

DIU EIL bloc 5

Graphes - Définitions, algorithmes élémentaires

Laure Gonnord

<http://laure.gonnord.org/pro/>

Laure.Gonnord@univ-lyon1.fr

DIU EIL, Dpt Info UCBL

2019-2020



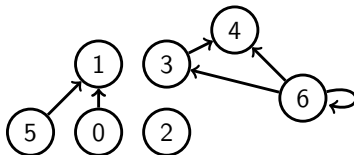
Credits : O. Bournez pour Polytechnique : définition, parcours.

- 1 Définitions
 - Graphes, graphes orientés
 - Arbres
- 2 Deux représentations des graphes
- 3 Parcours
- 4 Clôture transitive

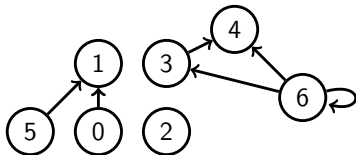
- 1 Définitions
 - Graphes, graphes orientés
 - Arbres
- 2 Deux représentations des graphes
- 3 Parcours
- 4 Clôture transitive

Un graphe orienté

- Un *graphe orienté* (digraph) est donné par un couple $G = (V, E)$, où
 - ▶ V est un ensemble.
 - ▶ $E \subset V \times V$.
- Exemple:
 - ▶ $V = \{0, 1, \dots, 6\}$.
 - ▶ $E = \{(0, 1), (3, 4), (5, 1), (6, 3), (6, 4), (6, 6)\}$.



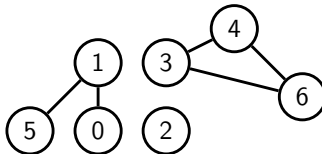
Vocabulaire



- Les éléments de V sont appelés des *sommets* (parfois aussi des *nœuds*).
- Les éléments e de E sont appelés des *arcs*.
- Si $e = (u, v)$, u est appelé *la source* de e , v est appelé *la destination* de e .
- Remarque:
 - ▶ Les boucles (les arcs (u, u)) sont autorisées.

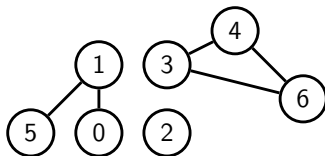
Un graphe

- Un *graphe*¹ est donné par un couple $G = (V, E)$, où
 - ▶ V est un ensemble.
 - ▶ E est un ensemble de paires $\{u, v\}$ avec $u, v \in V$.
- On convient de représenter une paire $\{u, v\}$ par (u, v) ou (v, u) .
- Autrement dit, (u, v) et (v, u) dénotent la même arête.
- Exemple:
 - ▶ $V = \{0, 1, \dots, 6\}$
 - ▶ $E = \{(0, 1), (3, 4), (5, 1), (6, 3), (6, 4)\}$.



¹Lorsqu'on ne précise pas, par défaut, un graphe est non-orienté.

Vocabulaire



- Les éléments e de E sont appelés des *arêtes*. Si $e = (u, v)$, u et v sont appelés les *extrémités* de e .
- Remarque: (sauf autre convention explicite)
 - ▶ Les boucles ne sont pas autorisées.

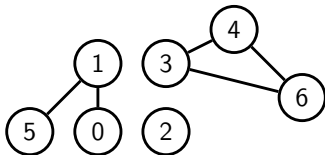
Graphes orientés

- sommets
- arc (u, v)
tuple
- boucles autorisées (u, u)
- cycles possibles

Graphes non orientés

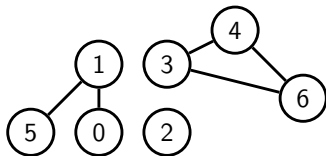
- extrémités
- arête $\{u, v\}$
ensemble
- boucles non autorisées $\{u, u\}$
- cycles possibles
(sans cycles c'est un arbre)

Vocabulaire (cas non-orienté)



- u et v sont dits *voisins* s'il y a une arête entre u et v .
- Le degré de u est le nombre de voisins de u .

