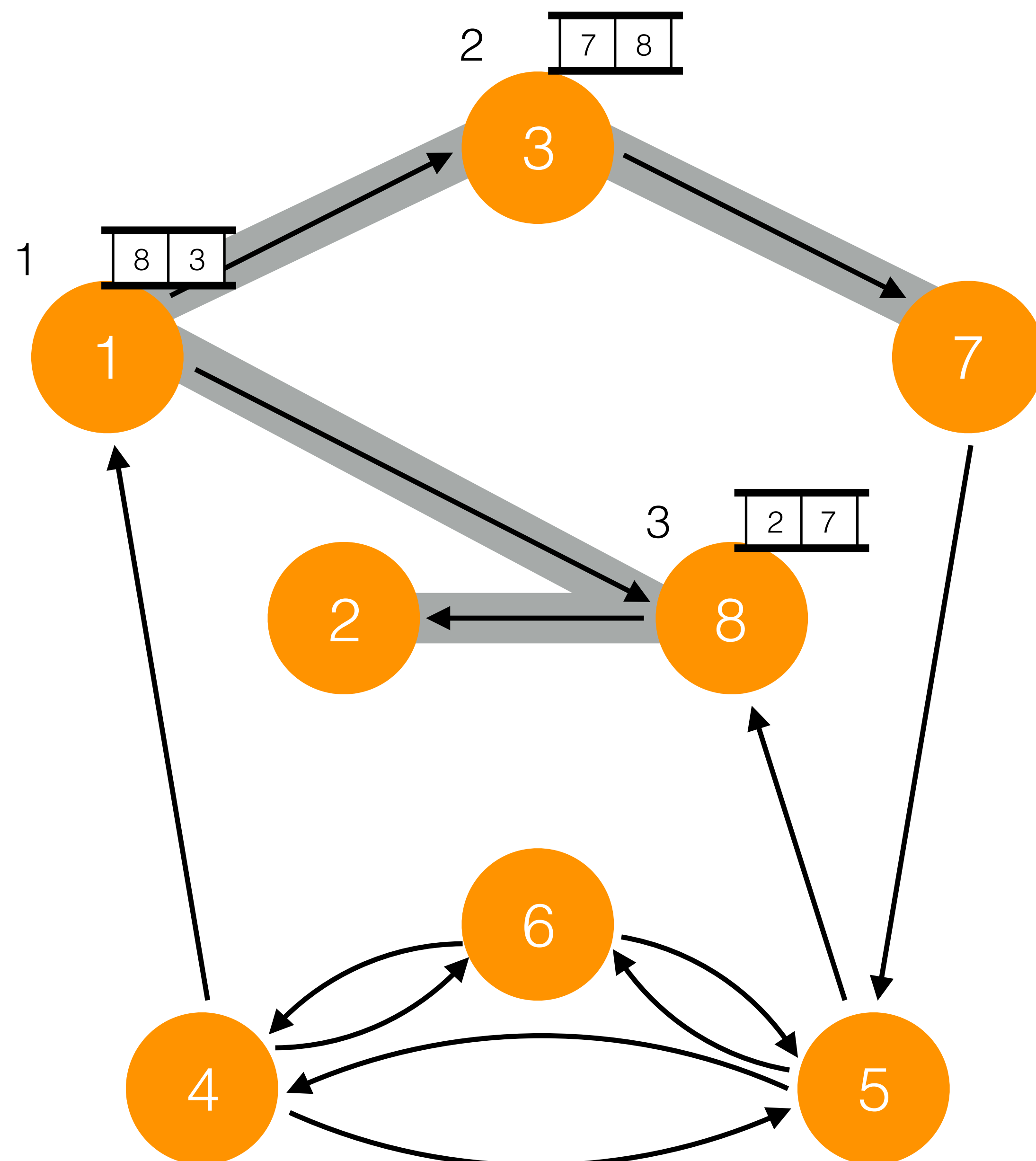

```
VISITE(i) =
    Q <- empty()
    VU[i] <- vraie
    enqueue(Q,i)
    tant que non vide?(Q)
        k <- dequeue(P)
        DEBUT[k] <- date
        date <- date + 1
        pour tout j ∊ Adj[k]
            si non VU[j] alors
                VU[j] <- vraie
                enqueue(Q,j)
```

