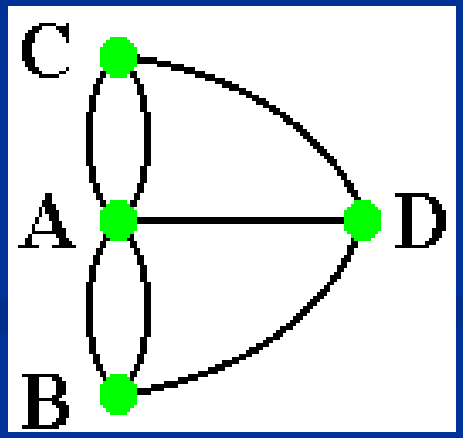
Graphs

- Depth First Search
- Breadth First Search
- Topological Sort
- Minimum Spanning Tree
- All-Pairs-Shortest Paths
- Two coloring

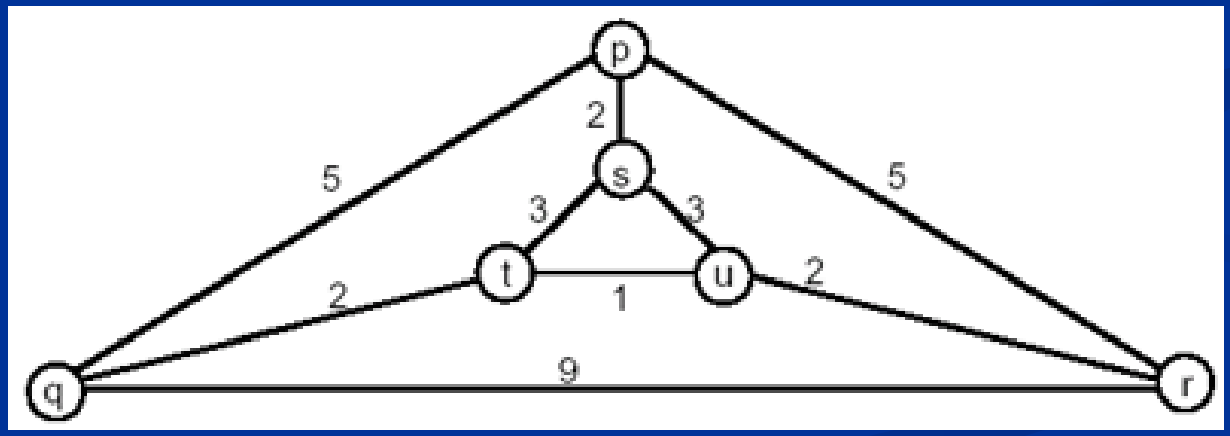
Introduction

- Graphs are a collection of vertices (nodes) and edges (arcs)
  - The solid circles are the vertices A, B, C and D
  - The lines are the edges



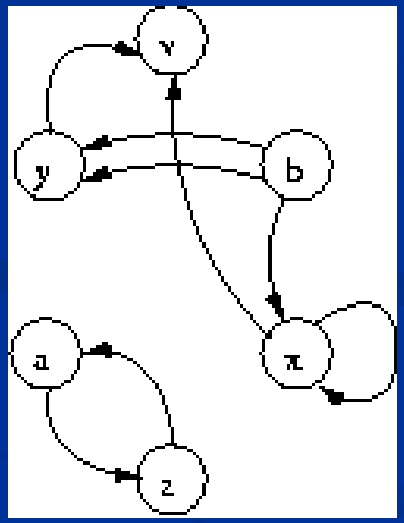
Weighted Graphs

- We can also assign values to the edges (a cost related to connecting the two vertices together) and form a *weighted graph*



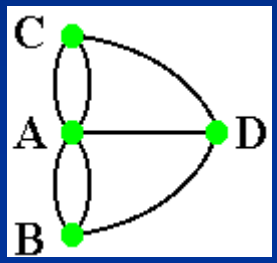
Directed Graph

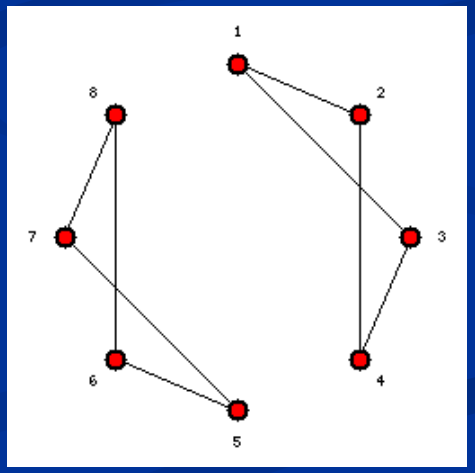
- There's also a *directed graph* (sometimes called a *digraph*) that makes the edges one-way
  - Since edges are only one-way, **b** has an edge to **x**, but **x** does not have an edge to **b**
- As you can probably guess, you can have a weighted, directed graph as well



Terminology: Connected

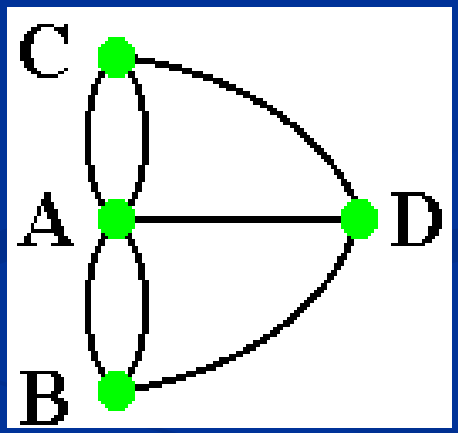
- A graph is said to be *connected* if there is a path from every vertex to every other vertex in the graph
- If a graph is not connected, then we call it *disconnected*





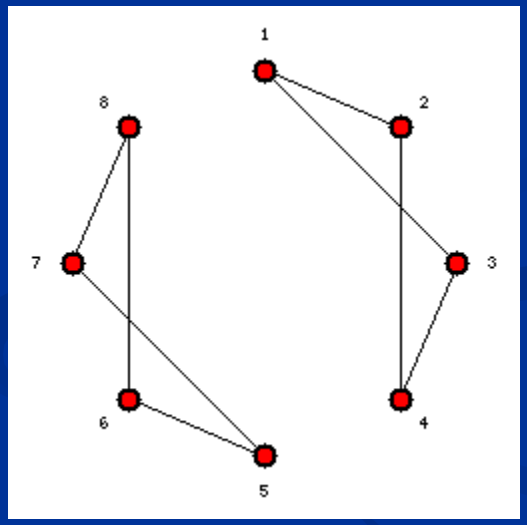
Terminology: Path

- A *path* is a list of vertices in which successive vertices are connected by edges in the graph. In this graph, one example path is
  - A→B→D→A→C
- A *simple path* is a path in which no vertex is repeated.
  - Hence, A→B→D→A→C is not a simple path, but A→B→D→C is



