Ian Piper
CSCI203

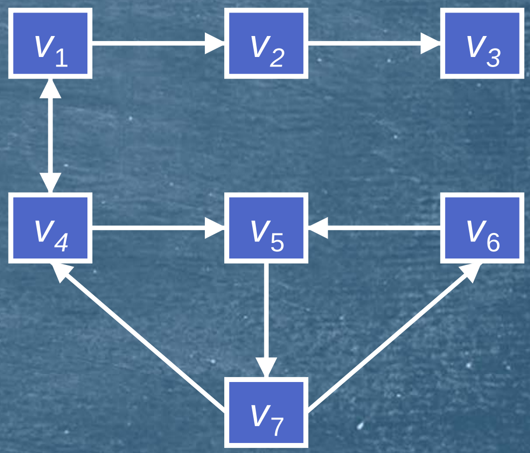
Algorithms and Data Structures

Week 8 – Lecture A

Graphs

Graphs

- A graph consists of a set, V, of points, called *Vertices* or *Nodes*, and a set, E, of *Edges*, also called *Arcs*, each of which contains a pair of vertices.
- $G=(V, E)$
  - $V=\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$
  - $E=\{\{v_1, v_2\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_4, v_5\}, \{v_4, v_7\}, \{v_5, v_6\}, \{v_5, v_7\}, \{v_6, v_7\}\}$
    - This is an *undirected* graph
  - $E=\{(v_1, v_2), (v_1, v_4), (v_2, v_3), (v_4, v_1), \{v_4, v_5\}, (v_5, v_7), (v_6, v_5), (v_7, v_4), (v_7, v_5)\}$
    - This is a *directed* graph.

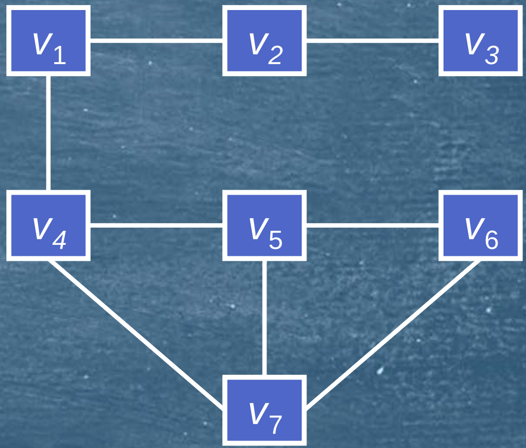


Representing a Graph

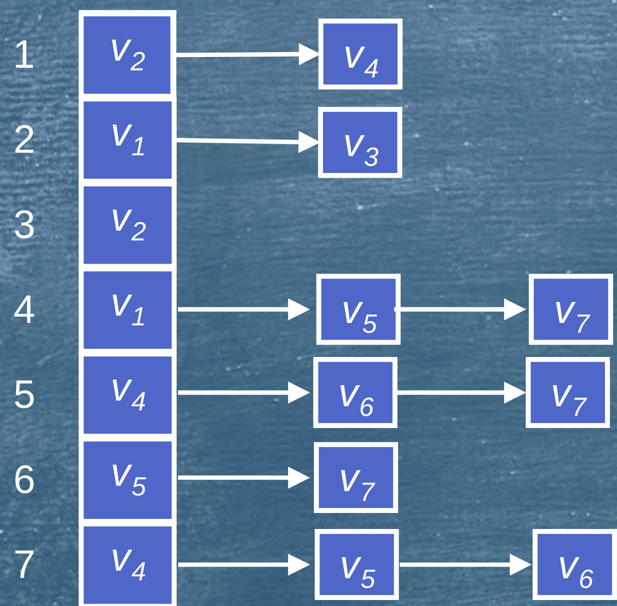
- The best way to represent a graph in a computer depends on how the graph is to be used.
- The obvious representation—an array of vertices and an array of edges—is probably the worst possible way!
- Two common representations are:
  - Adjacency List;
  - Adjacency Matrix.

