Chapter 28 Graphs and Applications

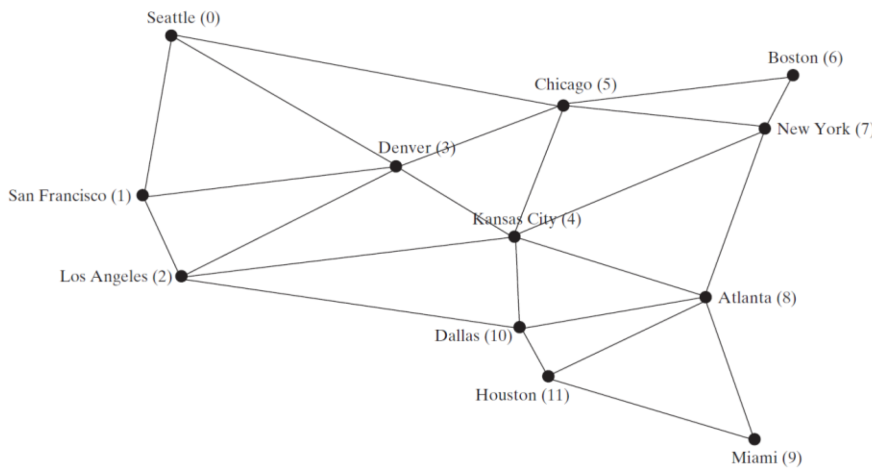
CS2: Data Structures and Algorithms
Colorado State University

Original slides by Daniel Liang
Modified slides by Chris Wilcox,
Wim Bohm, and Russ Wakefield

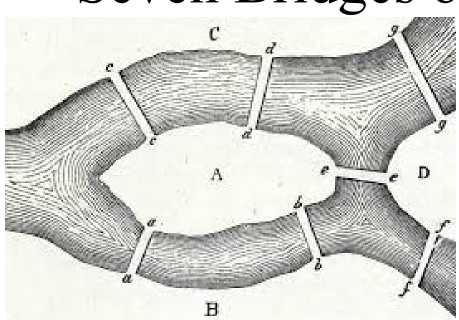
Objectives

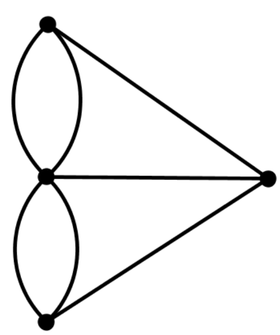
- To model real-world problems using graphs and explain the Seven Bridges of Königsberg problem (§28.1).
- To describe the graph terminologies: vertices, edges, simple graphs, weighted/unweighted graphs, and directed/undirected graphs (§28.2).
- To represent vertices and edges using lists, edge arrays, edge objects, adjacency matrices, and adjacency lists (§28.3).
- To model graphs using the **Graph** interface, the **AbstractGraph** class, and the **UnweightedGraph** class (§28.4).
- To display graphs visually (§28.5).
- To represent the traversal of a graph using the **AbstractGraph.Tree** class (§28.6).
- To design and implement depth-first search (§28.7).
- To solve the connected-circle problem using depth-first search (§28.8).
- To design and implement breadth-first search (§28.9).
- To solve the nine-tail problem using breadth-first search (§28.10).

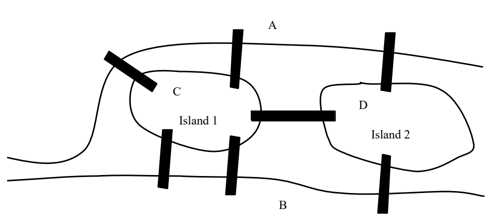
Modeling Using Graphs



Seven Bridges of Königsberg







Basic Graph Terminologies

What is a graph? G=(V, E) = vertices (nodes) and edges

Weighted vs. **Unweighted** graphs

Directed vs. **Undirected** graphs

Adjacent vertices share an edge (**Adjacent edges)**

A vertex is **Incident** to an edge that it joins

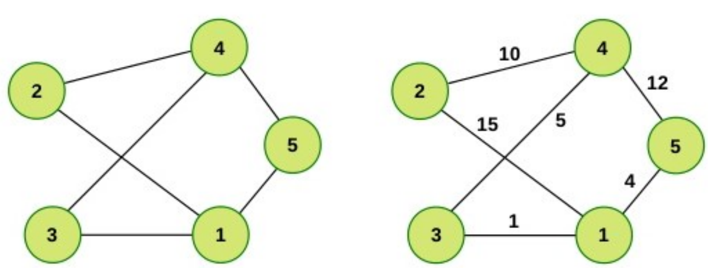
Degree of a vertex = number of edges it joins

