

# Graphes et algorithmique sur les graphes

Bloc 5: cours 3

Ces transparents sont inspirés par ceux d'Elisa Fromont, IRISA et  
Université Rennes 1

DIU « Enseigner l'Informatique au Lycée »

# Programme de Terminale NSI

- Structure de données - Graphes : structures relationnelles. Sommets, arcs, arêtes, graphes orientés ou non orientés.

Bloc  
4

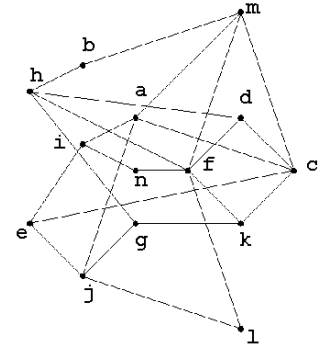
- Modéliser des situations sous forme de graphes.  
Écrire les implémentations correspondantes d'un graphe : matrice d'adjacence, liste de successeurs/de prédécesseurs. Passer d'une représentation à une autre.
- On s'appuie sur des exemples comme le réseau routier, le réseau électrique, internet, les réseaux sociaux. Le choix de la représentation dépend du traitement qu'on veut mettre en place : on fait le lien avec la rubrique « algorithmique ».

Bloc  
5

## Algorithmes sur les graphes

- Parcourir un graphe en profondeur d'abord, en largeur d'abord. Repérer la présence d'un cycle dans un graphe.
- Chercher un chemin dans un graphe.
- Le parcours d'un labyrinthe et le routage dans internet sont des exemples d'algorithmes sur les graphes. L'exemple des graphes permet d'illustrer l'utilisation des classes en programmation.

# Une structure de données utile : le graphe !



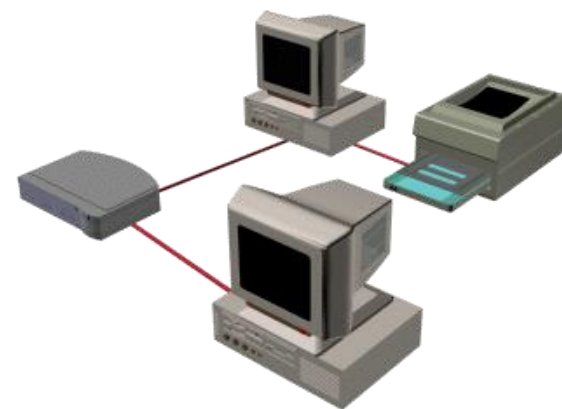
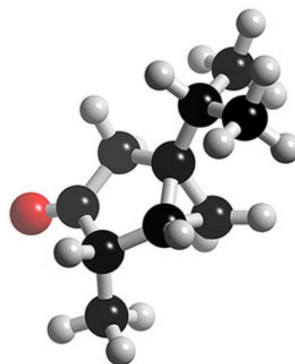
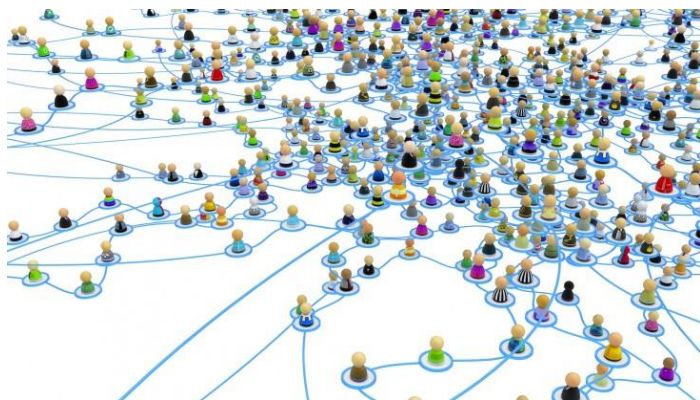
- Vous avez dit « structure de données » ???
  - En informatique, une **structure de données** est une structure logique destinée à contenir des données, afin de leur donner une organisation permettant de simplifier leur traitement.

•

- Singletons (constantes, variables, ensembles, ....)
- Tableaux
- Listes
- Arbres (déjà vu)
- Graphes (une généralisation des arbres)

•

# Des graphes naturellement partout !!



# Analyse de réseaux (sociaux)

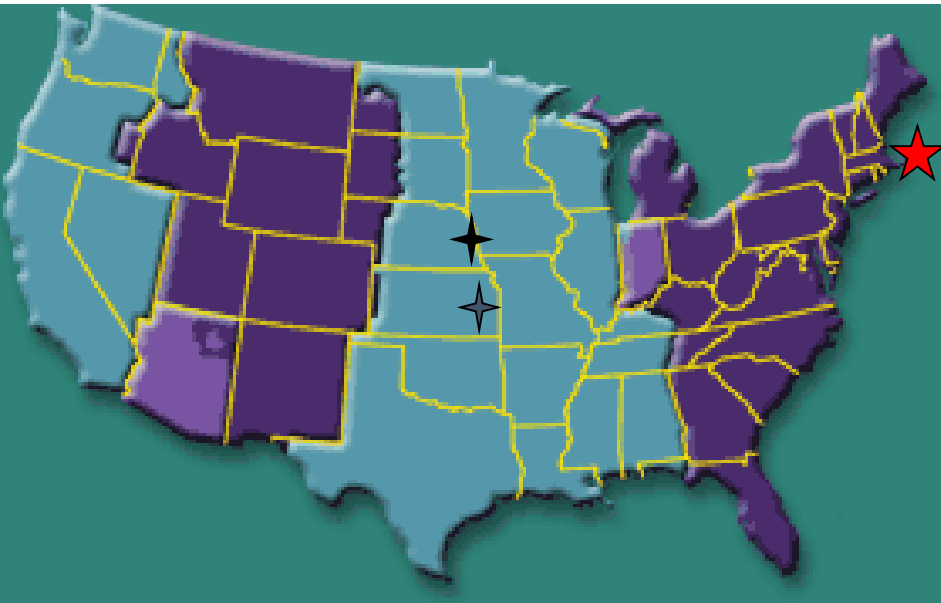
## Exemple de problèmes :

- Identifier des communautés dans les réseaux
- Identifier les « hubs » (les personnes « importantes ») dans les réseaux
- Prédire les « liens » dans le réseau

## Pour :

- Publicité personnalisée
- Marketing viral, marketing “temps réel”
- Contrôle des épidémies
- Fidélisation des visiteurs d’un site “social” par “animation de communauté”

# Réseaux sociaux: l'expérience de Milgram

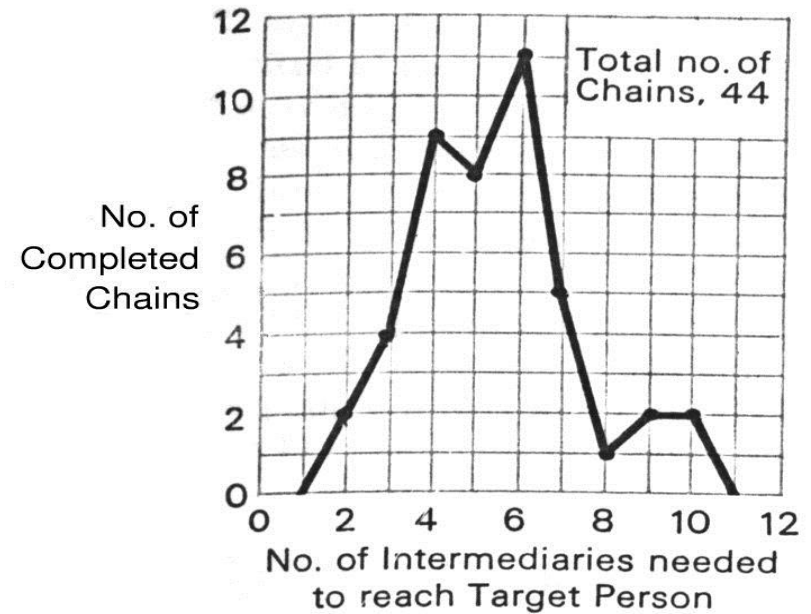
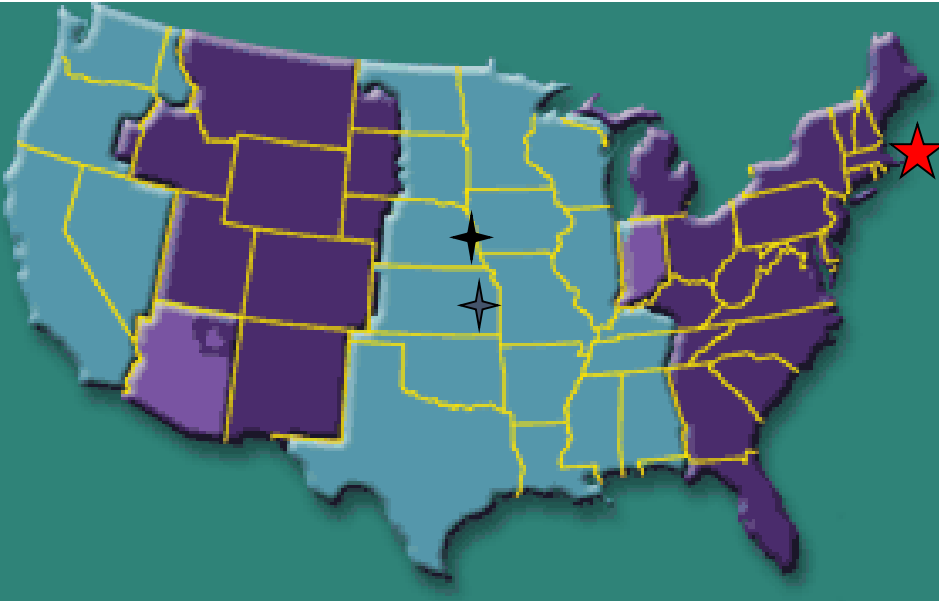


Milgram, *Psych Today* **2**, 60 (1967)

Dodds et al., *Science* **301**, 827 (2003)



# Réseaux sociaux: l'expérience de Milgram



**In the Nebraska Study the chains varied from two to 10 intermediate acquaintances with the median at five.**

**“Six degrés de séparation”:  
“petit-monde”**

# Etude de la propriété des graphes: le « Bacon number »

<https://oracleofbacon.org/>

- Sommets = acteurs
- Arêtes entre acteurs ayant joué dans un même film...



