# MATH-H-404 - Mise à niveau en algorithmique

Bernard Fortz  Nikita Veshchikov

Université Libre de Bruxelles
Département d'informatique
bernard.fortz@ulb.ac.be - nikita.veshchikov@ulb.ac.be

2011-2012

# Plan du cours

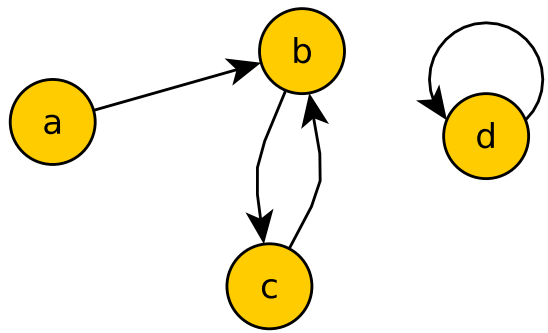
Les graphes

Les plus courts chemins

# Les graphes

# Introduction

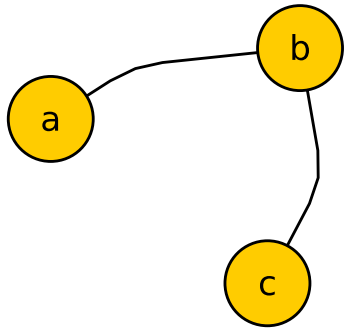
- Graphe : structure composée de sommets et soit d'arcs, soit d'arêtes.
- Sommets $\equiv$ objets, arcs / arêtes $\equiv$ relations entre ces objets.

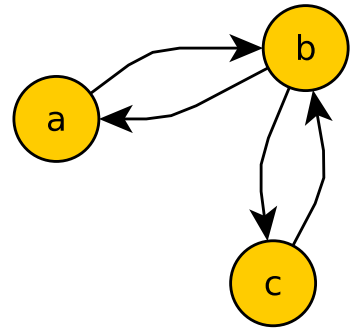


- Arc : relation dirigée entre deux sommets.


- Arête : relation non dirigée entre deux sommets.


- Arête entre deux sommets $\equiv$ deux arcs de sens opposés.



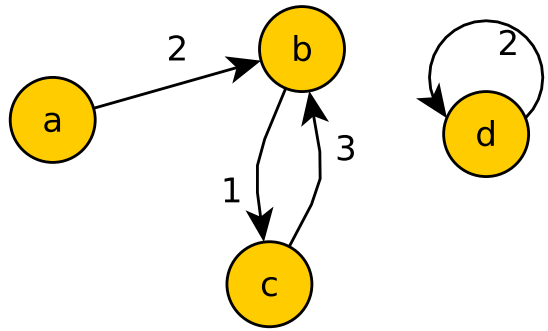



## Définition (Graphe)

Un graphe est défini comme une structure formée de deux ensembles :

- un ensemble $S$ de sommets ($\{a, b, c, d\}$, par exemple) ;
- un ensemble $A$ de couples de sommets ($\{(a, b), (b, c), (c, b), (d, d)\}$, par exemple).

- Ordre des couples de sommets importe : arcs, relation dirigée du premier sommet vers le second. Graphe dirigé ou orienté.
- Sinon, arêtes, relation non dirigée ($(a, b)$ équivalent à $(b, a)$). Graphe non dirigé.
- Eventuellement : étiquette associée à chaque arc / arête.



Quelques définitions utiles :

- l'ordre d'un graphe est le nombre de sommets qu'il contient ;
- un prédécesseur ou père d'un sommet $i$ est un sommet $j$ tel qu'il existe un arc ou une arête du sommet $j$ vers le sommet $i$ ;
- un successeur ou fils d'un sommet $i$ est un sommet $j$ tel qu'il existe un arc ou une arête du sommet $i$ vers le sommet $j$ ;
- deux sommets $i$ et $j$ qui sont des fils d'un même sommet $k$, sont dits frères ;
- le degré entrant d'un sommet est le nombre de ses prédécesseurs ;
- le degré sortant d'un sommet est le nombre de ses successeurs ;

- deux sommets d'un graphe sont adjacents s'ils sont distincts et s'il existe un arc ou une arête entre les deux sommets ;
- un sommet d'un graphe est isolé s'il n'est adjacent à aucun autre sommet ;
- deux arcs ou arêtes sont adjacents s'ils sont distincts et ont au moins une extrémité commune ;
- lorsque le graphe est tel qu'au plus $m$ arcs ou arêtes vont d'un sommet à un autre sommet, alors le graphe est aussi appelé un $m$-graphe ;

- une chaîne est une succession non vide d'arêtes contiguës ;
- un chemin est une succession orientée et non vide d'arcs contigus ;
- la longueur d'un chemin ou d'une chaîne est égale au nombre d'arcs ou d'arêtes qui le composent ;
- un chemin où n'apparaît pas deux fois le même arc est appelé un chemin simple ;
- un chemin qui ne passe pas plus d'une fois par chacun de ses sommets est appelé un chemin élémentaire ;
- un cycle, dans un graphe non dirigé, est une chaîne d'un sommet $a$ jusqu'à ce même sommet $a$ qui ne contient pas deux fois le même sommet (autre que $a$) :

- un cycle dans un graphe dirigé, appelé aussi un circuit, est un chemin d'un sommet *a* jusqu'à ce même sommet *a* qui ne contient pas deux fois le même sommet (autre que *a*) ;
- un chemin d'un sommet *a* jusqu'à ce même sommet *a* où chaque arc du graphe n'apparaît qu'une seule fois est appelé un cycle Eulerien ;
- un circuit qui comprend tous les sommets du graphe est appelé un cycle Hamiltonien ;
- un graphe qui ne contient aucun chemin qui part et arrive à un même sommet est appelé un graphe acyclique ;
- un graphe non dirigé qui contient une arête entre chaque paire de sommets est appelé un graphe complet.

# Graphes, arcs et arêtes

- Nombre maximum d'arêtes dans un graphe non dirigé de $n$ sommets : $\frac{n(n+1)}{2}$.
- Nombre maximum d'arcs d'un graphe dirigé de $n$ sommets : $n^2$
- $2^m$ graphes différents possibles avec $m$ arêtes.

| $n$ | # graphes non dirigés | # graphes dirigés |
|---|---|---|
| 2 | 8 | 16 |
| 3 | 64 | 512 |
| 4 | 1024 | 65536 |
| 5 | 32768 | 33554432 |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $i$ | $2^{\frac{i(i+1)}{2}}$ | $2^{i^2}$ |

- Graphe qui contient beaucoup moins d'arcs ou d'arêtes que le nombre maximal d'arcs ou d'arêtes qu'il peut contenir : graphe clairsemé (sparse graph).
- Graphe qui contient un nombre d'arcs ou d'arêtes proche du nombre maximal d'arcs ou d'arêtes qu'il peut contenir : graphe dense (dense graph).

# Mise en oeuvre

Implémentation d'un graphe :

- statique : matrice d'adjacence ou d'incidence ;
- dynamique : liste de prédécesseurs et/ou de successeurs.