Exemple



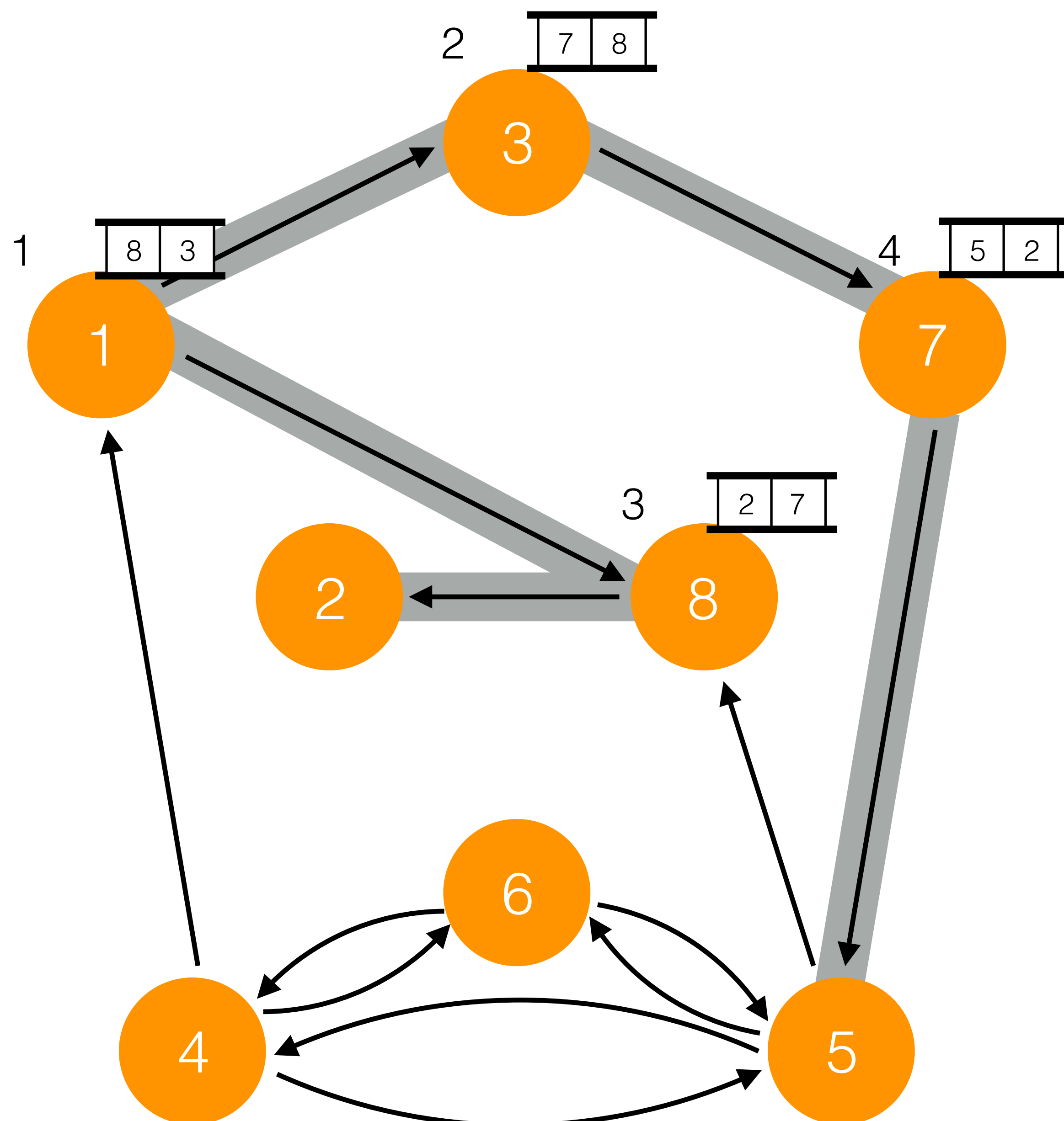

```
VISITE(i) =
    Q <- empty()
    VU[i] <- vraie
    enqueue(Q,i)
    tant que non vide?(Q)
        k <- dequeue(P)
        DEBUT[k] <- date
        date <- date + 1
        pour tout j ∈ Adj[k]
            si non VU[j] alors
                VU[j] <- vraie
                enqueue(Q,j)
```

