Graphes orientés (§12.4)

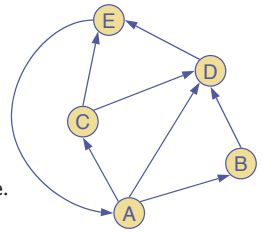
Terminologie

- Un graphe orienté est un graphe G=(S,A) dont toutes les arêtes sont orientés
- Étant donné un sommet d'un graphe orienté, on va considérer deux degrés pour ce sommet
  - inDeg(s), est le nombre d'arêtes entrant dans s, i.e.

$$|\{t \mid (t,s) \in A\}|$$

  - outDeg(s), est le nombre d'arêtes sortant de s, i.e.

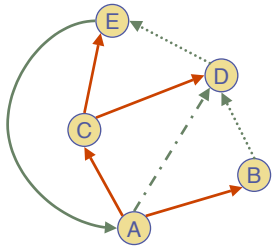
$$|\{t \mid (s,t) \in A\}|$$

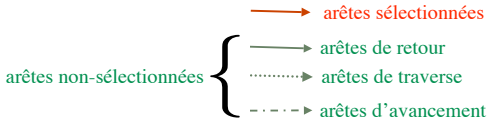


- Proposition: $\sum_{s \in S} inDeg(s) = \sum_{s \in S} outDeg(s) = m$

Parcours d'arbres orientés

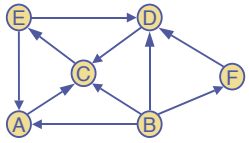
- On spécialise les parcours (en profondeur et en largeur) de graphes aux graphes orientés en traversant les arêtes seulement selon leur direction.
- Lorsqu'on exécute un algorithme de parcours à partir d'un sommet s d'un graphe orienté, les sommets visités représentent l'ensemble des sommets accessibles du sommet s



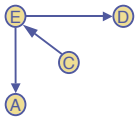


Sommets accessibles

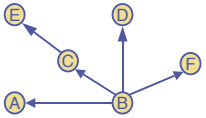
- L'arbre composé des arêtes sélectionnées et des sommets visités lors d'un parcours de graphe à partir du sommet s, représente l'ensemble des sommets accessibles de s



Sommets accessibles de c:

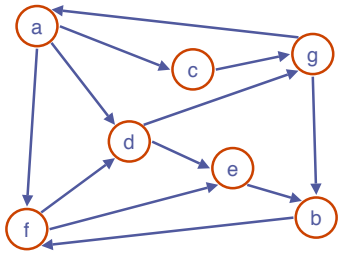


Sommets accessibles de b:



Graphes fortement connexes

- Graphes orientés dans lequel de chaque sommet on peut atteindre tous les autres sommets



Algorithmes pour tester si un graphe est fortement connexe

Première idée: utiliser l'algorithme de parcours en profondeur à partir de chacun des sommets du graphe

1) Pour chaque sommet s du graphe, exécute DFS(G,s); les sommets visités sont les sommets accessibles de s

2) Si, pour chaque sommet l'algorithme DFS(G,s) visite les n sommets du graphe alors le graphe est fortement connexe

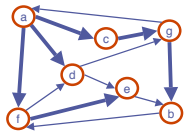
Complexité en temps: $O(n(n+m))$

Algorithmes pour tester si un graphe est fortement connexe

Meilleure idée: on peut tester si un graphe est fortement connexe en exécutant deux parcours en profondeur

1) On choisit un sommet s dans G et on exécute DFS(G,s)
   - Si on à moins de n sommets visités après l'exécution de l'algorithme, on retourne NON

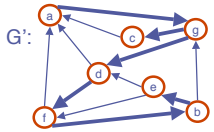
G:



2) On construit le graphe G' qui est le graphe G dans lequel toutes les arêtes ont été renversées

3) On exécute DFS(G,s)
   - Si on à moins de n sommets visités après l'exécution de l'algorithme, on retourne NON
   - Sinon retourner OUI

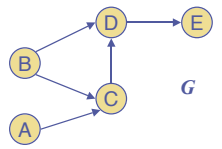
G':



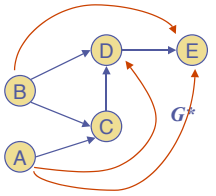
Complexité en temps: $O(n+m)$

Fermeture transitive d'un graphe orienté

- Étant donné un graphe orienté G, la fermeture transitive de G est un graphe orienté G* tel que:
  - G* a les mêmes sommets que G
  - S'il y a un chemin dans G de u à v ($u \neq v$), alors il y a une arête dans G* de u à v