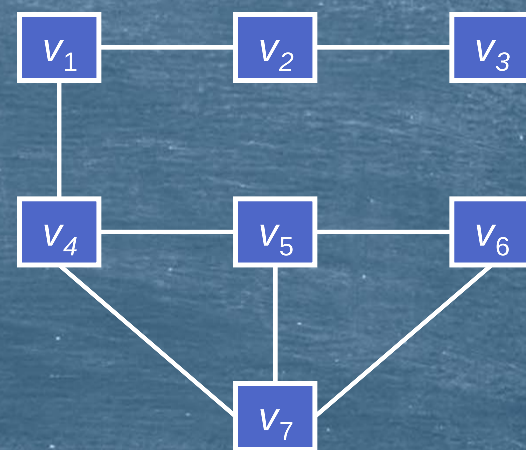
Adjacency Lists

- An adjacency list, L, is an array (or hash table), of length $|V|$, of linked lists where:
  - The list stored in L_i consists of all the vertices directly reachable from vertex i.
- The graph shown to the right…
- …has the following adjacency list representation:



Adjacency List



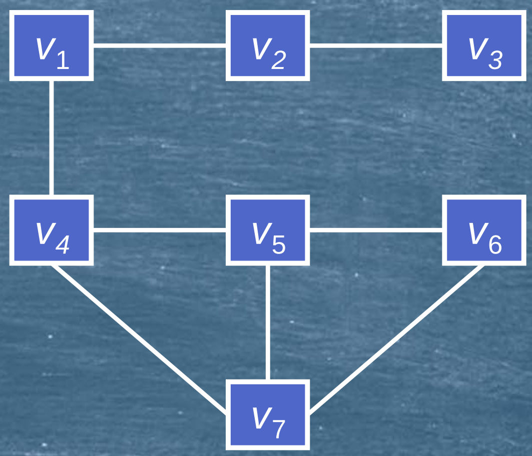


Adjacency Matrix

- An adjacency matrix is a $|V| \times |V|$ array containing zeros and ones.
  - Note: $|V|$ is the number of vertices in V.
  - A zero is stored in location (i, j) if there is no edge between v_i and v_j.
  - A one is stored in location (i, j) if there is an edge between v_i and v_j.
- Note:
  - If G is an nondirected graph, the array is symmetric.
  - $a(i, j)=a(j, i)$.

Adjacency Matrix

0	1	0	1	0	0	0
1	0	1	0	0	0	0
0	1	0	0	0	0	0
1	0	0	0	1	0	1
0	0	0	1	0	1	1
0	0	0	0	1	0	1
0	0	0	1	1	1	0



Abstract Notation

- If we ignore the detail of how, exactly, we store the graph we can represent it using the following abstract notation.
- G={Adj(v_1), Adj(v_2), ..., Adj($v_{|V|}$)}
- Here, G is defined as the addressable set of elements Adj(v_i) where Adj(v_i) is the set of vertices directly reachable from vertex v_i.
- This representation allows us to use any appropriate data structure to implement the graph;
  - Arrays;
  - Hash tables;
  - Matrices.

Graph Search

Graph Search

- A common problem involving graphs is *Graph Search*.
- This involves 'exploring' the graph in some systematic way starting at vertex *s* and visiting the other reachable vertices by following edges from one vertex to the next.
- This can be done in more than one way, as we shall see.
- Graph search has many real life applications, including:
  - Web crawling;
  - Network broadcast;
  - Social networking;
  - Garbage collection;
  - Solving puzzles and games.

Breadth-First Search (BFS)

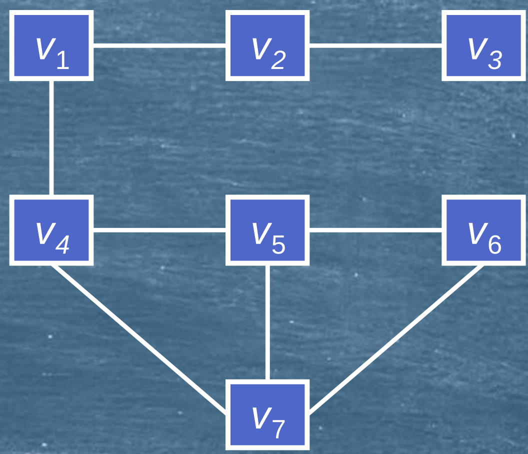
- Our goal is to list all the vertices that are reachable from some starting node $s \in V$ by following edges.
- To do this in $\Theta(|V|+|E|)$ time.
- Strategy:
  - List all the nodes reachable from s in 0 moves;
  - List all the new nodes reachable from s in 1 move;
  - List all the new nodes reachable from s in 2 moves;
  - …
  - List all the new nodes reachable from s in n moves.
- Note: a new node is one that has not already been visited.
- This avoids duplicate nodes in our result.