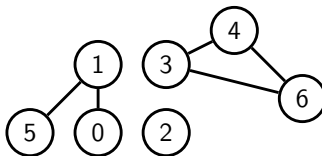
Vocabulaire (cas non-orienté)



- Un *chemin du sommet s vers le sommet t* est une suite e_0, e_1, \dots, e_n de sommets telle que $e_0 = s$, $e_n = t$, $(e_{i-1}, e_i) \in E$, pour tout $1 \leq i \leq n$.
 - ▶ n est appelé la *longueur* du chemin, et on dit que t est *joignable* à partir de s .
 - ▶ Le chemin est dit *simple* si les e_i sont distincts deux-à-deux.
 - ▶ Un *cycle* est un chemin de longueur non-nulle avec $e_0 = e_n$.
- Le sommet s est dit à *distance n* de t s'il existe un chemin de longueur n entre s et t , mais aucun chemin de longueur inférieure.

Composantes connexes (cas non-orienté)

- Prop. La relation “être joignable” est une relation d'équivalence.
- Les classes d'équivalence sont appelées les *composantes connexes*.



- Un graphe est dit *connexe* s'il n'y a qu'une seule classe d'équivalence.
 - ▶ Autrement dit, tout sommet est joignable à partir de tout sommet.

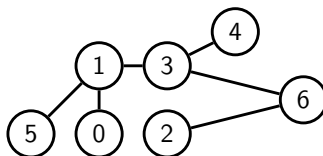
Les graphes sont partout!

- Beaucoup de problèmes se modélisent par des objets et des relations entre objets.
- Exemples:
 - ▶ Le graphe routier.
 - ▶ Les réseaux informatiques.
 - ▶ Le graphe du web.
- Beaucoup de problèmes se ramènent à des problèmes sur les graphes.
- Théorie des graphes:
 - ▶ Euler, Hamilton, Kirchhoff, König, Edmonds, Berge, Lovász, Seymour,...
- Les graphes sont omniprésents en informatique.

- 1 Définitions
 - Graphes, graphes orientés
 - Arbres
- 2 Deux représentations des graphes
- 3 Parcours
- 4 Clôture transitive

Les arbres

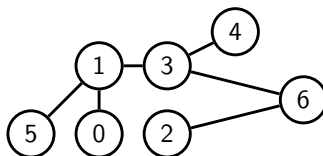
- Un graphe² connexe sans cycle est appelé un *arbre* (libre).
- Un graphe² sans-cycle est appelé une *forêt*:
 - ▶ chacune de ses composantes connexes est un arbre.



²non-orienté.

Les arbres sont partout!

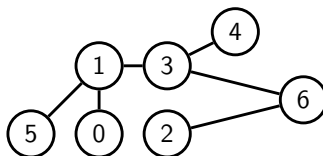
- Un graphe² connexe sans cycle est appelé un *arbre* (libre).
- Un graphe² sans-cycle est appelé une *forêt*:
 - ▶ chacune de ses composantes connexes est un arbre.
- Dès qu'on a des objets, des relations entre objets, et pas de cycle, on a donc un arbre ou une forêt.



²non-orienté.

Les arbres sont partout!

- Un graphe² connexe sans cycle est appelé un *arbre* (libre).
- Un graphe² sans-cycle est appelé une *forêt*:
 - ▶ chacune de ses composantes connexes est un arbre.
- Dès qu'on a des objets, des relations entre objets, et pas de cycle, on a donc un arbre ou une forêt.



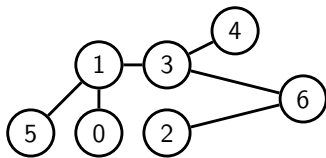
- Les arbres sont omniprésents en informatique.

²non-orienté.

Une caractérisation & Quelques propriétés

Soit $G = (V, E)$ un graphe³. Les propriétés suivantes sont équivalentes:

- G est un arbre (libre).
- Deux sommets quelconques de V sont connectés par un unique chemin simple.
- G est connexe, mais ne l'est plus si on enlève n'importe laquelle de ses arêtes.
- G est connexe, et $|E| = |V| - 1$.
- G est sans cycle, et $|E| = |V| - 1$.
- G est sans cycle, mais ne l'est plus si l'on ajoute n'importe quelle arête.