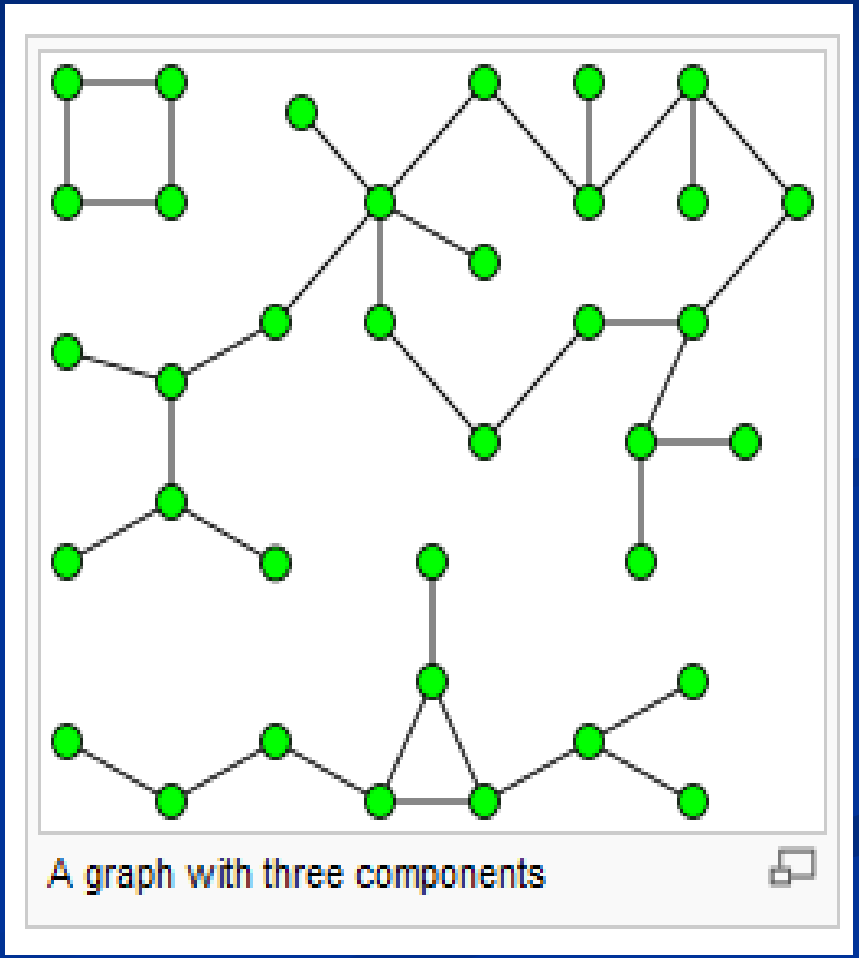
Terminology: Cycle

- A *cycle* is a path that is simple except that the first and last vertices are the same
- In this graph, the path 7→5→6→8→7 is a cycle



Terminology: Connected Component

- A **connected component** or **component** is a maximal connected subgraph

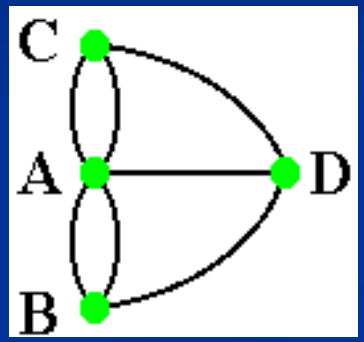


Graph Representation

- Adjacency Matrix
- Adjacency List
- Edge List
- Adjacency Function

Adjacency Matrix

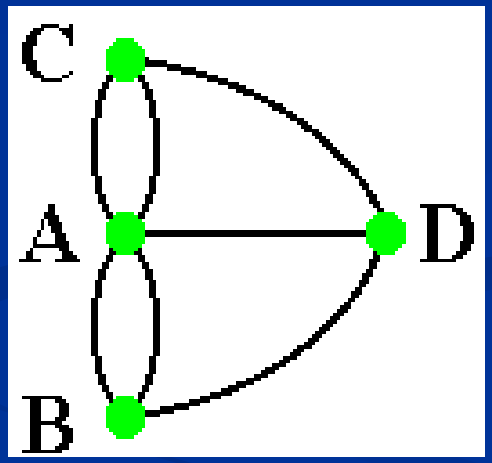
- An adjacency matrix is simply an *N* x *N* matrix (where *N* is the number of vertices in the graph).
  - Can store boolean (is there an edge?)
  - Can store number (how many edges connect nodes)
  - Can store weights (for weighted graph)



	A	B	C	D
A	0	2	2	1
B	2	0	0	1
C	2	0	0	1
D	1	1	1	0

Adjacency List

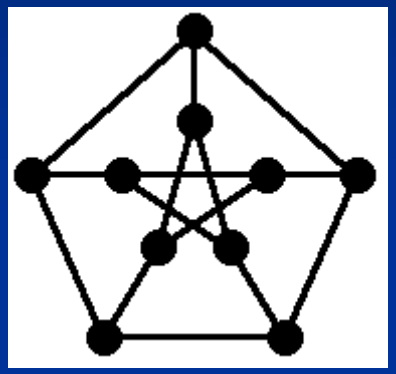
- The adjacency list simply keeps a linked list for each vertex that contains the vertices that are neighbors to it (that is, the list of vertices such that there exists an edge from the subject node to each of the vertices)
  - A : B→C→D
  - B : A→D
  - C : A→D
  - D : A→B→C

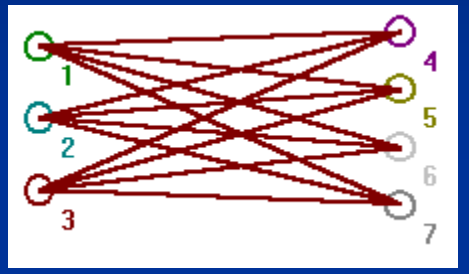


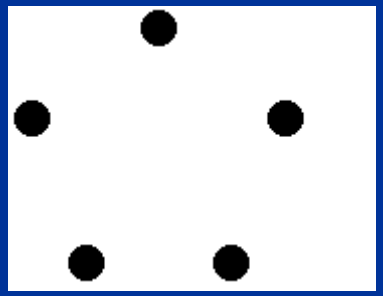
Motivation

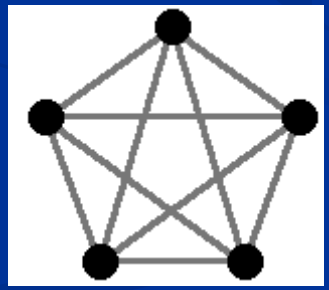
- Graphs are very useful for representing a large array of problems and for solving everyday "real world" problems, on which the contest loves to focus.
  - Map intersections to vertices and roads to edges, and now you have a representation to help you route fire trucks to fires.
  - Map a network of computers and routers to vertices, physical connections to edges, and you have a way to find the minimum number of hops between two given computers.
  - Map a set of rooms that need water to vertices, all combinations of distances between pairs of rooms to edges, and then you can determine the smallest amount of pipe needed to connect all the rooms together.
  - Map the buildings on the UCF campus to vertices, the sidewalks between the buildings to edges, and then you'll be able to find the shortest walking distance from Harris Engineering Center to any other building.
  - Many, many others!

Testing









Algorithms

- DFS
- BFS
- Topological Sort
- Kruskal's Algorithm (Minimum Spanning Tree)
- Floyd-Warshall's Algorithm (All Pairs Shortest Path)
- Two Coloring