# Mise en oeuvre

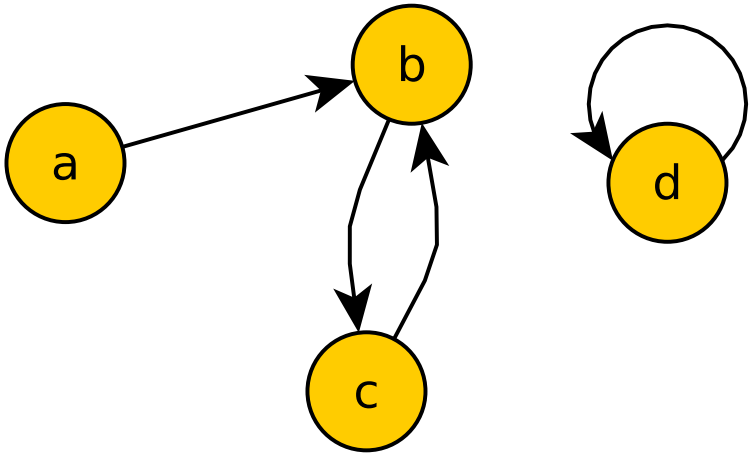
Implémentation d'un graphe :

statique : matrice d'adjacence ou d'incidence ;

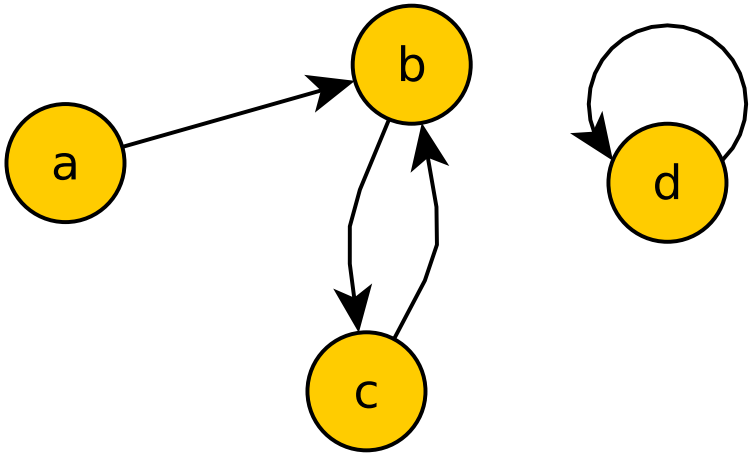
dynamique : liste de prédécesseurs et/ou de successeurs.

## Définition (Matrice d'adjacence)

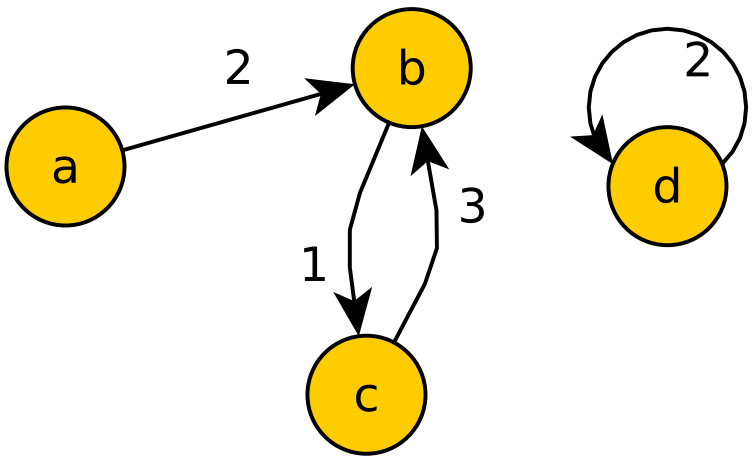
Matrice carrée $n \times n$ contenant des valeurs booléennes ou des étiquettes. La matrice d'adjacence donne de l'information concernant un lien (via un arc ou une arrête) entre deux sommets.




| | *a* | *b* | *c* | *d* |
|---|---|---|---|---|
| *a* | *F* | *V* | *F* | *F* |
| *b* | *F* | *F* | *V* | *F* |
| *c* | *F* | *V* | *F* | *F* |
| *d* | *F* | *F* | *F* | *V* |




| | *a* | *b* | *c* | *d* |
|---|---|---|---|---|
| *a* | *F* | *V* | *F* | *F* |
| *b* | *F* | *F* | *V* | *F* |
| *c* | *F* | *V* | *F* | *F* |
| *d* | *F* | *F* | *F* | *V* |



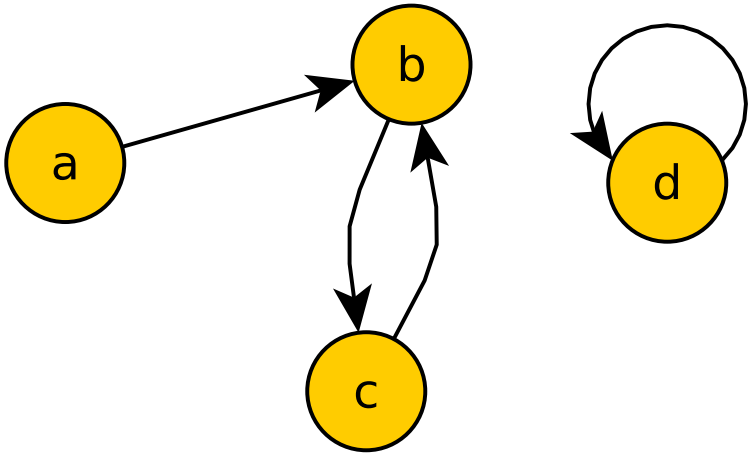

| | *a* | *b* | *c* | *d* |
|---|---|---|---|---|
| *a* | 0 | 2 | 0 | 0 |
| *b* | 0 | 0 | 1 | 0 |
| *c* | 0 | 3 | 0 | 0 |
| *d* | 0 | 0 | 0 | 2 |

## Définition (Matrice d'incidence)

Matrice donnant de l'information sur la relation entre un sommet et un arc ou une arête. L'information se trouvant aux indices $i$ et $j$ indique à la fois si l'arc ou l'arête $j$ pointe vers le sommet $i$ et/ou provient du sommet $i$.

## Définition (Matrice d'incidence)

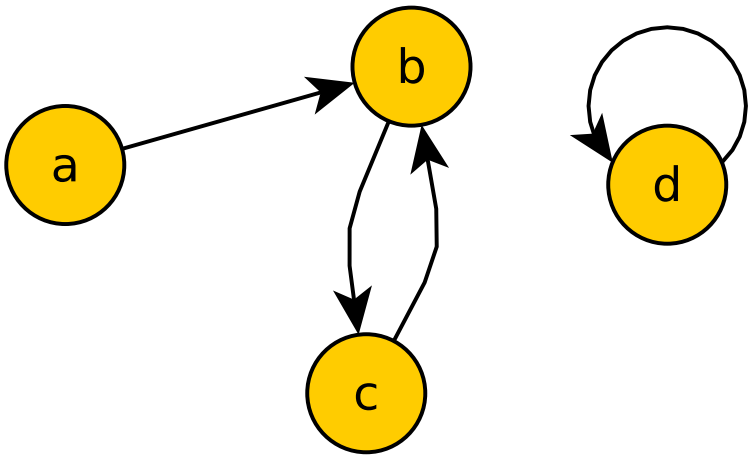
Matrice donnant de l'information sur la relation entre un sommet et un arc ou une arête. L'information se trouvant aux indices *i* et *j* indique à la fois si l'arc ou l'arête *j* pointe vers le sommet *i* et/ou provient du sommet *i*.




| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| *a* | $(V, F)$ | $(F, F)$ | $(F, F)$ | $(F, F)$ |
| *b* | $(F, V)$ | $(V, F)$ | $(F, V)$ | $(F, F)$ |
| *c* | $(F, F)$ | $(F, V)$ | $(V, F)$ | $(F, F)$ |
| *d* | $(F, F)$ | $(F, F)$ | $(F, F)$ | $(V, V)$ |