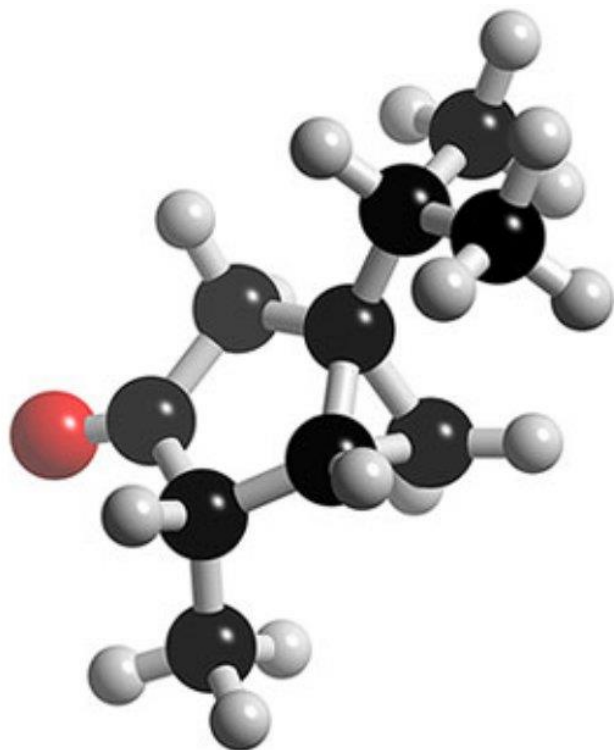
**Propriété :**  
**Tout acteur est à distance**  
**au plus 6 de Kevin**  
**Bacon !...**

Angelina Jolie Pitt (2) → Lara Croft Tomb Raider: The Cradle of Life (2003) avec Djimon Hounsou → Beauty Shop (2005) avec Kevin Bacon

Emma Stone (1) → Crazy, Stupid, Love. (2011) with Kevin Bacon

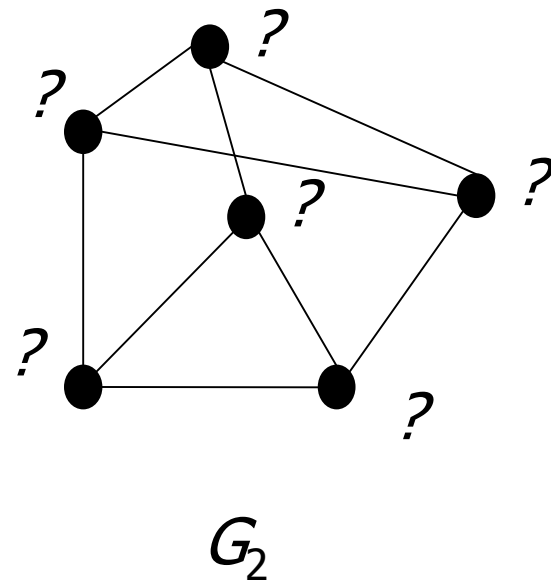
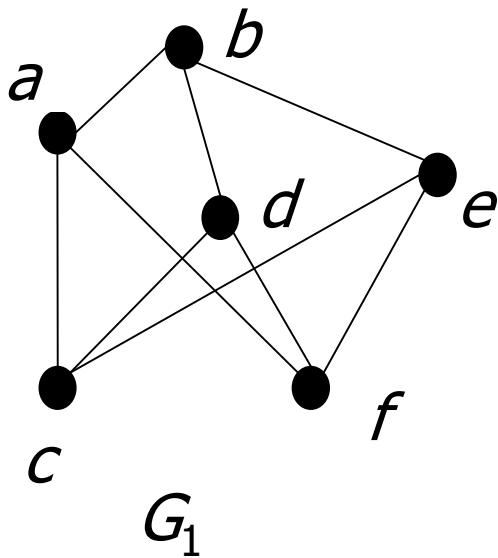


# Découvertes pharmaceutiques

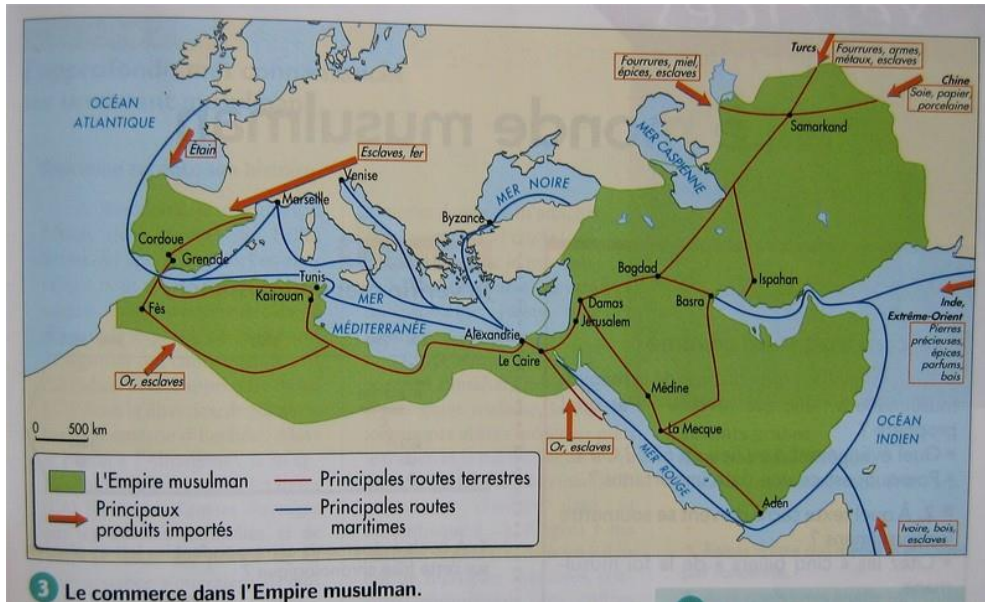


Chercher des sous parties de  
molécules communes dans  
certains médicaments  
(Isomorphismes de sous-graphes)

# Isomorphes ?

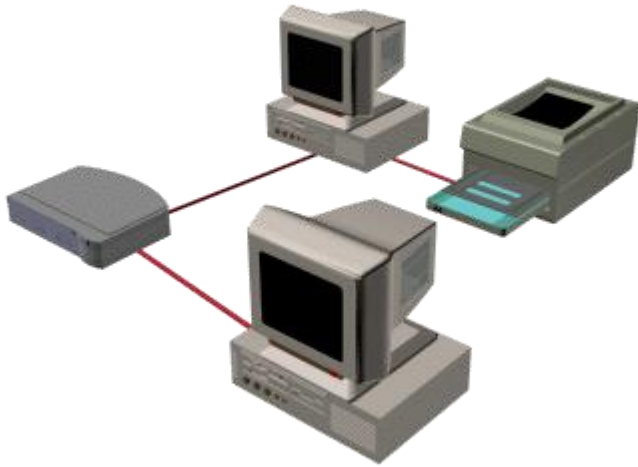


# Logistique



Plannings, ordonnancements... (des algorithmes que vous verrez beaucoup plus tard)

# Réseaux



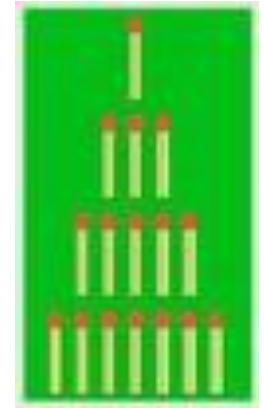
## Routage dans les réseaux

réseaux classiques,  
machines parallèles  
(communications entre  
processeurs), réseaux  
optiques, etc.

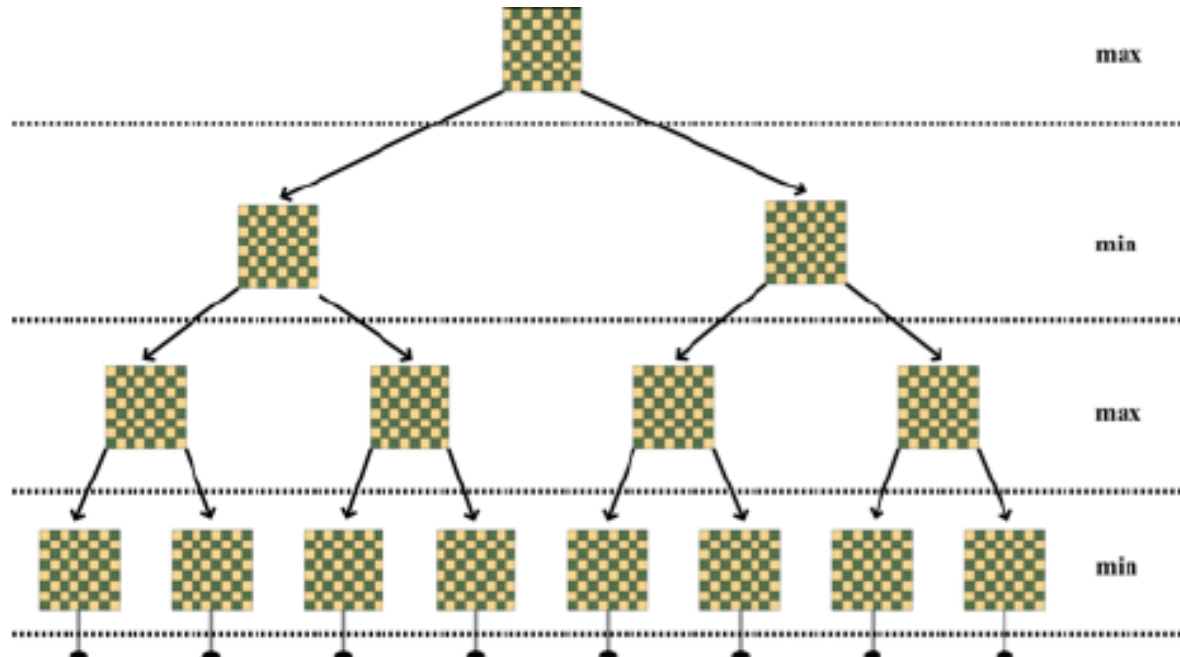
# Jeux

- Catégories des Jeux déterminés

Ex : jeux de Nim (chacun son tour tire le nombre d'allumettes qu'il désire dans le tas qu'il désire, le but étant de prendre la dernière allumette pour gagner)



## ■.... ou, plus généralement, jeux à deux joueurs





## Un autre exemple introductif

- Une famille désire traverser un ravin par un pont qui ne peut transporter que deux personnes à la fois. Et en plus il faut avoir la seule torche pour ne pas tomber dans le ravin. Chaque membre de la famille prend un temps pour traverser :
  - le père, 1 mn
  - la mère, 2 mn
  - le fils, 5 mn
  - la fille, 10 mn
- Quelle est la meilleure façon de faire traverser toute la famille ?
- Par exemple :
  - Le père et le fils partent, le père revient (5+1)
  - La mère et la fille partent, la mère revient (10+2)
  - Le père et la mère rejoignent les enfants ( 2)
- Total 20 minutes. Pouvez-vous faire mieux ? Et ensuite prouver que votre solution est optimale ?