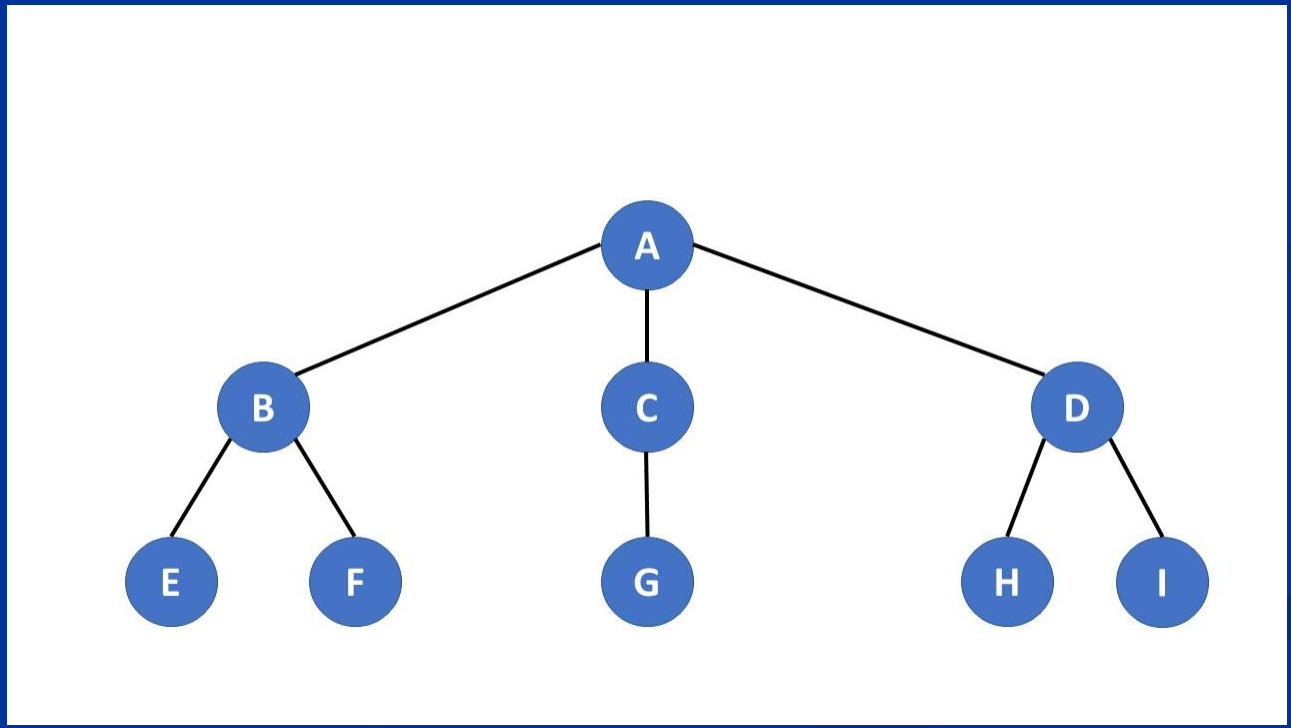
Depth First Search

A B E F C G D H I

A C G …

A D H I …

A D I H …



Depth First Search

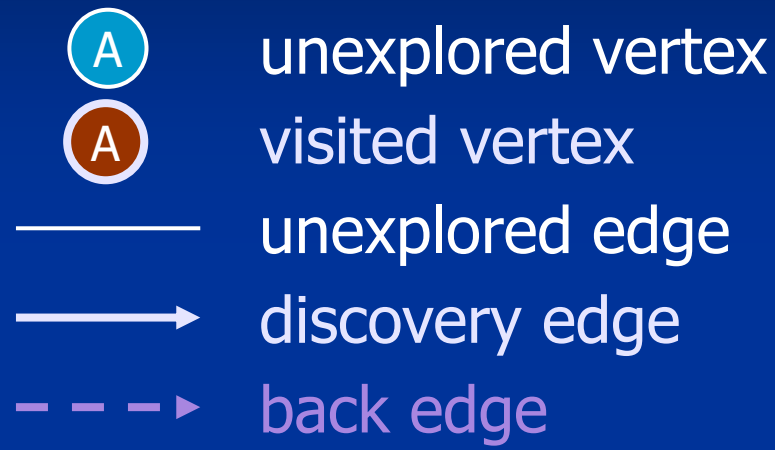
- The name "Depth-First Search" comes from the fact that you search as deep as you can in the graph first
  - We need an adjacency matrix and an array to keep track of whether we visited a given node yet

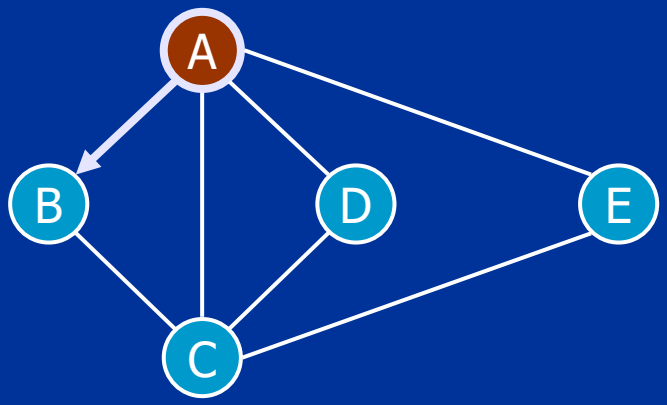
```
adj:array[1..N, 1..N] of Boolean
visited:array[1..N] of Boolean
   (initialized to all false)

DFS(node)
{
   visited[node] = true;
   if (node is target node) then
     process accordingly

   for I = 1 to N
   {
        if (adj[node][I]) and
           (not visited[I]) then
           DFS(I);
   }
}
```

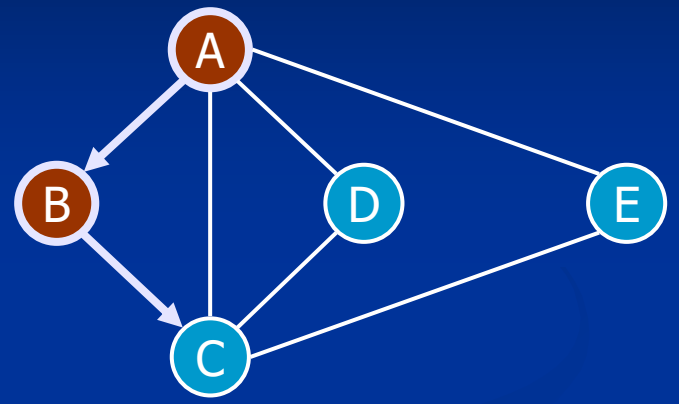
DFS Motivation

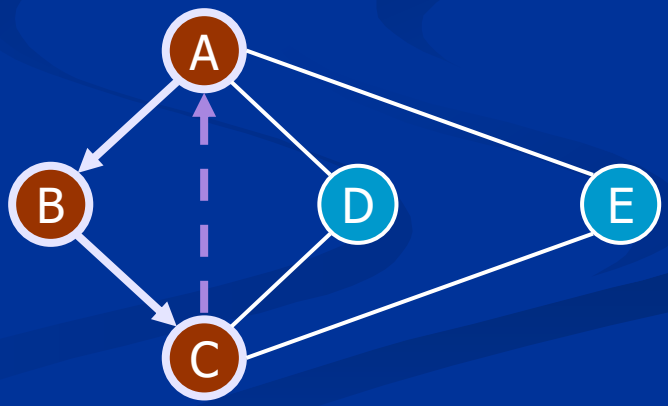
- Most basic searching algorithm
- Mazes
- Path finding
- Cycle detection