BFS Algorithm

```
Procedure BFS(s, adj):
    v: set of vertex
    q: queue of vertex
    v={}
    q.enqueue(s)
    while q is not empty:
         current = q.dequeue()
         add current to v
         for each n in adj(current)
              if n is not in v then
                   q.enqueue(n)
              fi
         rof
    elihw
    return v
end BFS
```

BFS, an Example

- BFS(v_1,Adj)



- Current node: **c** v_6
- Visited nodes: **v** v_1 v_2 v_4 v_3 v_5 v_7 v_6
- Pending nodes: **q** v_1 v_2 v_4 v_3 v_5 v_7 v_6

Keeping Track

- Sometimes we wish to keep track of how we got to each node in a graph.
- To achieve this we need only make a small change to the BFS algorithm.
  - We add an array, or hash table, *P*, indexed by each vertex we reached, containing the *parent* of each vertex, the vertex from which we reached it.
  - Clearly:
    - The parent of our starting vertex *s* is empty; $p(s)$ = null
    - The parent of each vertex directly connected to *s* is *s*.
  - The modified BFS is shown on the next slide.

BFS with Parents

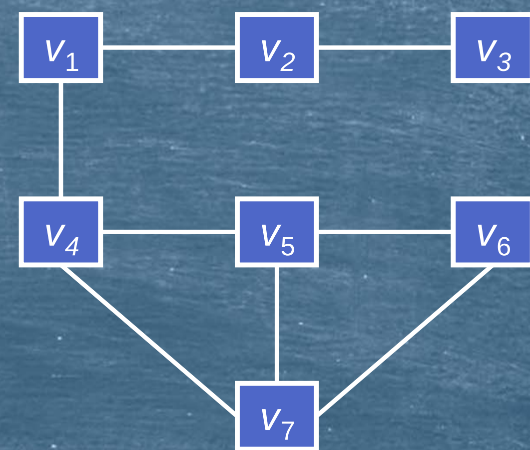
```
Procedure BFS_Parent(s, adj):
    v: set of vertex
    q: queue of vertex
    p[vertex]: array of vertex
    v={}
    q.enqueue(s)
    p[s]=null
    while q is not empty:
        current = q.dequeue()
        add current to v
        for each n in adj(current)
            if n is not in v then
                q.enqueue(n)
                p[n]=current
            fi
        rof
    elihw
    return v
end BFS_Parent
```

Using the Parents

- To find the path from vertex s to any given vertex, d, we simply list the sequence:
  - d;
  - $p[d]$
  - $p[p[d]]$
  - …
  - $p[p[\ldots p[[d]]\ldots]]$
  - Until we reach s.
- The reverse of this list is the path we seek.

BFS with Parent Tracking, an Example

- BFS(v_1,Adj)



- Current node: **c** v_6
- Visited nodes: **v** v_1 v_2 v_4 v_3 v_5 v_7 v_6
- Pending nodes: **q** v_1 v_2 v_4 v_3 v_5 v_7 v_6
- Parent node: **p** ϕ v_1 v_1 v_2 v_4 v_4 v_5

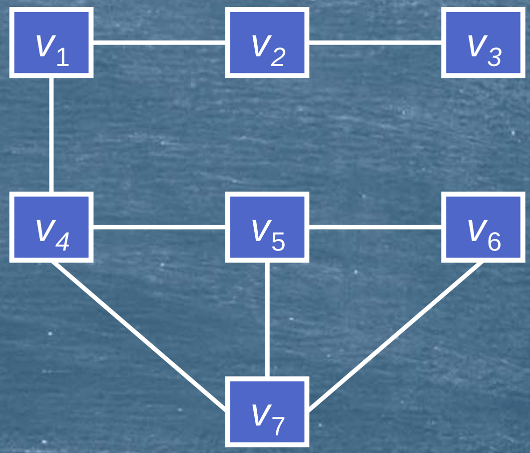
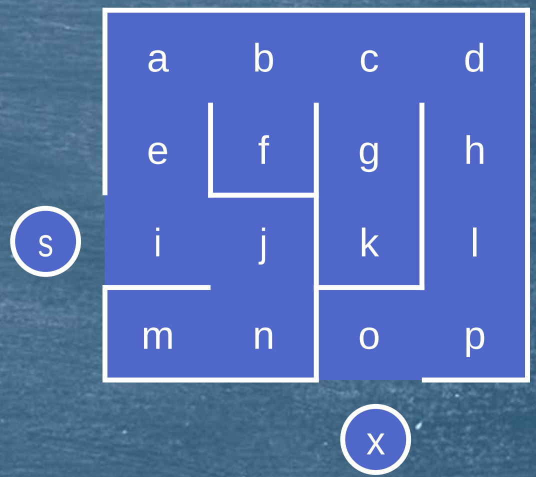
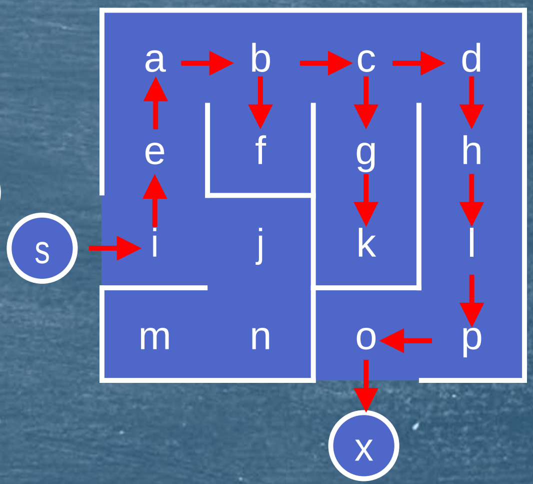
Efficiency of BFS

