CSCI203

Algorithms and Data Structures

Graphs (I)

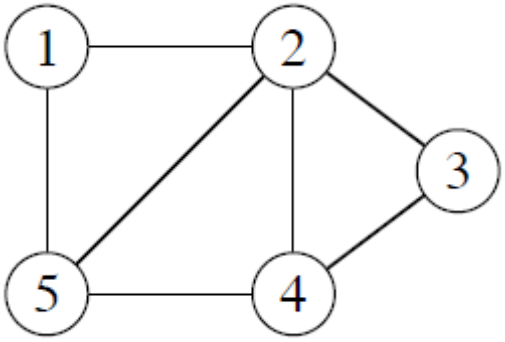
Lecturer: Dr. Fenghui Ren

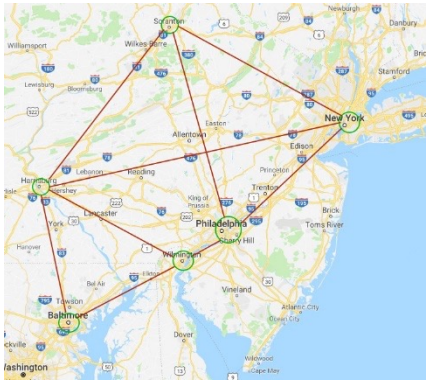
Room 3.203

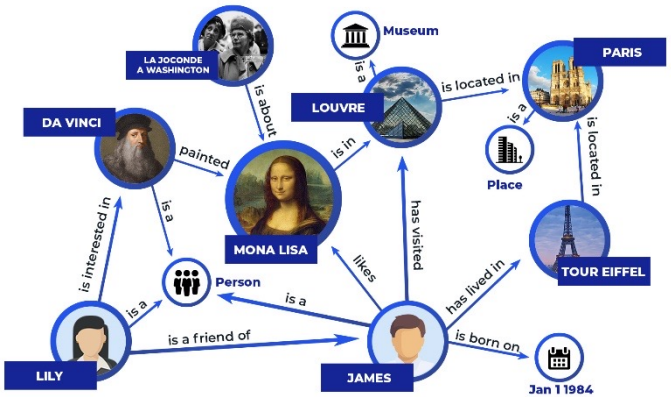
Email: fren@uow.edu.au

Graphs

- A graph, $G = (V, E)$, 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

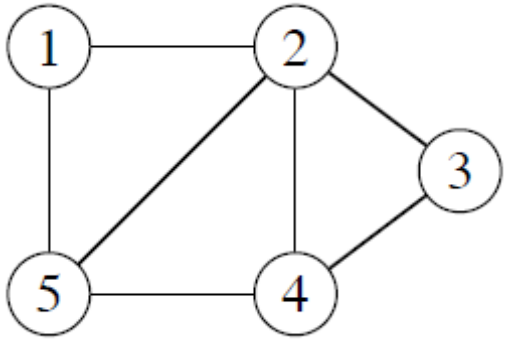


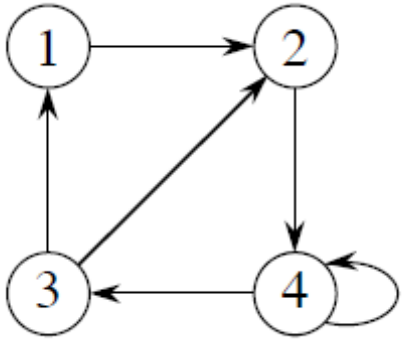




Graph $G = (V, E)$

- Undirected graph
  - $V = \{v_1, v_2, v_3, v_4, v_5\}$
  - $E = \{\{v_1, v_2\}, \{v_1, v_5\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_2, v_5\}, \{v_3, v_4\}, \{v_4, v_5\}\}$
- Directed graph
  - $V = \{v_1, v_2, v_3, v_4\}$
  - $E = \{(v_1, v_2), (v_2, v_4), (v_3, v_1), (v_3, v_2) (v_4, v_3), (v_4, v_4)\}$





Sparse vs Dense Graphs

- Sparse graphs
  - Those for which $|E|$ is much less than $|V|^2$
- Dense graphs
  - Those for which $|E|$ is close to $|V|^2$

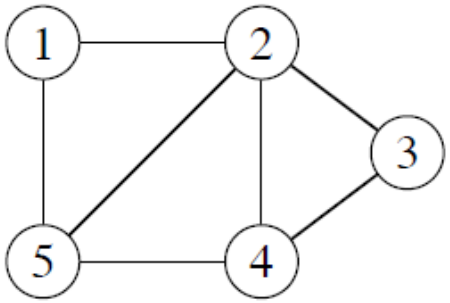
- When expressing the running time of an algorithm, it is often in terms of both $|V|$ and $|E|$. In asymptotic notion, we drop the cardinality, e.g. $O(V + E)$

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.