DFS Example

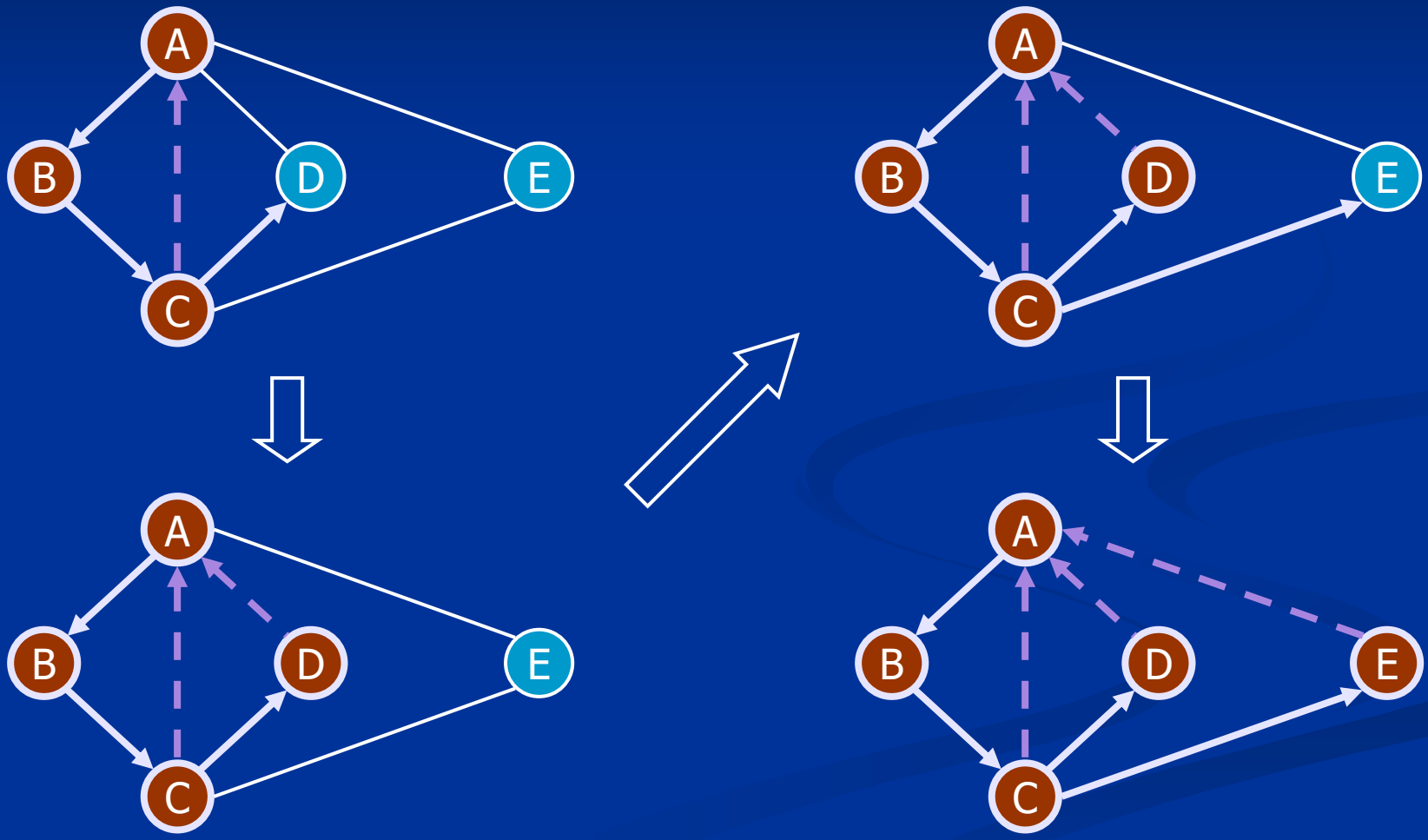

A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
back edge


A
B
D
E
C


A
B
D
E
C


A
B
D
E
C

DFS Example



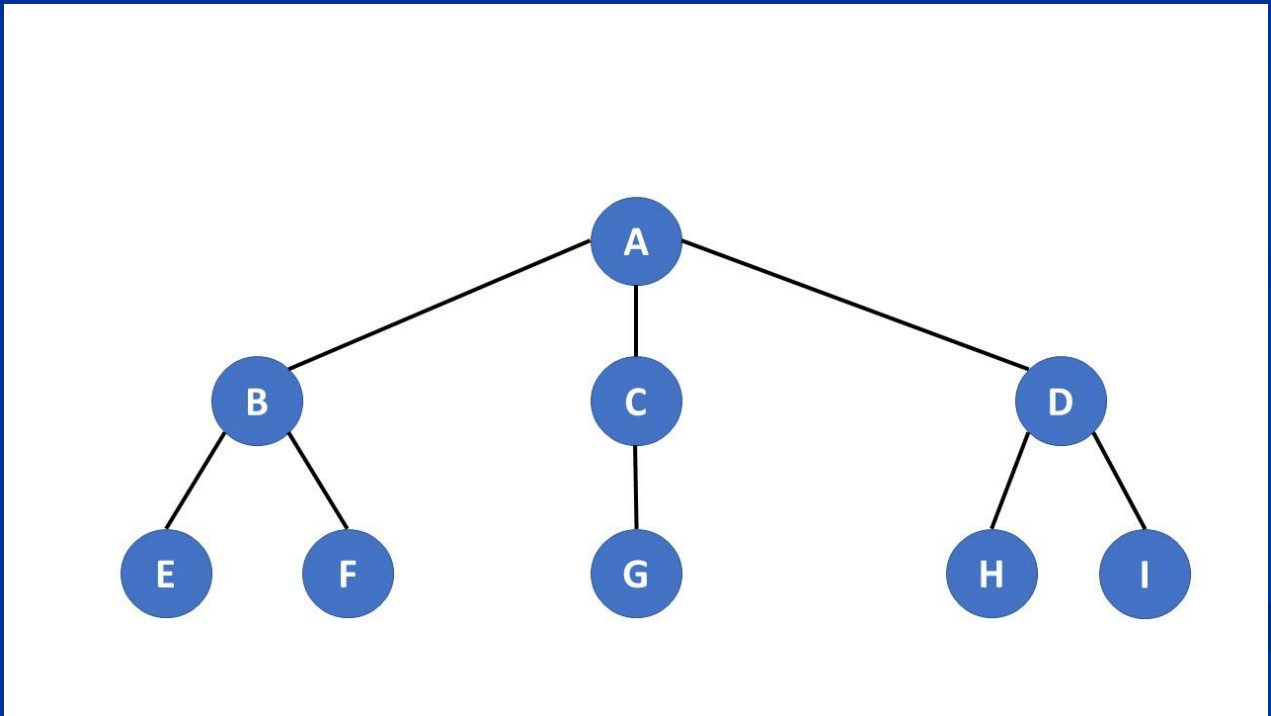
Breadth First Search

A B C D E F G H I

A C B D G …

A C D B G …

A D …



Breadth First Search

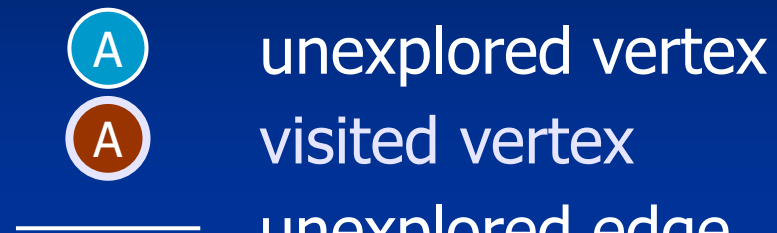
- The name "Breadth-First Search" comes from the idea that you search as wide as possible at each step
  - Again, we need the adjacency matrix and visited array. We will also need a queue data structure

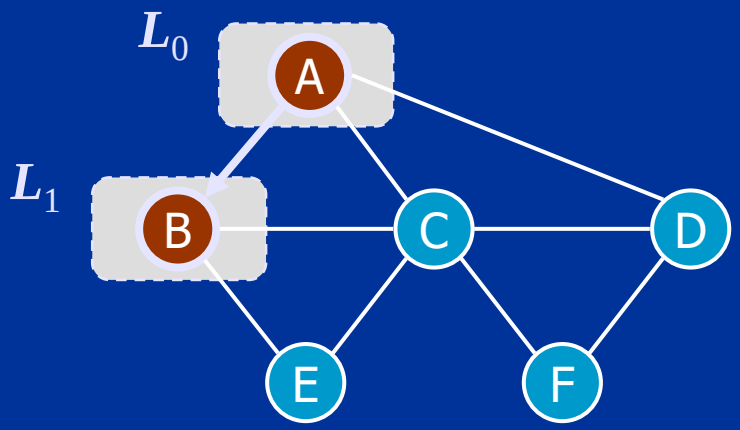
```
adj:array[1..N, 1..N] of Boolean
visited:array[1..N] of Boolean
   (initialized to all false)
BFS(node)
{
   enqueue(node)
   visited[node] = true
   while (queue is not empty)
   {
      t = dequeue()
      for I = 1 to N
      {
         if (adj[t][I]) and
            (not visited[I]) then
            enqueue(I)
            visited[I] = true
      }
   }
}
```