- We note that, in the worst case, we visit all vertices in the graph.
- For each vertex in the graph we traverse each of its edges.
- The complexity of BFS is, therefore, $O(|V| + |E|)$.
  - Strictly speaking, for undirected graphs, $O(|V| + 2\times|E|)$
  - For directed graphs, $O(|V| + |E|)$.

Depth First Search (DFS)

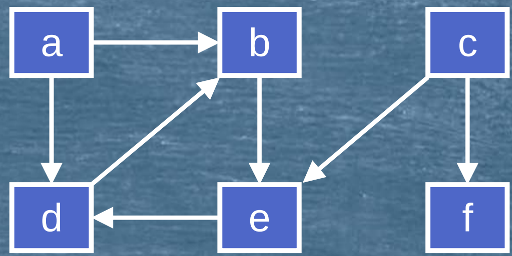
- The goal of DFS is the same as BFS:
  - List all vertices reachable from the starting vertex *s*.
- Strategy:
  - For each vertex *v* in Adj(*s*);
    - Perform DFS on *v*.
  - This is a recursive implementation of DFS.
- We can also perform DFS using a non-recursive algorithm.
- All we need to do is replace the queue from BFS with a stack.

DFS Algorithm

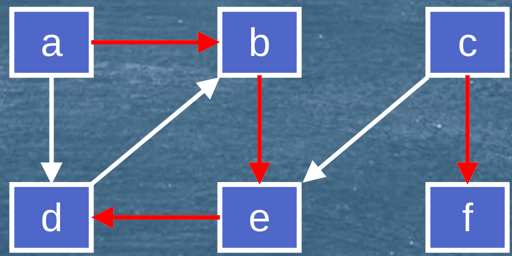
- ```
  Procedure DFS(s, adj):
       v: set of vertex
       k: stack of vertex
       p[vertex]: array of vertex
       v={}
       k.push(s)
       p[s]=null
       while k is not empty:
             current = k.pop()
             add current to v
             for each n in adj(current)
                   if n is not in v then
                         k.push(n)
                         p[n]=current
                   fi
             rof
       elihw
       return v
  end DFS
  ```
  ```

DFS, an Example

- DFS(v_1,Adj)



- Current node: **c** v_3
- Visited nodes: **v** v_1 v_4 v_7 v_6 v_5 v_2 v_3
- Pending nodes: **k** v_3 v_5 v_6

Some Notes

- BFS and DFS will visit the same vertices given the same starting vertex.
  - All that changes is the order they are visited.
- The edges traversed in performing a DFS or BFS form a tree.
  - If the graph is connected this is a *spanning* tree.
- BFS and DFS work on directed as well as on undirected graphs.

Traversing a Maze

- If we modify DFS to track parents we can use it as a means of finding our way through a maze.
- Consider the following maze:
- We can represent it as a graph with 18 vertices, a, b, ..., p, s and x.
- An edge exists in this graph if two vertices are adjacent and do not have a wall between them.
- We start the maze at vertex s and exit to vertex x.



A(maze)d

Traversing the Maze

- If we try directions in the sequence:
  - Down;
  - Up;
  - Left;
  - Right.
  - (we need to stack in the reverse order; R, L, U, D)
- The DFS proceeds as follows:



When DFS Fails

- If the graph is not connected, or is directed and not strongly connected, we may not reach every vertex with a single call to DFS.
- In this case we will need to call DFS repeatedly, once for each vertex we have not yet visited.
- We can do this easily with the following procedure, DFS_All:

DFS_All

