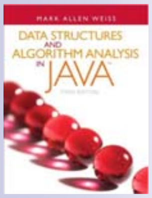
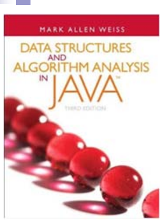


FLORIDA INTERNATIONAL UNIVERSITY

Some Problems on Graphs (II)

Dr. Antonio L. Bajuelos

FIU School of Computing & Information Sciences



COP-3530 - Data Structures

Module #7: Some Problems on Graphs (part II)

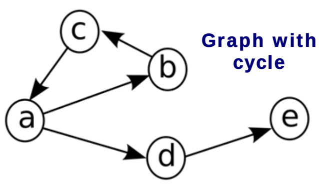
Outline:

- **Topological sort.**
- **Graphs traversals:**
  - **DFS**
  - **BFS**

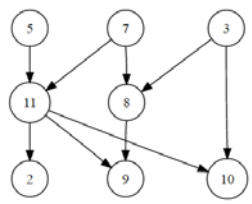
2

Graphs. Remember that …

- A **graph** G = <V,E> consists of a set of **vertices**, V, and a set of **edges**, E.
- Each edge is a pair (v,w), where v, w ∈ V. If the pair is ordered, then the graph is **directed**. Directed graphs are sometimes referred to as **digraphs**.
- Edges in the **digraphs** have a direction and in general for every (u,v) ∈ E ⇏ (v,u) ∈ E. The vertex u is the **source** and the vertex v the **destination.**
- **In-degree** of a vertex: edges where the vertex is the destination. **Out-degree** of a vertex: edges where the vertex is the source.
- A **cycle** is a path that begins and ends at the same node ($v_j = v_k$).



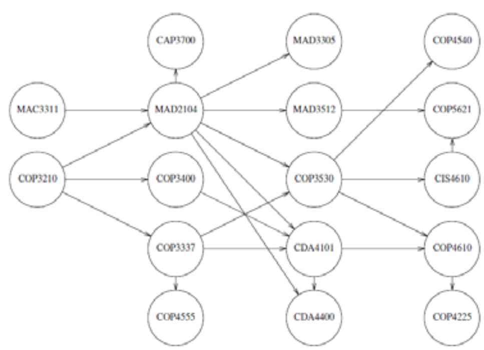
Graph with cycle



Graph without cycle (acyclic)

Topological Sort

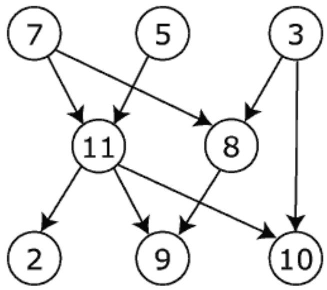
- A **topological sort** is an ordering of vertices in a **directed acyclic graph**, such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering.



- A directed edge (v,w) indicates that course v must be completed before course w may be attempted.
- A **topological ordering** of these courses is **any course sequence** that does not violate the **prerequisite requirement**.

Topological Sort (alternative definition)

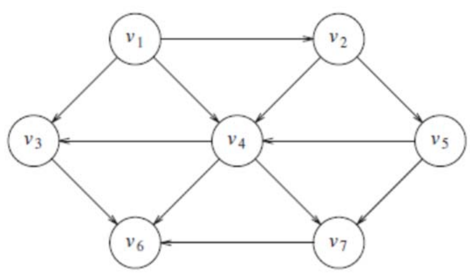
- A **topological sort** or **topological ordering** of a directed **acyclic** graph (DAG) is a linear ordering of its vertices such that for every directed edge **(u,v)** in E, the vertex **u comes before v** in the **topological ordering**.



- **Topological sort** example: **7, 5, 3, 11, 8, 2, 10, 9**

5

Topological Sort

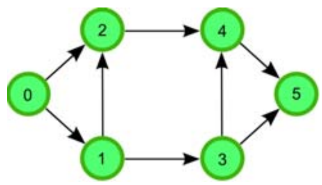


- **Topological ordering is not possible if the graph has a cycle**, since for two vertices v and w on the cycle, v precedes w and w precedes v.
- The **ordering is not necessarily unique**; any legal ordering will do.
  - Example:
    - $\mathbf{v_1, v_2, v_5, v_4, v_3, v_7, v_6}$ and $\mathbf{v_1, v_2, v_5, v_4, v_7, v_3, v_6}$ are both topological orderings.