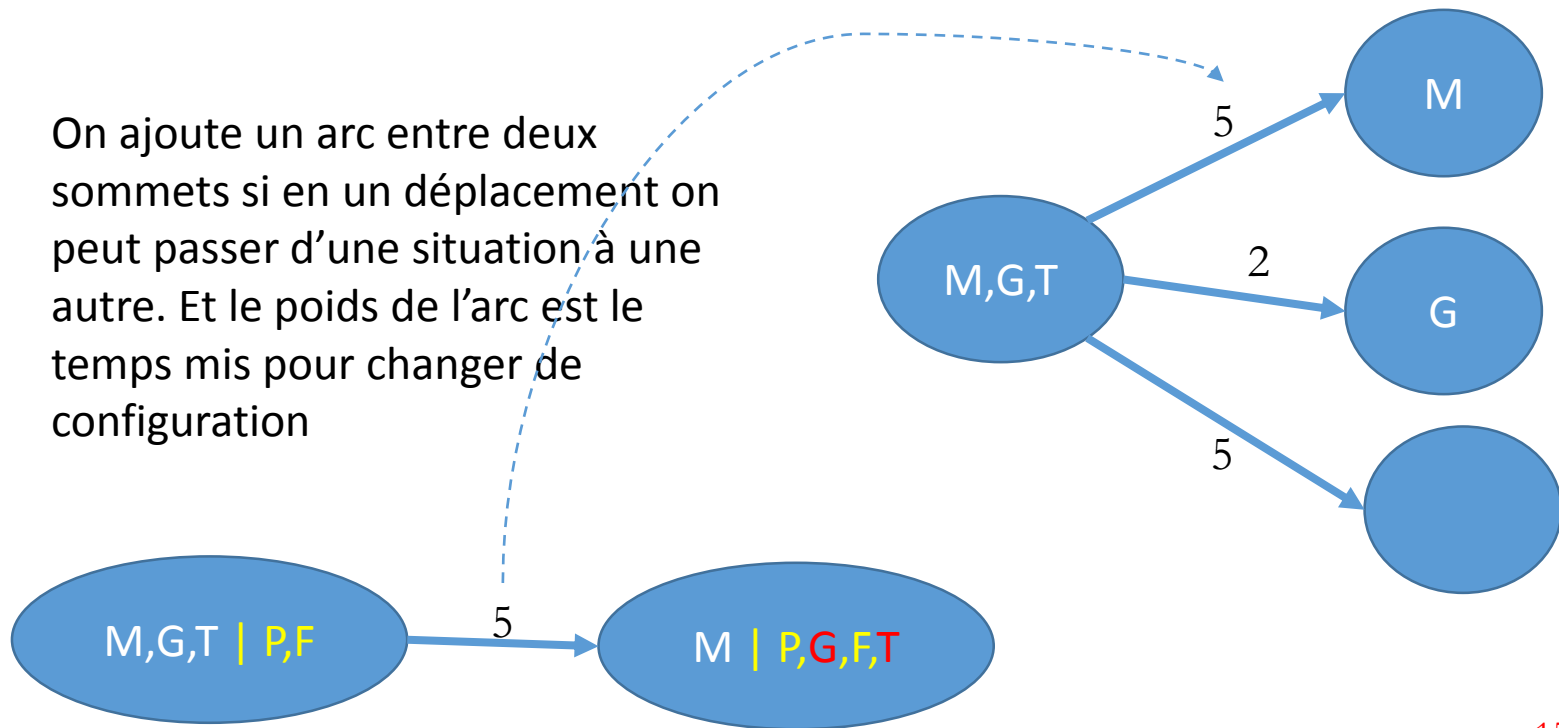


# Une modélisation par les graphes

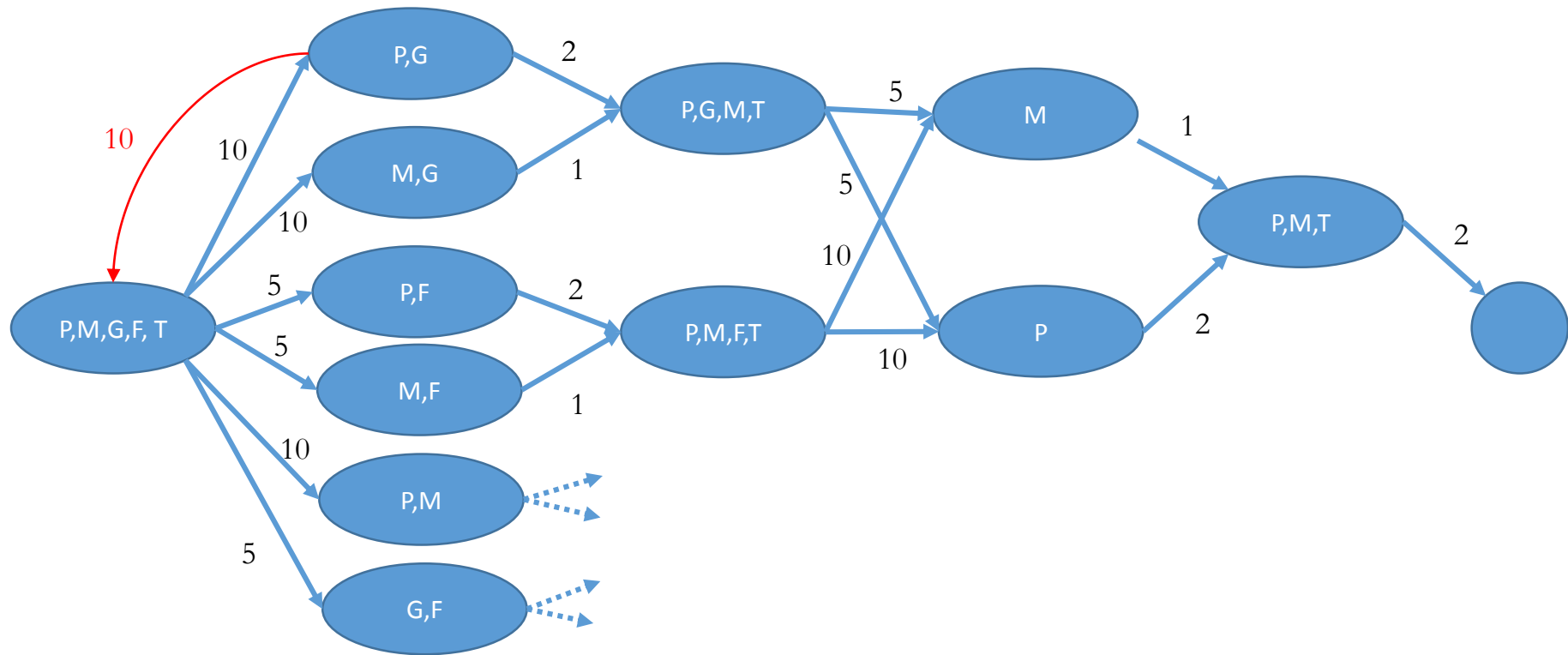
On représente la situation sur la rive de départ (P=père, M=mère, G=garçon, F=filles, T= torche) comme le sommet d'un graphe



On ajoute un arc entre deux sommets si en un déplacement on peut passer d'une situation à une autre. Et le poids de l'arc est le temps mis pour changer de configuration

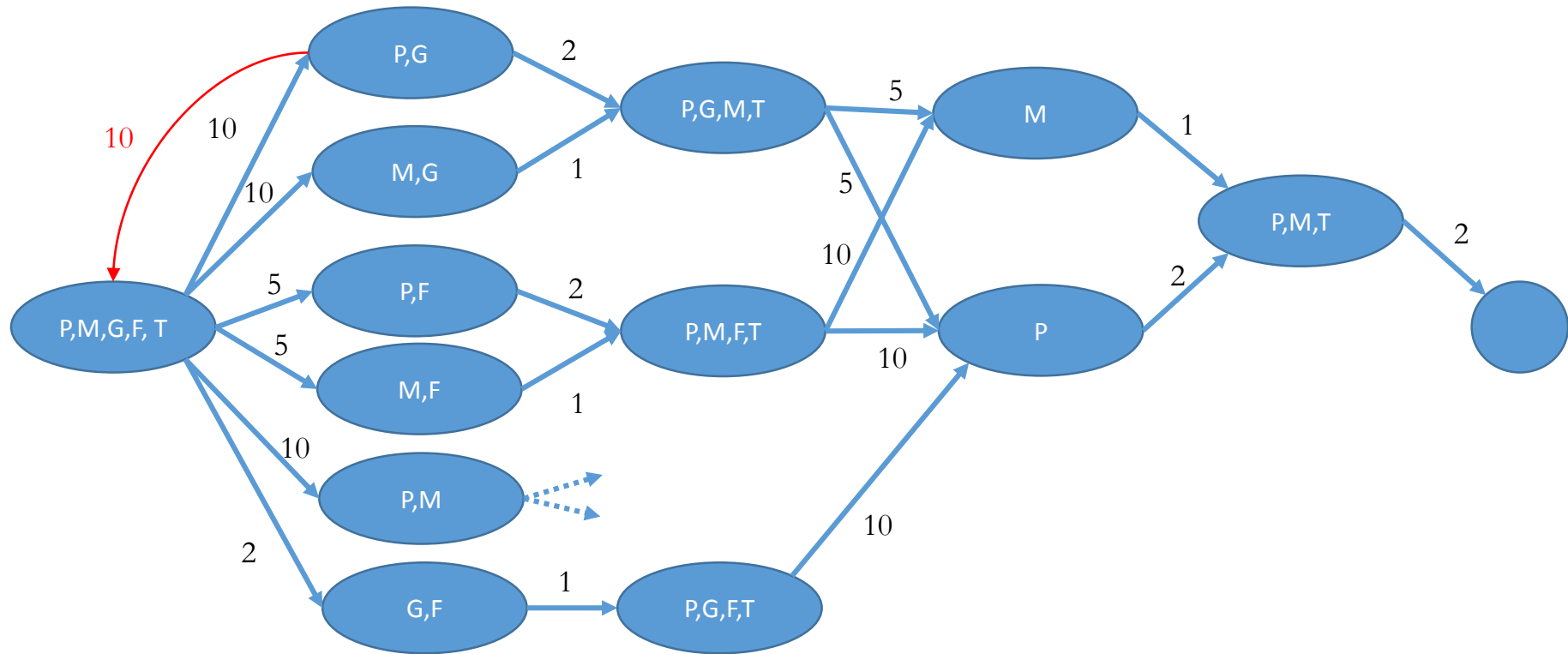


Une partie du graphe (c'est donc un sous-graphe partiel)





# Un sous graphe partiel un peu plus complet



# En quoi avons-nous prouvé que la solution à 17 est optimale ?

- Nous pouvons construire le graphe entier et donc représenter exhaustivement toutes les solutions
- On a donc gagné mais....
- Il n'y a rien qui vous a choqué ?

# Un exemple introductif

- Une famille désire traverser un ravin par un pont qui ne peut transporter que deux personnes à la fois. Et en plus il faut la seule torche pour ne pas tomber. Chaque membre de la famille prend un temps pour traverser :
  - le père, 1 mn
  - la mère, 2 mn
  - le fils, 5 mn
  - la fille, 10 mn
- ...

# Un exemple introductif

- Une famille désire traverser un ravin par un pont qui ne peut transporter que deux personnes à la fois. Et en plus il faut la seule torche pour ne pas tomber. Chaque membre de la famille prend un temps pour traverser :

- le père, 1 mn
- la mère, 2 mn
- le fils, 5 mn
- la fille, 10 mn

- ...



Quels  
boulets,  
ces filles !

# Parenthèse ODD 5

- L'informatique, en modélisant le monde, peut transmettre les biais et les préjugés. C'est aussi un bel endroit pour les mettre en avant et, les corriger
- Pourquoi pas...
  - le père, 2 mn
  - la mère, 1 mn
  - le fils, 10 mn
  - la fille, 5 mn



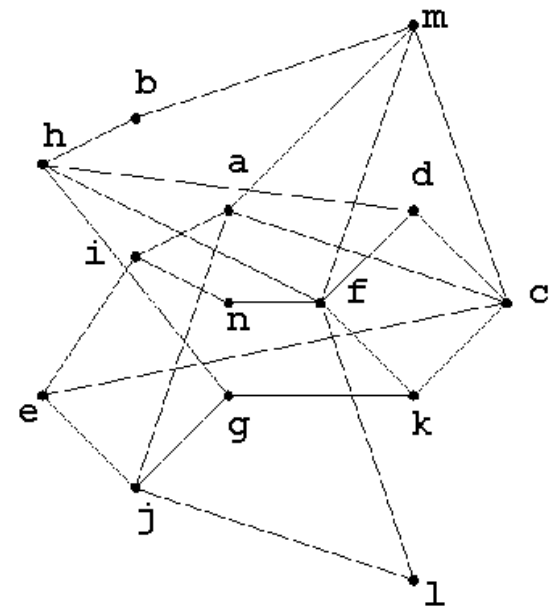
Rappel : il y a 17 objectifs de développement durable proposés par l'ONU



# Graphes non orientés (1)

(Graphe) Un graphe simple non orienté  $G$  est un couple formé de **deux ensembles** : un ensemble  $X = \{x_1, x_2, \dots, x_n\}$  dont les éléments sont appelés **sommets**, et un ensemble  $A = \{a_1, a_2, \dots, a_m\}$ , partie de l'ensemble  $P^2(X)$  des parties à deux éléments de  $X$ , dont les éléments sont appelés **arêtes**. On notera  $G = (X, A)$ .