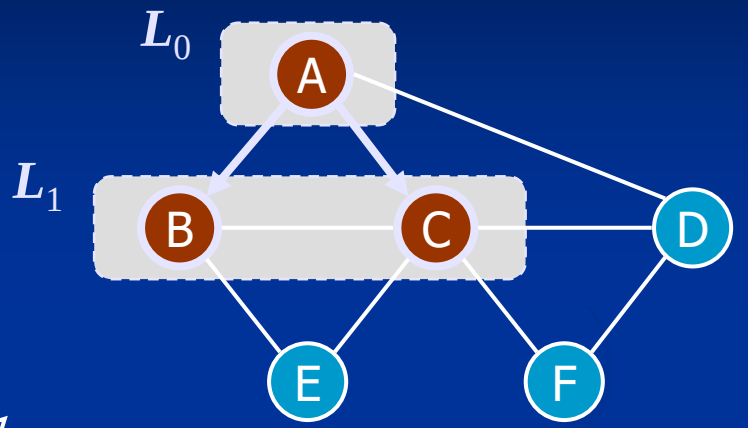
BFS Motivation

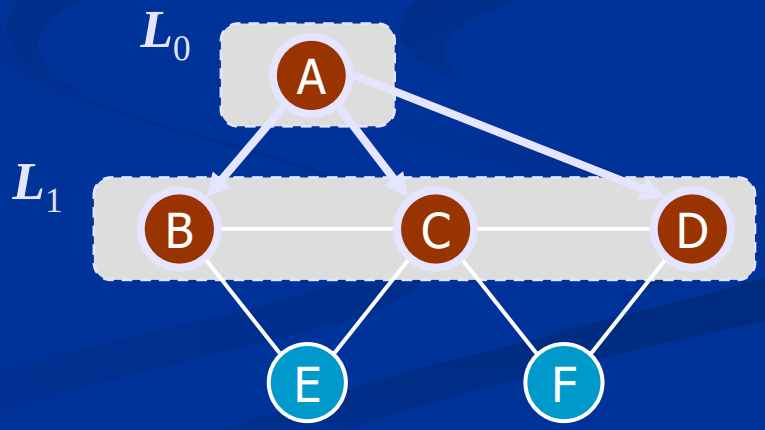
- Finding all connected components in a graph
- Finding the shortest path between two nodes *u* and *v* (in an unweighted graph)
- Testing a graph for bipartiteness