## Définition (Matrice d'incidence)

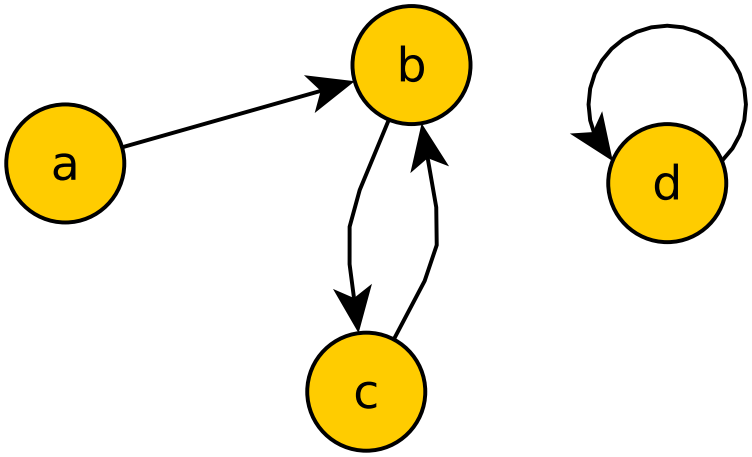
Matrice donnant de l'information sur la relation entre un sommet et un arc ou une arête. L'information se trouvant aux indices *i* et *j* indique à la fois si l'arc ou l'arête *j* pointe vers le sommet *i* et/ou provient du sommet *i*.

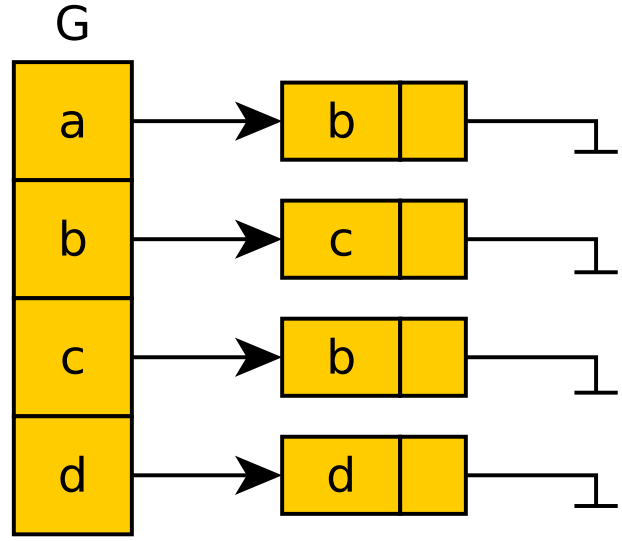


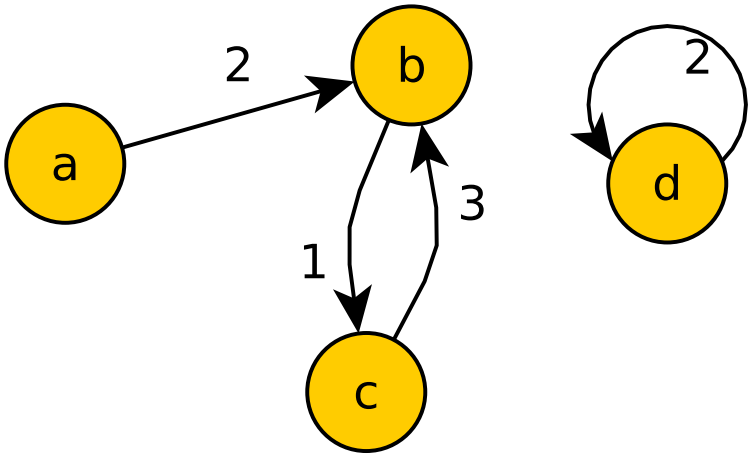

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| *a* | $(V, F)$ | $(F, F)$ | $(F, F)$ | $(F, F)$ |
| *b* | $(F, V)$ | $(V, F)$ | $(F, V)$ | $(F, F)$ |
| *c* | $(F, F)$ | $(F, V)$ | $(V, F)$ | $(F, F)$ |
| *d* | $(F, F)$ | $(F, F)$ | $(F, F)$ | $(V, V)$ |

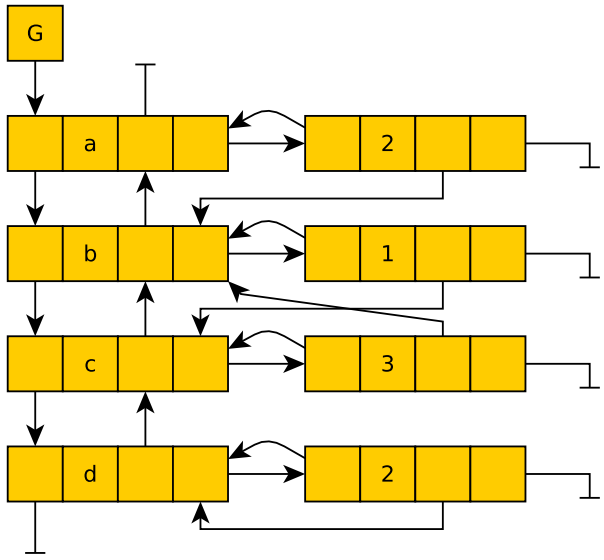
- Graphe non dirigé : un seul booléen est suffisant.
- Graphe dirigé sans boucle : valeurs ternaires
  - 1 si l'arc part du sommet,
  - -1 si l'arc arrive au sommet,
  - 0 si l'arc n'est pas liée au sommet.

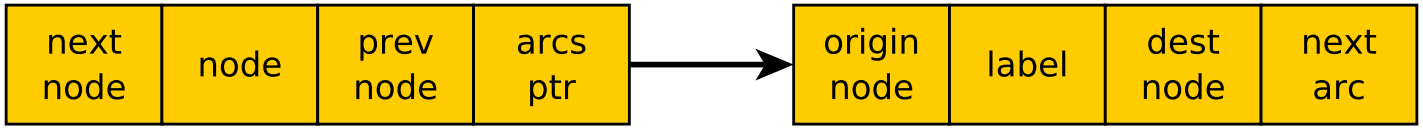
Une liste de successeurs peut être représentée par une matrice creuse.



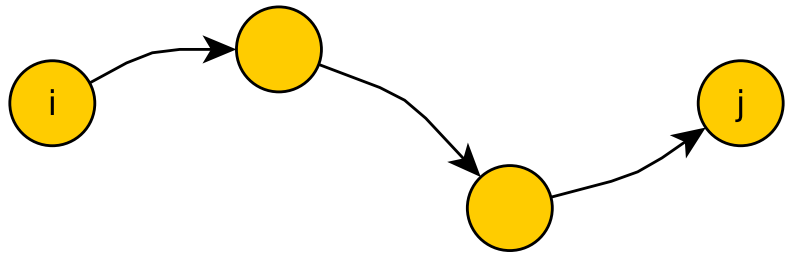

a
b
c
d
2
1
3
2


G
a
2
b
1
c
3
d
2


next node
node
prev node
arcs ptr
origin node
label
dest node
next arc

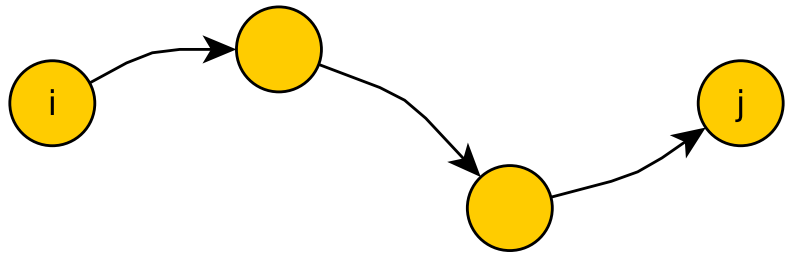
# Accessibilité

- Matrice d'adjacence $M$, $M_{ij}$ valeur booléenne indiquant s'il existe un arc ou une arête allant du sommet $i$ au sommet $j$.
- Existe-t-il un chemin ou une chaîne du sommet $i$ au sommet $j$ ?