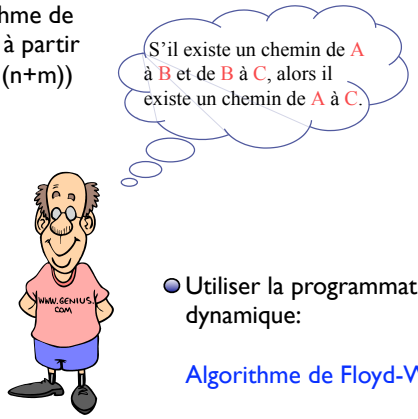


- La fermeture transitive d'un graphe contient toutes l'information sur les sommets accessibles du graphe



Calculer la fermeture transitive d'un graphe

- On peut utiliser l'algorithme de parcours en profondeur à partir de chaque sommet: $O(n(n+m))$

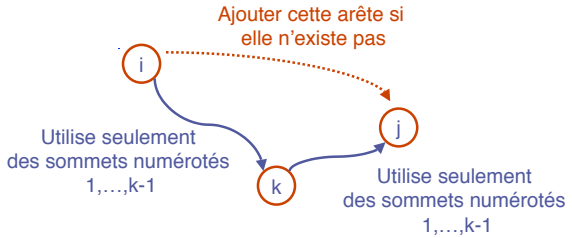


- Utiliser la programmation dynamique:

  Algorithme de Floyd-Warshall

Algorithme de Floyd-Warshall

- Idée 1: Numéroter les sommets de 1 à n
- Idée 2: À l'étape k, considérer les chemins utilisant seulement les sommets de 1 à k comme sommets intermédiaires

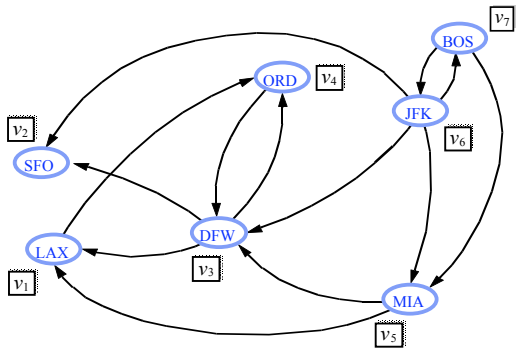


Algorithme de Floyd-Warshall

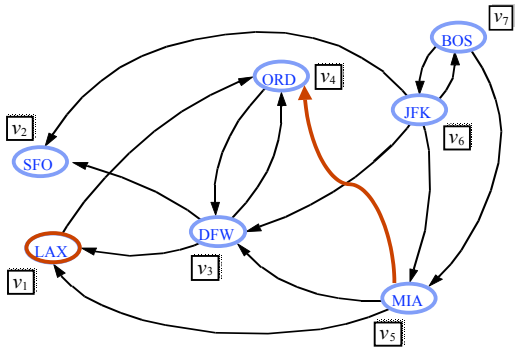
- L'algorithme de Floyd-Warshall numérote les sommets de 1 à n et calcule une séries de graphes orientés $G_0, G_1, \ldots, G_n$ tels que
  1) $G_0 = G$
  2) G_k a une arête dirigée (v_i, v_j) si G a un chemin de v_i à v_j dont les sommets intermédiaires sont dans $\{v_1, v_2, \ldots, v_k\}$
  3) $G_n = G^*$
- Complexité en temps: $O(n^3)$, si on assume que sontAdjacents peut être calculé en $O(1)$

```
Algorithme FloydWarshall(G)
  Entrée graphe orienté G
  Sortie fermeture transitive G* de G G_{k-1}
  i ← 1
  pour tout v ∈ G.sommets()
    numéroter v par v_i
    i ← i + 1
  G_0 ← G
  pour k ← 1 à n faire
    G_k ← G_{k-1}
    pour i ← 1 à n (i ≠ k) faire
      pour j ← 1 à n (j ≠ i, k) faire
        si G_{k-1}.sontAdjacents(v_i, v_k) ∧
            G_{k-1}.sontAdjacents(v_k, v_j)
          si ¬G_k.sontAdjacents(v_i, v_j)
            G_k.insérerArêteDirigée(v_i, v_j, k)
  retourner G_n
```