Example

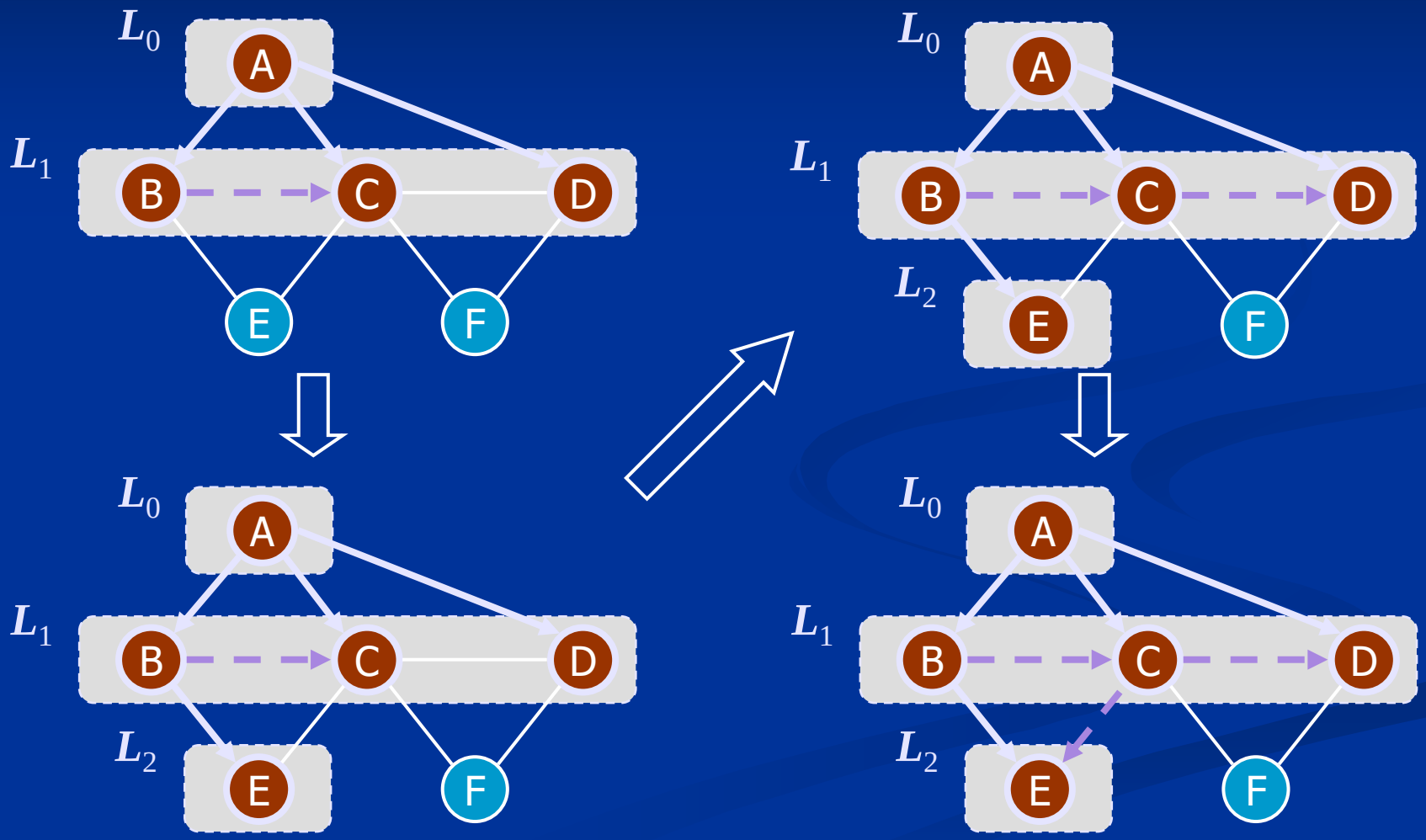

A
A
unexplored vertex
visited vertex
unexplored edge
discovery edge
cross edge


L_0
A
L_1
B
C
D
E
F

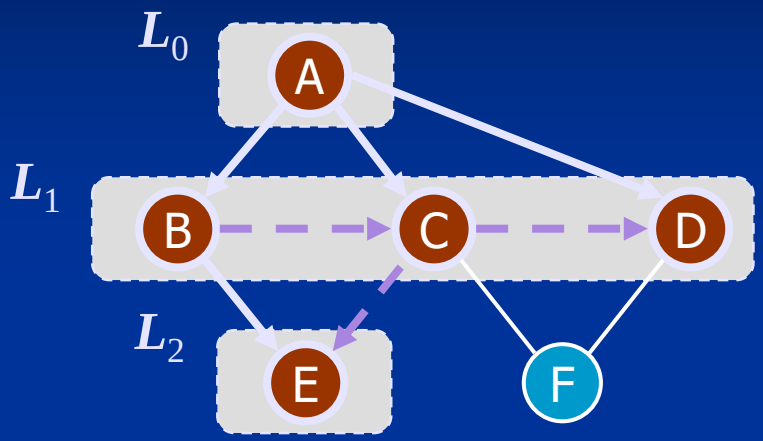

L_0
A
L_1
B
C
D
E
F

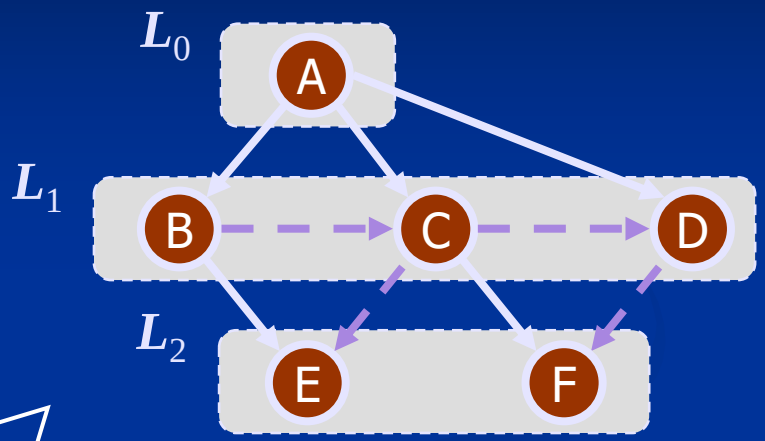

L_0
A
L_1
B
C
D
E
F

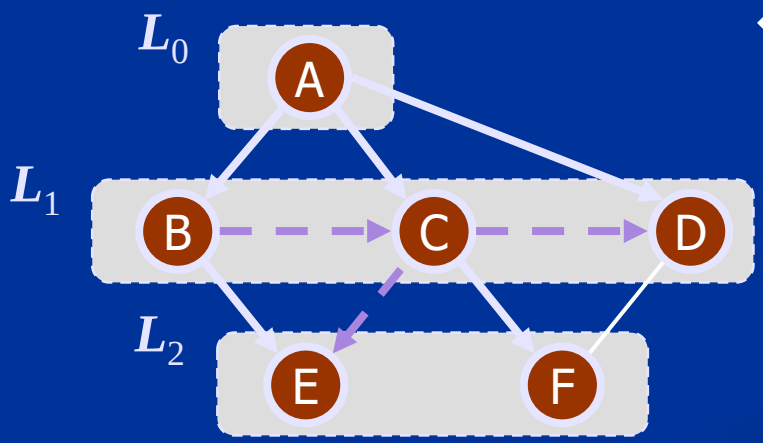
Example (cont.)



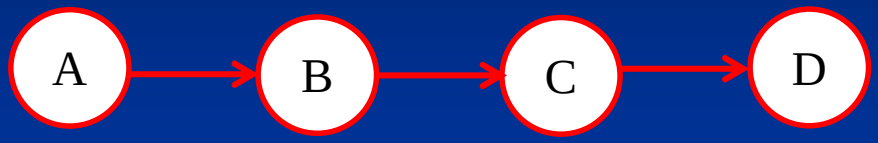
Example (cont.)



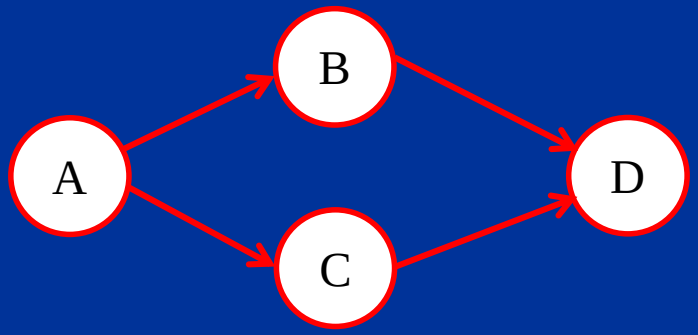




Topological Sort



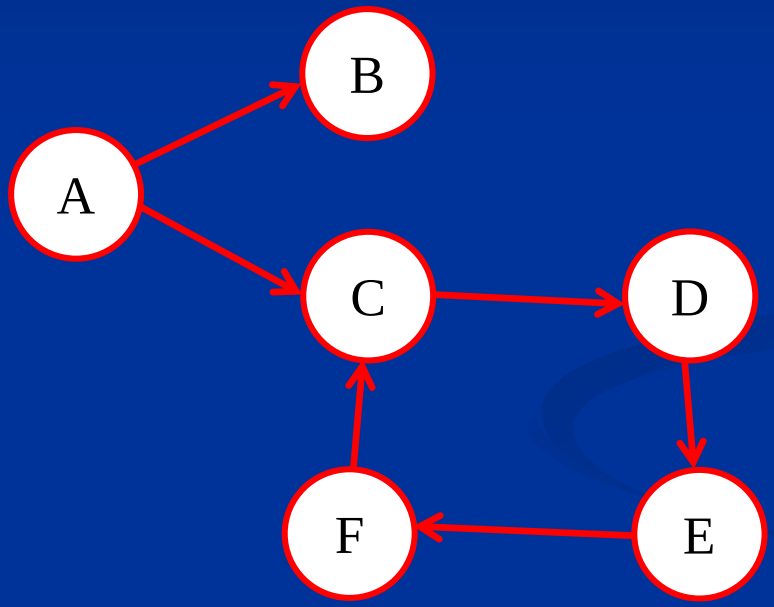
A B C D



A B C D

A C B D

Topological Sort



Topological Sort

loop

    find a vertex with no incoming arcs

    schedule this vertex

    remove the vertex and all its outgoing arcs

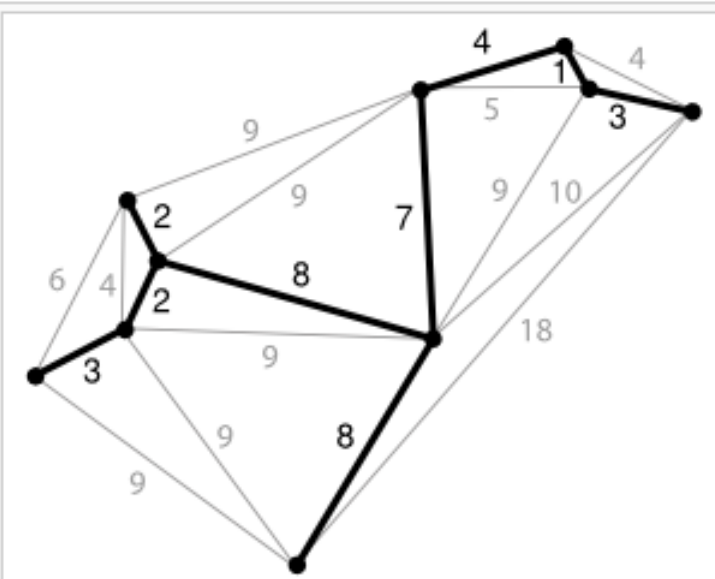
end loop

if (some vertices have not been scheduled) then

    there is a cycle

Minimum Spanning Tree

- A subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized
- Motivation
  - Connecting islands with cable
  - Minimum sidewalks to connect buildings
  - Many more!



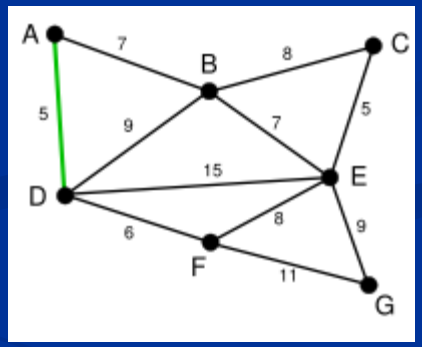
The minimum spanning tree of a planar graph. Each edge is labeled with its weight, which here is roughly proportional to its length.