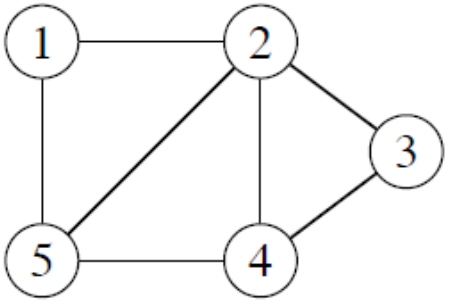
Adjacency Lists

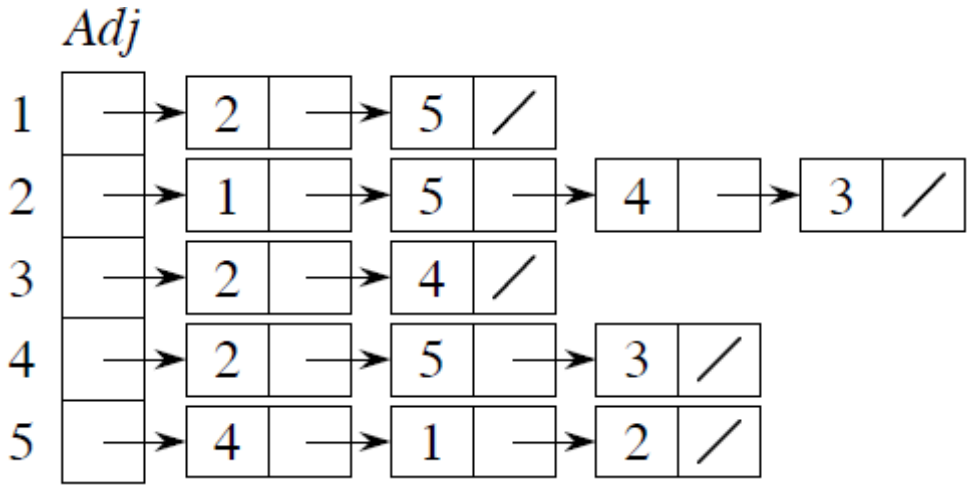
- An adjacency list, L, is an array (or hash table), of length $|V|$, of linked lists where:
  - The list stored in L_i consists of all the vertices directly reachable from vertex i.



- The graph shown to the right…
- … has the following adjacency list representation:

Adjacency List

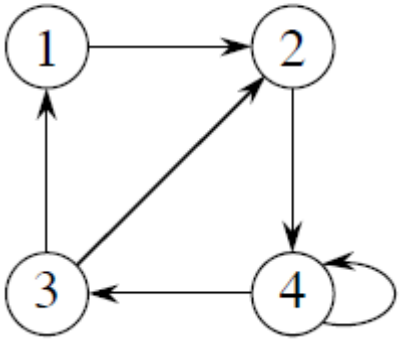


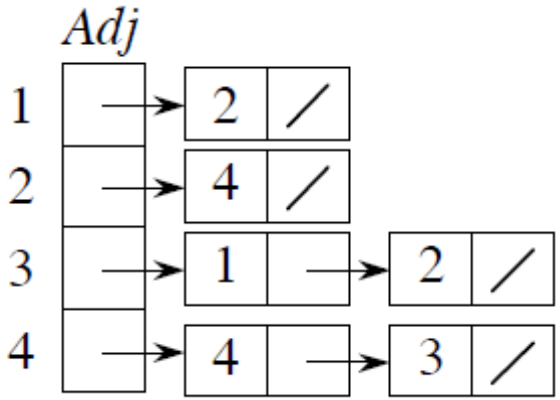


- Array Adj of $|V|$ lists, one per vertex. Vertex u's list has all vertices v such that $(u, v) \in E$. (Works for both directed and undirected graphs.)

- Space: $\Theta(V + E)$

Adjacency List





- Space complexity is the same as an undirected graph

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.

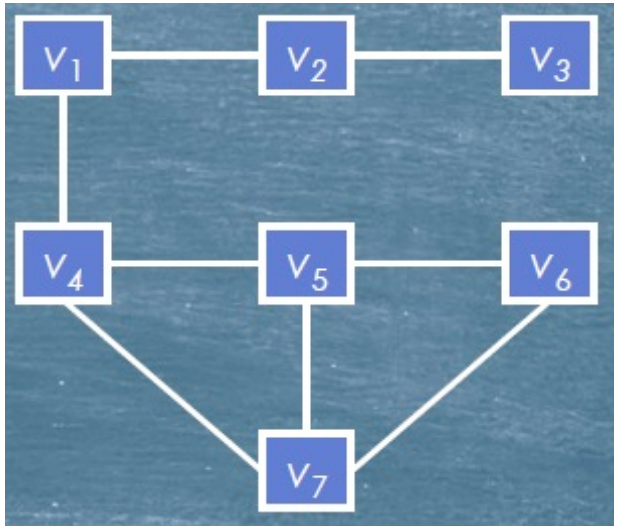
$$|V| \times |V| \text{ matrix } A = (a_{ij})$$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise}. \end{cases}$$

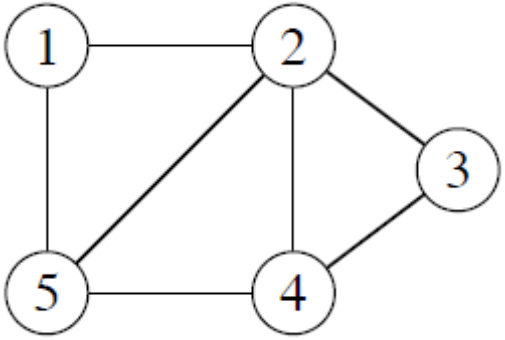
- Note: If G is a non-directed graph, the array is symmetric