³non-orienté.

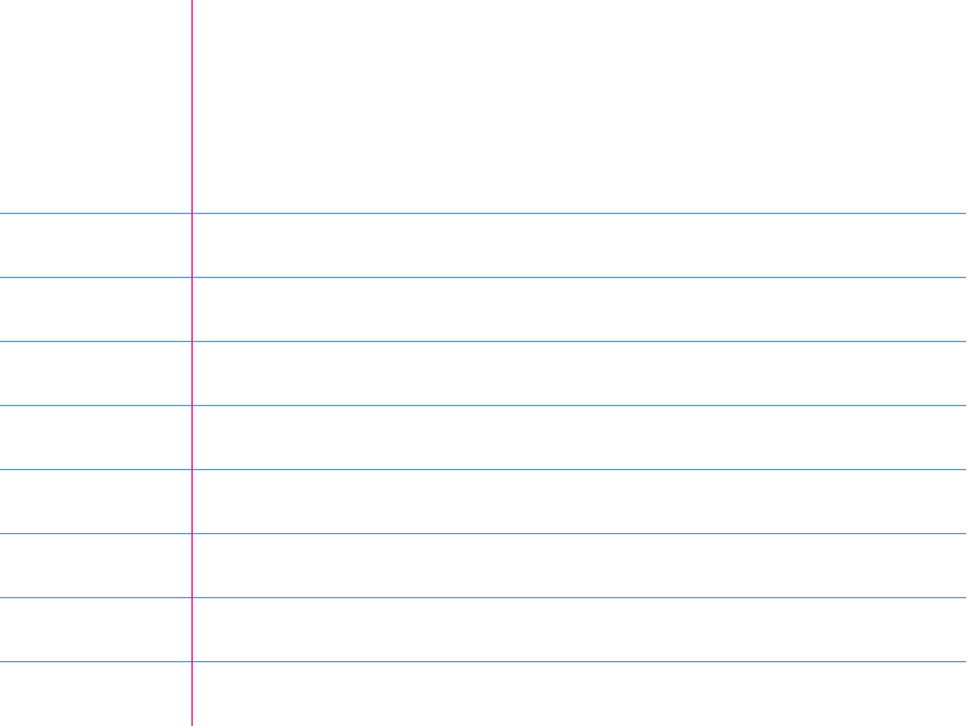
voir <https://www.enseignement.polytechnique.fr/informatique/INF431/X11-2012-2013/inf431-poly.pdf>

Théorème 6.4.1 Soit \mathcal{G} un graphe non-orienté de n sommets. Ces propriétés sont équivalentes et caractérisent le fait que \mathcal{G} est un arbre :

1. \mathcal{G} est connexe et sans cycles ;
2. \mathcal{G} est connexe et a $n - 1$ arêtes ;
3. \mathcal{G} n'a pas de cycles et $n - 1$ arêtes ;
4. étant donnés deux sommets de \mathcal{G} , ils sont reliés par une et une seule chaîne ;
5. \mathcal{G} est connexe et cette connexité disparaît dès que l'on efface une arête quelconque ;
6. \mathcal{G} n'a pas de cycles mais l'ajout d'une arête crée toujours un cycle ; ◇

Arborescence Lorsqu'un arbre est considéré comme un graphe non-orienté, on ne distingue plus quel sommet est la racine. C'est pourquoi on appelle parfois *arborescence* un graphe possédant la propriété d'être un arbre et muni d'une racine distinguée.