6

Topological Sort. The First Algorithm

- **Algorithm #1**
  - Mark each vertex with its **in-degree**
    - (via a data structure e.g. array)

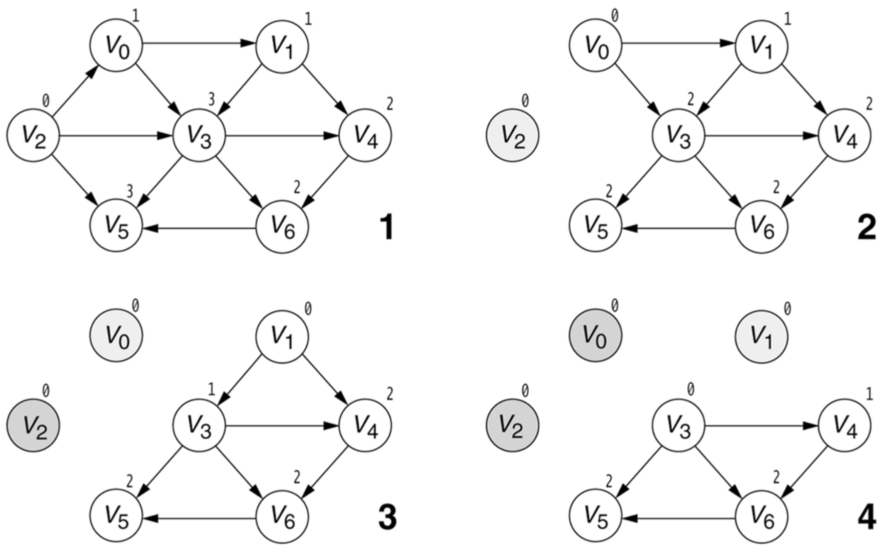


Vertex	in-degree
0	0
1	1
2	2
3	1
4	2
5	2

- While there are vertices not yet output:
  - Choose a vertex **v** with labeled with in-degree of 0
  - Output **v** and “remove” it from the graph
  - For each vertex **u** adjacent to **v,** decrement the in-degree of **u**

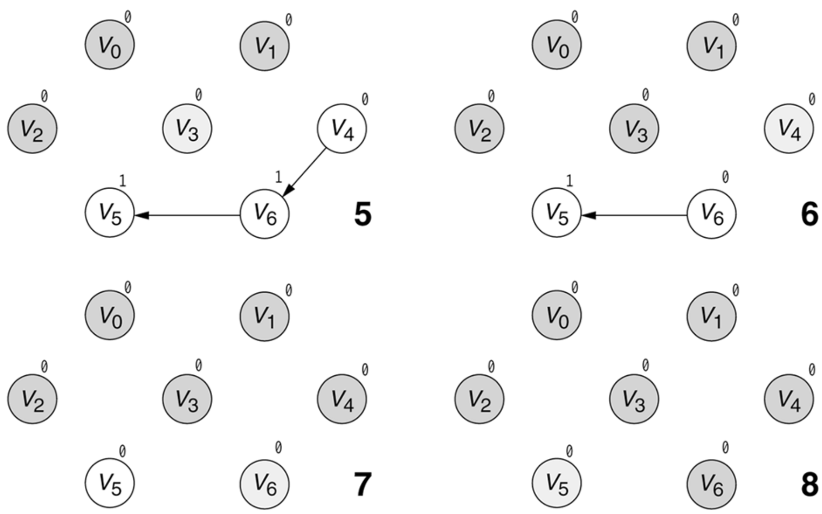
Topological Sort. The First Algorithm

- **Example**



Topological Sort. The First Algorithm

- **Example (cont…)**



9

Topological Sort. The First Algorithm

- **Algorithm Analysis**

```
labelEachVertexWithItsInDegree();
for(counter=0; counter < numVertices; counter++)
{
    v = findNewVertexOfDegreeZero();
    put v next in output
    for each w adjacent to v
        w.indegree--;
}
```

- Worst-case running time?
  - Initialization: **O(|V|+|E|)** (if we use an **adjacency list**)
  - Sum of all find-new-vertex: **O(|V|²)** (because each O(|V|))
  - Sum of all decrements: **O(|E|)** (assuming adjacency list)
  - Total: **O(|V|²)**
    - Not good for a graph with $|V|^2 >> |E|$ (**sparse graph**)

10

Topological Sort. The First Algorithm

- **Improved version. Algorithm #2**
  - The artifice is **to avoid searching for a zero-degree** vertex every time!
  - How we can do to do this?
    - Keep the “pending” zero-degree vertices in a list, stack, queue, bag, table, etc.
  - **Example:** We case use a **queue**:
    - Label each vertex with its in-degree, enqueue 0-degree vertices.
    - While queue is not empty:
      - v = **dequeue**()
      - Output v and “remove” it from the graph.
      - For each vertex u adjacent to v, decrement the in-degree of u, if new degree is 0, **enqueue** it.

Topological Sort.