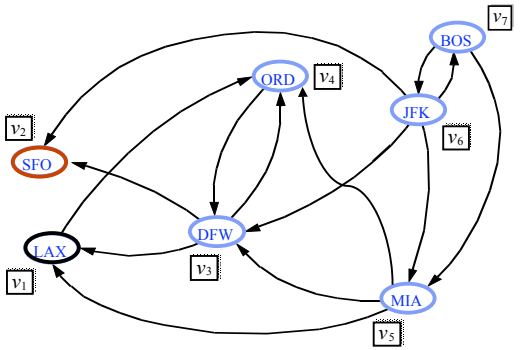

Algorithme de Floyd-Warshall (exemple)



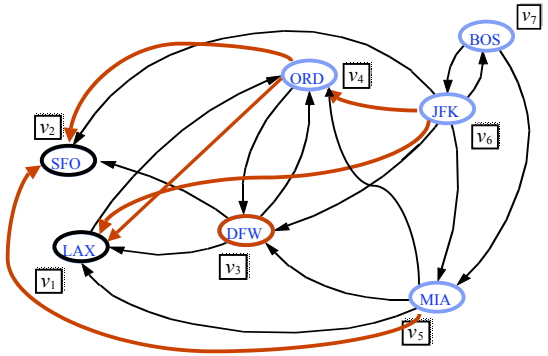
Floyd-Warshall (itération 1)



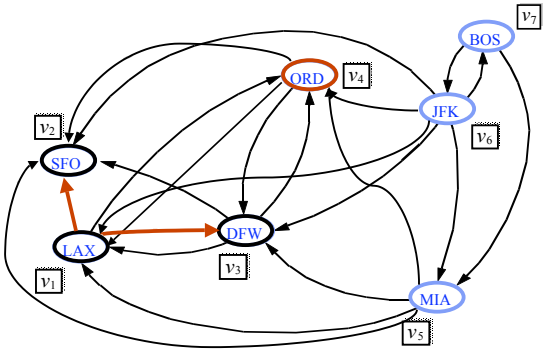
Floyd-Warshall (itération 2)



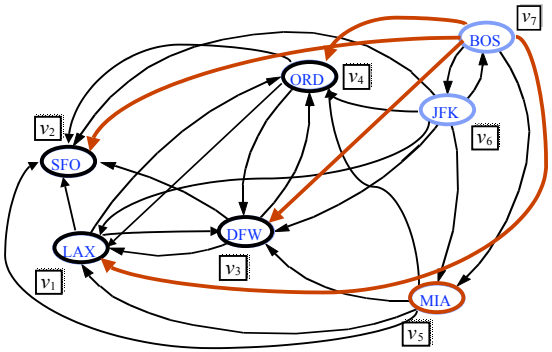
Floyd-Warshall (itération 3)



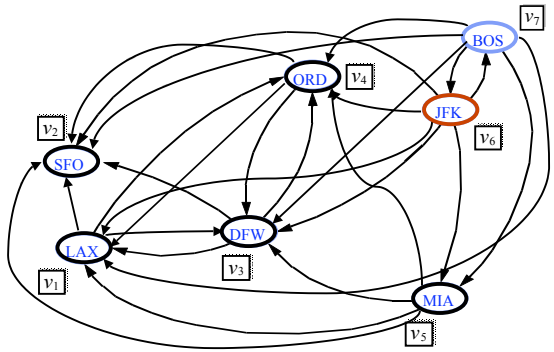
Floyd-Warshall (itération 4)



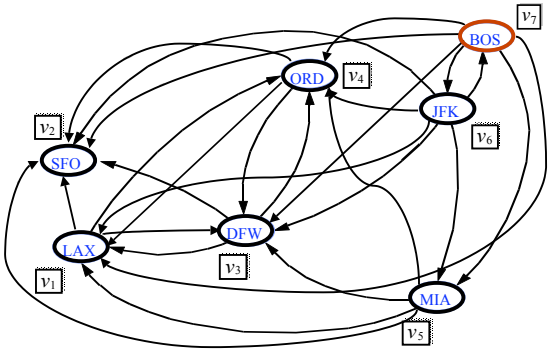
Floyd-Warshall (itération 5)



Floyd-Warshall (itération 6)



Floyd-Warshall (conclusion)

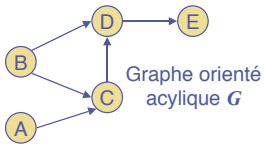


Graphes orientés acycliques et ordre topologique

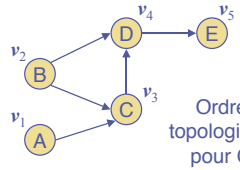
- Un graphe orienté acyclique est un graphe orienté qui n'a aucun cycle orienté
- Un ordre topologique pour graphe orienté est une numérotation des sommets

  $$v_1, \ldots, v_n$$

  telle que pour chaque arête (v_i, v_j) du graphe on a $i < j$
- Théorème: Un graphe orienté admet un ordre topologique si et seulement si le grpahe est acyclique.