# Accessibilité

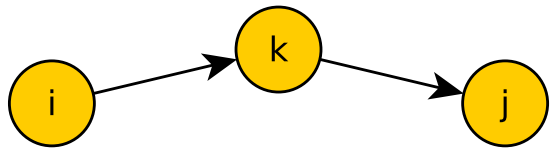
- Matrice d'adjacence $M$, $M_{ij}$ valeur booléenne indiquant s'il existe un arc ou une arête allant du sommet $i$ au sommet $j$.
- Existe-t-il un chemin ou une chaîne du sommet $i$ au sommet $j$ ?



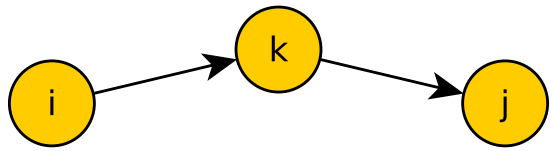

## Définition (Matrice d'accessibilité)

Matrice $A$ telle que $A_{ij}$ indique si le sommet $j$ est accessible à partir du sommet $i$.

$$M^2_{ij} = (M_{i1} \texttt{ and } M_{1j}) \texttt{ or } \ldots \texttt{ or } (M_{in} \texttt{ and } M_{nj}) = \bigvee_{k=1}^{n} M_{ik} \wedge M_{kj}$$



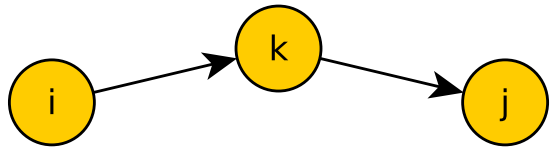
$$M^2_{ij} = (M_{i1} \texttt{ and } M_{1j}) \texttt{ or } \ldots \texttt{ or } (M_{in} \texttt{ and } M_{nj}) = \bigvee_{k=1}^{n} M_{ik} \wedge M_{kj}$$




$$M^r_{ij} = (M_{i1} \texttt{ and } M^{r-1}_{1j}) \texttt{ or } \ldots \texttt{ or } (M_{in} \texttt{ and } M^{r-1}_{nj})$$

Si $M^r_{ij}$ est vrai, il existe un chemin de longueur $r$ allant du sommet $i$ vers le sommet $j$.

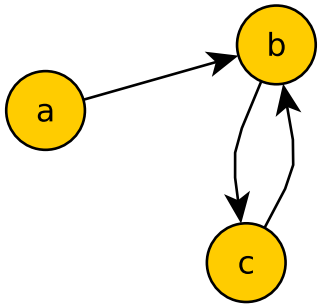
$$M^2_{ij} = (M_{i1} \texttt{ and } M_{1j}) \texttt{ or } \ldots \texttt{ or } (M_{in} \texttt{ and } M_{nj}) = \bigvee_{k=1}^{n} M_{ik} \wedge M_{kj}$$




$$M^r_{ij} = (M_{i1} \texttt{ and } M^{r-1}_{1j}) \texttt{ or } \ldots \texttt{ or } (M_{in} \texttt{ and } M^{r-1}_{nj})$$

Si $M^r_{ij}$ est vrai, il existe un chemin de longueur $r$ allant du sommet $i$ vers le sommet $j$.

$$A = M^0 \vee M^1 \vee M^2 \vee \cdots = \bigvee_{r=0}^{n-1} M^r$$




$$M = \begin{array}{c|ccc} & a & b & c \\ \hline a & 0 & 1 & 0 \\ b & 0 & 0 & 1 \\ c & 0 & 1 & 0 \end{array} \qquad M^2 = \begin{array}{c|ccc} & a & b & c \\ \hline a & 0 & 0 & 1 \\ b & 0 & 1 & 0 \\ c & 0 & 0 & 1 \end{array}$$

$$A = M^* = M^0 \vee M^1 \vee M^2 = \begin{array}{c|ccc} & a & b & c \\ \hline a & 1 & 1 & 1 \\ b & 0 & 1 & 1 \\ c & 0 & 1 & 1 \end{array}$$

# Algorithme de Roy-Warshall

```
// A = matrice d'adjacence du graphe

for (k=0 ; k < ordre du graphe ; ++k)
  for (i=0 ; i < ordre du graphe ; ++i)
     if (A[i][k])
       for (j=0 ; j < ordre du graphe ; ++j)
         A[i][j]=A[i][j] or A[k][j]
```

## Algorithme de Roy-Warshall

```
// A = matrice d'adjacence du graphe

for (k=0 ; k < ordre du graphe ; ++k)
  for (i=0 ; i < ordre du graphe ; ++i)
     if (A[i][k])
       for (j=0 ; j < ordre du graphe ; ++j)
         A[i][j]=A[i][j] or A[k][j]
```

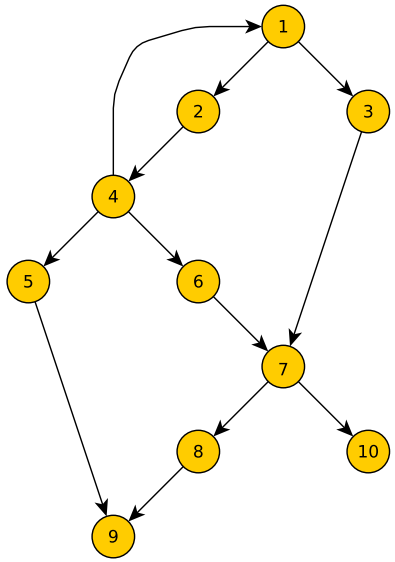
## Complexité

$O(n^3)$ (alors que $A = \bigvee_{r=0}^{n-1} M^r$ est en $O(n^4)$).

# Parcours de graphes

Deux manières classiques pour parcourir un graphe :

- parcours en profondeur (depth-first) ;
- parcours en largeur (breadth-first).

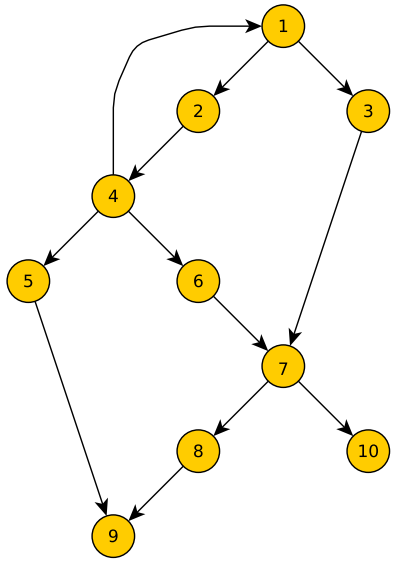



## Parcours en profondeur

Le parcours en profondeur découvre les sommets du graphe en essayant toujours de descendre, en suivant les arcs ou les arêtes, le plus profondément dans le graphe.

$$1, 2, 4, 5, 9, 6, 7, 8, 10, 3$$

A chaque fois, retour au dernier sommet pour lequel il reste encore des successeurs à visiter.