Neighborhood = subgraph with all adjacent vertices

A **Loop** is an edge that connects a vertex to itself

A **Cycle** is a path from a vertex to itself via other vertices

Weighted vs Unweighted Graph

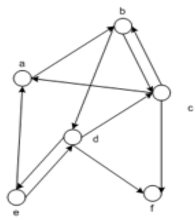


unweighted graph

weighted graph

https://www.slideshare.net/emersonferr/20-intro-graphs

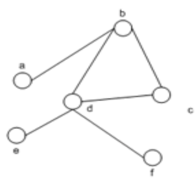
Directed vs Undirected Graph



Directed Graph (a)

	a	b	c	d	e	f
a		✓				
b			✓	✓		
c		✓				✓
d			✓		✓	✓
e	✓			✓		
f						

Adjacency Matrix Representation (a)



Undirected Graph (b)

	a	b	c	d	e	f
a		✓				
b	✓		✓	✓		
c		✓		✓		
d		✓	✓		✓	✓
e				✓		
f				✓		

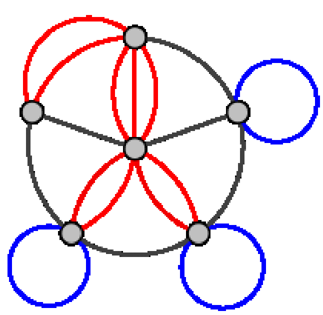
Adjacency Matrix Representation (b)

https://msdn.microsoft.com/en-us/library/ms379574(v=vs.80).aspx

More Graph Terminology

Parallel edge: two edges that share the same vertices, also called multiple edges:

- Multiple edges in red, loops in blue



https://en.wikipedia.org/wiki/Multigraph

More Graph Terminology

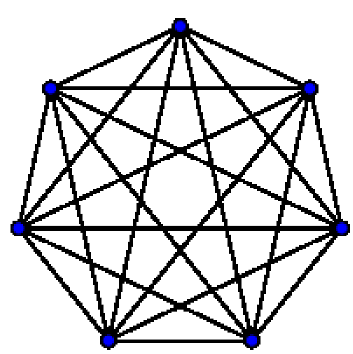
Simple graph: undirected, unweighted, no loops, no parallel edges:

simple graph — *nonsimple graph with multiple edges* — *nonsimple graph with loops*

http://mathworld.wolfram.com/SimpleGraph.html

More Graph Terminology

Complete graph: simple graph where every pair of vertices are connected by an edge:



https://en.wikipedia.org/wiki/Complete_graph

Representing Graphs

Representing Vertices

Representing Edges: Edge Array

Representing Edges: Edge Objects

Representing Edges: Adjacency Matrices

Representing Edges: Adjacency Lists

Representing Vertices (Nodes)

```
String[] vertices = {“Seattle“, “San Francisco“, “Los Angles”, … };
List<String> vertices;
```

or

```
public class City {
    String name;
}
City[] vertices = {city0, city1, … };
```

Representing Edges (Arcs)

```
int[][] edges = {{0, 1}, {0, 3} {0, 5}, {1, 0}, {1, 2}, … };
```

or

```
public class Edge {
  int u, v;
  public Edge(int u, int v) {
    this.u = u;
    this.v = v;
}
List<Edge> list = new ArrayList<>();
list.add(new Edge(0, 1)); list.add(new Edge(0, 3)); …
```


Representing Edges: Adjacency Matrix

```
int[][] adjacencyMatrix = {
  {0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0}, // Seattle
  {1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0}, // San Francisco
  {0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0}, // Los Angeles
  {1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0}, // Denver
  {0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0}, // Kansas City
  {1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0}, // Chicago
  {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0}, // Boston
  {0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0}, // New York
  {0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1}, // Atlanta
  {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1}, // Miami
  {0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1}, // Dallas
  {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0}  // Houston
};
```

