Graphs

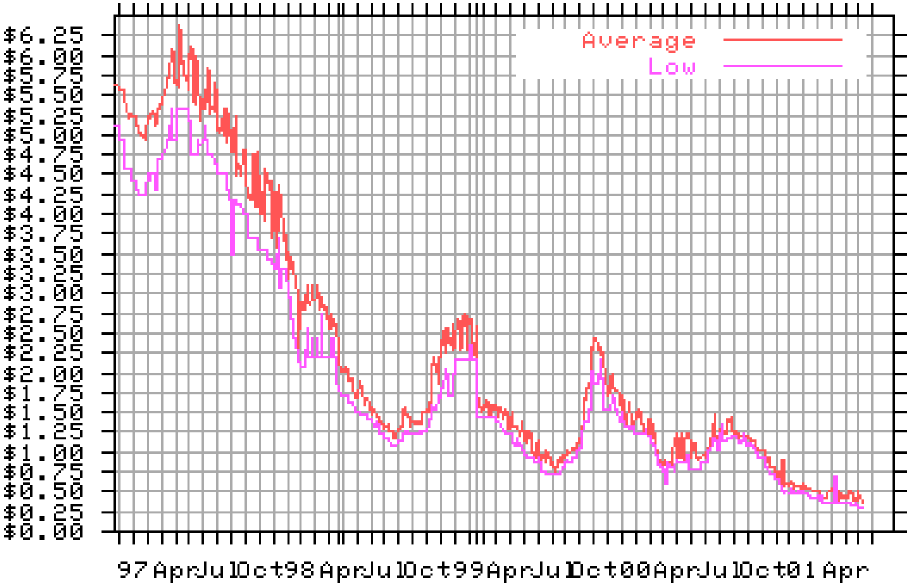
COL 106

Slide Courtesy : http://courses.cs.washington.edu/courses/cse373/

Douglas W. Harder, U Waterloo

What are graphs?

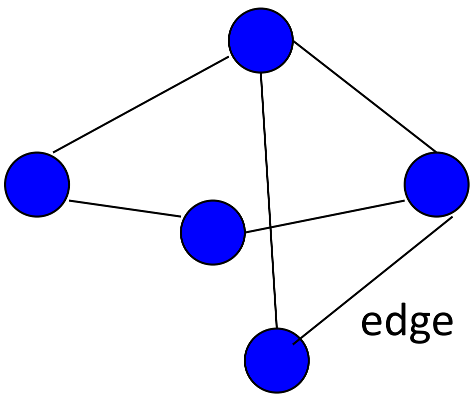
- Yes, this is a graph….



- But we are interested in a different kind of “graph”

Graphs

- Graphs are composed of
  - Nodes (vertices)
  - Edges (arcs)

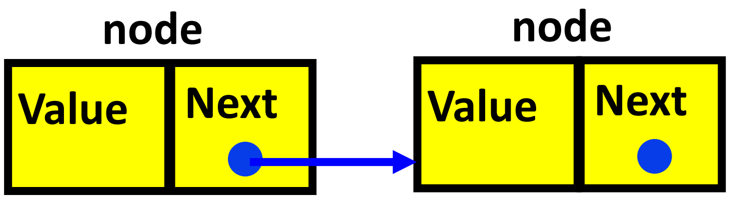


Varieties

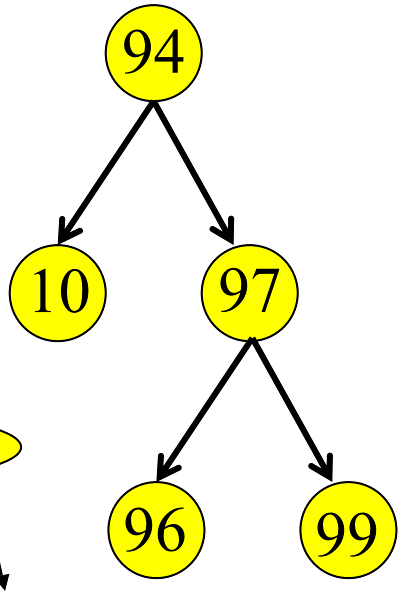
- Nodes
  - Labeled or unlabeled
- Edges
  - Directed or undirected
  - Labeled or unlabeled