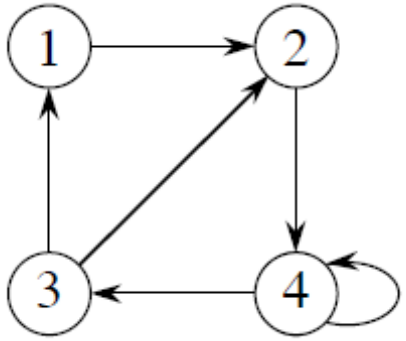
  i.e., $a(i, j) = a(j, i)$.
- Space: $\Theta(|V|^2)$

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





	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1

Abstract Notation

- If we ignore the detail of how, exactly, we store the graph we can represent it using the following abstract notation.
- $G = \{Adj(v_1), Adj(v_2), \ldots, Adj(v_{|V|})\}$
- 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

- 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)

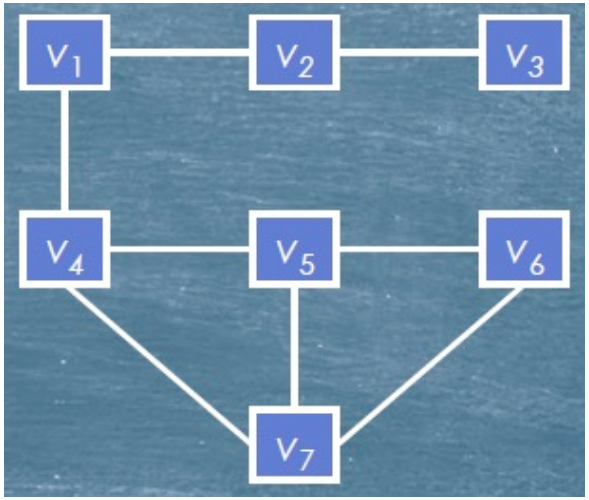
- Our goal is to list all the vertices that are reachable from some starting node $s \in V$ by following edges.
- To do this in $\Theta(|V| + |E|)$ time.
- Strategy:
  - List all the nodes reachable from s in 0 moves;
  - List all the new nodes reachable from s in 1 move;
  - List all the new nodes reachable 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 vertices
        q: queue of vertices
        c[vextex]: visited or not
        v={}
        q.enqueue(s); c[s] = visited
        while q is not empty:
                current = q.dequeue()
                add current to v
                for each n in adj(current)
                        if n is not in v and not in c then
                                q.enqueue(n)
                                c[n]=visited
                        fi
                rof
        elihw
        return v
end BFS
```

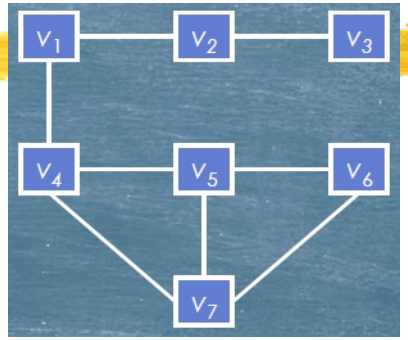
BSF, an Example

- BFS(v_1,Adj)



- Current node: c
- 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

BSF, an Example



c	v	q
	$\{\}$	$\{v_1\}$
v_1	$\{v_1\}$	$\{v_2, v_4\}$
v_2	$\{v_1, v_2\}$	$\{v_4, v_3\}$
v_4	$\{v_1, v_2, v_4\}$	$\{v_3, v_5, v_7\}$
v_3	$\{v_1, v_2, v_4, v_3\}$	$\{v_5, v_7\}$
v_5	$\{v_1, v_2, v_4, v_3, v_5\}$	$\{v_7, v_6\}$
v_7	$\{v_1, v_2, v_4, v_3, v_5, v_7\}$	$\{v_6\}$
v_6	$\{v_1, v_2, v_4, v_3, v_5, v_7, v_6\}$	$\{\}$

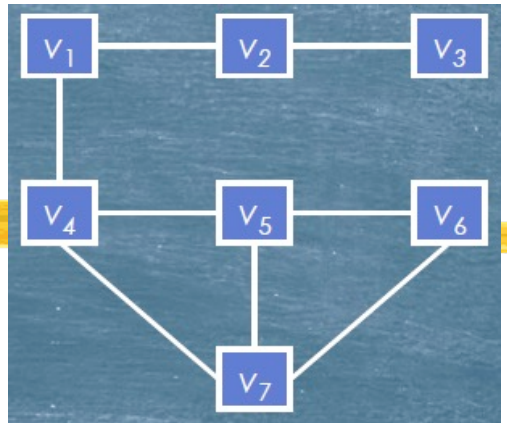
Keeping Track

- 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
        c[vextex]: visited or not
        p[vertex]: array of vertex
        v={}
        q.enqueue(s); c[s]=visited
        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 and not visited then
                                q.enqueue(n)
                                c[n] = visited
                                p[n]=current
                        fi
                rof
        elihw
        return v
end BFS_Parent
```