Graphe orienté acylique *G*



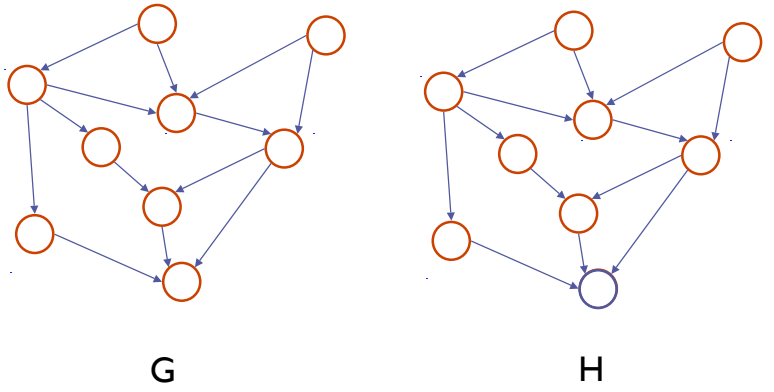
Ordre topologique pour G

Un algorithme pour trouver un ordre topologique

```
Algorithme OrdreTopologique(G)
    H ← G          // Copy temporaire de G
    n ← G.nombreSommets()
    tant que il existe un sommet v dans H tel que OutDeg(v)=0 faire
        Étiquetter v par n
        n ← n - 1
        Enlever v de H
```

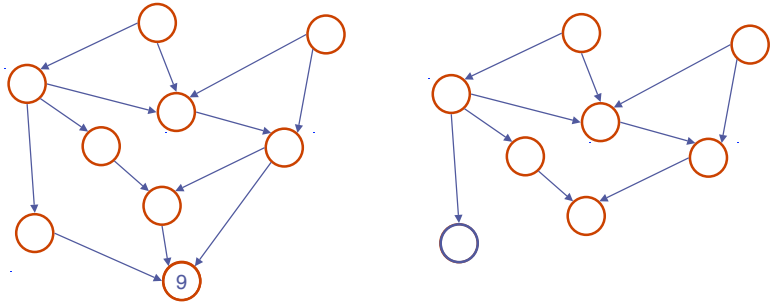
- Cet algorithme est différent de celui dans votre livre
- Si à la fin de l'algorithme, il reste encore des sommets dans H, cela implique que le graphe contient un cycle orienté

ordre topologique (exemple)



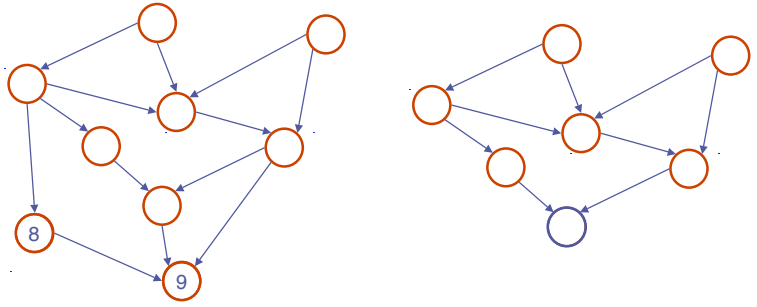
G H

ordre topologique (exemple)



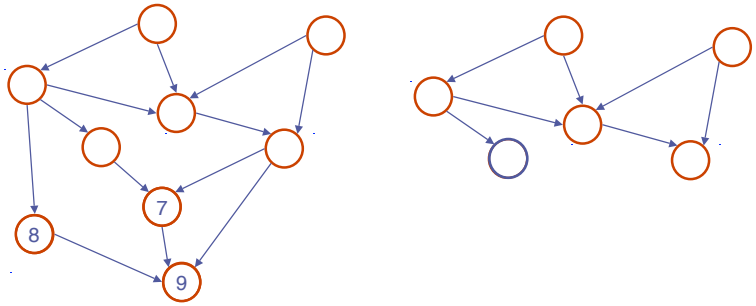
G H

ordre topologique (exemple)