Lorsque  $a = (x, y) \in A$ , on dit que  $a$  est l'arête de  $G$  **d'extrémité**  $x$  et  $y$ , ou que  $a$  joint  $x$  et  $y$ , ou que  $a$  passe par  $x$  et  $y$ .

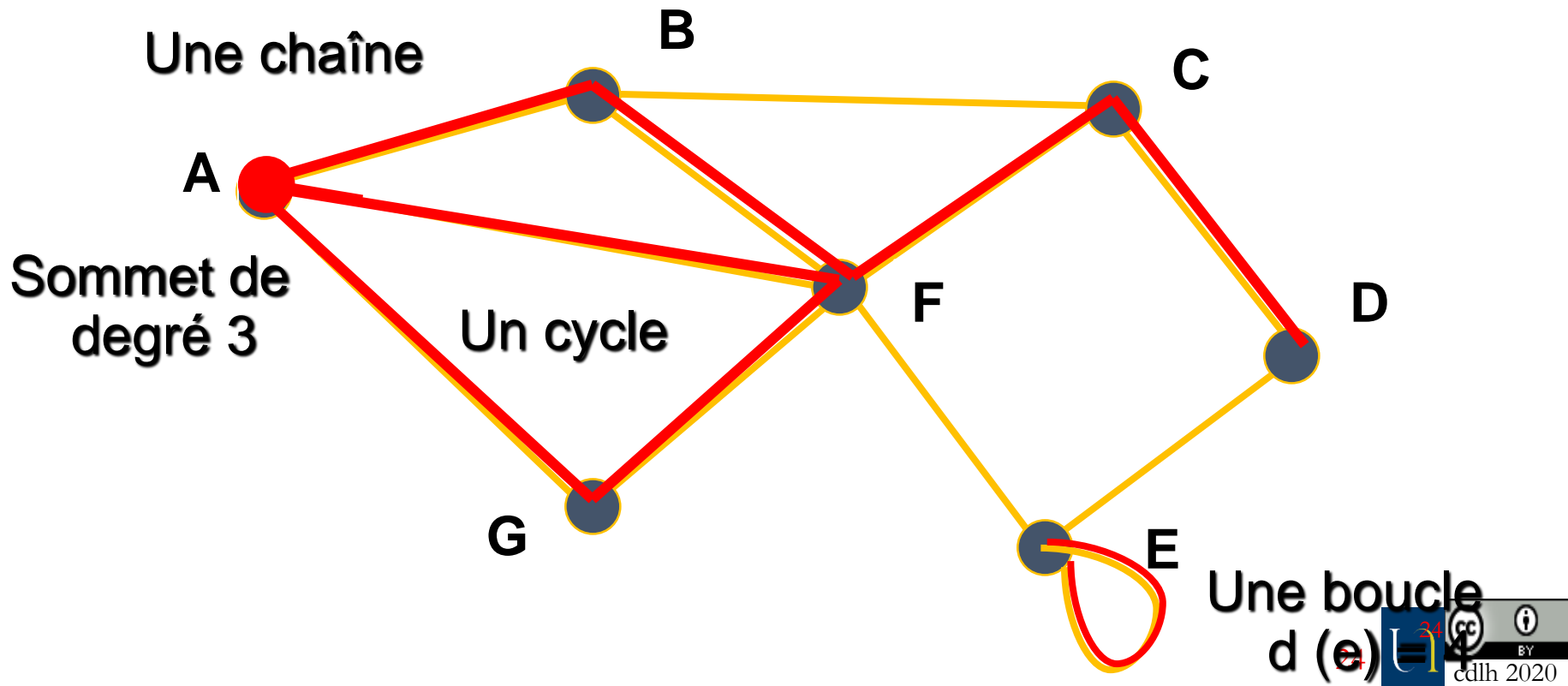


# Graphes non orientés (2)

- Sommets adjacents : deux sommets sont adjacents s'ils sont reliés entre eux par une arête. On dit que l'arête est **incidente** aux deux sommets.
- Degré d'un sommet  $x$  ( $d(x)$ ) : nombre d'arêtes incidentes au sommet.
- Chaîne : séquence ordonnée d'arêtes telle que chaque *arête* a une extrémité en commun avec l'arête suivante.
- Cycle : dans un graphe non orienté, un cycle est une suite d'arêtes consécutives (**chaîne**) dont les deux sommets extrémités sont identiques.
- Boucle : il peut exister des arêtes entre un sommet  $x$  et lui-même. Elles sont appelés « boucles ».

# Exemple

- Un ensemble de sommets  $\{A, B, C, D, E, F, G\}$
- Un ensemble d'arêtes  $\{(A, B), (B, C), \dots\}$





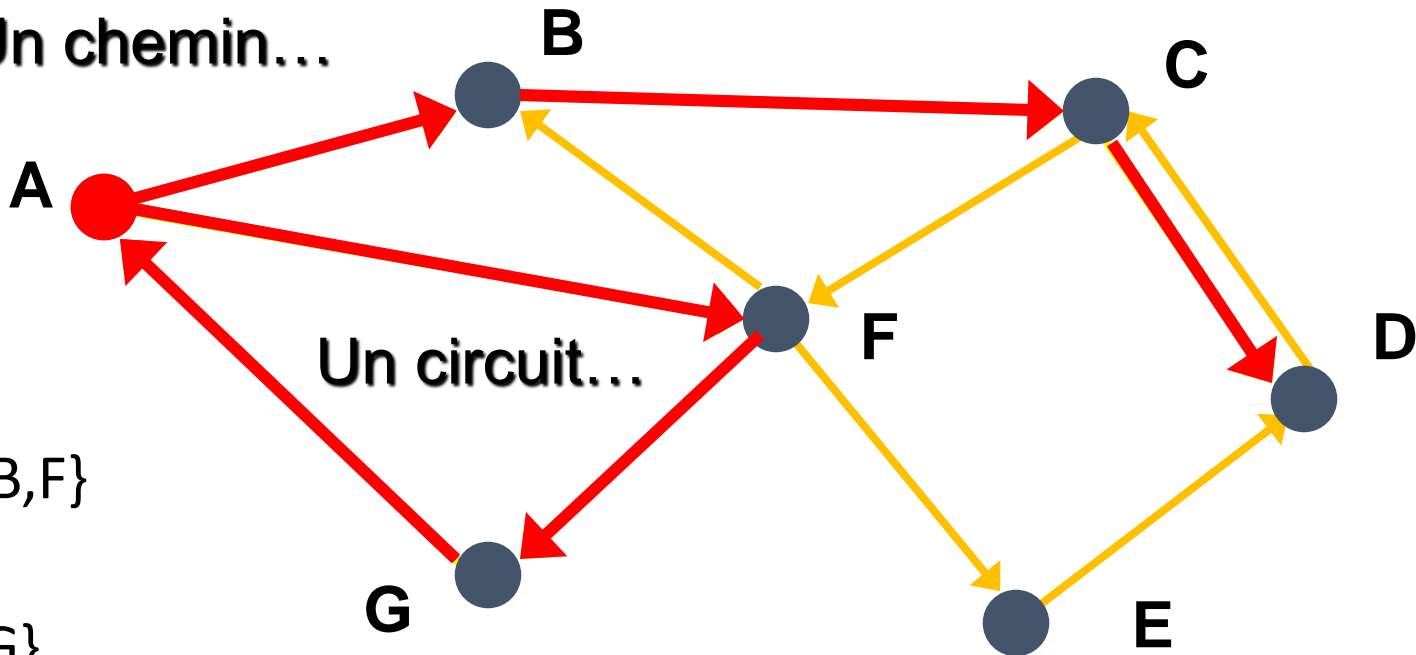
# Graphes orientés

- Un graphe est orienté si les arêtes ont un sens. Elles sont alors appelées **arcs**. Par exemple  $(x, y) \in A$  indique qu'il y a un arc d'origine  $x$  et d'extrémité finale  $y$ .
- (Successeurs et prédécesseurs d'un sommet  $x$ ) Dans un graphe orienté  $G = (X, A)$ , on appelle l'ensemble des sommets successeurs d'un sommet  $x$ , (noté  $\Gamma_+(x)$ ), l'ensemble des sommets  $y$  tels que  $(x, y) \in A$ . De la même façon, on appelle l'ensemble des sommets prédécesseurs d'un sommet  $x$ , (noté  $\Gamma_-(x)$ ), l'ensemble des sommets  $w$  tels que  $(w, x) \in A$ .
- (Circuit) Dans un graphe orienté, un circuit est une suite d'arcs consécutifs (**chemin**) dont les deux sommets extrémités sont identiques.
- Les notions de **degré** et **boucle** existent aussi dans le cas de graphes orientés. Le **degré entrant** ( $d_-$ ) d'un sommet  $x$  est  $\text{card}(\Gamma_-(x))$ . Le **degré sortant**,  $d_+$  d'un sommet  $x$  est  $\text{card}(\Gamma_+(x))$ . Le **degré**  $d$  d'un sommet  $x = d_+ + d_-$ .

# Graphes orientés

- Un ensemble de sommets
- Un ensemble d'arcs

Un chemin...



Un circuit...