BFS with Parent Tracking, an Example



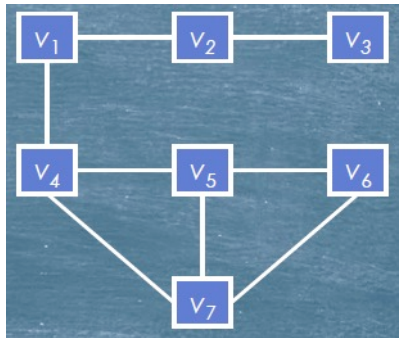
BFS(v_1,Adj)

c	v	q	p
	$\{\}$	$\{v_1\}$	$\{\phi,?,?,?,?,?,?\}$
v_1	$\{v_1\}$	$\{v_2,v_4\}$	$\{\phi,v_1,?,v_1,?,?,?\}$
v_2	$\{v_1,v_2\}$	$\{v_4,v_3\}$	$\{\phi,v_1,v_2,v_1,?,?,?\}$
v_4	$\{v_1,v_2,v_4\}$	$\{v_3,v_5,v_7\}$	$\{\phi,v_1,v_2,v_1,v_4,?,v_4\}$
v_3	$\{v_1,v_2,v_4,v_3\}$	$\{v_5,v_7\}$	$\{\phi,v_1,v_2,v_1,v_4,?,v_4\}$
v_5	$\{v_1,v_2,v_4,v_3,v_5\}$	$\{v_7,v_6\}$	$\{\phi,v_1,v_2,v_1,v_4,v_5,v_4\}$
v_7	$\{v_1,v_2,v_4,v_3,v_5,v_7\}$	$\{v_6\}$	$\{\phi,v_1,v_2,v_1,v_4,v_5,v_4\}$
v_6	$\{v_1,v_2,v_4,v_3,v_5,v_7,v_6\}$	$\{\}$	

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[…p[[d]]…]]`
  - Until we reach s.



$\{\phi, v_1, v_2, v_1, v_4, v_5, v_4\}$

- The reverse of this list is the path we seek.

Efficiency and Property 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, $O(|V| + 2x|E|)$
  - For directed graphs, $O(|V| + |E|)$.

- 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.

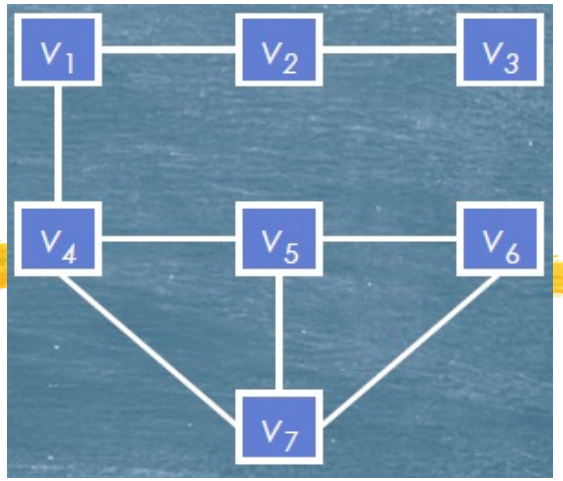
Depth First Search (DFS)