Le lemme précédent dit essentiellement qu'un arbre est un graphe possédant juste assez d'arêtes pour préserver la connexité. Inversement, on peut obtenir un arbre en retirant des arêtes d'un graphe connexe :



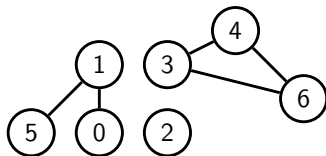
- 1 Définitions
- 2 Deux représentations des graphes
- 3 Parcours
- 4 Clôture transitive

Représentation des graphes: matrice d'adjacence

- Matrice d'adjacence. Pour $G = (V, E)$, $V = \{1, 2, \dots, n\}$, on représente G par une matrice M $n \times n$.

$$M_{i,j} = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{sinon} \end{cases}$$

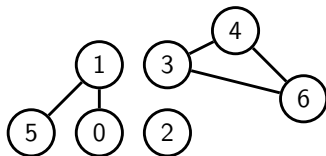
- Si graphe valué: $M_{i,j} = v(i,j)$.



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

Représentation des graphes: liste de successeurs

- On associe à chaque sommet i , la liste des sommets j tels que $(i, j) \in E$.



- $L[0] = (1)$
- $L[1] = (0, 5)$
- $L[2] = ()$
- $L[3] = (4, 6)$
- $L[4] = (3, 6)$
- $L[5] = (1)$
- $L[6] = (3, 4)$

Meilleure représentation?

- Matrice: mémoire $O(n^2)$ (mieux pour graphes denses).
- Listes: mémoire $O(n + m)$ (mieux pour graphes creux).
- où n nombre de sommets, m nombre d'arêtes.
- Le mieux?: cela dépend du contexte.

Matrice vs. liste d'adjacence

Pour $G = (S, A)$:

- Liste d'adjacence : chacun connaît ses voisins.
- Matrice d'adjacence : toutes les transitions/distances directes.

- Complexités spatiales différentes.

- Lien entre $|A|$ et $||$?

- Toujours :

$$|A| \leq ||^2$$

- Connexes :

$$|| - 1 \leq |A|$$

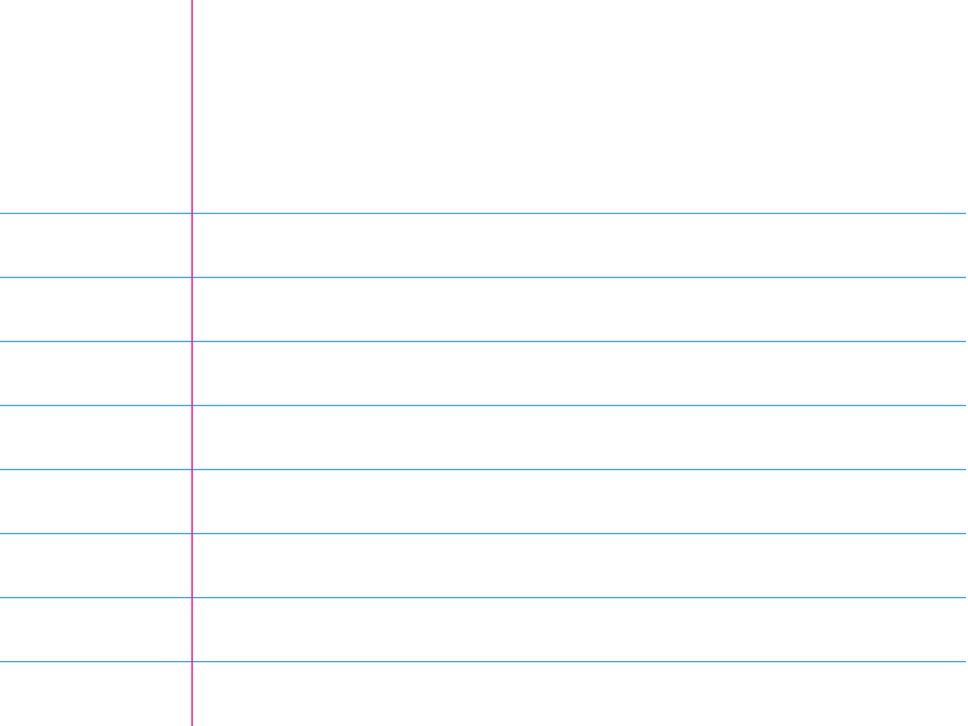
— borne inférieure atteinte pour les arbres