Representing Edges: Adjacency Vertex List

List<Integer>[] neighbors = new List[12];

Seattle	neighbors[0]	1	3	5			
San Francisco	neighbors[1]	0	2	3			
Los Angeles	neighbors[2]	1	3	4	10		
Denver	neighbors[3]	0	1	2	4	5	
Kansas City	neighbors[4]	2	3	5	7	8	10
Chicago	neighbors[5]	0	3	4	6	7	
Boston	neighbors[6]	5	7				
New York	neighbors[7]	4	5	6	8		
Atlanta	neighbors[8]	4	7	9	10	11	
Miami	neighbors[9]	8	11				
Dallas	neighbors[10]	2	4	8	11		
Houston	neighbors[11]	8	9	10			

List<List<Integer>> neighbors = new ArrayList<>();

Representing Edges: Adjacency Edge List

List<Edge>[] neighbors = new List[12];

Seattle	neighbors[0]	Edge(0, 1)	Edge(0, 3)	Edge(0, 5)			
San Francisco	neighbors[1]	Edge(1, 0)	Edge(1, 2)	Edge(1, 3)			
Los Angeles	neighbors[2]	Edge(2, 1)	Edge(2, 3)	Edge(2, 4)	Edge(2, 10)		
Denver	neighbors[3]	Edge(3, 0)	Edge(3, 1)	Edge(3, 2)	Edge(3, 4)	Edge(3, 5)	
Kansas City	neighbors[4]	Edge(4, 2)	Edge(4, 3)	Edge(4, 5)	Edge(4, 7)	Edge(4, 8)	Edge(4, 10)
Chicago	neighbors[5]	Edge(5, 0)	Edge(5, 3)	Edge(5, 4)	Edge(5, 6)	Edge(5, 7)	
Boston	neighbors[6]	Edge(6, 5)	Edge(6, 7)				
New York	neighbors[7]	Edge(7, 4)	Edge(7, 5)	Edge(7, 6)	Edge(7, 8)		
Atlanta	neighbors[8]	Edge(8, 4)	Edge(8, 7)	Edge(8, 9)	Edge(8, 10)	Edge(8, 11)	
Miami	neighbors[9]	Edge(9, 8)	Edge(9, 11)				
Dallas	neighbors[10]	Edge(10, 2)	Edge(10, 4)	Edge(10, 8)	Edge(10, 11)		
Houston	neighbors[11]	Edge(11, 8)	Edge(11, 9)	Edge(11, 10)			

Representing Adjacency Edge List Using ArrayList

```
List<ArrayList<Edge>> neighbors = new ArrayList<>();
neighbors.add(new ArrayList<Edge>());
neighbors.get(0).add(new Edge(0, 1));
neighbors.get(0).add(new Edge(0, 3));
neighbors.get(0).add(new Edge(0, 5));
neighbors.add(new ArrayList<Edge>());
neighbors.get(1).add(new Edge(1, 0));
neighbors.get(1).add(new Edge(1, 2));
neighbors.get(1).add(new Edge(1, 3));
...
...
neighbors.get(11).add(new Edge(11, 8));
neighbors.get(11).add(new Edge(11, 9));
neighbors.get(11).add(new Edge(11, 10));
```


«interface» Graph<V>	The generic type V is the type for vertices.
+getSize(): int	Returns the number of vertices in the graph.
+getVertices(): List<V>	Returns the vertices in the graph.
+getVertex(index: int): V	Returns the vertex object for the specified vertex index.
+getIndex(v: V): int	Returns the index for the specified vertex.
+getNeighbors(index: int): List<Integer>	Returns the neighbors of vertex with the specified index.
+getDegree(index: int): int	Returns the degree for a specified vertex index.
+printEdges(): void	Prints the edges.
+clear(): void	Clears the graph.
+addVertex(v: V): boolean	Returns true if v is added to the graph. Returns false if v is already in the graph.
+addEdge(u: int, v: int): boolean	Adds an edge from u to v to the graph throws IllegalArgumentException if u or v is invalid. Returns true if the edge is added and false if (u, v) is already in the graph.
+addEdge(e: Edge): boolean	Adds an edge into the adjacency edge list.
+remove(v: V): boolean	Removes a vertex from the graph.
+remove(u: int, v: int): boolean	Removes an edge from the graph.
+dfs(v: int): UnWeightedGraph<V>.SearchTree	Obtains a depth-first search tree starting from v.
+bfs(v: int): UnWeightedGraph<V>.SearchTree	Obtains a breadth-first search tree starting from v.