- **Algorithm Analysis (improved version)**

```
labelAllAndEnqueueZeros();
for(counter=0; counter < numVertices; counter++)
{
    v = dequeue();
    put v next in output
    for each w adjacent to v
    {
        w.indegree--;
        if(w.indegree==0)
          enqueue(w);
    }
}
```

- Worst-case running time?
  - Initialization: **O(|V|+|E|)** (if we using an adjacency list).
  - Sum of all **enqueues** and **dequeues**: **O(|V|)**
  - Sum of all decrements: **O(|E|)** (assuming adjacency list)
  - Total: **O(|E| + |V|)**

Graph Traversals

- **The next problem:**
  - For an arbitrary graph and a starting vertex v, **traverse a graph**, i.e. **explore every vertex in the graph exactly once**.
  - Because there are many paths leading from one vertex to another, the hardest part about traversing a graph is making sure that you do not process some vertex twice.
- **Basic ideas:**

```
traverseGraph(Vertex start)
{
    Set pending = emptySet()
    pending.add(start)
    mark start as visited
    while(pending is not empty)
    {
      next = pending.remove()
      for each vertex u adjacent to next
        if(u is not marked)
        {
            mark u
            pending.add(u)
        }
    }
}
```


Graph Traversals

- **Important points:**
  - Note that if **add** and **remove** are O(1), entire traversal (from one vertex) is O(|E|)
  - The order we traverse depends entirely on **add** and **remove**
    - If we use and **stack**: **Depth First graph Search** (**DFS**)
    - If we use and **queue**: **Breadth First graph Search** (**BFS**)
  - So **DFS** recursively explore one part before going back to the other parts not yet explored and
  - **BFS** explore areas closer to the start node first
  - **Time complexity** of the **DFS** and **BFS** is **O(|V|+|E|)**

Graph Traversals. DFS

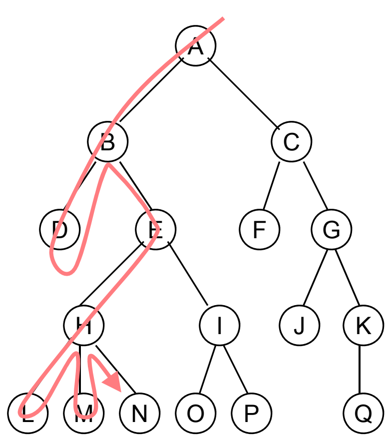
- **DFS**
  - So **DFS recursively** explore one part before going back to the other parts not yet explored
  - Pseudocode for **DFS**

```
void dfs(Vertex v)
{
   v.visited = true;
   for each vertex w adjacent to v
      if( !w.visited )
         dfs(w);
}
```

15

Graph Traversals. DFS

- **DFS**
  - So DFS **recursively** explore one part in **depth** before going back to the other parts not yet explored



- **DFS** explores a path all the way to a leaf before backtracking and exploring another path
- After searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order:

  A B D E H L M N I O P C F G J K Q

16

Graph Traversals. DFS

- **DFS (pseudocode using one stack)**

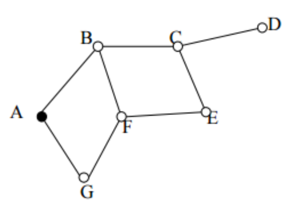
```
DFS(Vertex v)
{
    mark v visited
    print v
    make an empty stack S
    push all vertices adjacent to v onto S
    mark as visited all vertices adjacent to v
    while S is not empty do
    {
        vertex w is pop from S
        print w
        for all Vertex u adjacent to w do
        {
            if u is not visited then
            {
                mark u visited
                push u onto S
            }
        }
    }
}
```

17

Graph Traversals. DFS

- **Example of DFS using a stack**


```
DFS(Vertex v)
{
    mark v visited
    print v
    make an empty stack S
    push all vertices adjacent to v onto S
    mark as visited all vertices adjacent to v
    while S is not empty do
    {
        vertex w is pop from S
        print w
        for all Vertex u adjacent to w do
        {
            if u is not visited then
            {
                mark u visited
                push u onto S
            }
        }
    }
}
```

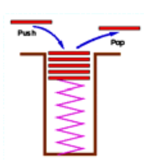



Output: A, G, F, E, C, D, B

Stack Status

S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
					D	~~D~~	~~D~~
				C	~~C~~	~~C~~	~~C~~
			E	~~E~~	~~E~~	~~E~~	~~E~~
		F	~~F~~	~~F~~	~~F~~	~~F~~	~~F~~
	G	~~G~~	~~G~~	~~G~~	~~G~~	~~G~~	~~G~~
	B	B	B	B	B	B	~~B~~