- ```
  Procedure DFS_All(g, adj)
      v: set of vertex
      v={}
      for each s in g
          if s not in v then
              DFS(s, adj)
          fi
      rof
  end DFS_All
  ```
- Note: we must remove the line **v={}** from procedure DFS.
  ```

DFS_All, an Example

- Consider the following graph:



- If we run DFS_All in alphabetical order we get the following result:



- v: | a | b | e | d | c | f |
- p: | ϕ | a | b | e | ϕ | c |

So, Why Both?

- If both BFS and DFS can be used to search a graph why bother with both of them.
- The answer is simply that, although they both perform graph search, they provide different information about the graph.
- For example, BFS will always find the *shortest* path from *s* to *x*, assuming a path exists, while DFS will not.
  - Shortest is the fewest vertices in the path.
- Many applications of DFS rely on the way it can be used to classify edges.

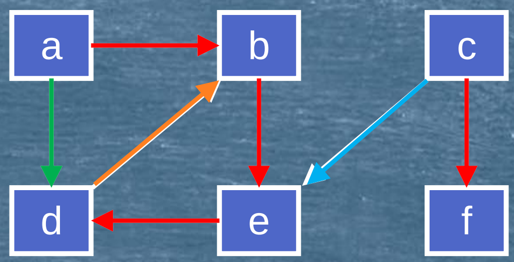
Edge Classification

Edge Classification

- If we perform a DFS on a graph we can classify the edges of a graph:
  - Tree edges: these form part of the search tree (or forest);
  - Forward edges: these lead from a vertex to a descendant;
  - Backward edges: these lead from a vertex to an ancestor;
  - Cross edges: these are all the edges that are left—they connect unrelated vertices.

Edge Classification: an Example

- In the Graph shown to the right;
- The edges are classified as follows:
- Tree edges (in red);
- Forward edges (in green);
- Backward edges (in orange);
- Cross edges (in cyan).



How to Tell

- How do we determine the type of an edge in our DFS procedure?
- The parent structure, p[], gives us the tree edges.
  - If p[i] =j then the edge from vertex j to vertex i is a tree edge.
- If we encounter a vertex that is currently in the recursion stack, k, then the corresponding edge is a backward edge.

How to Tell

- To identify forward and cross edges we need to record *when* we visit a vertex—the sequence in which the vertices are pushed onto the vertex stack, k.
- With this sequence number we can determine whether a remaining edge is a forward or a cross edge:
  - If the sequence number of the vertex we are visiting is larger than the sequence number of the current vertex this is a forward edge;
  - Otherwise this is a cross edge.
- We could also record the finishing time of the vertices—the sequence in which the vertices are popped from the vertex stack, k.

DFS Algorithm

```
Procedure DFS_Timed(s, adj):
     in: static integer = 0
     out:static integer = 0
     v: set of vertex
     k: stack of vertex
     p[vertex]: array of vertex
     k.push(s)
     s.start=in++
     p[s]=null
     while k is not empty:
            current = k.pop()
            s.end=out++
            add current to v
            for each n in adj(current)
                   if n is not in v then
                          k.push(n)
                          n.start=in++
                          p[n]=current
                   fi
            rof
     elihw
     return v
end DFS_Timed
```

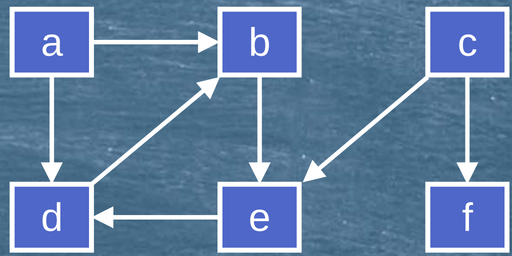
Why?

- Why is edge classification important?
- We will consider two problems in graph theory:
  - Cycle detection;
  - Topological sorting.
- Each of these depends on edge classification for its solution.

Cycle Detection

Cycle Detection

- We define a cycle as any sequence of edges that form a closed loop in a graph.
- E.g. in the graph to the right…
- The edges (b, e), (e, d) and (d, b) form a cycle.



- Finding these cycles is trivial once we have classified all the edges:
  - *G* has a cycle if, and only if, DFS(*G*, Adj) contains at least one backward edge.