- Planaires :

$$|A| \leq 3|| - 6$$

$|A| \rightarrow$ nombre d'arêtes
 $|| \rightarrow$ nombre de sommets

??
 ;



Python, liste d'adjacence

Liste d'adjacence implémentée par un dictionnaire :

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

Accéder à ma liste de voisins : `g["c"]`. *ici les étiquettes (*label*) sont des chaînes, ce qui n'est pas forcément une bonne idée*

Python, liste d'adjacence 2/2

Attention à l'ajout d'une arête dans le cas non-orienté :

```

{ if vertex1 in self.__graph_dict:
    self.__graph_dict[vertex1].append(vertex2)
  else:
    self.__graph_dict[vertex1] = [vertex2]
  if vertex2 in self.__graph_dict:
    self.__graph_dict[vertex2].append(vertex1)
  else:
    self.__graph_dict[vertex2] = [vertex1]

```

Pour le cas non orienté, si on ajoute l'arête $\{ \text{vertex1}, \text{vertex2} \}$, alors il faut rajouter vertex1 à la liste d'adjacence de vertex2 et vice-versa.

- 1 Définitions
- 2 Deux représentations des graphes
- 3 Parcours**
- 4 Clôture transitive

Parcours générique

ParcoursGenerique(G, s):

- $L := \{s\}$ \leftarrow déjà visités
- $B := \text{Voisins}(s)$ \leftarrow frontière
- Tant que $B \neq \emptyset$
 - ▶ choisir u dans B
 - ▶ $L := L \cup \{u\}$.
 - ▶ $B := B - \{u\}$.
 - ▶ $B := B \cup (\text{Voisins}(u) - L)$
 $\underbrace{\hspace{1.5cm}}$ voisins de u pas encore visités

On verra que les parcours de graphes peuvent répondre à des buts très divers ; dans tous les cas, le principe d'un parcours est de *visiter* l'ensemble de la composante connexe. A chaque étape on peut donc distinguer états pour les sommets :

1. les sommets pas encore visités,
2. la frontière, c'est-à-dire les sommets déjà visités, dont certains voisins n'ont pas été visités,
3. les sommets déjà traités, c'est-à-dire qu'ils ont été visités, ainsi que tous leurs voisins, visités.

Initialement, tous les sommets sont à l'état (1), sauf le sommet de départ qui est à l'état (2). Une étape du parcours consiste alors à :

- choisir un sommet s de la frontière (2),
- si tous les voisins de s ont été visités, c'est-à-dire qu'aucun n'est à l'état (1), alors s passe à l'état (3),
- sinon on choisit un de ses voisins de s non visités, qui passe de l'état (1) à l'état (2).

Le parcours se termine lorsque tous les voisins (de la composante connexe) sont à l'état (3). On a alors visité tous les sommets, dans un ordre qui dépend comment sont choisis les sommets visités à chaque étape.

Dans tous les cas, au cours d'un parcours, il faut se souvenir des sommets déjà parcourus. Très souvent, on munira les sommets d'une marque permettant d'indiquer leur état.
--

Application 1: numérotation des sommets

ParcoursNumerotation(G, s):

- $num := 0$
- $numero[s] := num; num := num + 1.$
- $L := \{s\}$
- $B := Voisins(s)$
- Tant que $B \neq \emptyset$
 - ▶ choisir u dans B
 - ▶ $numero[s] := num; num := num + 1.$
 - ▶ $L := L \cup \{u\}.$
 - ▶ $B := B - \{u\}.$
 - ▶ $B := B \cup (Voisins(u) - L)$

Application 2: composantes connexes

UneComposante(G, s):

- $L := \{s\}$
- $B := \text{Voisins}(s)$
- Tant que $B \neq \emptyset$
 - ▶ choisir u dans B
 - ▶ $L := L \cup \{u\}$.
 - ▶ $B := B - \{u\}$.
 - ▶ $B := B \cup (\text{Voisins}(u) - L)$
- Retourner L \rightarrow composante

ComposantesConnexes(G)