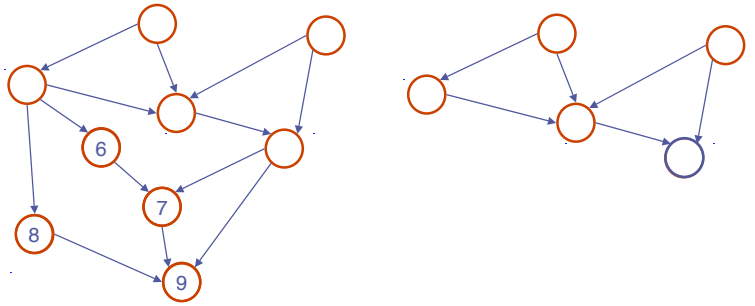
G H

ordre topologique (exemple)



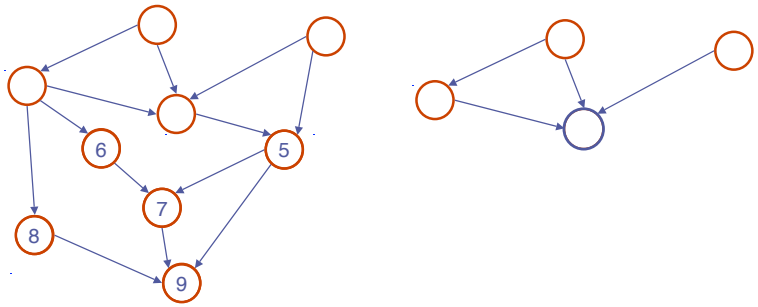
G H

ordre topologique (exemple)



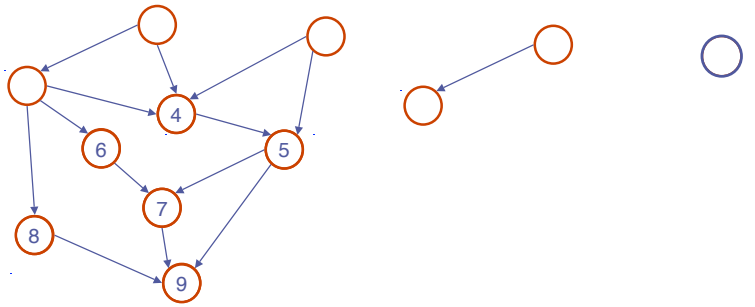
G H

ordre topologique (exemple)



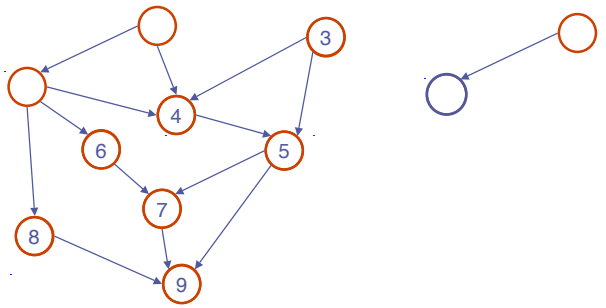
G H

ordre topologique (exemple)



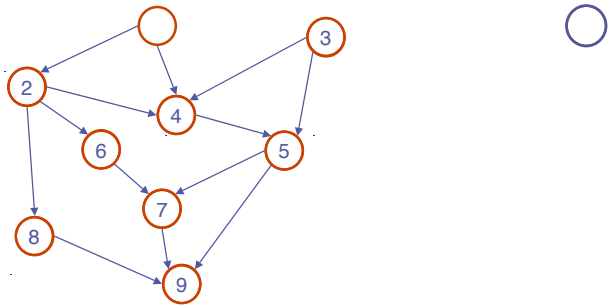
G H

ordre topologique (exemple)



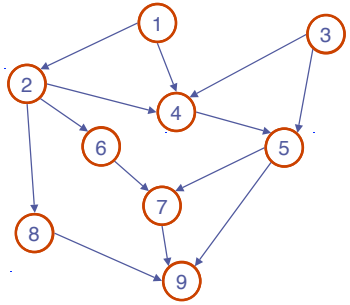
G H

ordre topologique (exemple)



G H

ordre topologique (exemple)



G H