Motivation for Graphs

- Consider the data structures we have looked at so far…

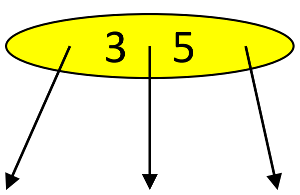


- Linked list: nodes with 1 incoming edge + 1 outgoing edge



- Binary trees/heaps: nodes with 1 incoming edge + 2 outgoing edges

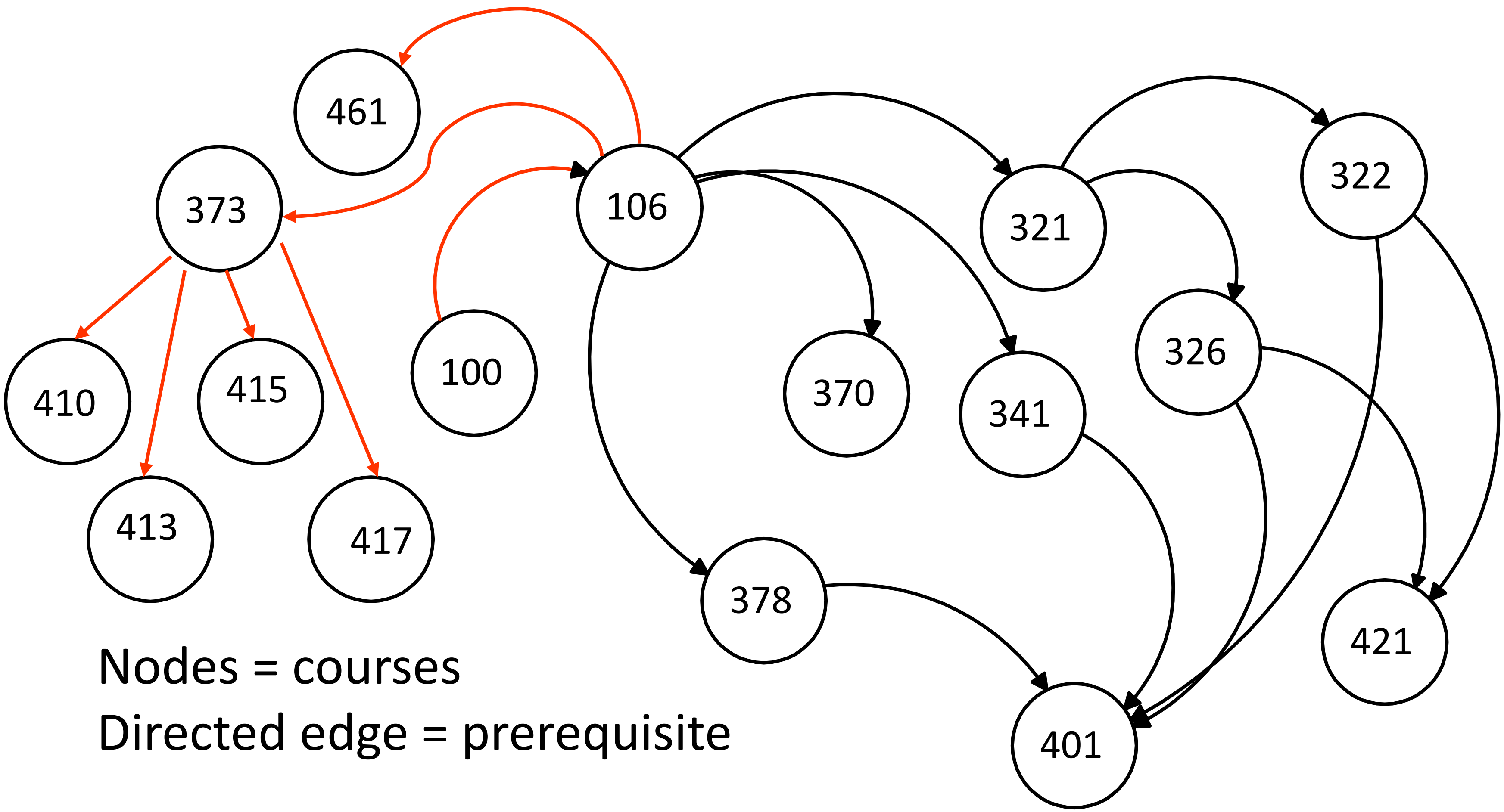
- B-trees: nodes with 1 incoming edge + multiple outgoing edges



Motivation for Graphs

- How can you generalize these data structures?
- Consider data structures for representing the following problems…