Exemple



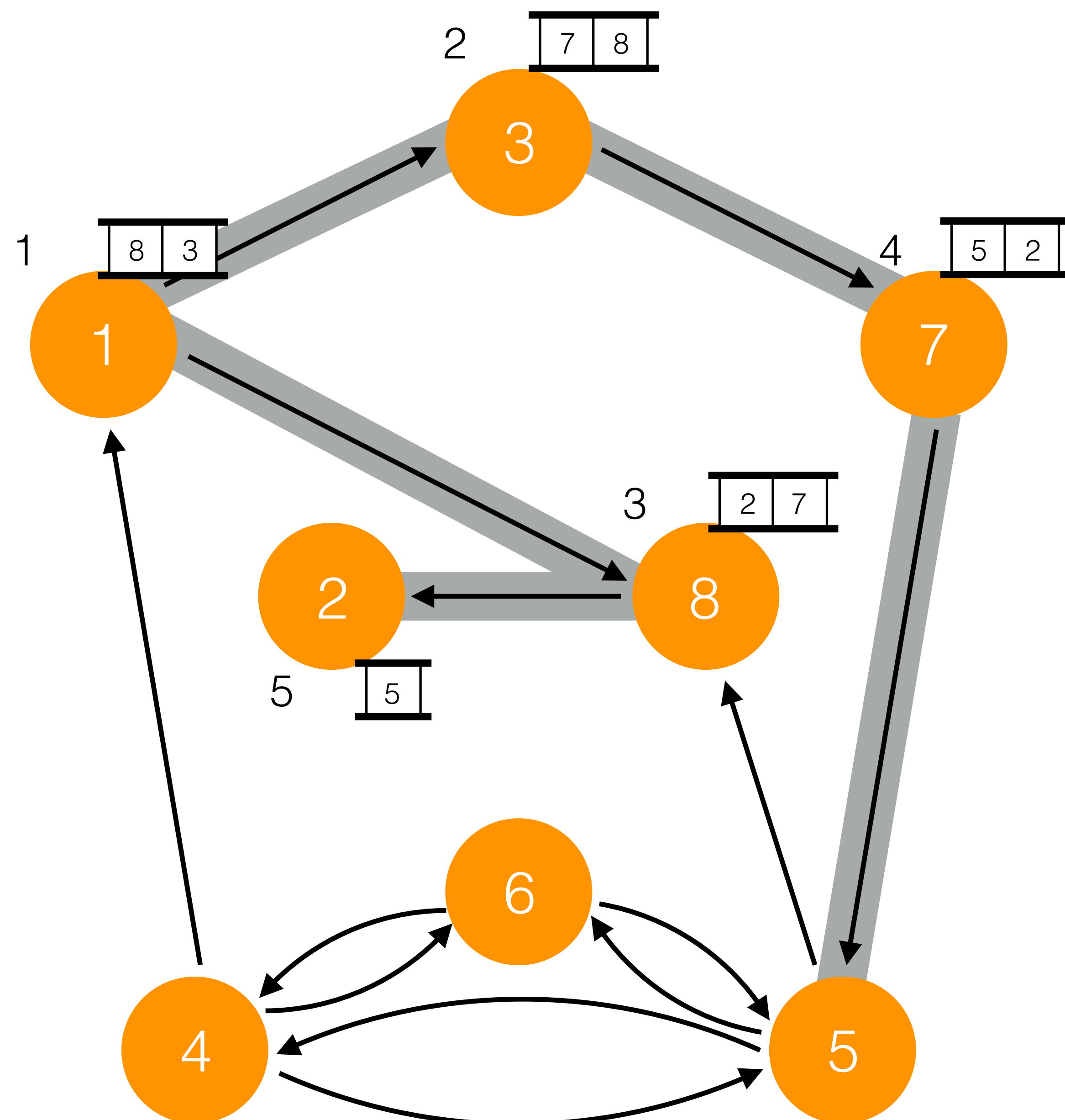

```
VISITE(i) =
    Q <- empty()
    VU[i] <- vraie
    enqueue(Q,i)
    tant que non vide?(Q)
        k <- dequeue(P)
        DEBUT[k] <- date
        date <- date + 1
        pour tout j ∈ Adj[k]
            si non VU[j] alors
                VU[j] <- vraie
                enqueue(Q,j)
```