UnweightedGraph<V>	
#vertices: List<V>	Vertices in the graph.
#neighbors: List<List<Edge>>	Neighbors for each vertex in the graph.
+UnweightedGraph()	Constructs an empty graph.
+UnweightedGraph(vertices: V[], edges: int[][])	Constructs a graph with the specified edges and vertices stored in arrays.
+UnweightedGraph(vertices: List<V>, edges: List<Edge>)	Constructs a graph with the specified edges and vertices stored in lists.
+UnweightedGraph(edges: int[][], numberOfVertices: int)	Constructs a graph with the specified edges in an array and the integer vertices 1, 2, ...
+UnweightedGraph(edges: List<Edge>, numberOfVertices: int)	Constructs a graph with the specified edges in a list and the integer vertices 1, 2, ….

Graph

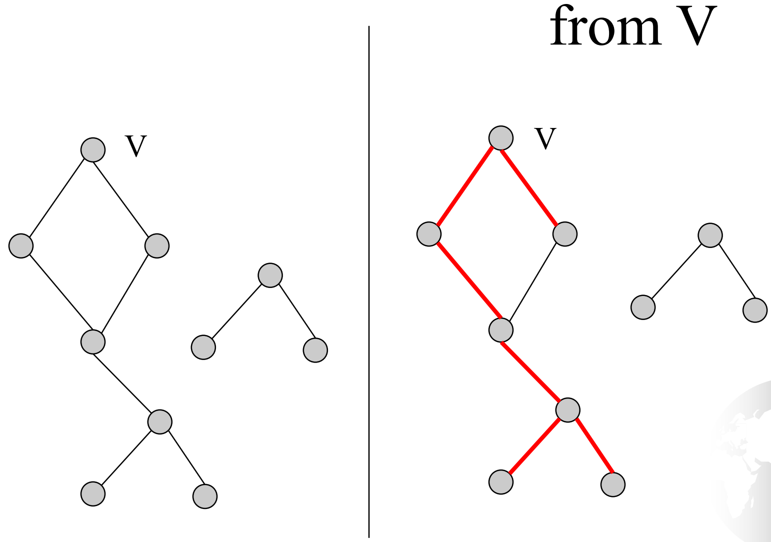
UnweightedGraph

TestGraph

Run

Graph

Spanning Tree from V



Graph Traversals

Depth-first search and breadth-first search

Both traversals result in a spanning tree, which can be modeled using a class.

UnweightedGraph<V>.SearchTree	
-root: int	The root of the tree.
-parent: int[]	The parents of the vertices.
-searchOrder: List<Integer>	The orders for traversing the vertices.
+SearchTree(root: int, parent: int[], searchOrder: List<Integer>)	Constructs a tree with the specified root, parent, and searchOrder.
+getRoot(): int	Returns the root of the tree.
+getSearchOrder(): List<Integer>	Returns the order of vertices searched.
+getParent(index: int): int	Returns the parent for the specified vertex index.
+getNumberOfVerticesFound(): int	Returns the number of vertices searched.
+getPath(index: int): List<V>	Returns a list of vertices from the specified vertex index to the root.
+printPath(index: int): void	Displays a path from the root to the specified vertex.
+printTree(): void	Displays tree with the root and all edges.

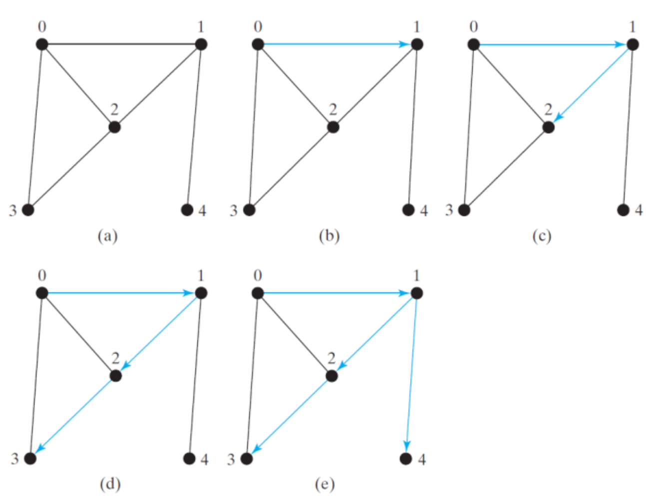
Depth-First Search

The depth-first search of a graph builds a “spanning tree” of all of the reachable edges from the starting vertex v, using marking of the visited nodes:

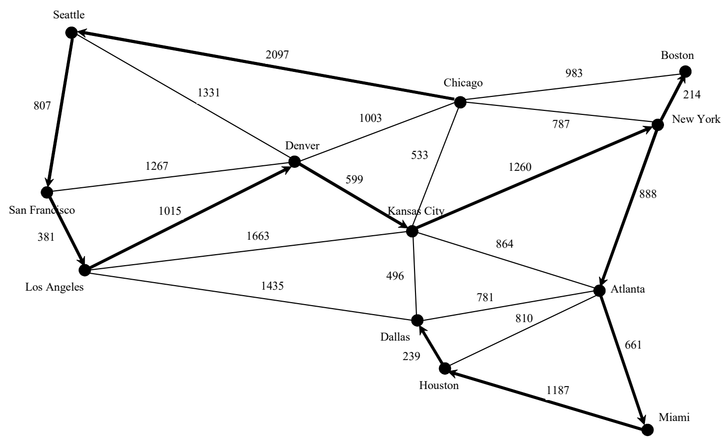
Input: G = (V, E) and a starting vertex v
Output: a DFS tree rooted at v

```
Tree dfs(vertex v) {
  visit v;
  for each neighbor w of v
    if (w has not been visited) {
      set v as the parent for w;
      dfs(w);
    }
}
```


Depth-First Search Example



Depth-First Search Example



Applications of the DFS

- Detecting whether a graph is connected. Search the graph starting from any vertex. If the number of vertices searched is the same as the number of vertices in the graph, the graph is connected. Otherwise, the graph is not connected.
- Detecting whether there is a path between two vertices.
- Finding a path between two vertices.
- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.
- Detecting whether there is a cycle in the graph, and finding a cycle in the graph.

DFS: Depth First Search

- Explores edges from the most recently discovered node; backtracks when reaching a dead-end. The book does not used white, grey, black, but uses visited (and implicitly unexplored). Recursive

```
DFS(u):
  mark u as visited
  for each edge (u,v) :
    if v is not marked visited :
       DFS(v)
```


DFS and cyclic graphs

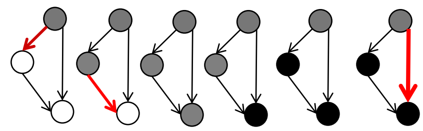
- When DFS visits a node for the first time it is white. There are two ways DFS can **revisit** a node:

- 1. DFS has already fully explored the node. **What color does it have then? Is there a cycle then?**

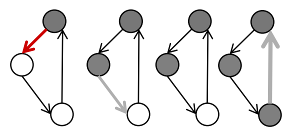
- 2. DFS is still exploring this node. What color does it have in this case? **Is there a cycle then?**

DFS and cyclic graphs

- There are two ways DFS can **revisit** a node:
- 1. DFS has already fully explored the node. **What color does it have then?**



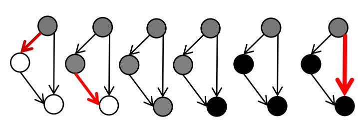
- **Is there a cycle then?**
- 2. DFS is still exploring this node. **What color does it have in this case?**



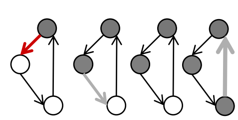
- **Is there a cycle then?**

DFS and cyclic graphs

- There are two ways DFS can **revisit** a node:
- 1. DFS has already fully explored
- the node. **What color does it have**
- **then? Is there a cycle then?**
- No, the node is revisited
- from outside.