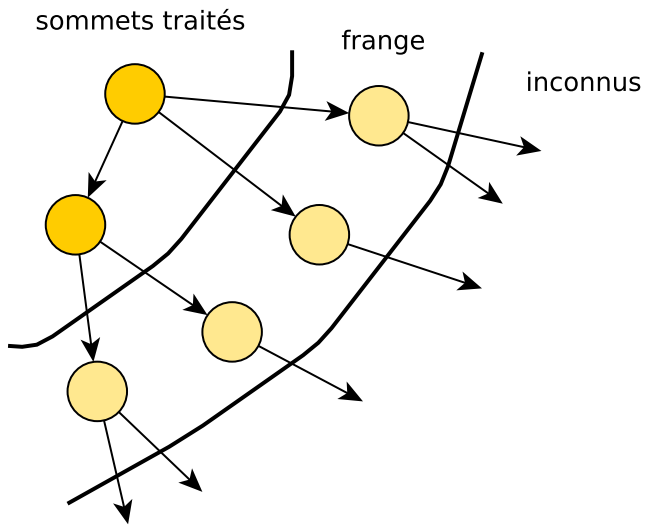



## Parcours en largeur

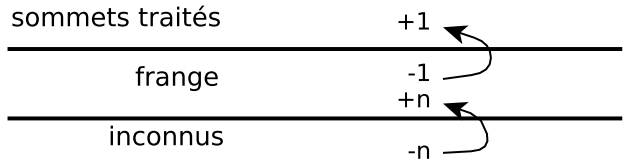
Le parcours en largeur explore les sommets du graphe par niveau, en découvrant d'abord les sommets à une distance 1, en terme du nombre d'arcs ou d'arêtes, du sommet de départ, puis les sommets à distance 2, etc.

$$1, 2, 3, 4, 7, 5, 6, 8, 10, 9$$

- L'ordre d'exploration dépend aussi de l'ordre dans lequel on considère les fils d'un sommet (dépendant de l'implémentation).
- A tout moment, classification des sommets en trois ensembles :
  1. l'ensemble des sommets déjà traités ;
  2. l'ensemble des sommets frontaliers, appelés aussi la frange ;
  3. l'ensemble des sommets encore inconnus.



- A chaque itération, nous sélectionnons un élément de la frange pour être traité et un certain nombre, $n$, d'éléments inconnus entrent dans la frange :




- La manière de choisir les éléments de la frange qui seront traités détermine l'algorithme :

  parcours en profondeur : sommet frontalier que l'on a rencontré en dernier.

  parcours en largeur : sommets frontaliers dans l'ordre dans lequel on les a rencontrés.

## Structures de données

parcours en profondeur : pile.

parcours en largeur : file (ou queue).

## Structures de données

parcours en profondeur : pile.

parcours en largeur : file (ou queue).

## Parcours en profondeur récursif

```
void DF(sommet s)
{
  traiter s
  marquer s
  a=s->1er_arc
  while(a existe)
  {
    traiter a
    if(a->extremite n est pas marquee)
      DF(a->extremite)
    a=a->arc_suivant
  }
}
```

# Parcours en profondeur itératif

```
void DF(sommet s) {
  do {
     while(s n est pas marque) {
       traiter s
       marquer s
       a=s->1er_arc
       if(a existe) {
         traiter a
         s=a->extremite
         a=a->arc_suivant
         if(a existe) push(a)
       }
     }
     if(stack n est pas vide) {
       a=pop()
       traiter a
       s=a->extremite
       a=a->arc_suivant
       if(a existe) push(a)
     }
  }
  while(s n est pas marque or stack n est pas vide)
}
```

# Parcours en largeur

```
void BF(sommet s) {
  traiter s
  marquer s
  enqueue(s)
  while(queue n est pas vide) {
    s=dequeue()
    a=s->1er_arc
    while(a existe) {
      traiter a
      s=a->extremite
      if(s n est pas marque) {
        traiter s
        marquer s
        enqueue(s)
      }
      a=a->arc_suivant
    }
  }
}
```

# Parcours en largeur au moyen d'un vecteur

```
void BF() {
  n=0
  val=(0,...,0)
  enqueue(1)
  while(queue n est pas vide) {
    ++n
    k=dequeue()
    val[k]=n
    t=1er successeur du sommet k
    while(t existe) {
      if(val[t] == 0) {
        enqueue(t)
        val[t] = -1
      }
      t=successeur suivant de t
    }
  }
}
```

# Parcours généralisés

- Il est possible qu'un graphe possède plusieurs sommets origine.
- On dit alors qu'il est formé de plusieurs composantes.
- Lancer les parcours pour chaque sommet origine de chaque composante.

# Parcours généralisés

- Il est possible qu'un graphe possède plusieurs sommets origine.
- On dit alors qu'il est formé de plusieurs composantes.
- Lancer les parcours pour chaque sommet origine de chaque composante.

## Parcours en profondeur généralisé

```
void DepthFirst(graphe g)
{
  demarquer g
  n=ordre de g
  for(s=1 ; s <= n ; ++s)
     if(sommet s n est pas marque)
        DF(s)
}
```

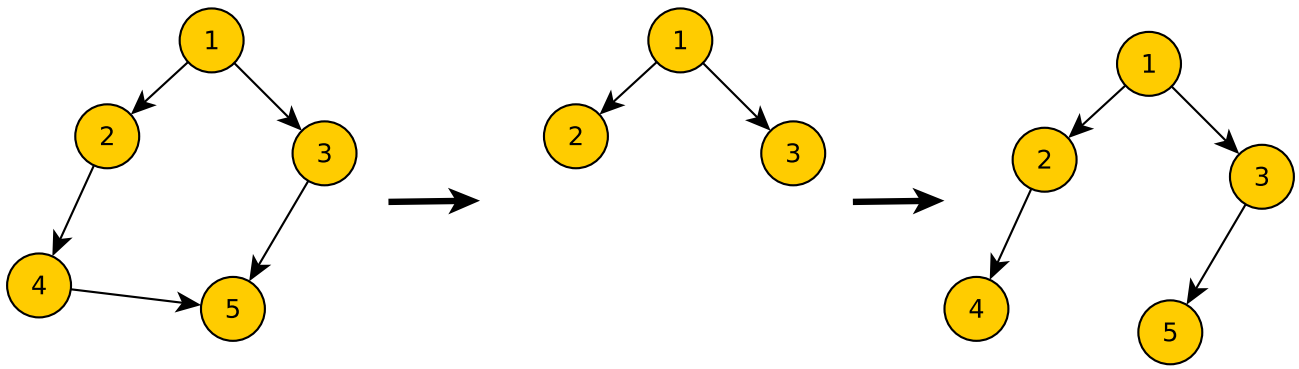
# Parcours en largeur généralisé

```
void BreadthFirst(graphe g)
{
  demarquer g
  n=ordre de g
  for(s=1 ; s <= n ; ++s)
     if(sommet s n est pas marque)
        BF(s)
}
```

# Les plus courts chemins

# Introduction

- Trouver le plus court chemin (shortest path) entre deux sommets d'un digraphe pondéré consiste en la recherche du chemin dont la somme des pondérations des arcs le composant est minimale, parmi tous les chemins reliant les deux sommets donnés.
- Cas particulier : digraphe où tous les poids sont égaux. Le parcours en largeur permet de trouver le plus court chemin entre un sommet donné et tous les autres sommets du graphe.