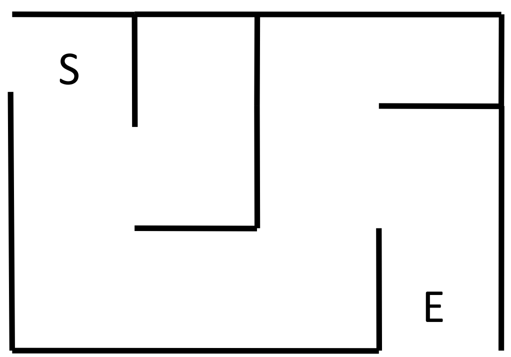
CSE Course Prerequisites

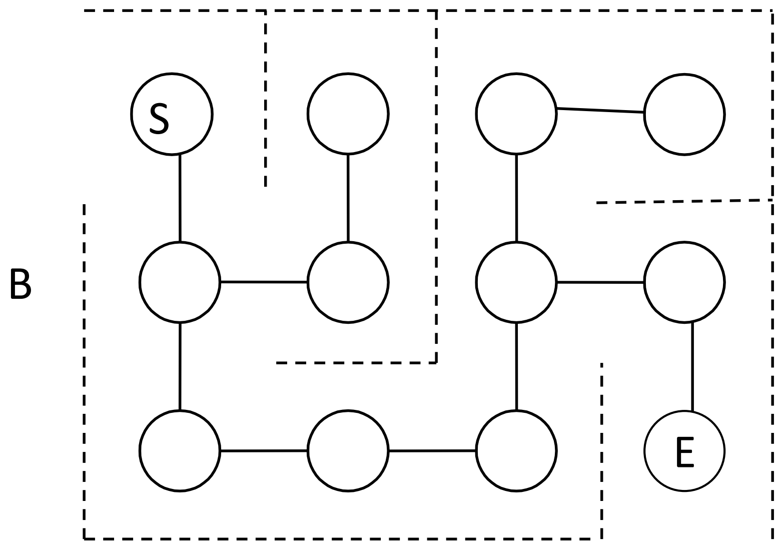


Nodes = courses

Directed edge = prerequisite

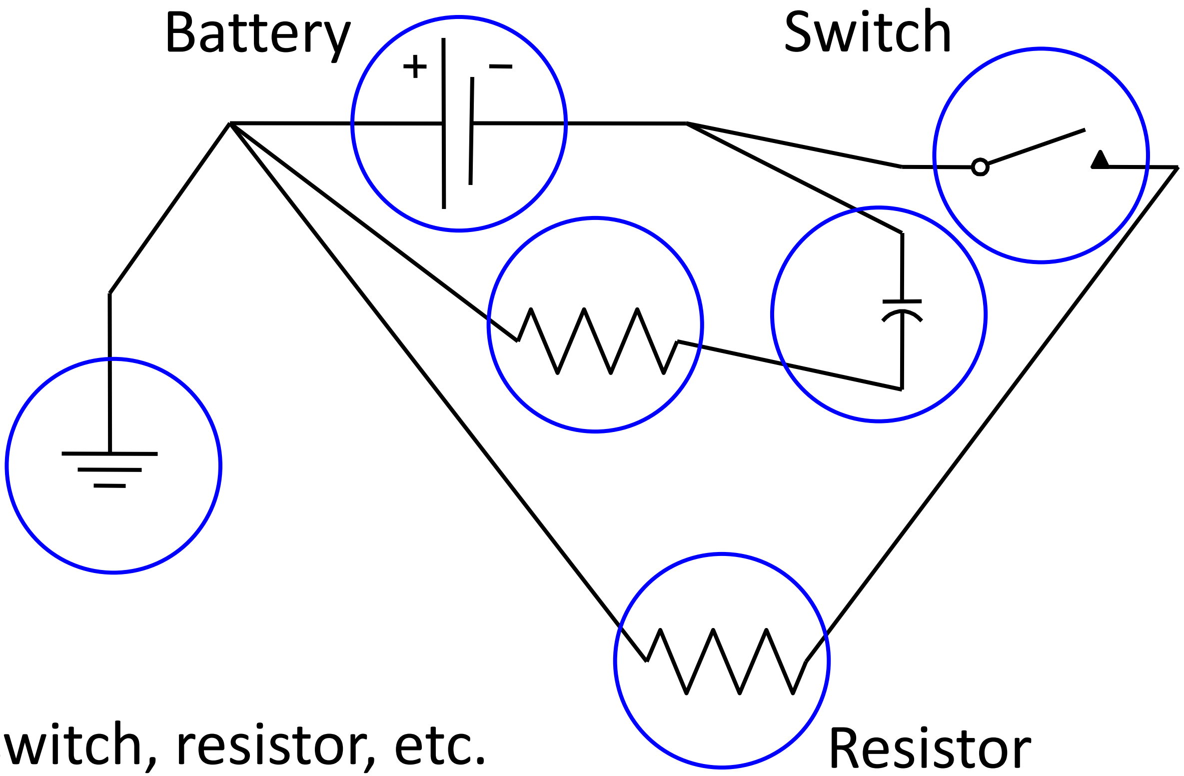
Representing a Maze





Nodes = rooms
Edge = door or passage

Representing Electrical Circuits

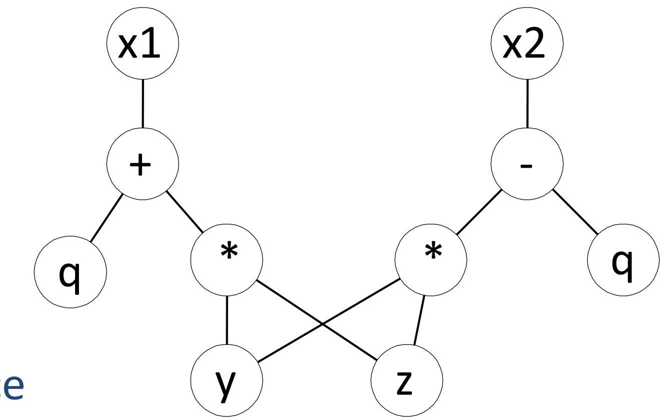


Nodes = battery, switch, resistor, etc.
Edges = connections

Program statements

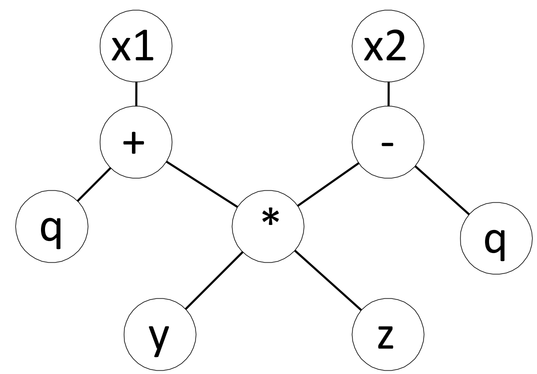
```
x1=q+y*z
x2=y*z-q
```

Naive:



y*z calculated twice

common subexpression eliminated:



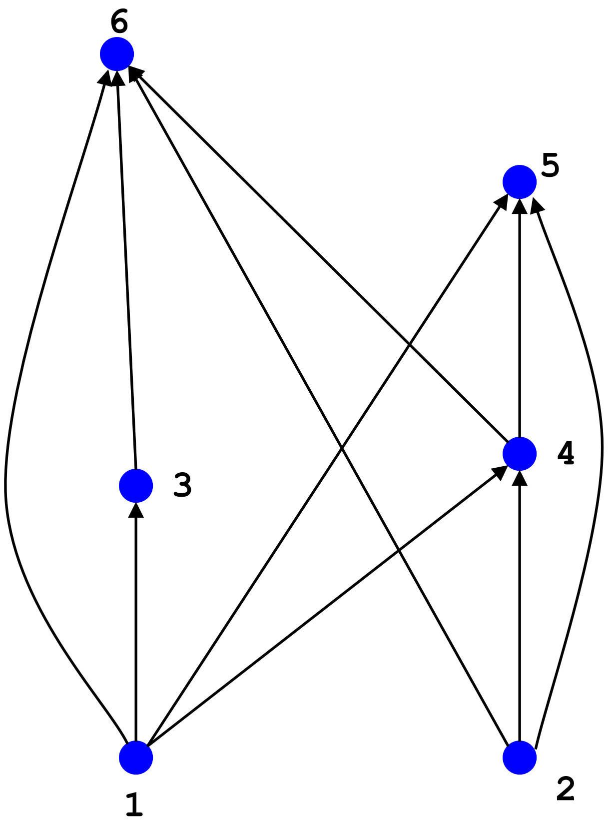
Nodes = symbols/operators
Edges = relationships

Precedence

```
S1  a=0;
S2  b=1;
S3  c=a+1
S4  d=b+a;
S5  e=d+1;
S6  e=c+d;
```

Which statements must execute before S_6?