Degré 3

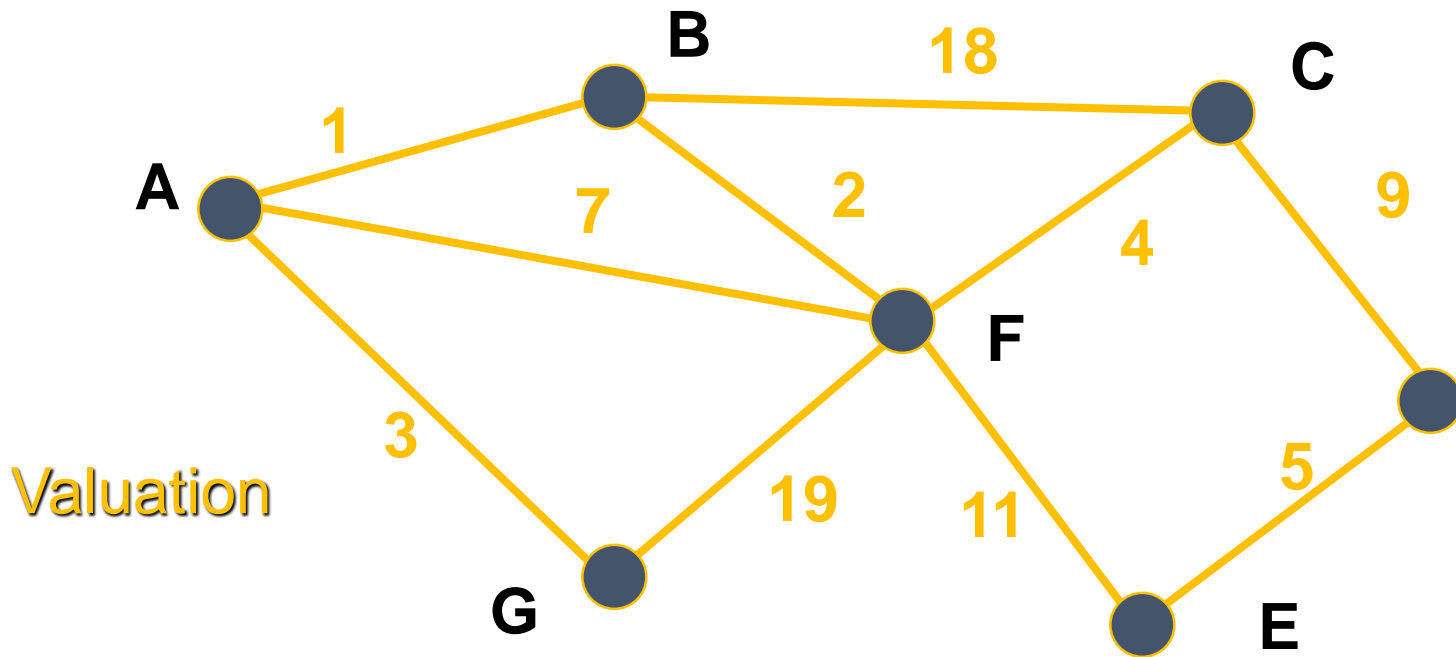
$$\Gamma_+(A) = \{B, F\}$$

$$d_+(A) = 2$$

$$\Gamma_-(A) = \{G\}$$

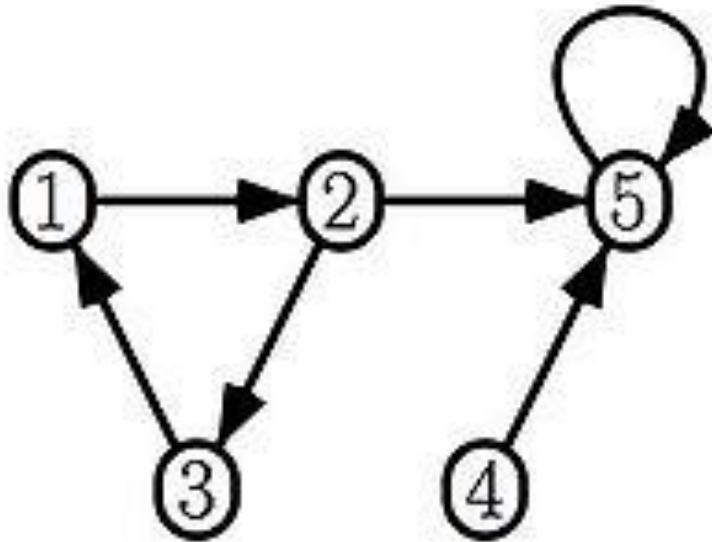
$$d_-(A) = 1$$

# Graphe valué



- On associe un coût (valuation) à toutes les arêtes/arcs du graphe

# Exercice 1



Sommets ?

Arcs ?

Circuits ?

Boucles ?

$\Gamma_+(2)$  ?

$\Gamma_-(5)$  ?

$\Gamma_-(4)$  ?

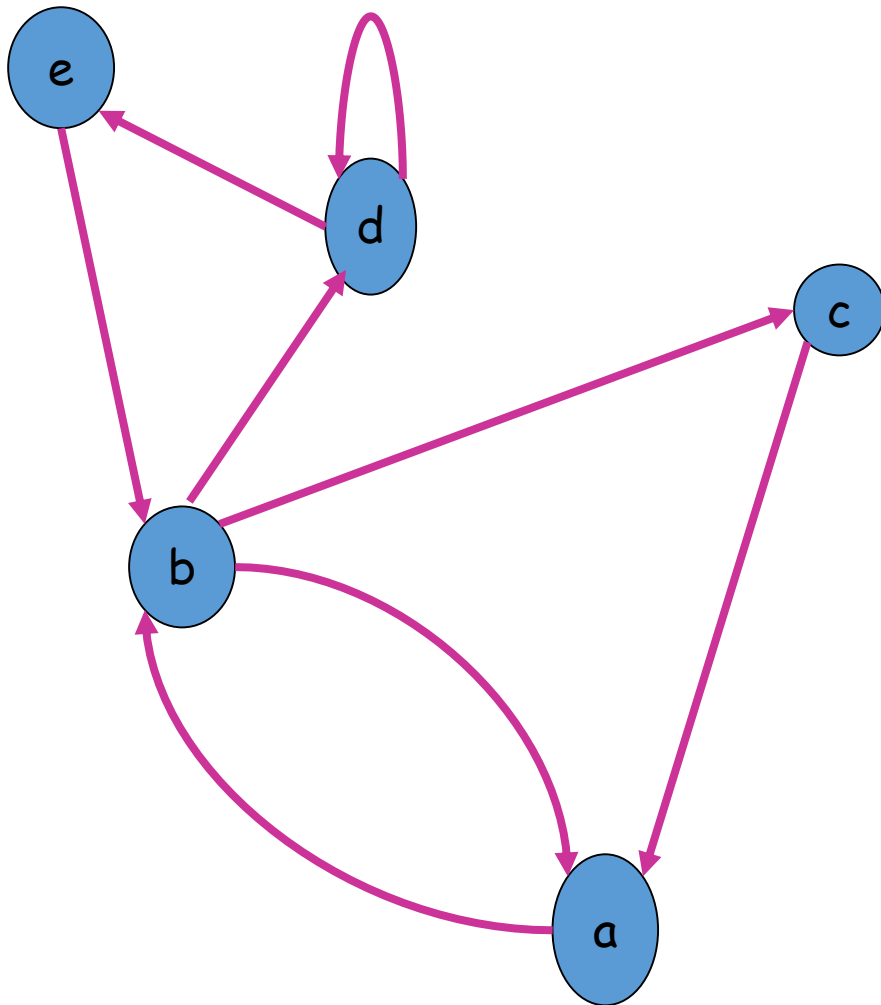
Matrice d'adjacence ?

# Une représentation possible : la **matrice d'adjacence**

- On représente un graphe (orienté ou non) par un tableau (matrice) :
  - Lignes (d'indice  $i$ ) : tous les sommets
  - Colonnes (d'indice  $j$ ) : tous les sommets
  - $\text{case}(i,j)$  : 1 (ou la valeur de l'arête  $(i,j)$ ) s'il existe un arc (ou une arête) entre le sommet  $i$  et le sommet  $j$  dans le graphe ; 0 sinon.

Remarque: dans le **cas non orienté**, le tableau est **symétrique**  
 $\text{case}(i,j) = \text{case}(j,i)$

# Graphe et matrice d'adjacence

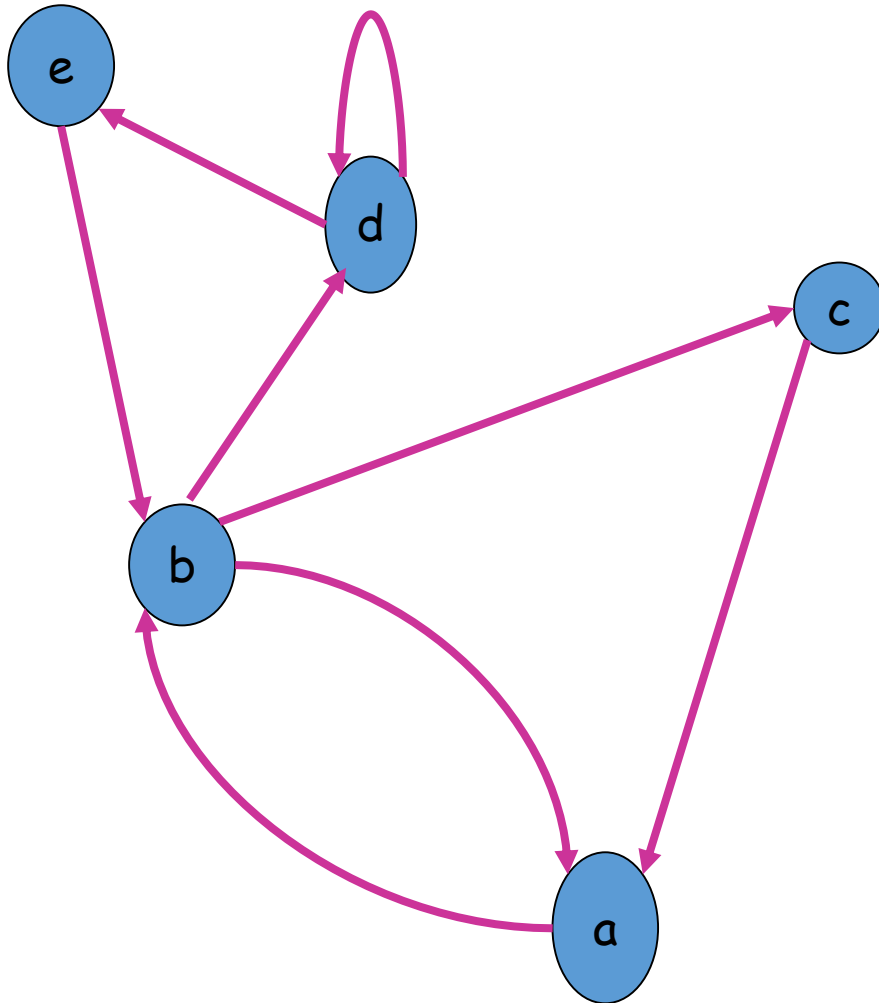


	a	b	c	d	e
a	0	1	0	0	0
b	1	0	1	1	0
c	1	0	0	0	0
d	0	0	0	1	1
e	0	1	0	0	0

# Une seconde représentation possible : listes d'adjacence

- On représente un graphe (orienté ou non) par une **liste de listes** :
  - Une liste de sommets
  - Et pour chaque sommet une liste de successeurs
- Remarque : dans Python on peut avantageusement utiliser un dictionnaire. Et certains utilisent le type ensemble pour la liste des successeurs

# Graphe et liste d'adjacence dans Python



```
{"a":["b"],  
"b":["a", "c", "d"],  
"c":["a"],  
"d":["d", "e"],  
"e":["b"]}
```



# Avec un dictionnaire

```
Entrée [19]: mongraphe = { "a" : ["b", "c"],  
                           "b" : ["c", "e"],  
                           "c" : ["b", "d", "e"],  
                           "d" : ["e"],  
                           "e" : [],  
                           "f" : []  
                           }
```

```
Entrée [20]: for voisin in mongraphe["c"]:  
              print(voisin)
```

b  
d  
e

```
Entrée [21]: def ajouterSommet(graphe, u):  
              graphe[u] = []
```

```
Entrée [22]: ajouterSommet(mongraphe, "g")
```

```
Entrée [23]: def ajouterArc(graphe, u, v):  
              graphe[u].append(v)
```

```
Entrée [24]: ajouterArc(mongraphe, "f", "g")
```

```
Entrée [25]: mongraphe
```

```
Out[25]: {'a': ['b', 'c'],  
          'b': ['c', 'e'],  
          'c': ['b', 'd', 'e'],  
          'd': ['e'],  
          'e': [],  
          'f': ['g'],  
          'g': []}
```

```
Entrée [26]: import random
             random.choice(mongraphe["c"])
```

Out[26]: 'e'

```
Entrée [27]: def randomWalk(graphe,u,n):
             if n==0:
                 w=u
             elif graphe[u]==[]:
                 w=u
             else:
                 print(u)
                 w=randomWalk(graphe,random.choice(mongraphe[u]),n-1)
             return(w)
```

```
Entrée [34]: randomWalk(mongraphe,"a",5)
```

a  
c  
d

Out[34]: 'e'



## Exercices 2

1. Quel est le nombre maximum d'arcs d'un graphe simple orienté à  $n$  sommets avec boucles ?
2. ....d'un graphe simple orienté sans boucle ?
3. Quel est le nombre maximum d'arêtes d'un graphe simple non orienté avec boucles ?
4. ...d'un graphe simple non orienté sans boucle ?
5. Combien peut on construire de graphes simples orientés différents à  $n$  sommets ?
6. ... à  $n$  sommets et  $m$  arcs ?