- Tant que $V \neq \emptyset$ \rightarrow V ensemble des sommets
tant qu'il y a des sommets
 - ▶ Choisir s dans V
 - ▶ $L := \text{UneComposante}(G, s)$
 - ▶ Afficher L
 - ▶ $V := V - L$

Plan

Rappels: les graphes

Rappels: Les arbres

Les arbres binaires

Parcours d'arbres

Représentation des graphes

Matrice d'adjacences

Liste de successeurs

Parcours de graphes

Parcours générique

Parcours en largeur BFS

Parcours en profondeur DFS

Calcul de distances

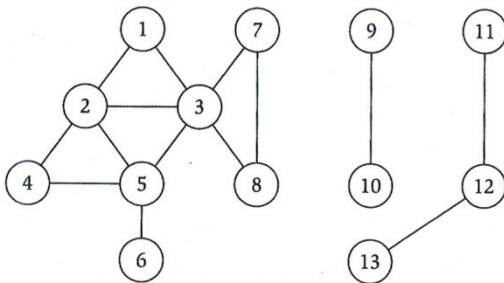
Algorithme de Bellman-Ford

Algorithme de Dijkstra

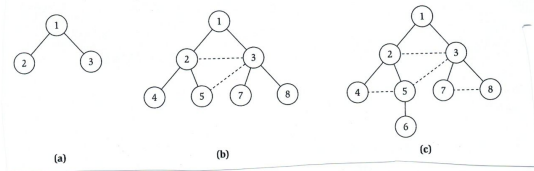
Algorithme de Floyd Warshall

Parcours en largeur BFS

- B est implémenté par une file (FIFO = First In First Out).
- On explore les sommets par niveau: le niveau k correspond aux sommets à distance k du sommet s .
- Exemple:



BFS sur l'exemple précédent: (a), (b) et (c) montre les niveaux successifs qui sont considérés. Les sommets sont visités dans l'ordre 1, 2, 3, 4, 5, 7, 8, 6.



- Pour chaque $k \geq 1$, le niveau k correspond aux sommets à distance exactement k de s .
- Il y a un chemin de s vers t si et seulement si t apparaît dans un certain niveau. Si t apparaît au niveau k , alors t est à distance k de s .
- Soit x et y deux sommets aux niveaux L_i et L_j avec $(x, y) \in E$. Alors i et j diffèrent d'au plus 1.

Plan

Rappels: les graphes

Rappels: Les arbres

Les arbres binaires

Parcours d'arbres

Représentation des graphes

Matrice d'adjacences

Liste de successeurs

Parcours de graphes

Parcours générique

Parcours en largeur BFS

Parcours en profondeur DFS

Calcul de distances

Algorithme de Bellman-Ford

Algorithme de Dijkstra

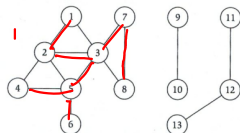
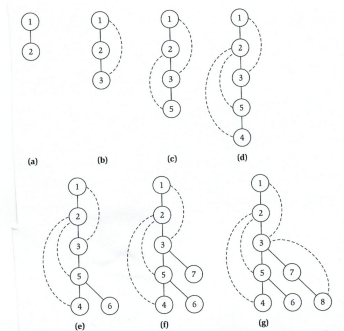
Algorithme de Floyd Warshall

Parcours en profondeur DFS

DFS(s)

- Marquer s comme “exploré” et ajouter s à L .
- Pour chaque arête $(s, v) \in E$
 - ▶ si v n'est pas marqué “exploré” alors
 - appeler récursivement DFS(v)

Sur le graphe précédent, les sommets sont visités cette fois dans l'ordre 1, 2, 3, 5, 4, 6, 7, 8. Le dessin plus bas montre les sommets dans l'ordre où ils sont découverts.



arbre couvrant
en rouge obtenu
par DFS

en pointillés les arêtes
qui ne sont pas utilisées

Si on appelle T l'arbre couvrant produit (celui qui contient "l'histoire du parcours": on décide que v a u comme père si l'arête (u, v) a permis de découvrir v), on a:

- Pour chaque appel récursif $\text{DFS}(u)$, tous les sommets qui sont marqués "explorés" entre l'invocation de l'appel et son retour sont des descendants de u dans l'arbre T produit.
- Soit T un arbre DFS, et x et y deux sommets dans T , avec $(x, y) \in E$ qui n'est pas une arête de T . Alors soit x est un ancêtre de y , soit le contraire.