# Plus courts chemins dans un digraphe

- Trouver les plus courts chemins reliant un sommet $x$ d'un digraphe donné à tous les autres sommets de ce digraphe.
- Pondérations positives.
- Conséquence : jamais de cycle dans les courts chemins entre deux sommets.
- La construction des plus courts chemins depuis un sommet du graphe conduit à produire un arbre recouvrant du digraphe.
- Construction de l'arbre des plus courts chemins à partir d'un sommet $x$ : ajouter à chaque étape le sommet de la frange le plus proche du sommet $x$.

Illustration :

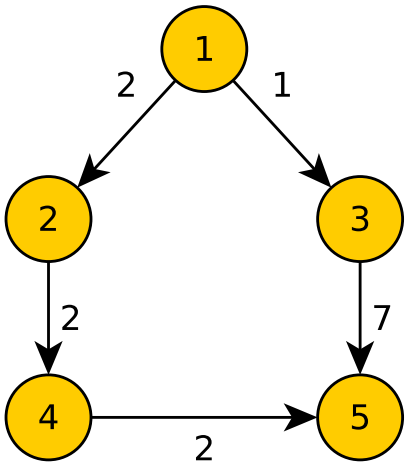
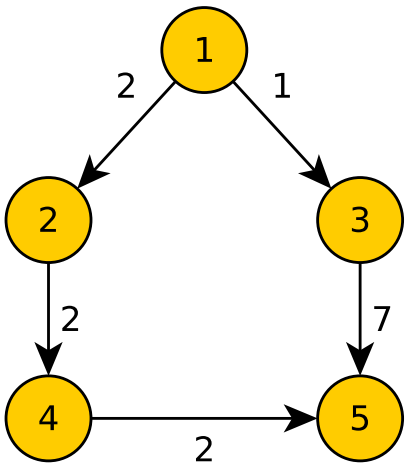
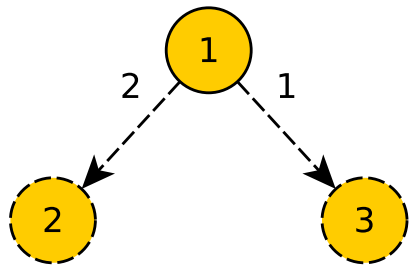
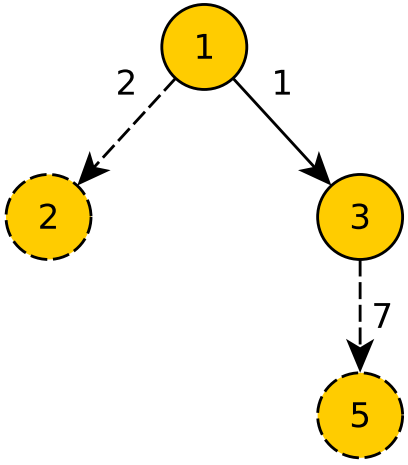
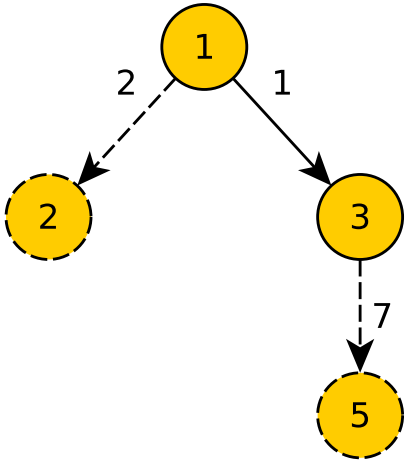


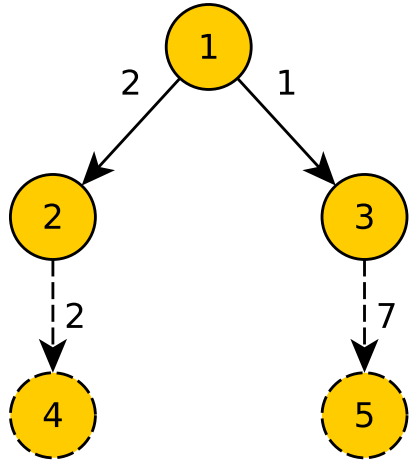
Illustration :