Exemple



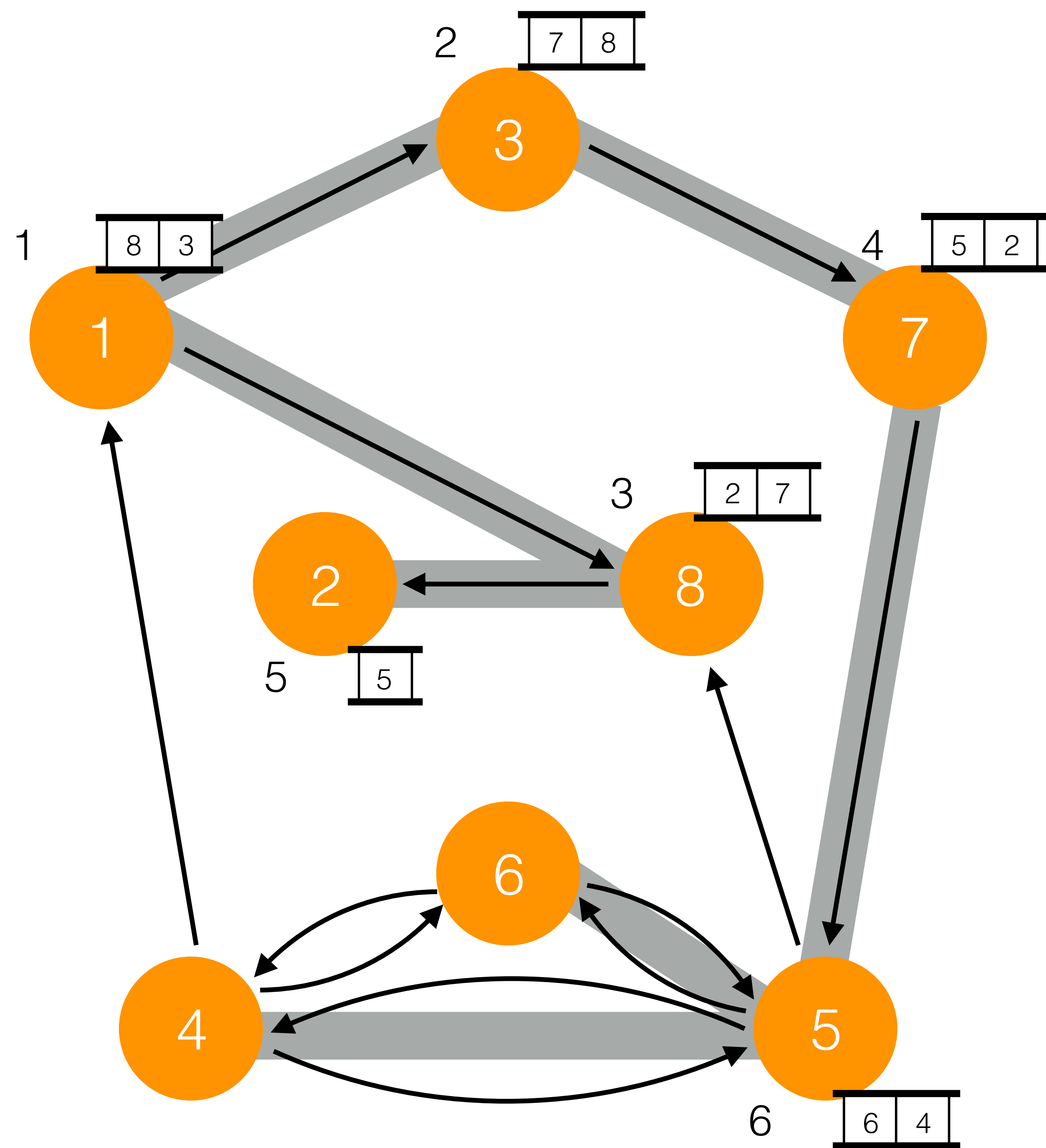

```
VISITE(i) =
    Q <- empty()
    VU[i] <- vraie
    enqueue(Q,i)
    tant que non vide?(Q)
        k <- dequeue(P)
        DEBUT[k] <- date
        date <- date + 1
        pour tout j ∈ Adj[k]
            si non VU[j] alors
                VU[j] <- vraie
                enqueue(Q,j)
```