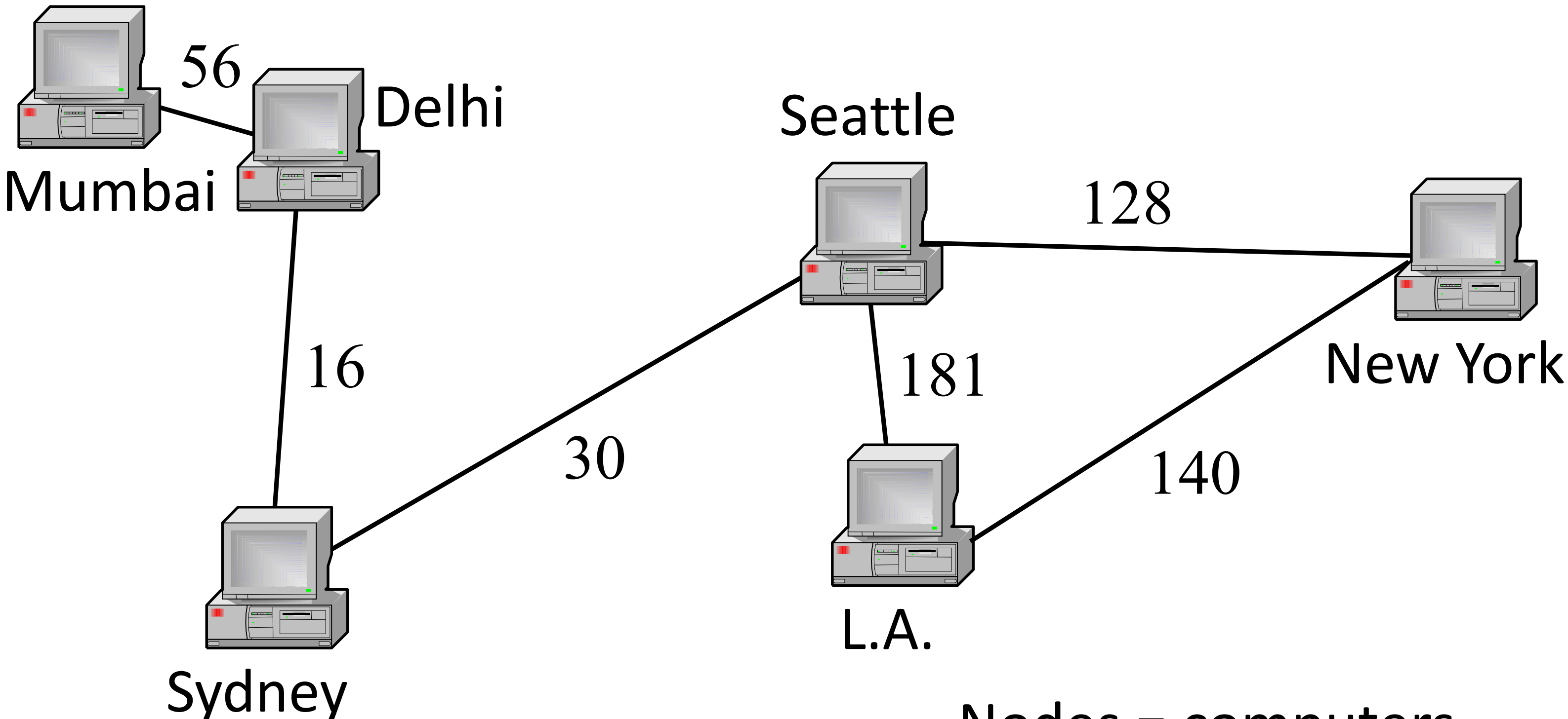
S_1, S_2, S_3, S_4



Nodes = statements

Edges = precedence requirements

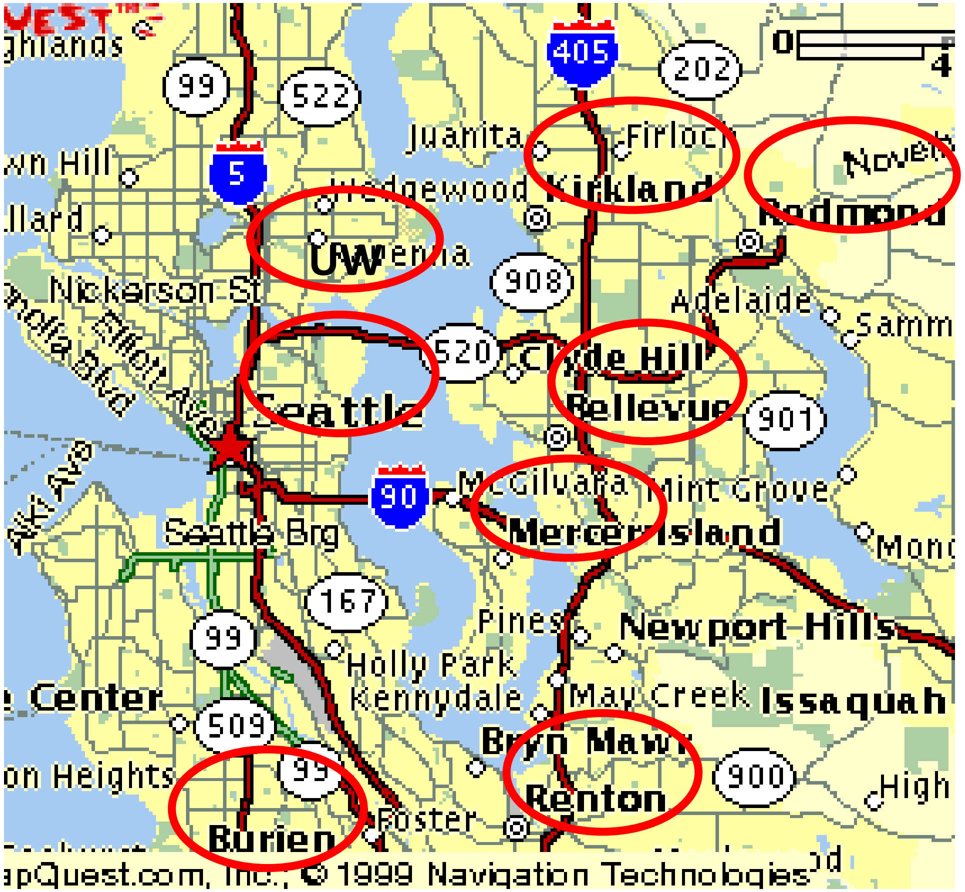
Information Transmission in a Computer Network



Nodes = computers

Edges = transmission rates

Traffic Flow on Highways



Nodes = cities

Edges = # vehicles on connecting highway

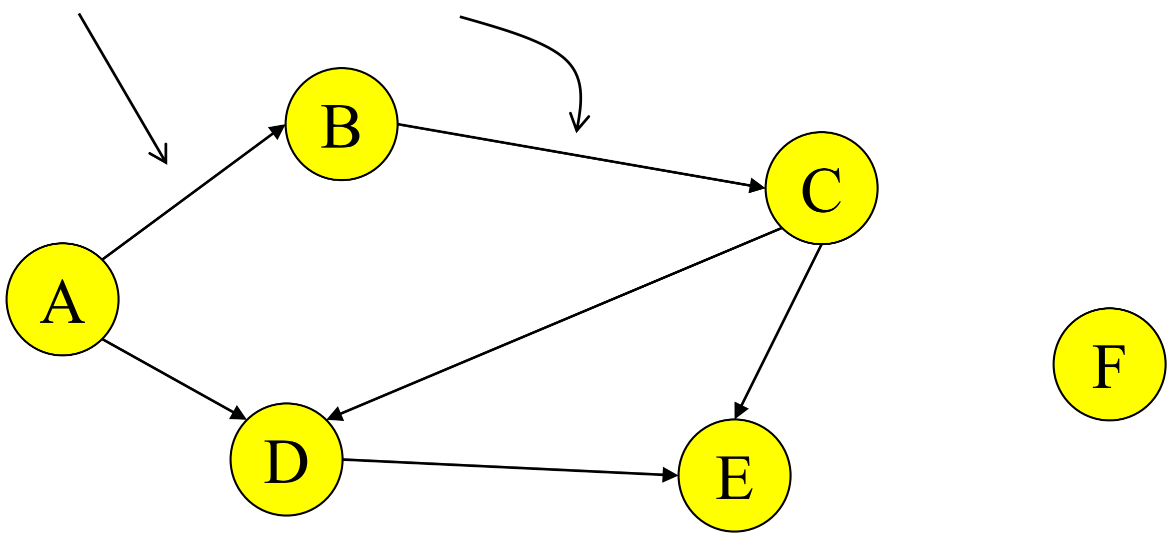
Graph Definition

- A graph is simply a collection of nodes plus edges
  - Linked lists, trees, and heaps are all special cases of graphs
- The nodes are known as vertices (node = "vertex")
- Formal Definition: A graph G is a pair (V, E) where
  - V is a set of vertices or nodes
  - E is a set of edges that connect vertices