# Le cas particulier des arbres

Soit  $G = (X, A)$  un graphe (non) orienté tel que  $|X| = n$ .  $G$  est **un arbre si et seulement si** :

- $G$  est connexe et  $G$  est sans cycle.
- $G$  est sans cycle avec  $n-1$  arêtes
- $\forall x, y \in X$ , il existe une chaîne unique (ne passant pas deux fois par le même sommet) joignant  $x$  à  $y$

(toutes ces définitions sont équivalentes)



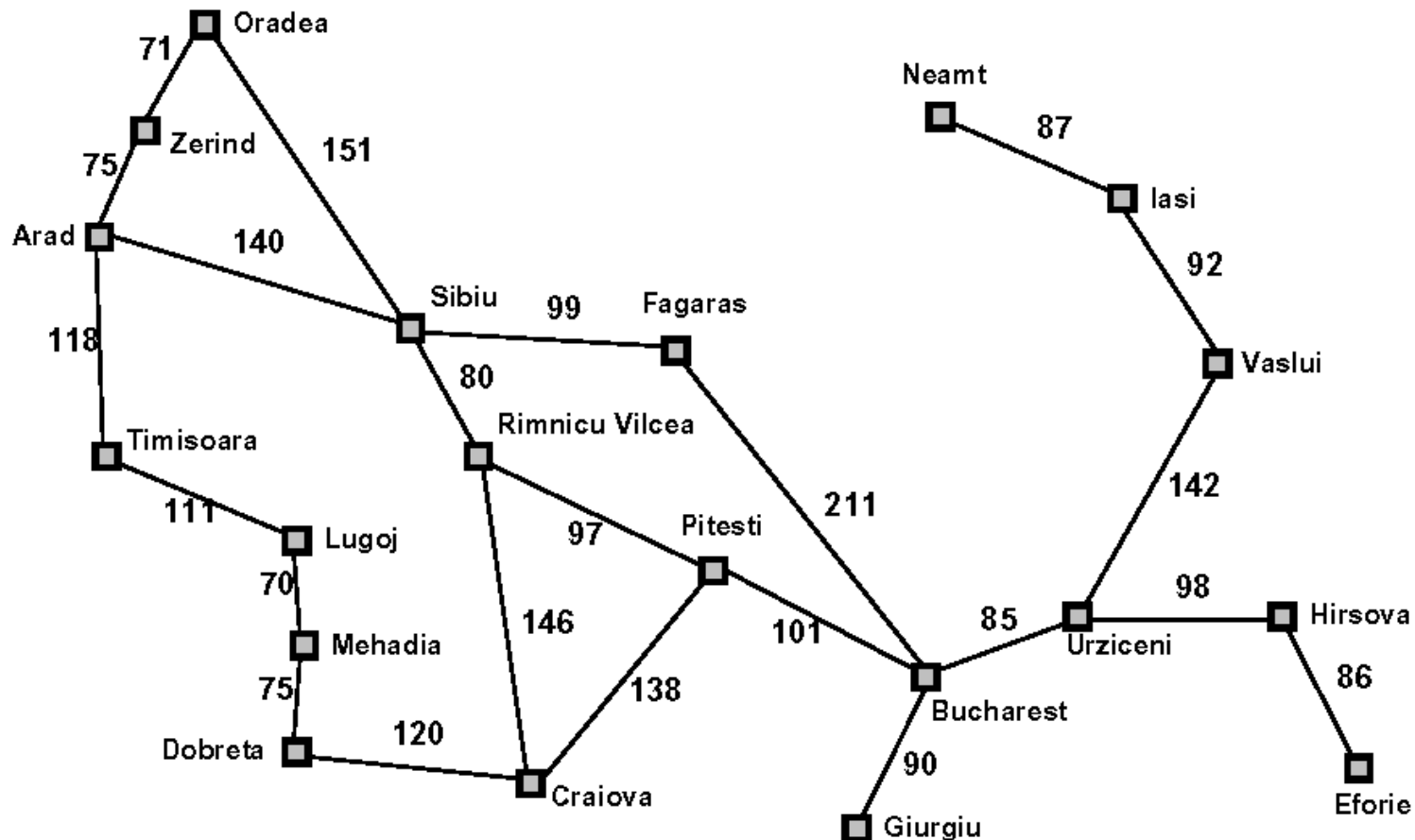
# Quelques algorithmes sur les graphes

# Explorer/parcourir un graphe

- Nous avons vu un premier algorithme : le *random walk*. Utilisé en pratique sur des grands graphes pour bénéficier de propriétés statistiques.
- Mais si on a un graphe qu'on veut explorer systématiquement, par exemple pour connaître la liste des sommets ?

# Exemple introductif : on part d'Arad

## Voyage en Roumanie (d'après S. Russel et P. Norvig)



# Difficultés

- Ne pas tourner en rond

→ penser au petit poucet : marquer les sommets par lesquels on est passé

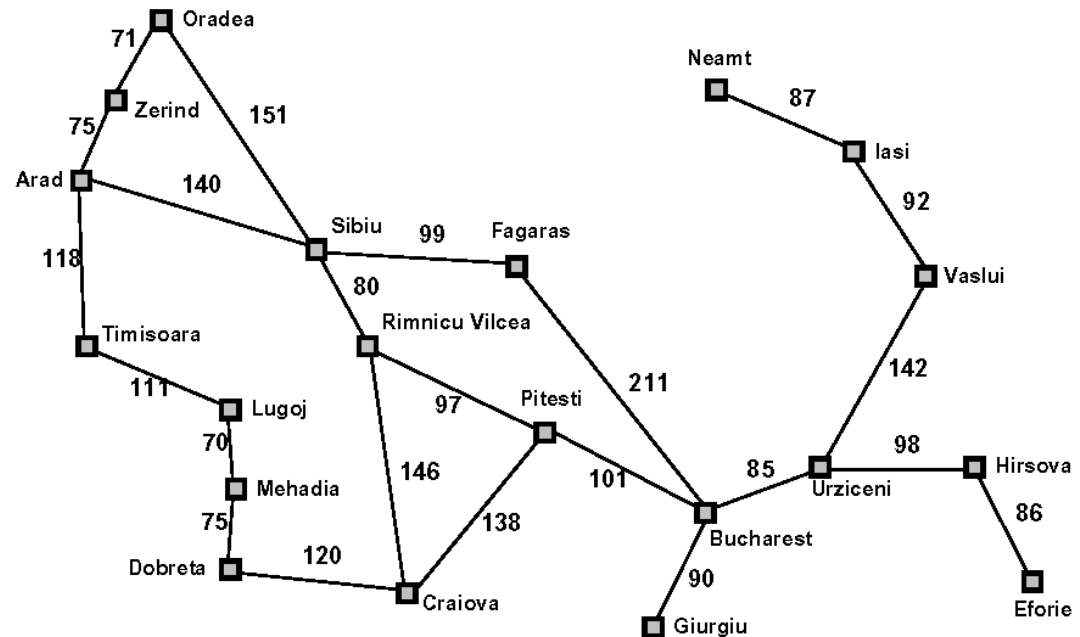
- Ne pas oublier de sommets

→ ça nous oblige à utiliser tous les arcs (ou ici toutes les arêtes)



# Exemple introductif : on part d'Arad

- Donc on explore Arad (on marque Arad), et on a le choix entre Zerind, Sibiu et Timisoara.
- Idée : on stocke Zerind, Sibiu et Timisoara, puis on prend la 1<sup>ère</sup> (Zerind) et on explore. La question clé est « après Zerind on va à Oradea ou à Sibiu ? »



# Algorithme 1 : recherche en profondeur d'abord

(vous avez choisi Oradea)

- On va donc stocker dans une pile les gares découvertes en suivant les arêtes.
- Pile= Last in First out
- Le nom traditionnel de l'algorithme est **DFS : Depth-First Search**

# Algorithme DFS

```
fonction dfs(graphe, debut)
Variables Dictionnaire visite , Pile p
    visite ← {}
    p ← [debut]
    Tant que taille(p) > 0 faire
        noeud ← sommet(p)
        dépiler(p)
        si non estDans(visite, noeud)
            alors
                associer(visite, noeud, true)
                pour chaque voisin de noeud faire
                    si non estDans(visite, voisin)
                        alors
                            empiler(p, voisin)
                finsi
            finpour
        finsi
    fintantque
    retourner visite
```



# Quelques éléments de discussion

- `Visite` est un dictionnaire. Avec `estDans` on peut savoir si un élément a été ajouté au dictionnaire. Mais on n'a pas vraiment besoin de la valeur (`true`). Toute valeur aurait marché. On aurait donc pu prendre une SAD plus simple : l'ensemble (cf 2<sup>nd</sup> regroupement)
- Si on applique l'algorithme sur la Roumanie, ça marche. Si on l'applique sur la France (en partant de Nantes) ça ne marche pas.
- A cause de la Corse (et la Martinique, Guadeloupe...). Il faut des propriétés de connexité : que toute gare soit accessible à partir de Nantes.
- Une façon de faire s'il n'y a pas de connexité est de supposer qu'on connaît tous les sommets (c'est souvent le cas) et dans ce cas on initialise le dictionnaire `visite` à `false` pour tous les sommets.
  - Modifiez l'algorithme pour cela
  - Expliquer à quoi ça sert d'explorer si on connaît déjà tous les sommets

# Complexité

- On suppose qu'on a  $n$  sommets et  $m$  arcs.
- Chaque arc n'est visité qu'une fois quand le sommet origine est marqué.
- Le nombre d'éléments empilés est au plus  $m$ .
- Donc, malgré les deux boucles imbriquées, la complexité est en  $O(n+m)$
- En fait, c'est plus subtil que ça :  $O(m)$  suffit. On a l'habitude d'écrire  $O(n+m)$  pour gérer les deux « pires » cas :
  - Celui où  $m=n^2$
  - Celui où il n'y a que très peu d'arcs et où il faut quand même parcourir tous les sommets pour s'en rendre compte: or dans le **dfs** proposé ce ne sera pas nécessaire

# En Python

```
Entrée [30]: def dfs(graphe, debut):
               visite, pile = {}, [debut]
               while pile:
                   noeud = pile.pop()
                   if noeud not in visite:
                       visite[noeud] = True
                       print(noeud)
                       for voisin in graphe[noeud]:
                           if voisin not in visite:
                               pile.append(voisin)
               return visite
```

```
Entrée [31]: dfs(mongraphe, "a")
```

```
a
c
e
d
b
```

```
Out[31]: {'a': True, 'c': True, 'e': True, 'd': True, 'b': True}
```

# Algorithme 2 : recherche en largeur d'abord

- (vous avez choisi Sibiu)
- On va donc stocker dans une file les gares découvertes en suivant les arêtes.
- File= First in First out
- Le nom traditionnel de l'algorithme est **BFS : Breadth-First Search**

# Algorithme BFS

```
fonction bfs(graphe, debut)
Variables Dictionnaire visite, File f
    visite ← {}
    f ← >debut>
    Tant que taille(f) > 0 faire
        noeud ← premier(f)
        défiler(f)
        si non estDans(visite, noeud)
            alors
                ecrire(noeud)
                associer(visite, noeud, true)
                pour chaque voisin de noeud faire
                    si non estDans(visite, voisin)
                        alors
                            enfiler(f, voisin)
                finsi
            finpour
        finsi
    fintantque
    retourner visite
```



```
Entrée [32]: def bfs(graphe, debut):
               visite, file = {}, [debut]
               while file:
                   noeud = file.pop(0)
                   if noeud not in visite:
                       visite[noeud] = True
                       print(noeud)
                       for voisin in graphe[noeud]:
                           if voisin not in visite:
                               file.append(voisin)
               return visite
```

```
Entrée [33]: bfs(mongraphe, "a")
```

```
a
b
c
e
d
```