Exemple



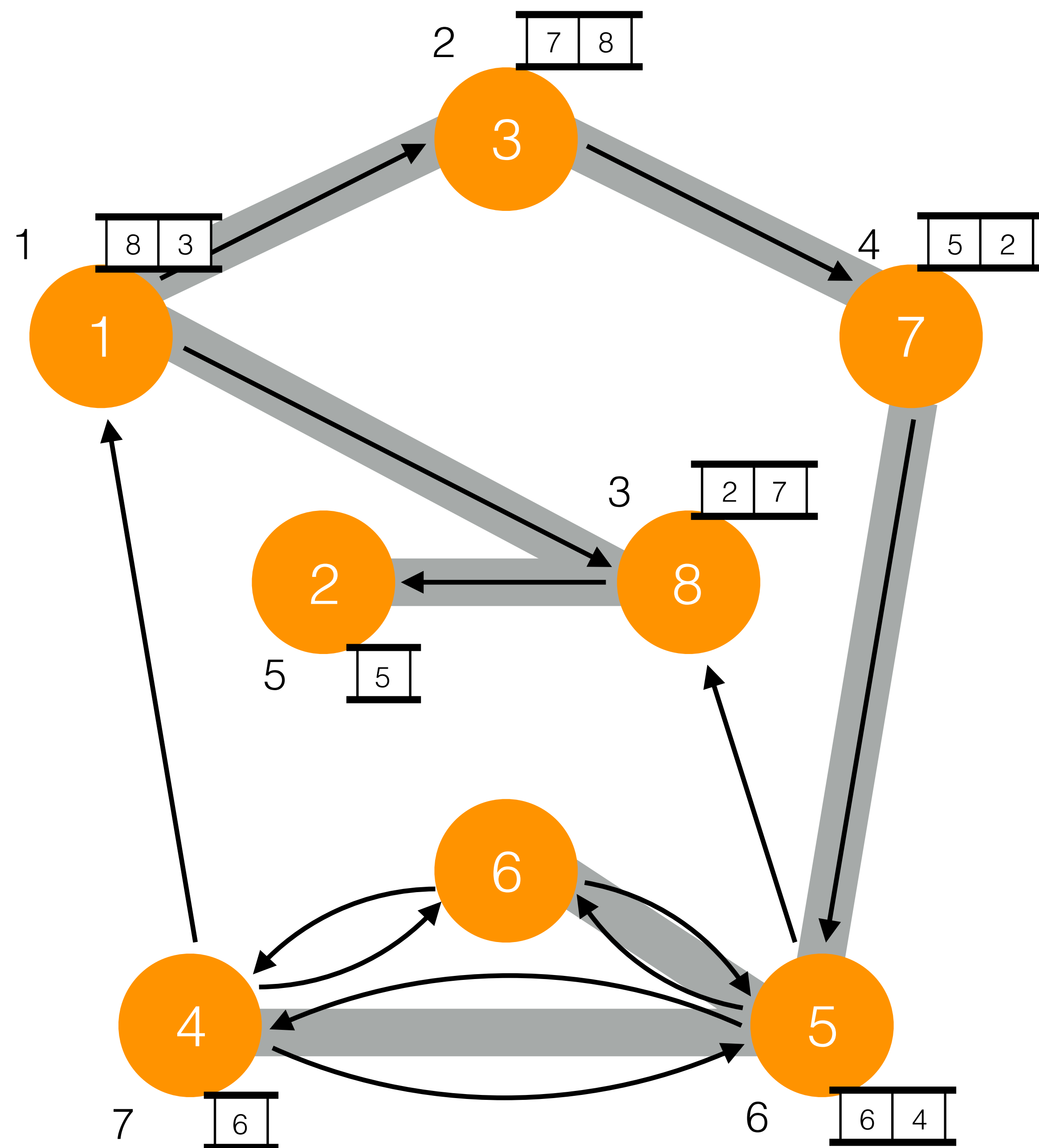

```
VISITE(i) =
    Q <- empty()
    VU[i] <- vraie
    enqueue(Q,i)
    tant que non vide?(Q)
        k <- dequeue(P)
        DEBUT[k] <- date
        date <- date + 1
        pour tout j ∊ Adj[k]
            si non VU[j] alors
                VU[j] <- vraie
                enqueue(Q,j)
```