Graph Example

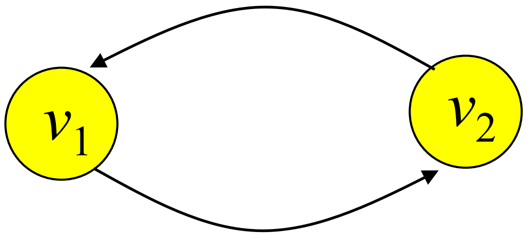
- Here is a directed graph $G = (V, E)$
  - Each edge is a pair (v_1, v_2), where v_1, v_2 are vertices in V
  - V = {A, B, C, D, E, F}

  E = {(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)}

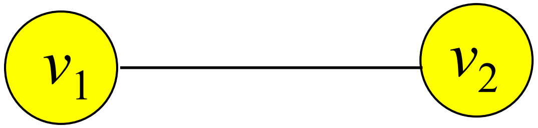


Directed vs Undirected Graphs

- If the order of edge pairs (v_1, v_2) matters, the graph is directed (also called a digraph): (v_1, v_2) ≠(v_2, v_1)



- If the order of edge pairs (v_1, v_2) does not matter, the graph is called an undirected graph: in this case, (v_1, v_2) = (v_2, v_1)



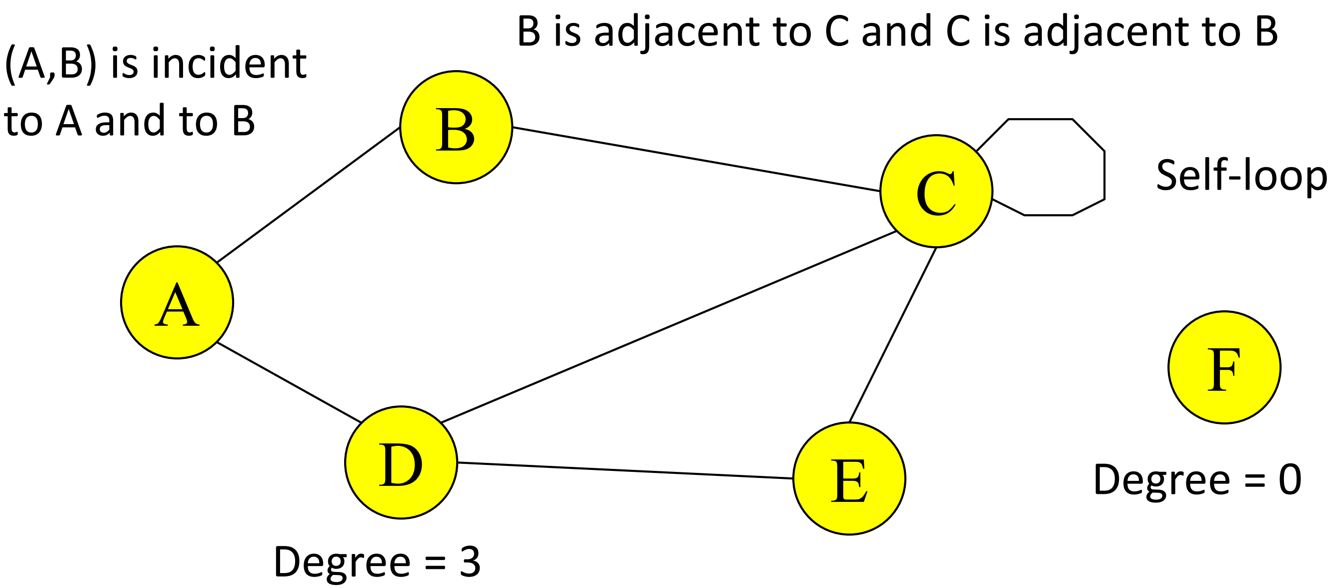
Undirected Terminology

- Two vertices u and v are adjacent in an undirected graph G if {u,v} is an edge in G
  - edge e = {u,v} is incident with vertex u and vertex v

- A graph is connected if given any two vertices u and v, there is a path from u to v

- The degree of a vertex in an undirected graph is the number of edges incident with it
  - a self-loop counts twice (both ends count)
  - denoted with deg(v)