- 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
        c[vextex]: visited or not
        p[vertex]: array of vertex
        v={}
        k.push(s); c[s]=visited;
        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 and not visited then
                                k.push(n)
                                p[n]=current
                                c[n] = visited
                        fi
                rof
        elihw
        return v
end DFS
```

DFS, An Example



- DFS(v_1,Adj)

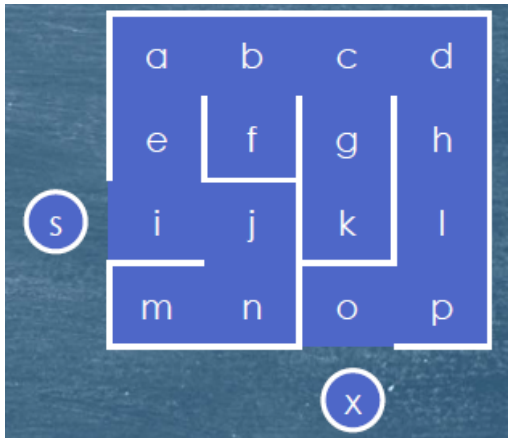
	c	v	k	p
→		$\{\}$	$\{v_1\}$	$\{\phi,?,?,?,?,?,?\}$
→	v_1	$\{v_1\}$	$\{v_2,v_4\}$	$\{\phi,v_1,?,v_1,?,?,?\}$
→	v_4	$\{v_1,v_4\}$	$\{v_2,v_5,v_7\}$	$\{\phi,v_1,?,v_1,v_4,?,v_4\}$
→	v_7	$\{v_1,v_4,v_7\}$	$\{v_2,v_5,v_6\}$	$\{\phi,v_1,?,v_1,v_4,v_7,v_4\}$
→	v_6	$\{v_1,v_4,v_7,v_6\}$	$\{v_2,v_5\}$	$\{\phi,v_1,?,v_1,v_4,v_7,v_4\}$
→	v_5	$\{v_1,v_4,v_7,v_6,v_5\}$	$\{v_2\}$	$\{\phi,v_1,?,v_1,v_4,v_7,v_4\}$
→	v_2	$\{v_1,v_4,v_7,v_6,v_5,v_2\}$	$\{v_3\}$	$\{\phi,v_1,v_2,v_1,v_4,v_7,v_4\}$
→	v_6	$\{v_1,v_4,v_7,v_6,v_5,v_2,v_3\}$	$\{\}$	

Some Notes

- BFS and DFS will visit the same set of 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.

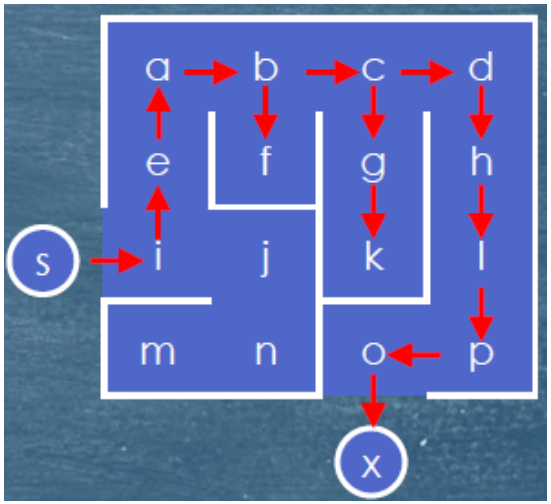
Traverse a Maze

- 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, $a, b, \ldots, p, s \; and \; x$.
- 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.



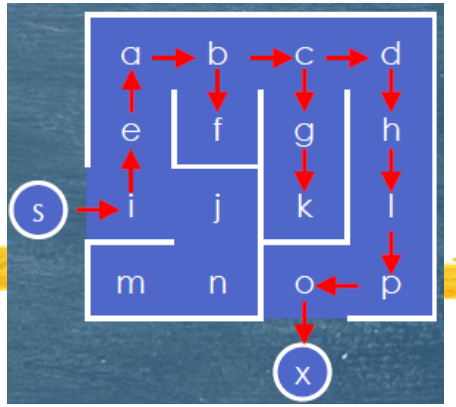
Traversing the Maze

- 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:



DFS, An Example

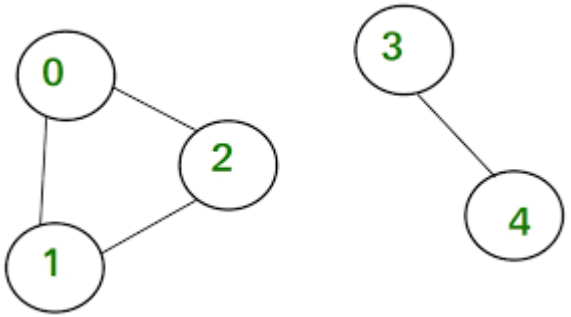
- DFS(s,Adj)
  - Down, Up, Left, Right.