Finding Cycles

- Once we find a backward edge, (v_0, v_k), finding the cycle it forms part of is easy:
  - Simply follow the parent sequence, back from v_0 until we reach v_k.
  - These edges plus the backward edge must form the cycle.
- Our next problem, *Topological Sort*, requires that the graph be acyclic:
  - It has no cycles.
  - We now know how to do this.
- We will look at topological sort by way of a simple application, *job scheduling*.

Topological Sort

Job Scheduling.

- Consider a directed, acyclic graph (DAG) in which:
  - The vertices each represent a task to be performed;
  - The edges represent an order in which the tasks may be performed.
    - If edge (v_i, v_j) exists in the graph, task v_i must be completed before we start task v_j.
- Our problem is to find a *feasible sequence* of tasks:
  - An order in which the tasks may be performed which does not conflict with the order required by all of the edges.
- You can only do one task at a time.
- Consider the task of getting dressed:

Getting Dressed: Vertices

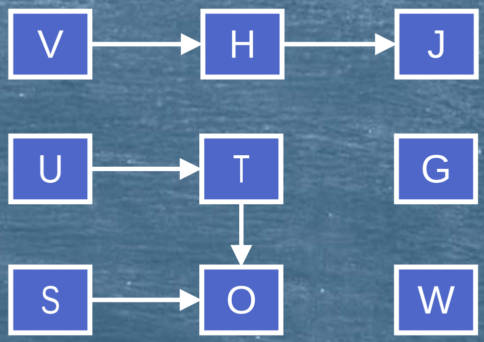
- We can label our tasks (vertices), in alphabetical order:
  - G: put on glasses;
  - H: put on shirt;
  - J: put on jacket;
  - O: put on shoes;
  - S: put on socks;
  - T: put on trousers.
  - U: put on underpants;
  - V: put on vest;
  - W: put on watch;
- We can then determine the edges:

Getting Dressed: Edges

- We can define our edges as follows:
  - (V, H): vest before shirt;
  - (U, T): underpants before trousers;
  - (S, O): socks before shoes;
  - (T, O): trousers before shoes:
  - (H, J): shirt before jacket.
- Note vertices G and W do not appear in the edge list:
  - We can do these tasks at any time.
- Taken together, the vertices and edges form a DAG.

Getting Dressed: the Graph

- The resulting graph is shown to the right:

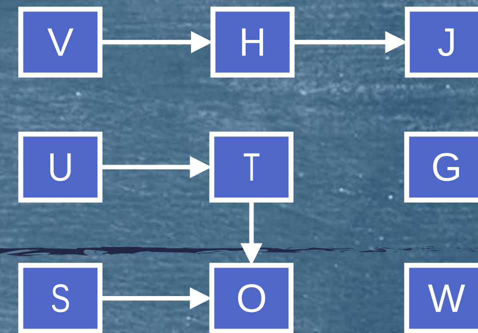


- We perform our topological sort as follows:
  - Conduct a DFS of the graph;
  - List the vertices in reverse order of finishing time.
- In practice we construct a list of the vertices as each is popped from the stack and then print the list backwards.

Why is it So?

- We can prove that the procedure we describe produces a correct ordering:
- Theorem: if G contains the edge (u, v) then vertex v will be popped before vertex u.
- Proof: consider two cases.
  - Case 1: u starts before v.
    - In this case we must visit v before we are finished with u.
    - Because edge (u, v) exists, vertex v will be visited while vertex u is still on the stack
  - Case 2: u starts after v.
    - In this case we cannot visit v while u is still on the stack.
    - Because the graph is acyclic, there is no path from v to u.

Topological Sort: an Example



- Given the “getting dressed” graph:
- Our DFS (performed in alphabetical order) goes like this:
- Push G: no edges—pop G.
- Push H: edge (H, J)
  - Push J: no edges—pop J
- Back at H: no more edges—pop H
- Push O: no edges—pop O
- Push S: no unvisited vertices—pop S
- Push T: no unvisited vertices—pop T
- Push U: no unvisited vertices—pop U
- Push V: no unvisited vertices—pop V
- Push W: no edges—pop W
- Our topological sort is:
  W, V, U, T, S, O, H, J, G.

One of Many.

- The example we just saw gives us one of many possible topological sorts for the “getting dressed” graph.
- If we visited the vertices in a different order we would probably get a different sort order.
- E.g. reverse alphabetical order results in the topological sort G, S, U, T, O, V, H, J, W.
- Verify:
  - This is really what you get;
  - This is a valid sequence of operations.