Implémenter BFS(s)

■ $Discovered[s] := true; Discovered[v] := false$ pour $v \neq s$.

■ $L[0] = \{s\}; i := 0$ // Layer counter

■ Set $T = \emptyset$ // Arbre

■ Tant que $L[i] \neq \emptyset$

▶ $L[i+1] := \emptyset$

▶ Pour chaque $u \in L[i]$

• Pour chaque arête $(u, v) \in E$

• Si $Discovered[v] == false$ alors

• $Discovered[v] := true;$

• Ajouter (u, v) à l'arbre T ($\pi(v) = u$).

• Ajouter v à $L[i+1]$

• Fin si

▶ Fin pour

▶ $i := i+1$

■ Fin tant que

$L[i+1]$ → ensemble des sommets au niveau $i+1$

on marque chaque voisin non marqué →

le père de v est u

Temps $O(n + m)$, où n est le nombre de sommets, et m le nombre d'arêtes avec une représentation par liste de successeurs.

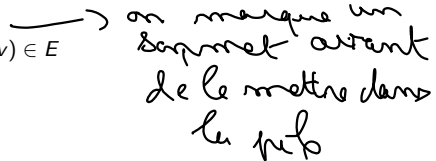
Remarques sur BFS.

- implémentation à l'aide d'une file
- complexité en $O(m + n)$

nb de
sommets

nb d'arêtes

Implémenter DFS(s)

- $Discovered[s] := true; Discovered[v] := false$ pour $v \neq s$.
- Créer S comme la pile contenant uniquement l'élément s
- Tant que $S \neq \emptyset$
 - ▶ Prendre le sommet u de S
 - ▶ Si $Discovered[u] == false$ alors
 - $Discovered[u] := true$ 
 - Pour chaque arête $(u, v) \in E$
 - Ajouter u à la pile S
 - Fin pour
 - ▶ Fin si
- Fin while

Temps $O(n + m)$, où n est le nombre de sommets, et m le nombre d'arêtes, avec une représentation par liste de successeurs.

Remarques sur DFS :

- on implémente naturellement en récursif
- en itératif on implémente avec une pile
- complexité en $O(n + m)$
 - nb de sommets
 - nb d'arêtes

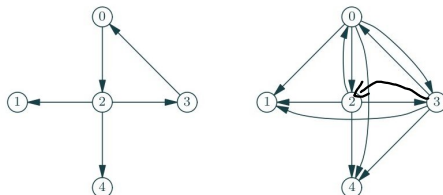
Applications des parcours

- Numérotation (vue plus haut).
- Découverte de cycles.
- Numérotation en niveaux ...

- 1 Définitions
- 2 Deux représentations des graphes
- 3 Parcours
- 4 Clôture transitive

Clôture transitive.

Attention il manque une flèche sur le dessin de droite !



Le graphe $G^+ = (V, E^+)$ de la clôture transitive du graphe orienté $G = (V, E)$ est tel que E^+ est le plus petit ensemble d'arêtes satisfaisant :

- $E \subseteq E^+$
- $(x, y) \in E^+$ and $(y, z) \in E^+ \Leftrightarrow (x, z) \in E^+$

Un premier algorithme pour la clôture

Attention ici la multiplication n'est pas forcément celle que l'on croit

Si E est la matrice d'adjacence du graphe initial :

$$E^+ = E + E^2 + \dots E^{n-1}.$$

► Cela donne un algo en $O(n^4)$.

► Mieux ? oui ; en $O(n^3)$, avec Floyd Warshall, voir plus tard !

Plus court chemin de toutes sources
vers toutes arrivées