MST Algorithms

- Two algorithms
  - Kruskal's
    - Adds edges that won't cause a cycle (stop after n-1 edges)
  - Prim's
    - Builds up a tree that includes all vertices (without cycles, obviously)

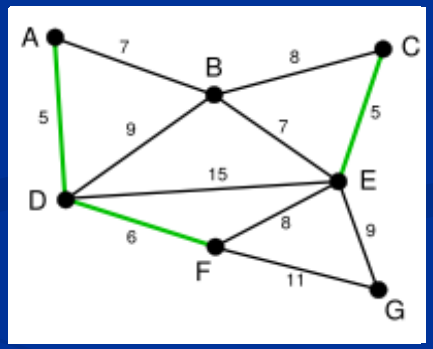
Kruskal's Example

- Here is our original graph
- We'll begin by finding the smallest edge
  - Both AD and CE are the smallest (length 5)
  - We'll arbitrarily choose AD



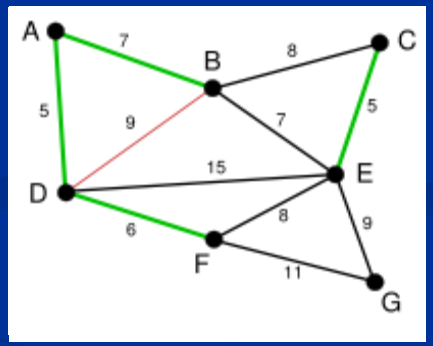
Kruskal's Example

- Then we consider the smallest edges left and find the one that won't create a cycle
- CE is of length 5 and won't create a cycle
- Then, DF (of length 6) is chosen in a similar fashion



Kruskal's Example

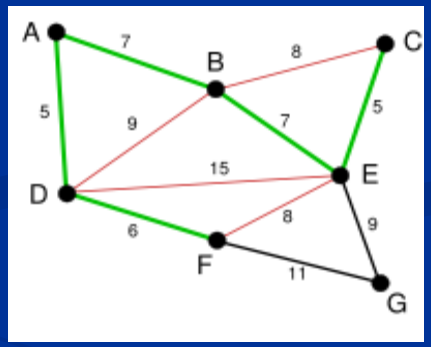
- The process continues…
- Both AB and BE are of length 7
- We arbitrarily choose AB



- Note that BD, if ever chosen, would create a cycle in our MST

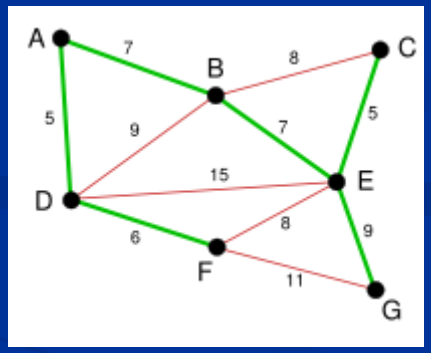
Kruskal's Example

- We then select BE (length 7) as our next edge since it does not create a cycle
- Many more edges would now cause a cycle to be created if used later
  - BC, DE and EF



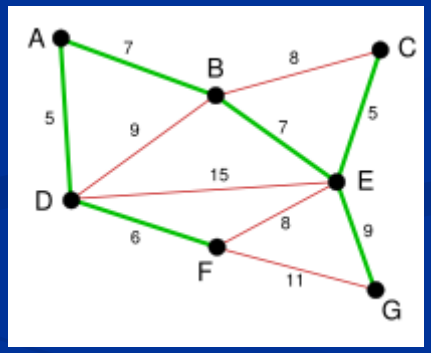
Kruskal's Example

- Finally, we will consider both BC and EF (of length 8) but they will be rejected as they would create cycles
- So we consider BD and EG (of length 9)
  - BD would create a cycle
  - EG will not so we choose it



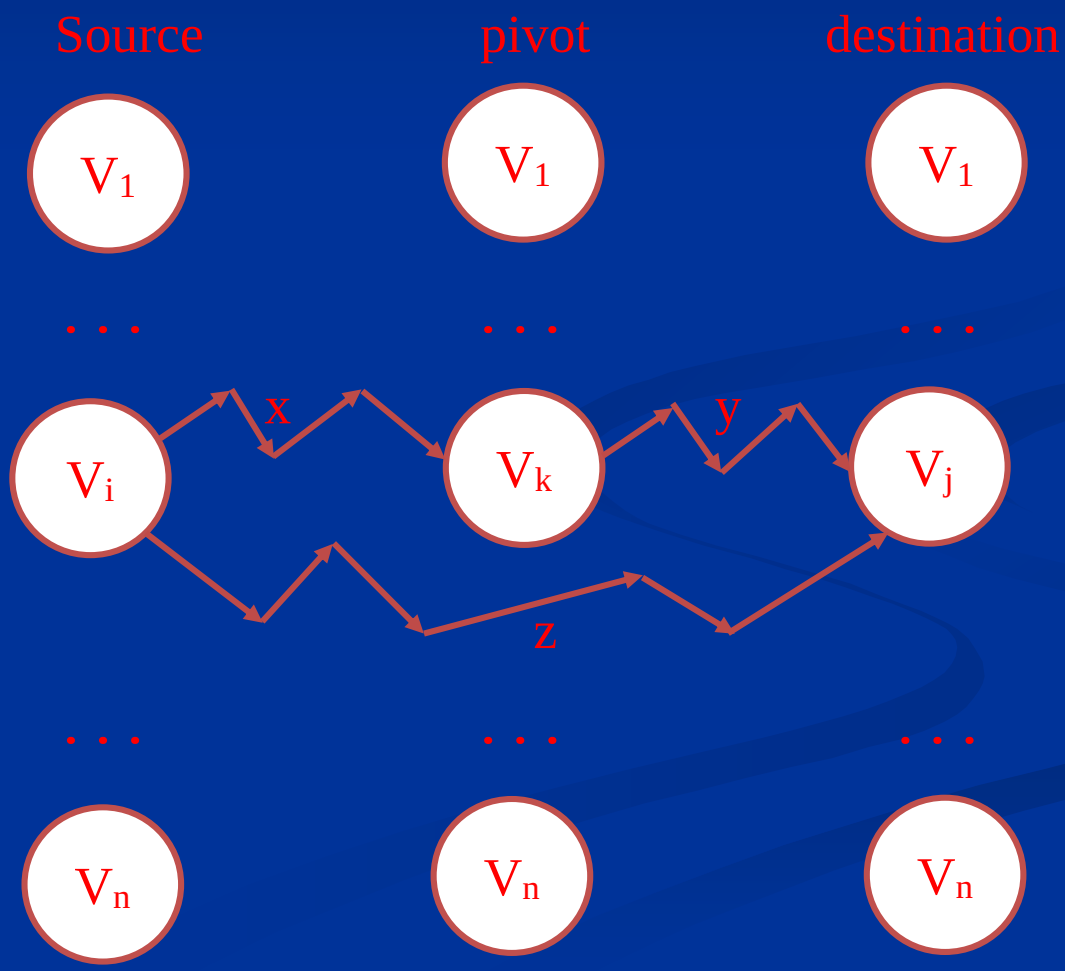
Kruskal's Example

- We've chosen n-1 edges so we've created our minimum spanning tree!



Floyd-Warshall's Algorithm

All-Pairs-Shortest Paths

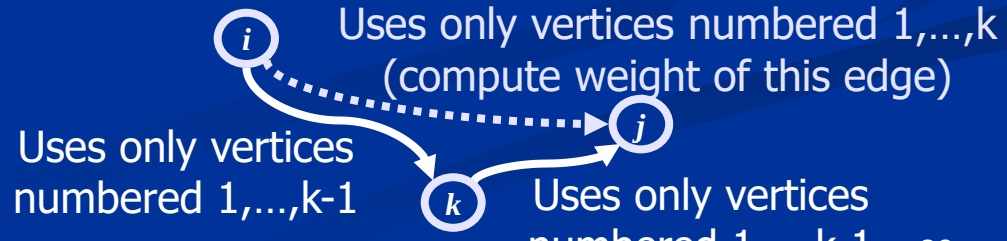


Floyd's All Shortest Paths

- This algorithm is known as "Floyd's" and is based on dynamic programming. Basically, the idea of the algorithm is to construct a matrix that gives the length of the shortest path between each pair of nodes.
- We'll initialize this matrix to our adjacency matrix and then perform N iterations. After each iteration k, the matrix will give the length of the shortest paths between each pair of nodes that use only nodes numbered 1 to k.
- At each iteration, the algorithm must check for each pair of nodes (i, j), whether or not there exists a path from i to j going through the node k that is better than the current shortest path using only nodes 1 to k-1.
- After n iterations, the matrix will give the length of the shortest paths using any of the nodes, which is the result we want.