Exemple



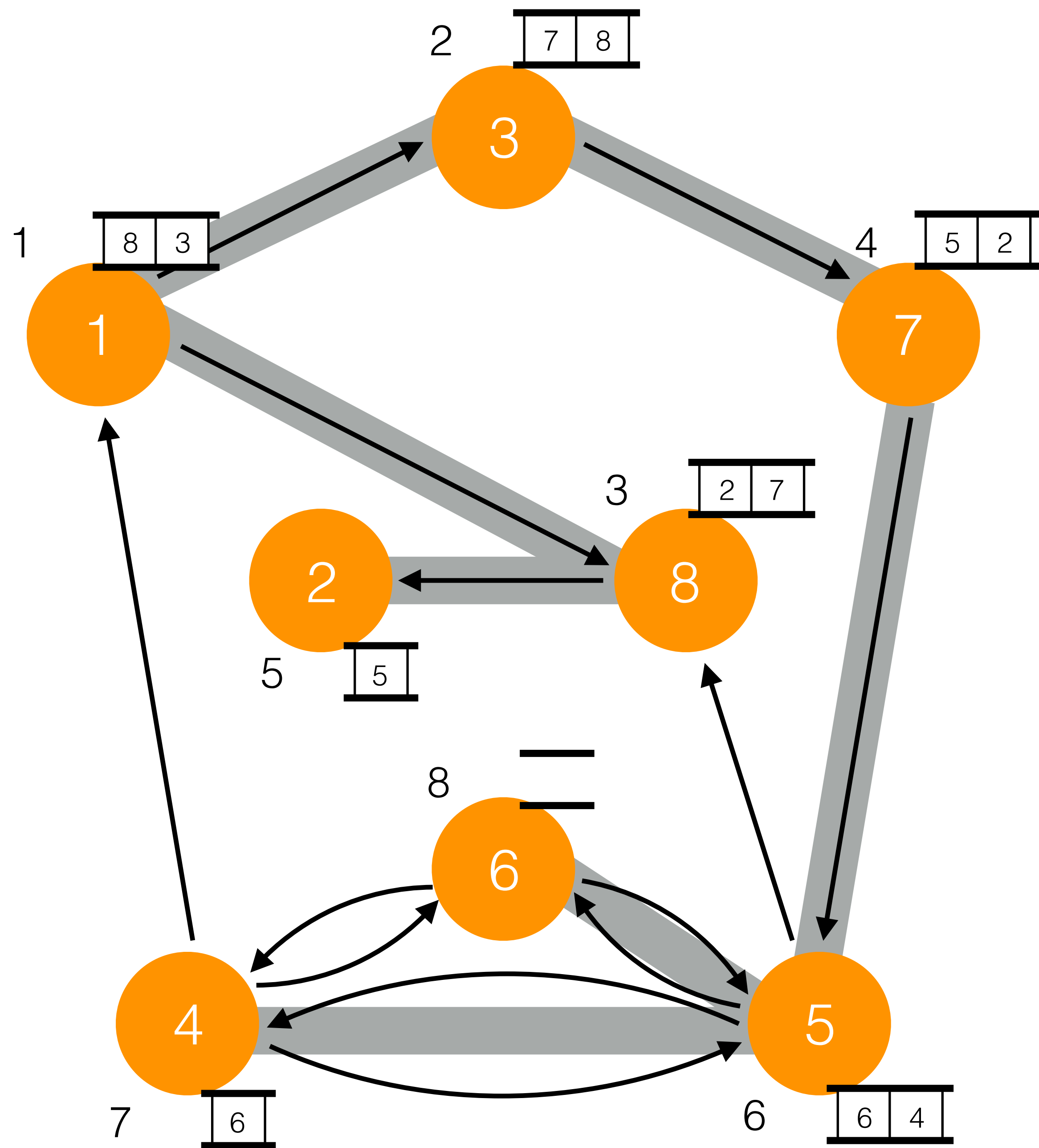

```
VISITE(i) =
    Q <- empty()
    VU[i] <- vraie
    enqueue(Q,i)
    tant que non vide?(Q)
        k <- dequeue(P)
        DEBUT[k] <- date
        date <- date + 1
        pour tout j ∈ Adj[k]
            si non VU[j] alors
                VU[j] <- vraie
                enqueue(Q,j)
```