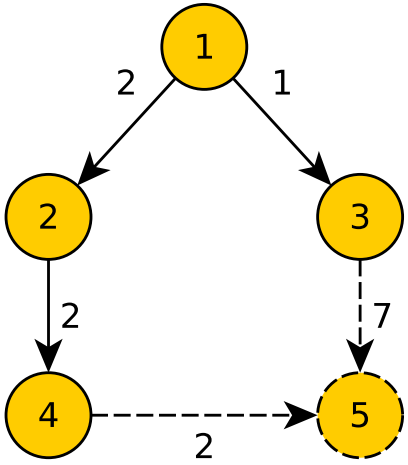


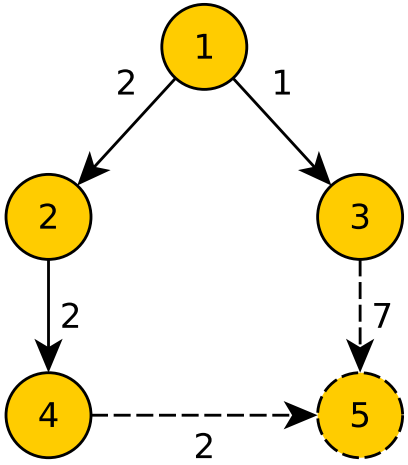



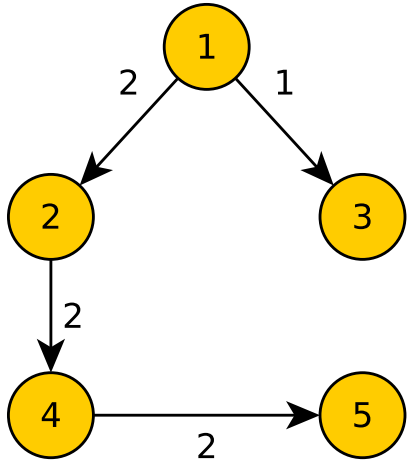

1
2
1
2
3
7
5


1
2
1
2
3
7
5

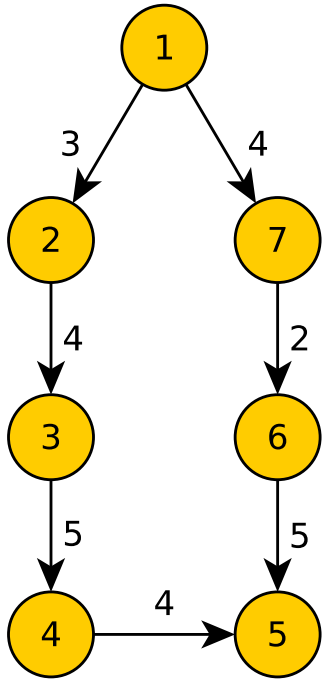

1
2
1
2
3
2
7
4
5


1
2
1
2
3
2
7
4
5
2

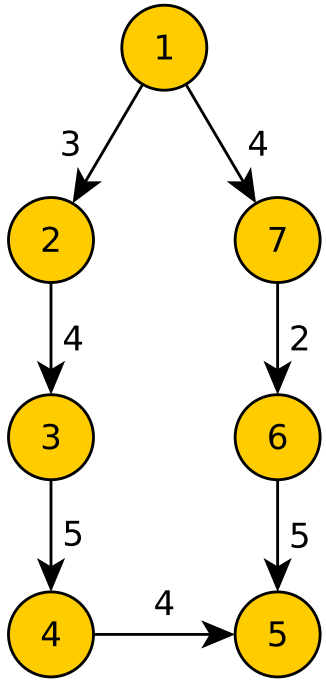

1
2
1
2
3
2
7
4
5
2

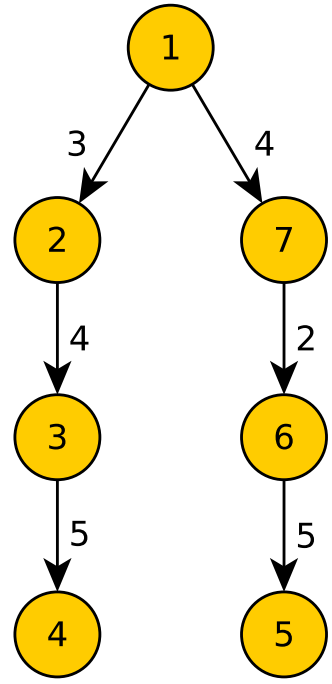

1
2
1
2
3
2
4
5
2

Autre exemple :



Autre exemple :



- Algorithme : stocke dans un vecteur `dist`, la distance de chaque sommet avec le sommet initial `x`.
- A l'ajour d'un sommet `k` : mise à jour de la frange en parcourant la liste de successeurs de `k`.
- Pour chaque sommet `t` de cette liste, la distance de `x` à `t` est égale à `dist[k] + distance de k à t`.
- Propriété fondamentale : dans un digraphe, le chemin le plus court entre `a` et `c` est composé du plus court chemin entre `a` et `b` et du plus court chemin entre `b` et `c`, pour tout sommet `b` se trouvant sur le plus court chemin entre `a` et `c`.

# Algorithme de Dijkstra

```
M=ensemble vide
for(i=1; i<= ordre du graphe; ++i) {
  dist[i]=poids arc(s,i); prec[i]=s
  rajouter le sommet i a M
}
retirer le sommet s de M
while(M n est pas vide) {
  m=sommet x appartenant a M tel que dist[x] est minimum
  retirer le sommet m de M
  if(dist[m] == infini)
    M=ensemble vide
  else {
    for(a=1er arc sortant de m ; a existe ; a=arc suivant) {
      y=extremite de a
      if(y appartient a M) {
        v=dist[m]+poids arc(m,y)
        if(v < dist[y]) {
          dist[y]=v
          prec[y]=m
        }
      }
    }
  }
}
```

- L'ensemble `M` peut être géré au moyen d'un heap de sommets dont la priorité de chaque sommet est inversement proportionnelle à la distance courante entre le sommet `s`. initial et le sommet considéré.
- Complexité totale : $O(|A| \log |S|)$.

# Tous les plus courts chemins

- On pourrait réaliser une recherche des plus courts chemins, tel que réalisé par l'algorithme de Dijkstra, en partant tour à tour de chaque sommet du digraphe.
- Autre approche : calculer la fermeture transitive du graphe en considérant que s'il existe un moyen pour aller de `x` à `y` en n'utilisant que des sommets d'indice inférieur ou égal à `k-1` et s'il existe un moyen pour aller de `k` à `y`, alors il existe un moyen pour aller de `x` à `y` en n'utilisant que des sommets d'indice inférieur ou égal à `k`.
- Il est ainsi espéré que le nouveau chemin considéré, passant par le sommet `k`, soit plus court que tous ceux déjà explorés.

# Algorithme de Floyd

```
for(k=0 ; k < ordre du graphe ; ++k)
  for(x=0 ; x < ordre du graphe ; ++x)
    if(mat[x][k] < infini)
      for(y=0 ; y < ordre du graphe ; ++y)
        if(mat[k][y] < infini)
          if(mat[x][k]+mat[k][y] < mat[x][y])
            mat[x][y]=mat[x][k]+mat[k][y]
```

Complexité : $O(|S|^3)$

# Plus courts chemins dans un DAG

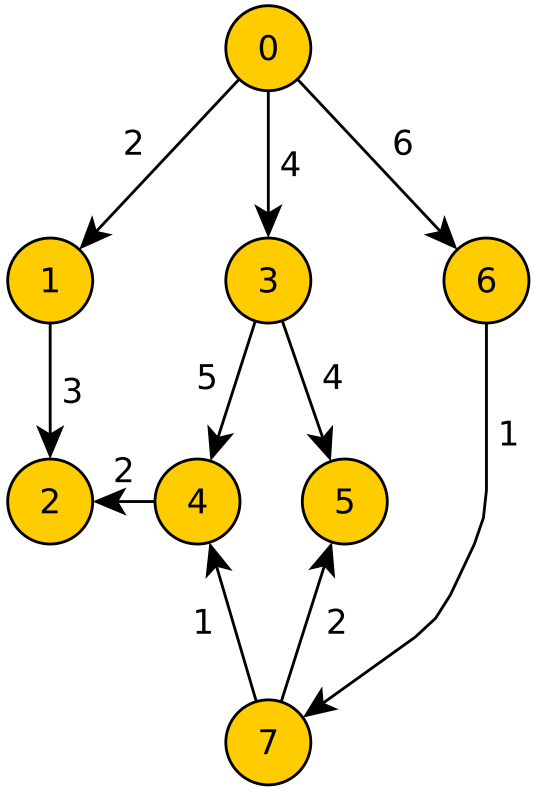
- Pour déterminer le plus court chemin depuis un sommet initial `s` jusqu'à un sommet `x`, il faut connaître le plus court chemin du sommet `s` jusqu'à chacun des prédécesseurs du sommet `x`. Ainsi, on ne traite un sommet que si tous ses prédécesseurs sont traités.
- Cela nous ramène au tri topologique.
- Une fois que tous les plus courts chemins entre `s` et tous les prédécesseurs d'un sommet `x` sont connus, il est facile de déterminer le plus court chemin de `s` à `x`.
- Pour chaque prédécesseur `y` de `x`, additionner la taille du plus court chemin entre les sommets `s` et `y` à la distance reprise sur l'arc entre les sommets `y` et `x`. Le résultat le plus petit indiquera le plus court chemin entre `s` et `x`.