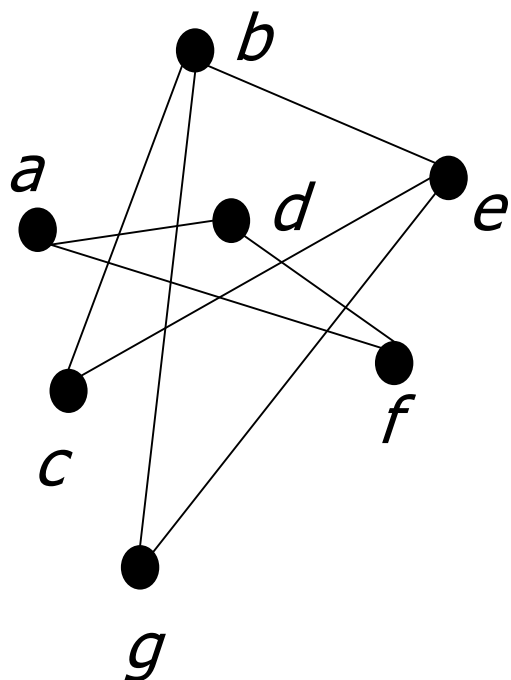
```
Out[33]: {'a': True, 'b': True, 'c': True, 'e': True, 'd': True}
```

# Correction et complexité

- Même analyse que pour DFS. Ce sont d'ailleurs les mêmes sommets qu'on obtient. Seul l'ordre va changer.
- Comme pour dfs, on peut ne pas tester `si non estDans(visite, voisin)` et obtenir la même complexité

# Connexité

# Cas des graphes non orientés



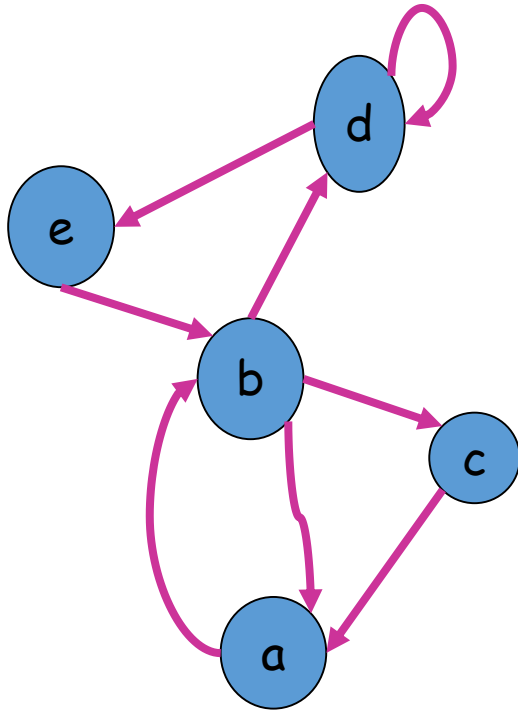
Ce graphe n'est pas connexe

# Définitions

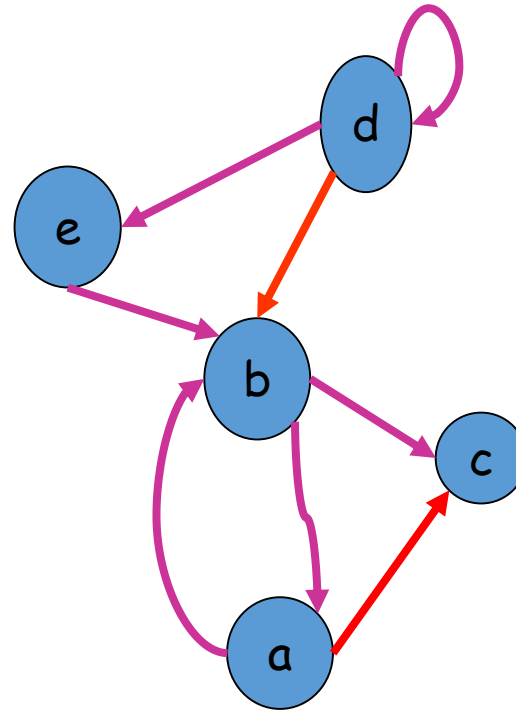
- Un graphe non orienté est connexe si entre deux sommets quelconques il existe une chaîne
- Un graphe orienté est connexe si entre deux sommets quelconques il existe une chaîne
- Un graphe orienté est fortement connexe si entre deux sommets quelconques il existe un chemin

(important : le vocabulaire, quand on parle de graphes, doit rester très précis)

# Cas des graphes orientés



Un graphe  
connexe et  
fortement connexe



Un graphe connexe  
mais non fortement  
connexe. Il possède 4  
composantes fortement  
connexes

# Théorème

- Un graphe  $G$  (non orienté) est connexe si il existe un sommet **début** tel que  $\text{dfs}(G, \text{début})$  visite tous les sommets.
- Un graphe  $G$  (non orienté) n'est pas connexe si il existe un sommet **début** tel que  $\text{dfs}(G, \text{début})$  ne visite pas tous les sommets
- On peut donc tester la connexité en  $O(n+m)$ ... car il suffit de tester pour un seul sommet **début**.
- On peut utiliser ce théorème pour régler le cas des graphes orientés
- Pour la forte connexité c'est un peu plus compliqué

# L'algorithme de Kosaraju (1)

## Théorème

- Un graphe est fortement connexe si et seulement si on peut explorer tout le graphe à partir de chaque sommet.
- Cela signifie que l'algorithme naïf Pour  $x$  dans nœuds faire  $\text{dfs}(G, x) \dots$  va fonctionner en  $O(n(n+m))$



# L'algorithme de Kosaraju (2)

- Mais on peut faire mieux avec l'algorithme de Kosaraju (qui n'est pas dans le programme) mais est très élégant et simple
  - Initialiser tous les sommets comme non visités ( $O(n)$ )
  - Choisir un sommet  $x$
  - Calculer  $\text{dfs}(G, x)$ . Si certains sommets ne sont pas visités retourner faux ( $O(n+m)$ )
  - Inverser le sens de tous les arcs de  $G$  ( $O(m)$ )
  - Réinitialiser tous les sommets comme non visités ( $O(n)$ )
  - Calculer  $\text{dfs}(G, x)$ . Si certains sommets ne sont pas visités retourner faux ( $O(n+m)$ )
  - Si (et seulement si) les deux dfs ont marqué tous les sommets, alors le graphe est fortement connexe
- Et la complexité est  $O(n+m)$ . En plus, on peut adapter pour obtenir toutes les cfc

(cfc= composante fortement connexe)

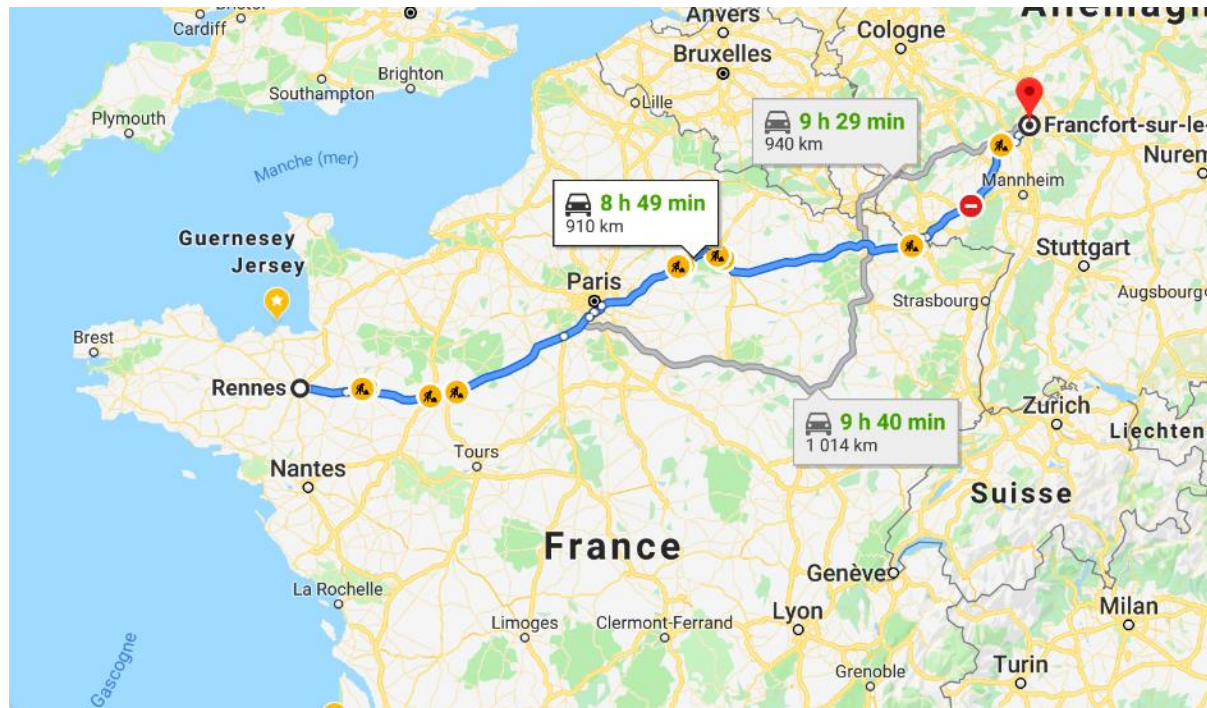
# Correction et Complexité

- On dira que  $y$  est **accessible** à partir de  $x$  s'il existe un chemin de  $x$  à  $y$ .
- On dira que  $y$  est **co-accessible** à partir de  $x$  s'il existe un chemin de  $y$  à  $x$ .
- Supposons que l'algorithme retourne « vrai ».
- Soient  $y$  et  $z$  deux nœuds de  $G$ . Comme  $y$  est co-accessible à partir de  $x$  il existe un chemin de  $y$  à  $x$ . Et comme  $z$  est accessible à partir de  $x$  il existe un chemin de  $x$  à  $z$ . Et donc un chemin (passant par  $x$ ) de  $y$  à  $z$ .
- On raisonne de façon similaire pour prouver l'existence d'un chemin de  $z$  à  $y$ .
- Conclusion :  $G$  est fortement connexe
- Supposons que l'algorithme retourne faux. C'est alors parce qu'un des dfs a échoué. Et dans ce cas, le graphe n'est pas fortement connexe.
- La complexité : 2 fois celle de dfs; donc en  $O(m+n)$

Plus court chemin

# Plus court chemin

Déterminer le chemin de coût minimum reliant un nœud **a** à un nœud **b**.





## Exercice

- Modifiez dfs pour qu'il retourne un chemin entre deux sommets  $x$  et  $y$
- Aurez-vous le plus court chemin ?
- Notez qu'on peut définir « plus court chemin » de deux façons différentes. Avis ?