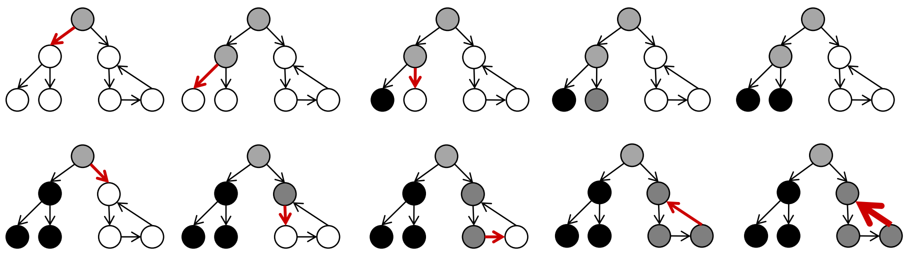


- 2. DFS is still exploring this node.
- **What color does it have in this**
- **case? Is there a cycle then?**
- Yes, the node is revisited on a
- path containing the node itself.



- **So DFS with the white, grey, black coloring scheme detects a cycle when a GREY node is visited**

Cycle detection: DFS + coloring



When a grey (frontier) node is visited, a cycle is detected.

Recursive / node coloring version

```
DFS(u):
  #c: color,  p: parent
  c[u]=grey
  forall  v in Adj(u):
    if c[v]==white:
      parent[v]=u
      DFS(v)
  c[u]=black
```

Breadth-First Search

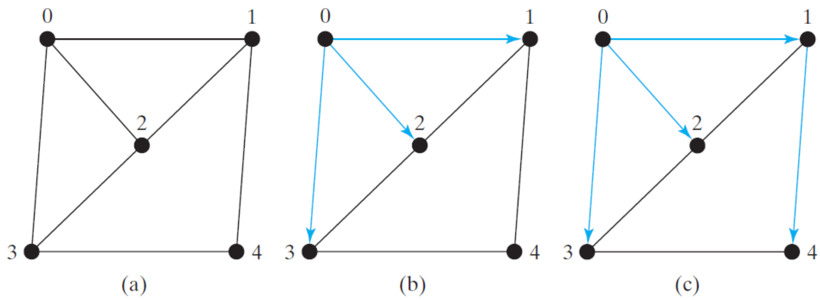
The breadth-first traversal of a graph is like the breadth-first traversal of a tree discussed in §25.2.3, "Tree Traversal."

With breadth-first traversal of a tree, the nodes are visited level by level. First the root is visited, then all the children of the root, then the grandchildren of the root from left to right, and so on.

Breadth-First Search

```
Input: G = (V, E) and a starting vertex v, Output: a BFS tree rooted at v
bfs(vertex v) {
  create an empty queue for storing vertices to be visited;
  add v into the queue;
  mark v visited;
  while the queue is not empty {
    dequeue a vertex, say u, from the queue
    process u;
    for each neighbor w of u
      if w has not been visited {
        add w into the queue;
        set u as the parent for w;
        mark w visited;
      }
  }
}
```