Undirected Terminology

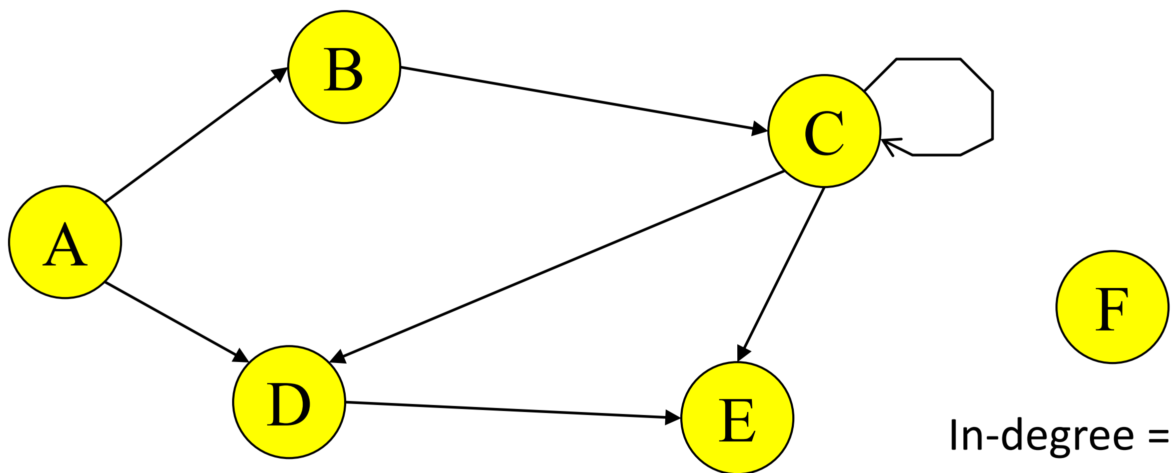


Directed Terminology

- Vertex u is adjacent to vertex v in a directed graph G if (u,v) is an edge in G
  - vertex u is the initial vertex of (u,v)
- Vertex v is adjacent from vertex u
  - vertex v is the terminal (or end) vertex of (u,v)
- Degree
  - in-degree is the number of edges with the vertex as the terminal vertex
  - out-degree is the number of edges with the vertex as the initial vertex

Directed Terminology

B adjacent to C and C adjacent from B

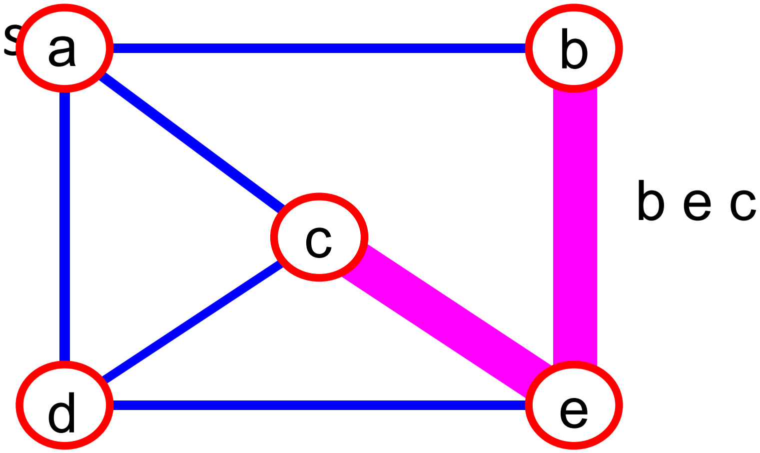


In-degree = 2
Out-degree = 1

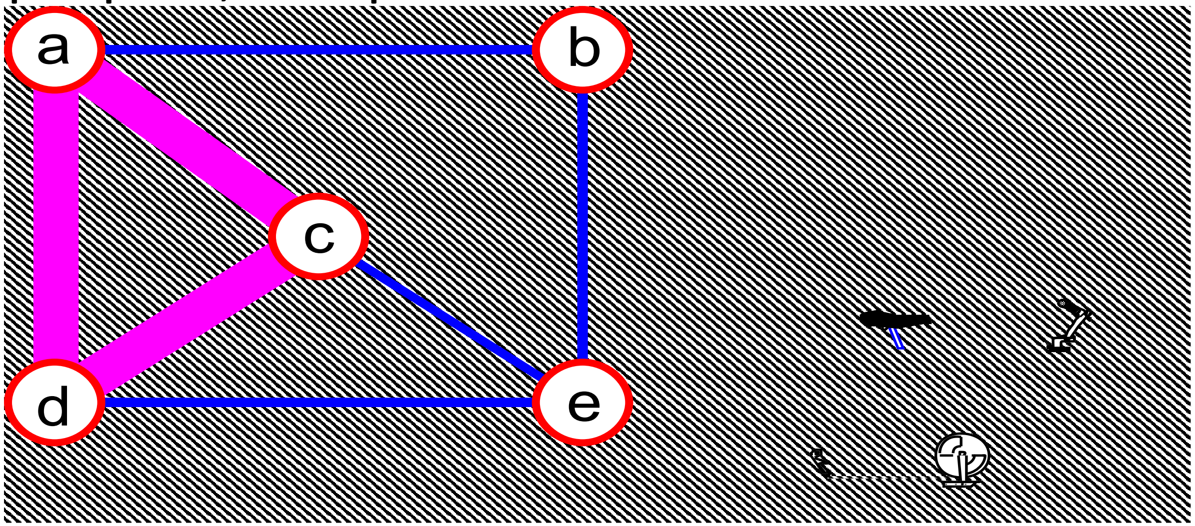
In-degree = 0
Out-degree = 0

More Graph Terminology

- simple path: no repeated vertices

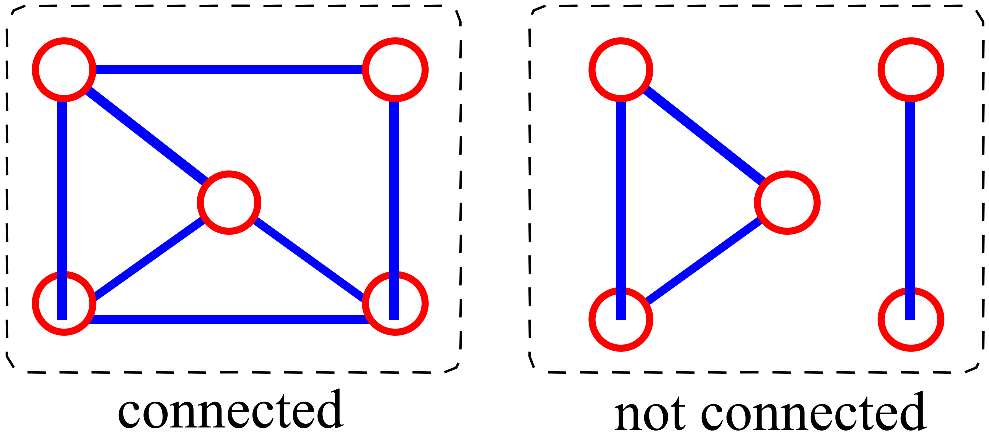


- cycle: simple path, except that the last vertex is the same as the first vertex