c	k	p
	$\{s\}$	$[s,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,x]$ $\{\phi,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?\}$
s	$\{i\}$	$\{\phi,?,?,?,?,?,?,?,?,s,?,?,?,?,?,?,?,?\}$
i	$\{j,e\}$	$\{\phi,?,?,?,?,i,?,?,?,s,?,?,?,?,?,?,?,?\}$
e	$\{j,a\}$	$\{\phi,e,?,?,?,i,?,?,?,s,?,?,?,?,?,?,?,?\}$
a	$\{j,b\}$	$\{\phi,e,a,?,?,i,?,?,?,s,?,?,?,?,?,?,?,?\}$
b	$\{j,c,f\}$	$\{\phi,e,a,b,?,i,b,?,?,s,?,?,?,?,?,?,?,?\}$
f	$\{j,c\}$	$\{\phi,e,a,b,?,i,b,?,?,s,?,?,?,?,?,?,?,?\}$
c	$\{j,d,g\}$	$\{\phi,e,a,b,c,i,b,c,?,s,?,?,?,?,?,?,?,?\}$
…	…	…

DFS of $G(V, E)$ – $DFS\_ALL(G)$

- 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, $DFS\_ALL(G)$:
- DSF of a graph can reveal important information of the structure of the graph



$DSF\_ALL(G)$

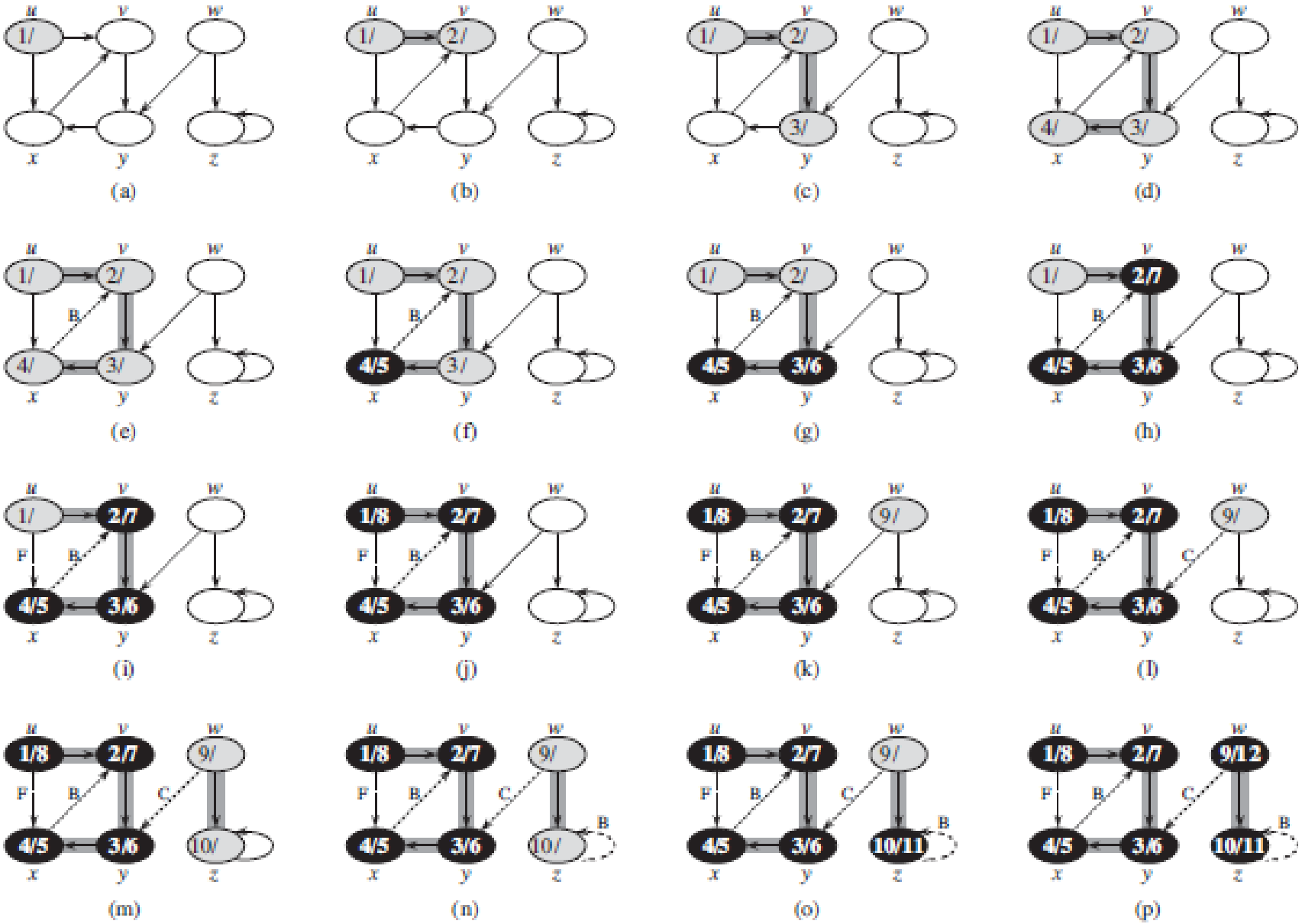
- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:** 2 timestamps on each vertex:
  - d[v] = discovery/exploring time
  - f [v] = finishing time
  - These will be useful for other algorithms
- Will methodically explore every edge.
  - Start over from different vertices as necessary.
- As soon as we discover a vertex, explore from it.
  - Unlike BFS, which puts a vertex on a queue so that we explore from it later.
- As DFS progresses, every vertex has a color:
  - WHITE = undiscovered
  - GRAY = discovered, but not finished (not done exploring from it)
  - BLACK = finished (have found everything reachable from it)
- Discovery and finish times:
  - Unique integers from 1 to $2|V|$.
  - For all v, $d[v] < f[v]$.
  - In other words, $1 \leq d[v] < f[v] \leq 2|V|$.

$DSF\_ALL(G)$

- Uses a global timestamp $time$.