# Algorithme de Bellman

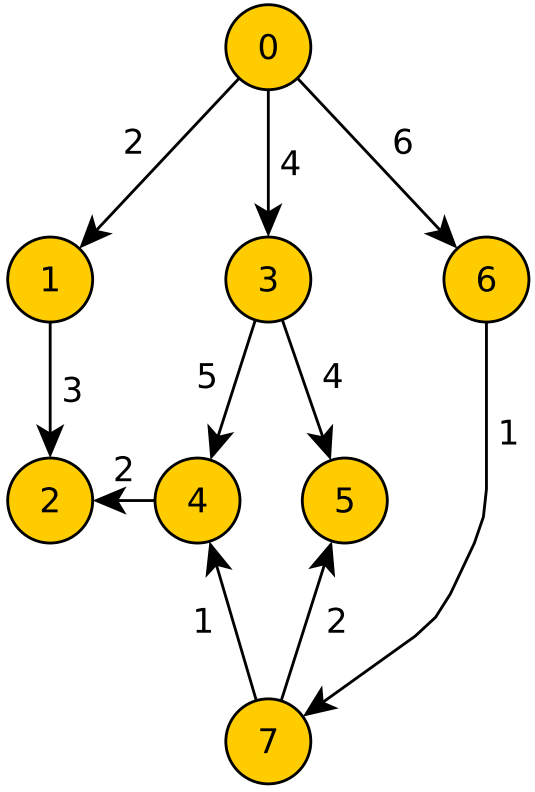
```
dist[0 ... n-1] = +infini
dist[s] = 0
renumeroter les sommets dans l ordre topologique
for(int k=1 ; k < ordre du graphe ; ++k)
{
   j = i tel que pour tous les arcs (i,k) dans le graphe
         on ait dist[i]+mat[i][k] qui soit minimum
   dist[k] = dist[j]+mat[j][k]
   pred[k]=j
}
```

Complexité : $O(|S| + |A|)$

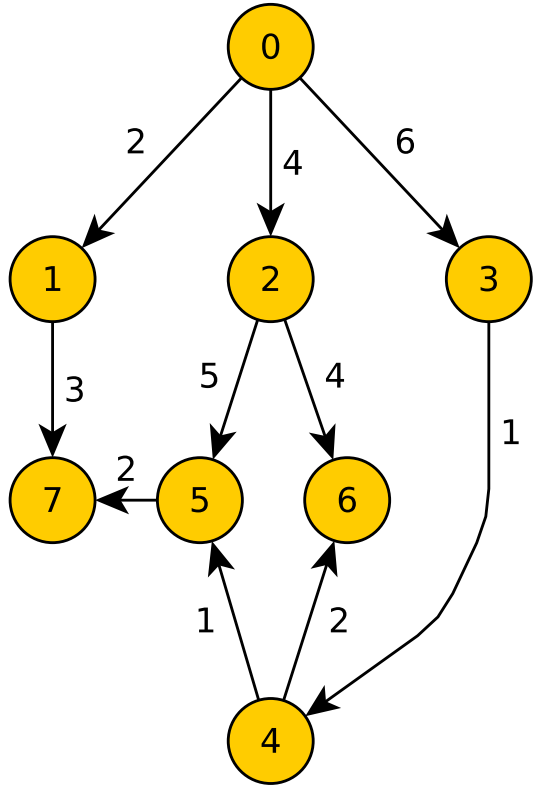
Exemple :



Exemple :

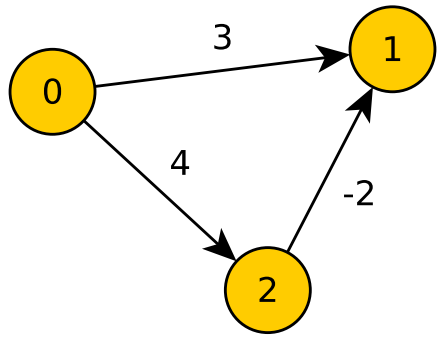



Après renumérotation :



# Digraphes ayant des poids négatifs

- Dijkstra ne fonctionne plus.




- L'algorithme de Dijkstra trouverait un chemin du sommet `0` au sommet `1` de longueur 3 et un chemin du sommet `0` au sommet `2` de longueur 4. Alors que pour aller du sommet `0` au sommet `1` le plus court chemin passe par le sommet `2` et est de longueur 2.

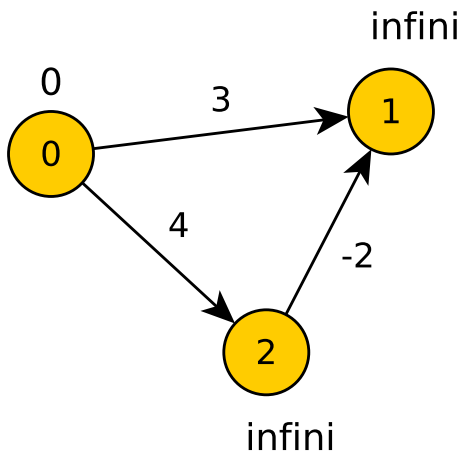
- Poids positifs : quand un sommet `t` est considéré, par l'algorithme de Dijkstra, comme faisant partie d'un plus court chemin, nous sommes sûr d'avoir trouvé la plus courte distance du sommet d'origine à ce sommet `t` .
- S'il existe des poids négatifs : même si un sommet `u` est plus éloigné du sommet d'origine que le sommet `t` , il peut être avantageux de passer par `u` puisque le plus court chemin de `u` à `t` pourrait être d'un poids total négatif.
- Poids d'un cycle : somme des poids des arcs qui le composent.
- Un graphe ayant un cycle de poids négatif n'a pas de plus court chemin. Nous considérons donc des graphes sans cycle de poids négatif.

# Algorithme de Bellman et Ford

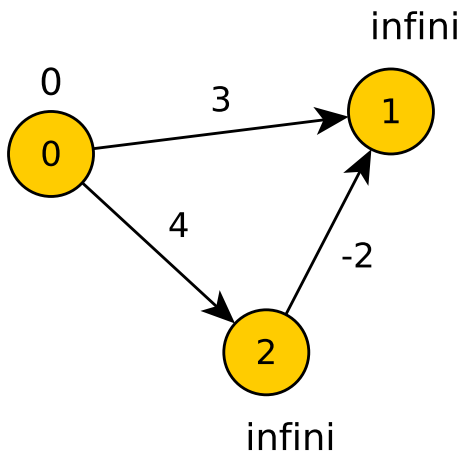
```
mettre le poids de tous les sommets a +infini
mettre le poids du sommet initial a 0

for(int i=1 ; i < ordre du graphe ; ++i)
{
  for((u,v)=1er arc du graphe; (u,v) existe ; (u,v)=arc suivant)
  {
    val=poids du sommet u + poids de l arc (u,v)
    if(val < poids du sommet v)
    {
      pred[v]=u
      poids du sommet v=val
    }
  }
}
for((u,v)=1er arc du graphe; (u,v) existe ; (u,v)=arc suivant)
  if(poids de u + poids de (u,v) < poids de v)
    return(false)
return(true)
```

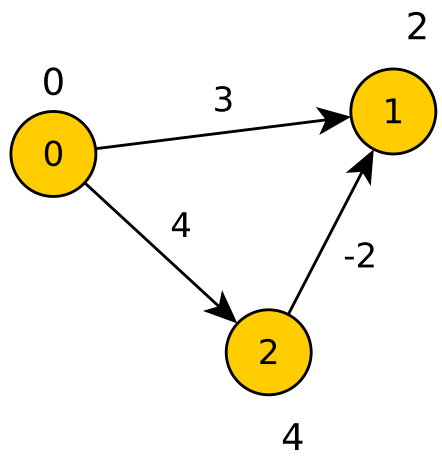
Complexité : $O(|S||A|)$

Exemple :



Exemple :



# Dernières remarques

- Le couple de sommets dans un digraphe qui est tel que le chemin entre les sommets de ce couple est le plus grand du digraphe définit le diamètre du digraphe. La valeur de ce diamètre est donc la longueur de ce plus long chemin.
- Si un DAG possède plusieurs sommets initiaux sans prédécesseur, nous pouvons le modifier en lui ajoutant une source unique qui a comme successeurs les sources initiales et dont les arcs ainsi ajoutés sont de poids nuls.