Breadth-First Search Example



Queue: 0

isVisited[0] = true

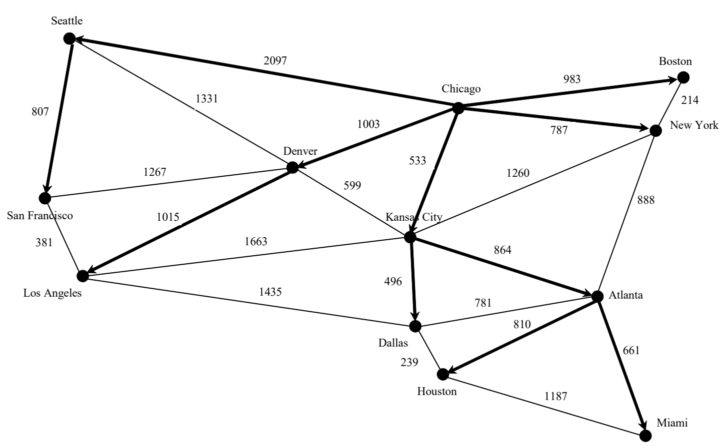
Queue: 1 2 3

isVisited[1] = true, isVisited[2] = true, isVisited[3] = true

Queue: 2 3 4

isVisited[4] = true

Breadth-First Search Example



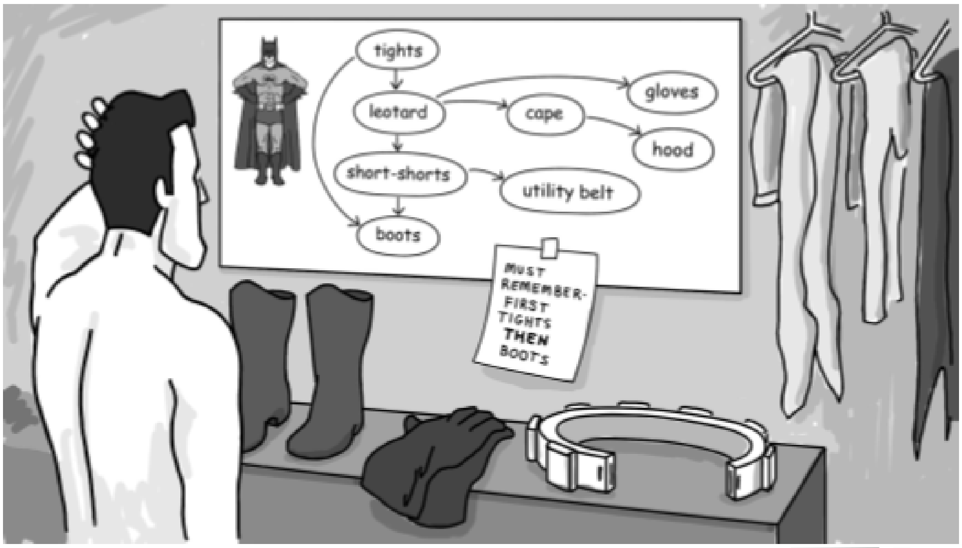
Applications of the BFS

- Detecting whether a graph is connected. A graph is connected if there is a path between any two vertices in the graph.
- Detecting whether there is a path between two vertices.
- Finding a shortest path between two vertices. You can prove that the path between the root and any node in the BFS tree is the shortest path between the root and the node.
- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.

Graphs Describing Precedence

- Edge from *x* to *y* indicates *x* should come before *y,* e.g.:
  – prerequisites for a set of courses
  – dependences between programs
  – set of tasks, e.g. building a computer

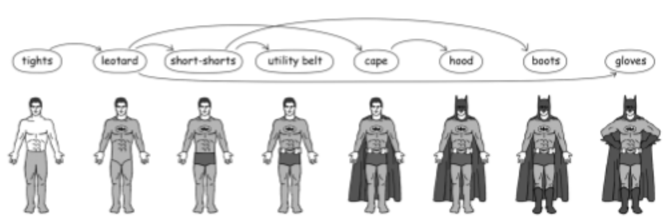
Graphs Describing Precedence



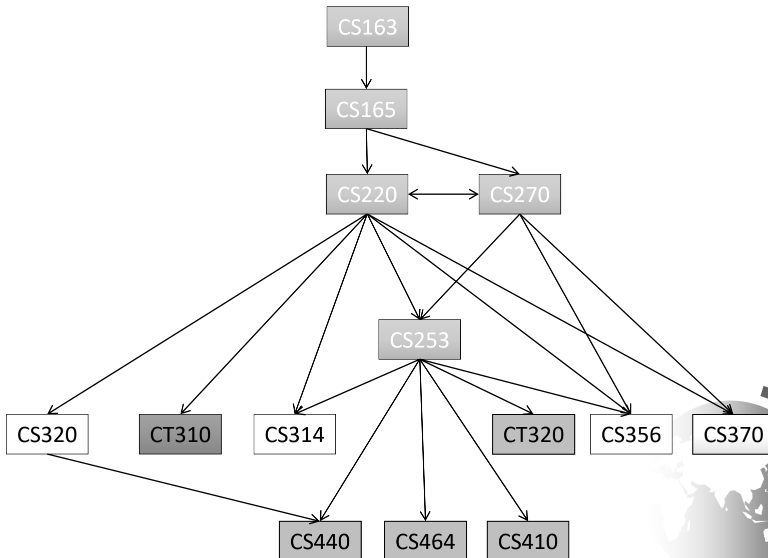
Batman images are from the book "Introduction to bioinformatics algorithms"

Graphs Describing Precedence

- Want an ordering of the vertices of the graph that respects the precedence relation
  - Example: An ordering of CS courses
- The graph must not contain cycles. **WHY?**