```
DFS_ALL(G)
    for each u ∈ V
        do color[u] ← WHITE
time ← 0
for each u ∈ V
        do if color[u] = WHITE then DFS-VISIT(u)

DFS-VISIT(G,u)
color[u] ← GRAY // discover u
time ← time+1
d[u] ← time
for each v ∈ Adj[u] // explore (u, v)
    do if color[v] = WHITE then DFS-VISIT(v)
color[u] ← BLACK    //
time ← time+1
f[u] ← time                 //finish u
```

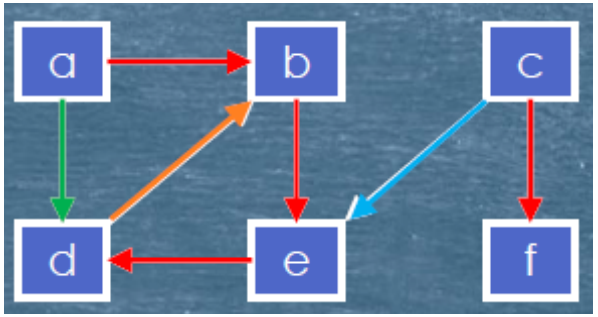

u v w
1/
x y z
(a)
1/ 2/
(b)
1/ 2/ 3/
(c)
1/ 2/ 4/ 3/
(d)
B
(e)
4/5 3/
(f)
4/5 3/6
(g)
2/7
(h)
F
(i)
1/8
(j)
9/
(k)
C
(l)
10/
(m)
(n)
10/11
(o)
9/12
(p)

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.
- Vertex v is descendant of vertex u in depth first search iff v is discovered during the time which u is gray

Edge Classification: an Example

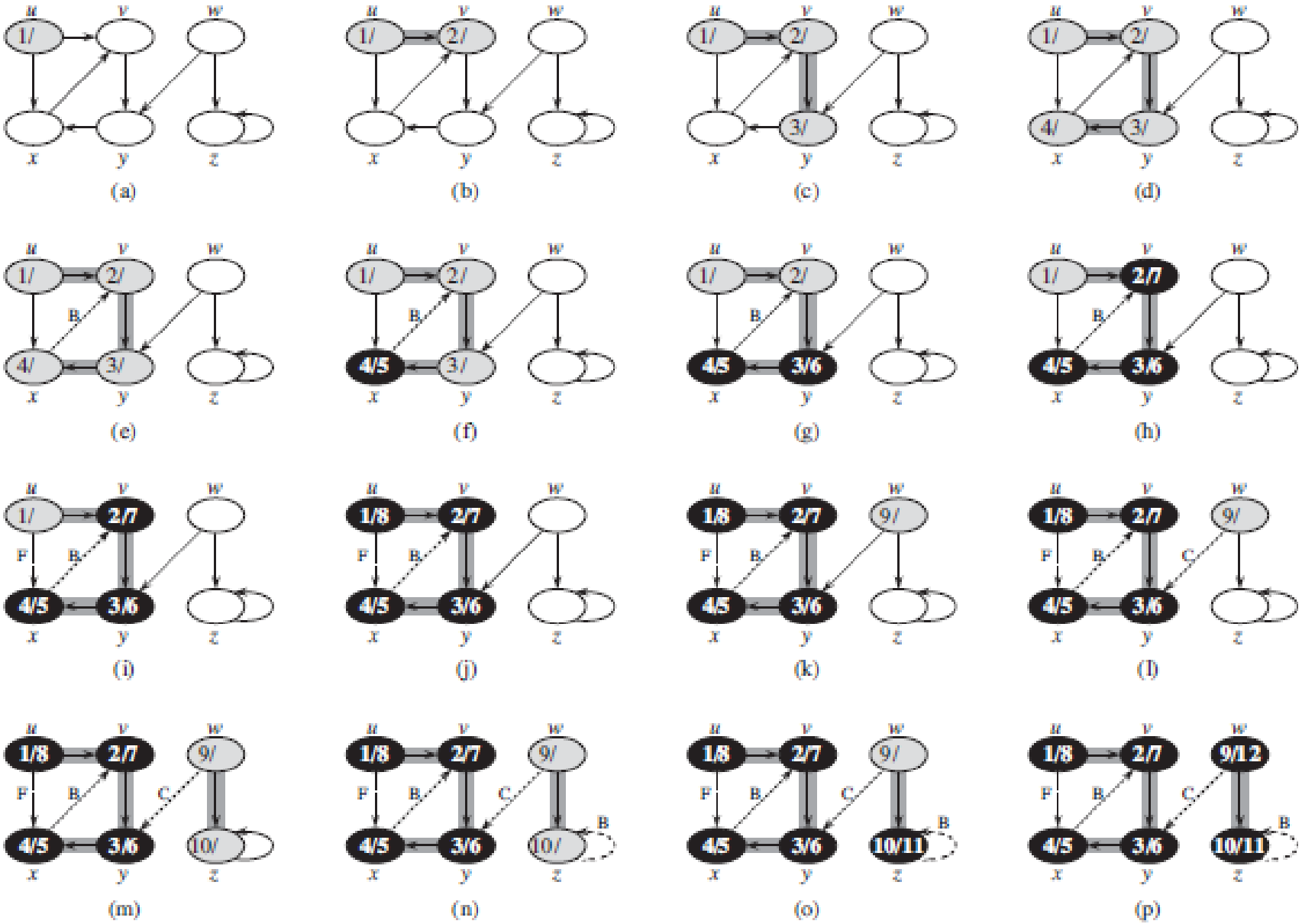
- In the Graph shown to the right;
- The edges are classified as follows:
  - Tree edges (in red);
  - Forward edges (in green);
  - Backward edges (in orange);
  - Cross edges (in cyan).



How to Tell?

- `DFS_ALL(G)` has enough information to classify some edges as it encounters them

- The key idea is that wen we first explore an edge (u, v), the color of vertex v tell us something about the edge (u, v):
  - WHITE indicates a tree edge (in the search tree)
  - GRAY indicates a back edge, and
  - BLACK indicate a forward or cross edge
    - $d[u] < f[v]$ indicates forward edge
    - $d[u] > f[v]$ indicates cross edge

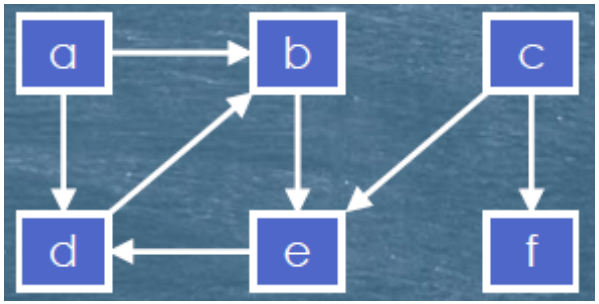

u v w
1/
x y z
(a)
1/ 2/
(b)
1/ 2/ 3/
(c)
1/ 2/ 4/ 3/
(d)
1/ 2/ B 4/ 3/
(e)
1/ 2/ B 4/5 3/
(f)
1/ 2/ B 4/5 3/6
(g)
1/ 2/7 B 4/5 3/6
(h)
1/ 2/7 F B 4/5 3/6
(i)
1/8 2/7 F B 4/5 3/6
(j)
1/8 2/7 9/ F B 4/5 3/6
(k)
1/8 2/7 9/ F B C 4/5 3/6
(l)
1/8 2/7 9/ F B C 4/5 3/6 10/
(m)
1/8 2/7 9/ F B C 4/5 3/6 10/ B
(n)
1/8 2/7 9/ F B C 4/5 3/6 10/11 B
(o)
1/8 2/7 9/12 F B C 4/5 3/6 10/11 B
(p)

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