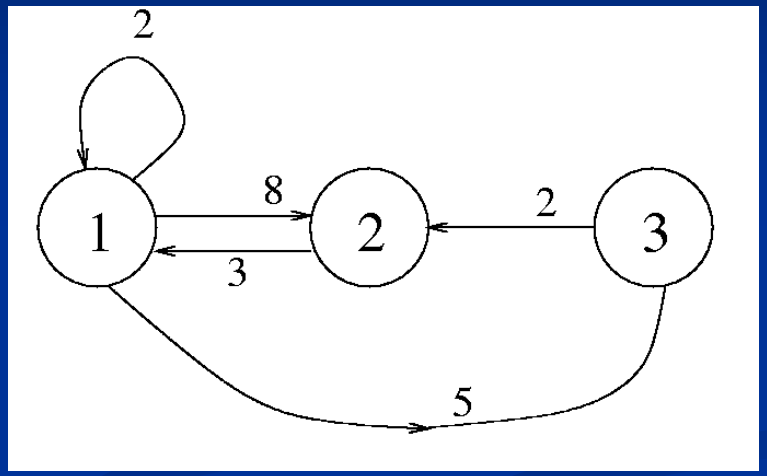
Floyd's Algorithm

```
adj:array[1..N, 1..N] of Integer
A:array[1..N, 1..N] of Integer  (A is our shortest path
   matrix)

Floyd()
{
   A = adj;
   For k=1 to N
   {
        for i=1 to N
        {
              for j=1 to N
                    A[i,j] = MIN(A[i,j], A[i,k]+A[k,j])
        }
   }
}
```



Floyd's Example

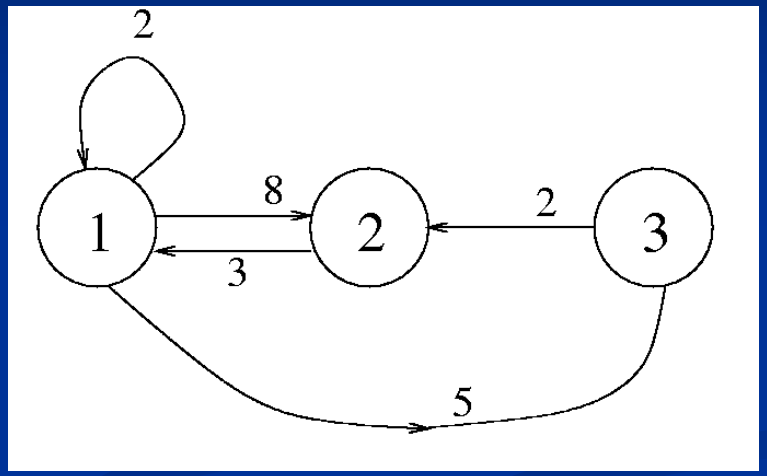
- Consider this graph with the adjacency matrix C
- Be careful with self-loops
- Need to zero out the diagonals
  - But may be problem dependent!
- So we'll start with a modified adjacency matrix A_0



$$A_0 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix}$$

Floyd's Example

- Then we go through a loop where we update the shortest path from each node i to node j going through nodes 1 to k



- So in step 1, we update each shortest path using only node 1

```
A[i,j] = MIN(A[i,j], A[i,k]+A[k,j])
A[2,3] = MIN(A[2,3], A[2,1]+A[1,3])
A[2,3] = MIN(Inf, 3+5) = 8
```

  - All others remain unchanged this loop through

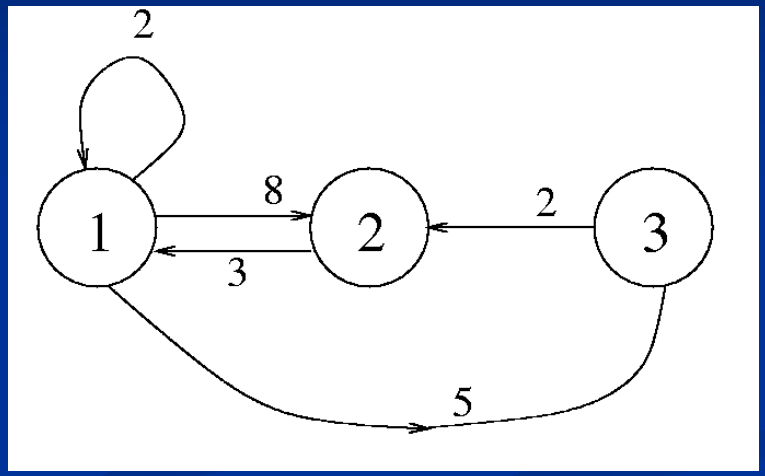
$$A_1 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ \infty & 2 & 0 \end{bmatrix}$$

Floyd's Example

- Loop k=2
- In step 2, we update each shortest path using only nodes 1 and 2

```
A[i,j] = MIN(A[i,j], A[i,k]+A[k,j])
A[3,1] = MIN(A[3,1], A[3,2]+A[2,1])
A[3,1] = MIN(Inf, 2+3) = 5
```

  - All others remain unchanged this loop through



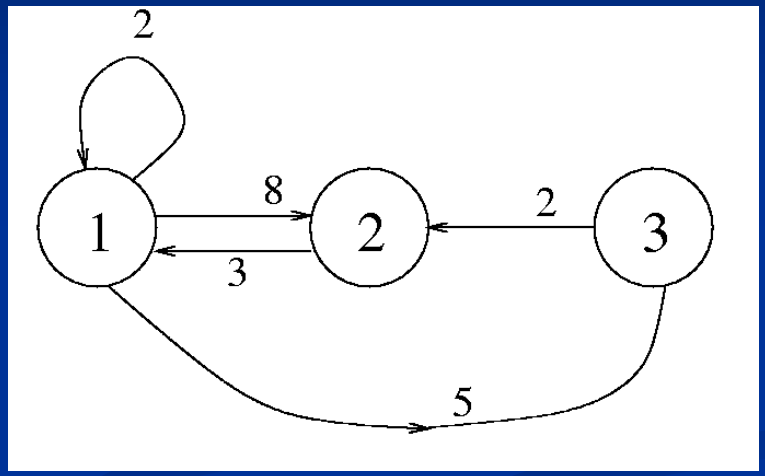
$$A_2 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}$$

Floyd's Example

- Loop k=3
- In step 3, we update each shortest path using only nodes 1, 2 and 3

```
A[i,j] = MIN(A[i,j], A[i,k]+A[k,j])
A[1,2] = MIN(A[1,2], A[1,3]+A[3,2])
A[1,2] = MIN(8, 5+2) = 7
```

  - All others remain unchanged this loop through



$$A_3 = \begin{bmatrix} 0 & 7 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}$$

Graph Coloring Problem

- Goal: Assign colors to each vertex in a graph such that no two vertices that are connected by an edge are the same color.
- The minimum number of colors necessary to complete a coloring for a particular graph is its chromatic number.

Two Color Problem

- The two color problem is as follows: Given a graph, determine whether or not two colors suffices to color it.
- There is a relatively quick solution to this problem.
- However, no such solution for three colors exists. The three-color problem is NP-Complete, which, for now, is a way of saying that the problem is quite difficult.

Two Color Algorithm

- Pick a start node arbitrarily and color it black.
- Using, essentially a BFS, color all neighbors of this node white.
- Continue the BFS, coloring all neighbors of black nodes white and all neighbors of white nodes black.
- If a conflict ever arises, (you need to color a node already assigned a color the opposite color), answer NO!

Two Color Alg. Cont.

- If all nodes are assigned a color, then clearly the answer is YES!
- One trick: If the graph isn't connected, you need to make sure and check all connected components. Thus, if the initial BFS doesn't hit all nodes, then you must start another BFS from any uncolored node. Repeat until all nodes have been colored.

Summary

- Graphs are useful for representing lots of different problems
  - A major topic area for the ICPC
- A variety of algorithms
  - We've covered the most important today but there are others
  - Get to know these algorithms very well!
  - Watch the run-time

Thanks!

- Thanks for images and other information to:
  - Wikipedia
  - Changhui (Charles) Yan
    - http://www.cs.usu.edu/~cyan/CS5050/