CS Courses Required for CS and ACT Majors

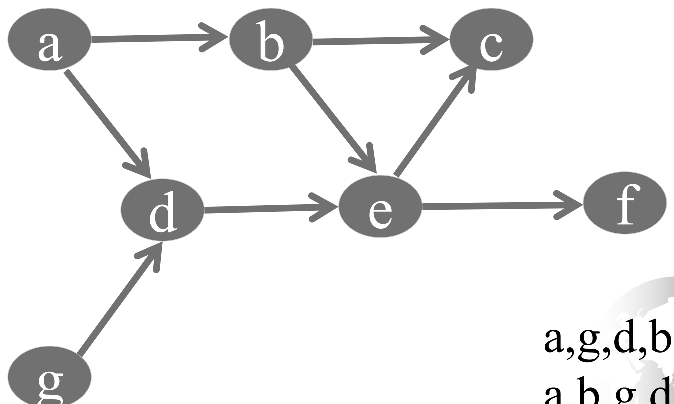


Topological Sorting of DAGs

- DAG: Directed Acyclic Graph
- Topological sort: listing of nodes such that if *(a,b)* is an edge, *a* appears before *b* in the list
- Is a topological sort unique?

Question: Is a topological sort unique?

A directed graph without cycles



a,g,d,b,e,c,f
a,b,g,d,e,f,c

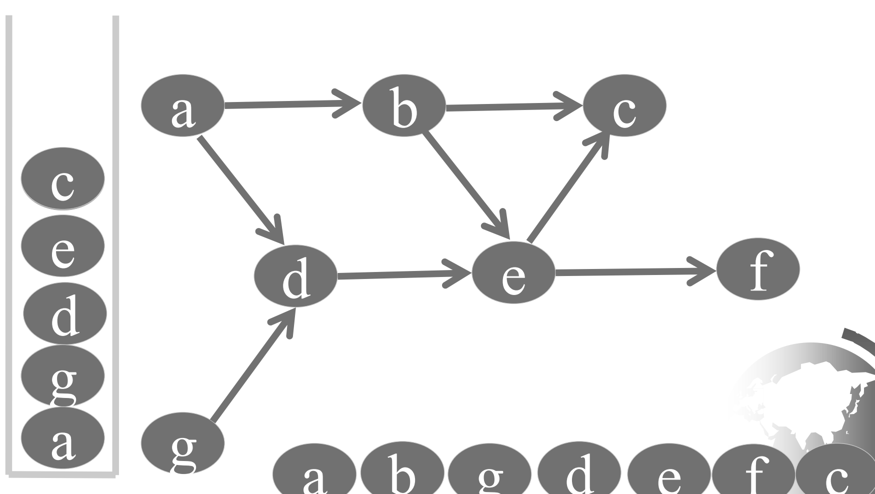
Topological Sort Algorithm

- Modification of DFS: Traverse tree using DFS starting from all nodes that have no predecessor.
- Add a node to the list when ready to backtrack.

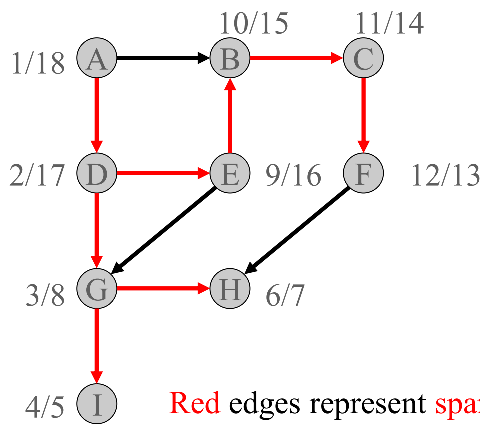
Topological Sort Algorithm

```
List toppoSort(Graph theGraph)
   // use stack stck and list lst
   // push all roots
   for (all vertices v in the graph theGraph)
        if (v has no predecessors)
                stck.push(v)
                Mark v as visited
   // DFS
   while (!stck.isEmpty())
        if (all vertices adjacent to the vertex on top of
                the stack have been visited)
                v = stck.pop()
                lst.add(0, v)
        else
                Select an unvisited vertex u adjacent to vertex v on
                top of the stack
                stck.push(u)
                Mark u as visited
                Set v as parent of u
   return lst
```


Algorithm 2: Example 1



Topological sorting solution



Red edges represent spanning tree