- 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:

Topological sort- Algorithm

- TOPOLOGICAL-SORT(G)
  1. call DFS_ALL(G) to compute finishing times $f[v]$ for each vertex v
  2. as each vertex is finished, insert it onto the front of a linked list
  3. return the linked list of vertices

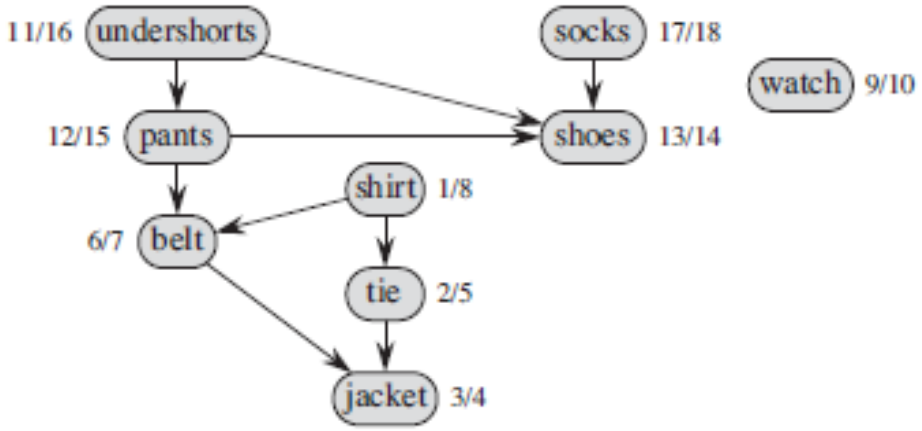
- We can perform a topological sort in time $\Theta(V+E)$ since depth-first search takes $\Theta(V+E)$ and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list

Getting Dressed: Vertices

- We can label our tasks (vertices), in alphabetical order:
  - Put on undershorts
  - Put on pants
  - Pub on belt
  - Put on shirt
  - Put on tie
  - Put on jacket
  - Put on socks
  - Pub on shoes
  - Put on watch
- We can then determine the edges:

Getting Dressed: Edges

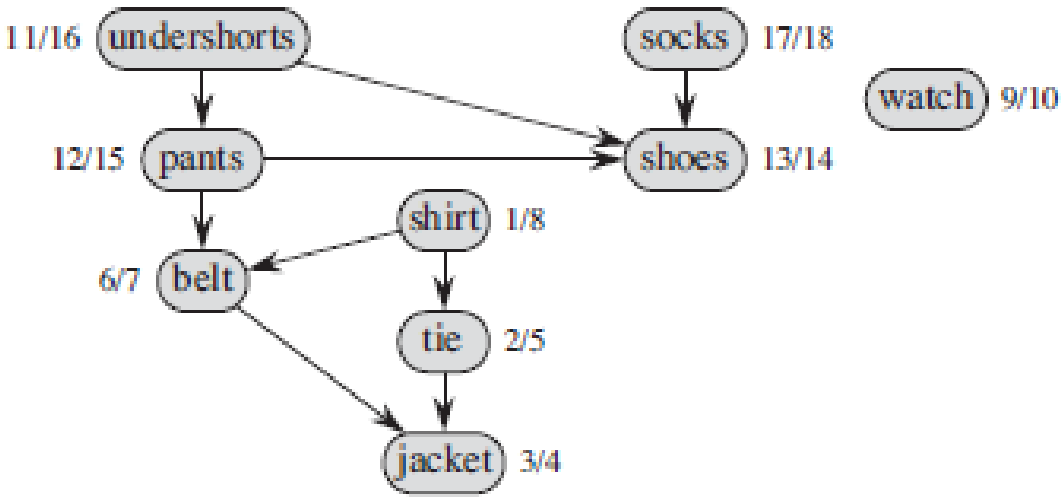
- We can define our edges as (a, b); perform a before b
  - (undershorts, shoes):
  - (undershorts, pants)
  - (pants, belt)
  - (pants, shoes)
  - (shirt, belt)
  - (shirt, tie)
  - (tie, jacket)
  - (socks, shoes)
  - (belt, jacket)



- Note vertex "watch" do not appear in the edge list:
  - We can do this task at any time.
- Taken together, the vertices and edges form a DAG.

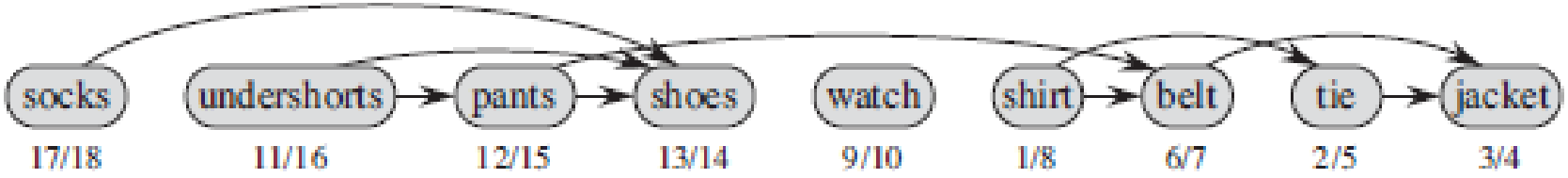
Getting Dressed: the Graph

- 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.



Getting Dressed: the Graph

- In practice we construct a list of the vertices as each is popped from the stack and then print the list backwards.



The same graph shown topologically sorted, with its vertices arranged from left to right in order of decreasing finishing time. All directed edges go from left to right.

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.

Related References

- Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
  - Chapters 1.4,3.5, 4.2
- Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 22.1-4