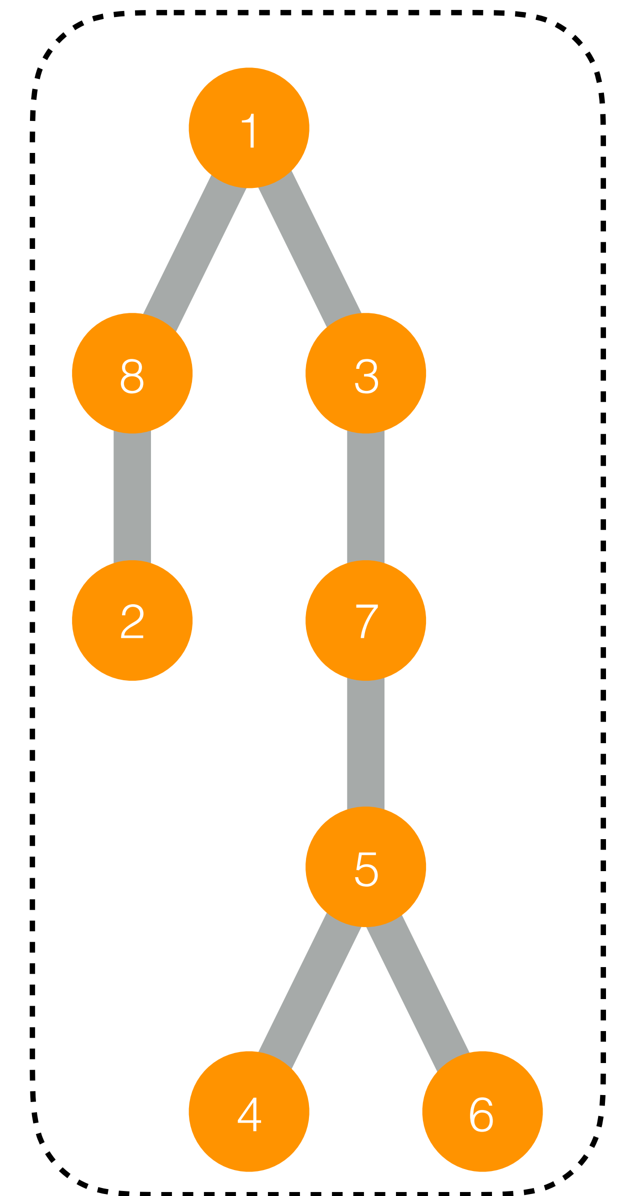
Exemple




```
VISITE(i) =
    Q <- empty()
    VU[i] <- vraie
    enqueue(Q,i)
    tant que non vide?(Q)
        k <- dequeue(P)
        DEBUT[k] <- date
        date <- date + 1
        pour tout j ∊ Adj[k]
            si non VU[j] alors
                VU[j] <- vraie
                enqueue(Q,j)
```

Exemple

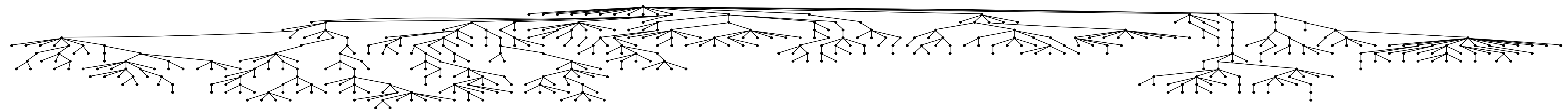




cf. fichier `maze_bfs.pdf`

Parcours en largeur

N=500



Arbre de parcours