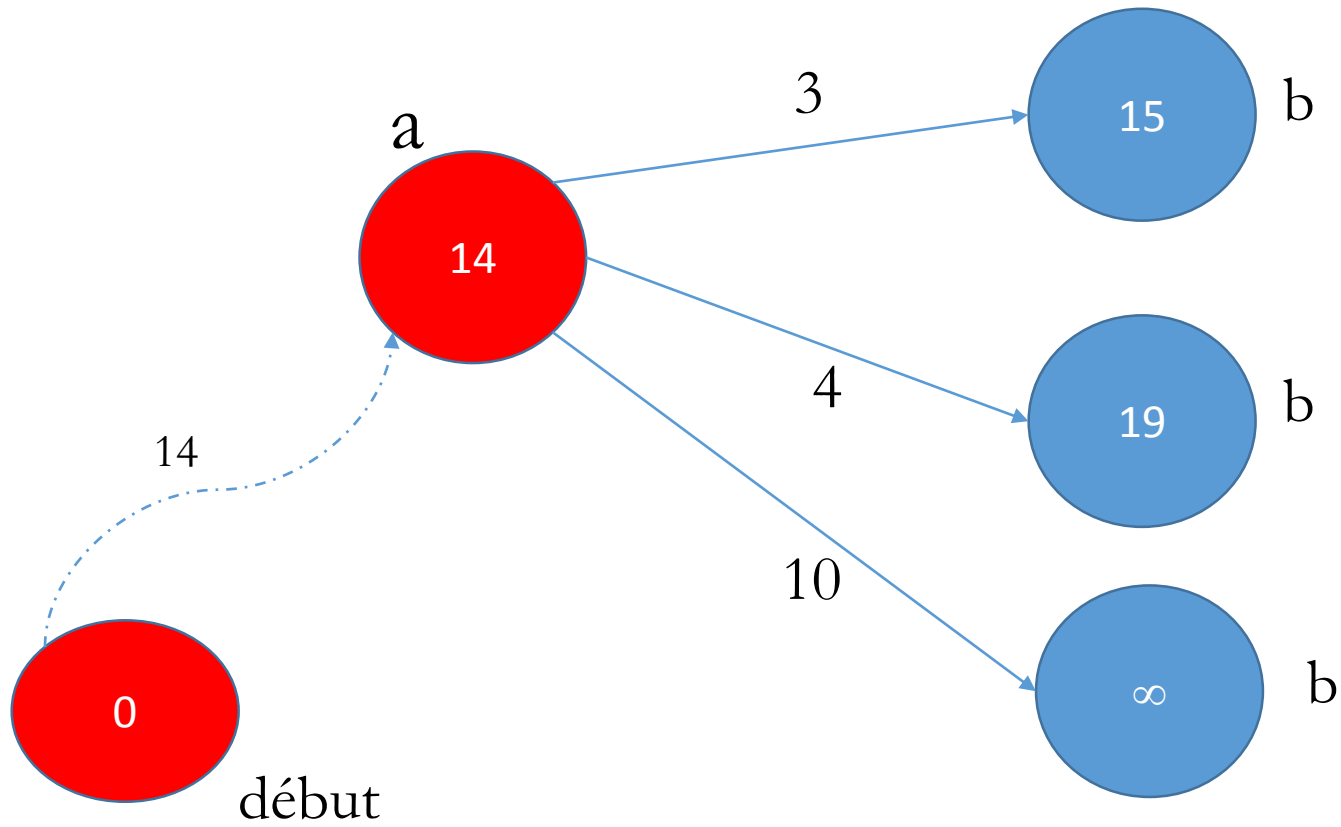
# Algorithme Dijkstra (1959)

- Entrées :  $G=(X,A)$  un graphe avec une valeur **positive**  $v$  des arêtes, **debut** un sommet de départ de  $X$  , **fin** un sommet d'arrivée de  $X$
- Sorties : **D** **tableau des distances minimales** de chaque sommet depuis debut, **un tableau P** qui contient pour chaque sommet le sommet qui le précède dans un plus court chemin de debut à fin.

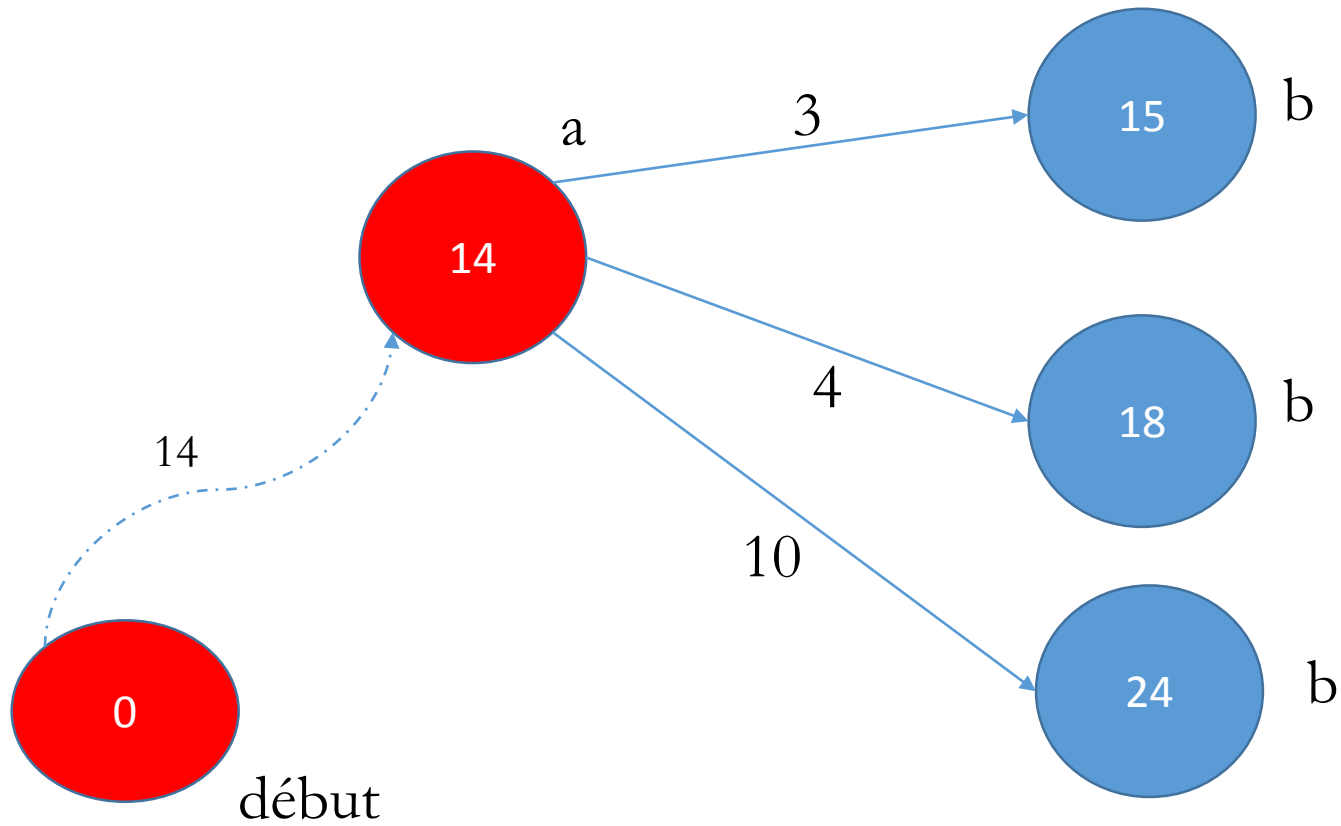
1. Initialiser tous les sommets comme *non marqué*
2. Affecter la valeur  $\infty$  à tout  $D$  sauf  $D(\text{debut})=0$
3. Tant qu'il existe un sommet non marqué :
  1. Choisir un sommet  $a$  non marqué de plus petite valeur  $D$
  2. Marquer  $a$  ;  
Pour chaque **sommet  $b$  non marqué voisin de  $a$**  faire  
    Si  $D(b) > D(a) + v(a,b)$   
        Alors  $D(b) \leftarrow D(a) + v(a,b)$   
         $P(b) \leftarrow a$

<https://www.youtube.com/watch?v=rHylCtXtdNs>

# Avant



Après



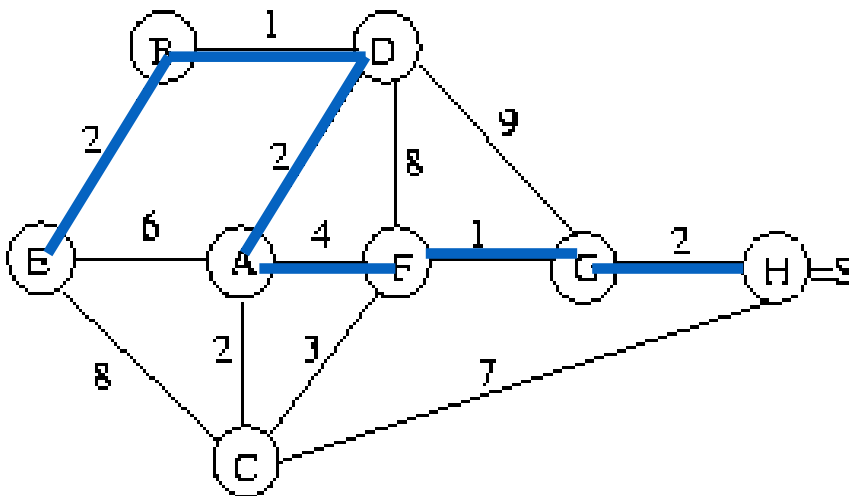


# Ex : algorithme du plus court chemin

Recherche d'une chaîne de valeur minimum de E à H (plus courte chaîne)

À chaque étape on choisit un nouveau sommet qui doit être

1. relié à un sommet déjà choisi
2. le plus proche possible de E parmi ceux qui restent à choisir



- Le sommet E est à la distance 0 de lui-même

B : 2    C : 8    A : 6

- Le sommet B est à la distance 2 du départ E

C : 8    D : 3    A : 6

- Le sommet D est à la distance 3 du départ E

C : 8    A : 5    F : 11    G : 12

- Le sommet A est à la distance 5 du départ E

C : 7    F : 9    G : 12

- Le sommet C est à la distance 7 du départ E

F : 9    G : 12    H : 14

- Le sommet F est à la distance 9 du départ E

G : 10    H : 14

- Le sommet G est à la distance 10 du départ E

H : 12

- Le sommet H est à la distance 12 du départ E

La chaîne la plus petite de E à H a pour valeur 12

# La fonction la plus compliquée

- **Trouve\_min** ( $G$ ) doit retourner le nœud non marqué de plus petite valeur (point 3.1 se l'algorithme du slide 62)
- L'implémenter efficacement a une grande influence sur la complexité
- Comme c'est une recherche de minimum, on peut penser que c'est en  $O(n)$ . Effectué  $n$  fois.
- Et dans ce cas la complexité totale est en  $O(n^2 + (n+m))$
- Mais si on implémente  $D$  comme une file ou une liste ordonnée (avec des sda adaptées : **tas**, **tas de Fibonacci**) on arrive à faire descendre la complexité à
  - $O((n+m) \log n)$ , voire
  - $O(m+n \log n)$

# Remarque

- L'algorithme de Dijkstra ne fonctionne qu'avec des poids positifs.
- Avec des poids négatifs (mais pas de circuits négatifs !!!) on peut utiliser d'autres algorithmes.

# Autres éléments et conclusions

# Les algorithmes de graphes sont importants

- Beaucoup de graphes explicites (la carte de France, le réseau social,...)
- Mais aussi beaucoup de graphes implicites :
  - Les sommets sont toutes les positions possibles aux échecs
  - Les arcs sont les coups pour passer d'une position à une autre
  - On aimerait explorer ce graphe
  - Et d'ailleurs on l'a fait (cf bloc 2)
- Typiquement, beaucoup de problèmes d'IA peuvent être vus comme des problèmes de recherche dans un grand espace implicite appelé graphe d'états