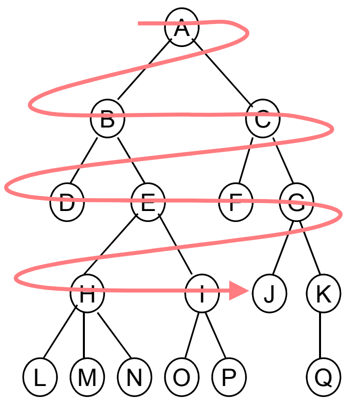
Remember that: **Stack…**

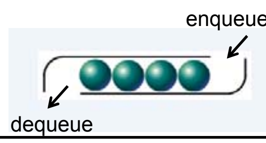


18

Graph Traversals. BFS

- **BFS** explore areas closer to the start node first



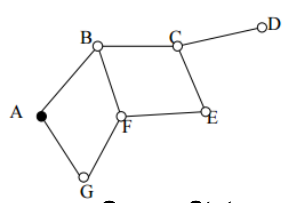


- A breadth-first search (**BFS**) explores nodes nearest the root before exploring nodes further away
- Pseudocode

```
BFS(Vertex v)
{
    make an empty queue Q
    enqueue v
    mark v as visited
    while (Q is not empty) do
    {
      vertex w is dequeue from Q
      print w
      for all Vextex u adjacent to w do
      {
        if (u is not visited) then
        {
          mark u as visited
          enqueue u onto Q
        }
      }
    }
}
```


Graph Traversals. BFS

- **Example of BFS using a queue**



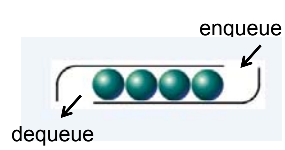
Queue Status

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
				E	E	E	~~E~~
				D	D	~~D~~	~~D~~
		F	F	F	~~F~~	~~F~~	~~F~~
		C	C	~~C~~	~~C~~	~~C~~	~~C~~
	G	G	~~G~~	~~G~~	~~G~~	~~G~~	~~G~~
	B	~~B~~	~~B~~	~~B~~	~~B~~	~~B~~	~~B~~
A	~~A~~	~~A~~	~~A~~	~~A~~	~~A~~	~~A~~	~~A~~

```
BFS(Vertex v)
{
    make an empty queue Q
    enqueue v
    mark v as visited
    while (Q is not empty) do
    {
      vertex w is dequeue from Q
      print w
      for all Vextex u adjacent to w do
      {
        if (u is not visited) then
        {
          mark u as visited
          enqueue u onto Q
        }
      }
    }
}
```

Output: A, B, G, C, F, D, E

Remember that: **Queue…**



Graph Traversals. DFS & BFS complexity

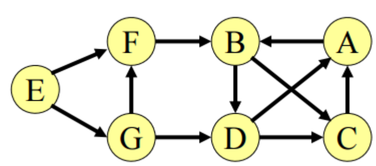
```
DFS(Vertex v)
{
   mark v visited
   print v
   make an empty stack S
   push all vertices adjacent to v onto S
   mark as visited all vertices adjacent to v
   while S is not empty do
   {
      vertex w is pop from S
      print w
      for all Vertex u adjacent to w do
      {
         if u is not visited then
         {
            mark u visited
            push u onto S
         }
      }
   }
}
```

```
BFS(Vertex v)
{
     make an empty queue Q
     enqueue v
     mark v as visited
     while (Q is not empty) do
     {
       vertex w is dequeue from Q
       print w
       for all Vextex u adjacent to w do
       {
          if (u is not visited) then
          {
            mark u as visited
            enqueue u onto Q
          }
       }
     }
}
```

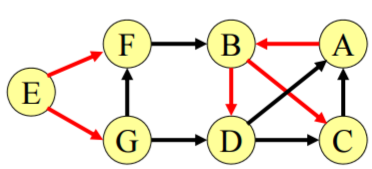
- To calculate the time complexity of the **DFS** & **BFS** algorithms, we observe that every node is "visited" exactly once during the execution of the algorithm.
- Also, every edge (u,v) is "crossed" twice:
  - one time when node v is checked from u to see if it is visited (if not visited, then v would be visited from u), and
  - another time, when we back track from v to u.
- Therefore, the running time of **DFS** and **BFS** is **O(|V| + |E|)**.

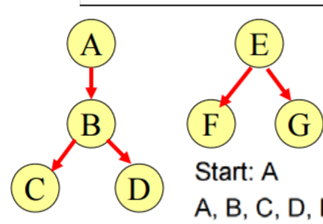
Graph Traversals. DFS & BFS for Digraphs

Not strongly connected graph!




```
DFS(Vertex v)
{
   mark v visited
   print v
   make an empty stack S
   push all vertices adjacent to v onto S
   mark as visited all vertices adjacent to v
   while S is not empty do
   {
      vertex w is pop from S
      print w
      for all Vertex u adjacent to w do
      {
         if u is not visited then
         {
            mark u visited
            push u onto S
         }
      }
   }
}
```





Start: A

A, B, C, D, E, F, G