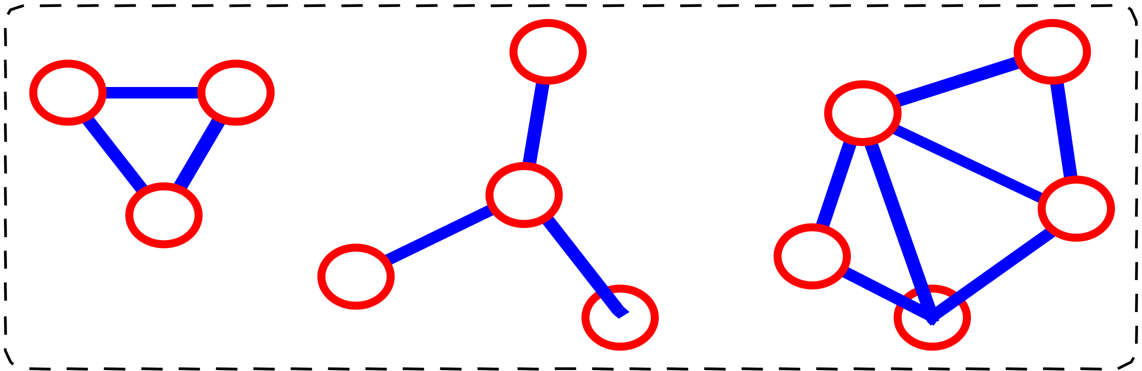


Even More Terminology

- connected graph: any two vertices are connected by some path

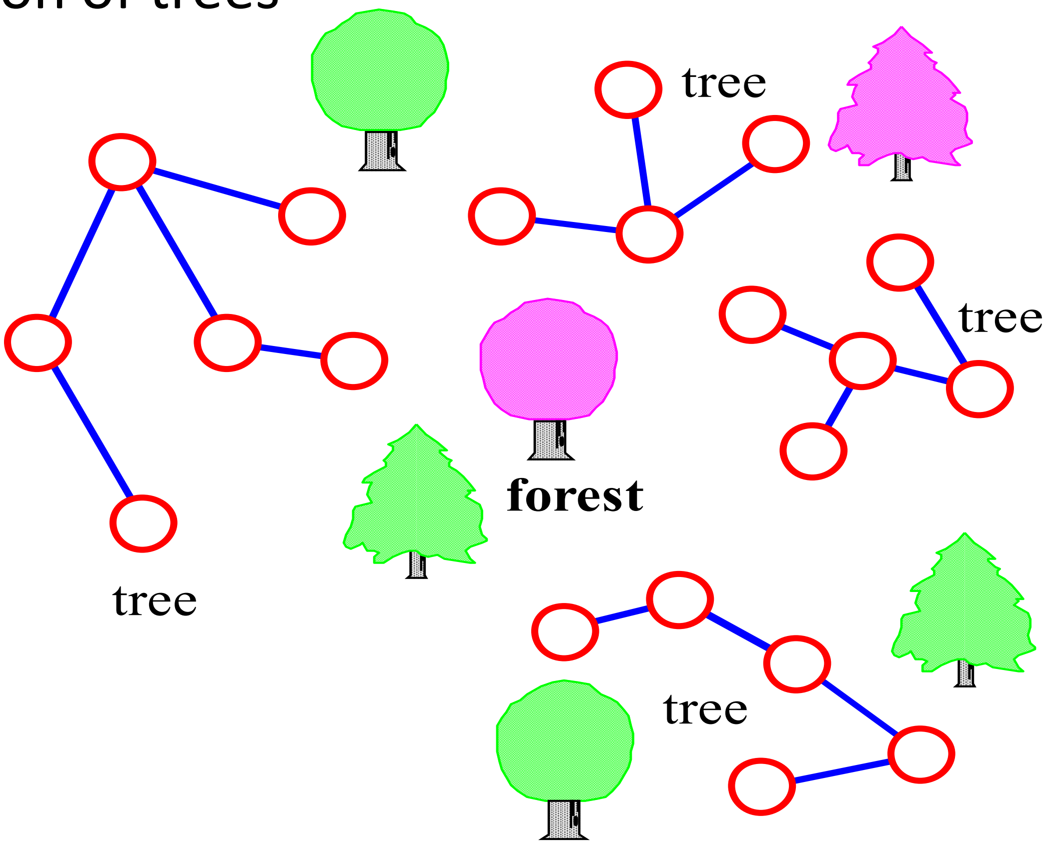


- subgraph: subset of vertices and edges forming a graph
- connected component: maximal connected subgraph. E.g., the graph below has 3 connected components.



Trees from Perspective of Graphs

- (free) tree - connected graph without cycles
- forest - collection of trees



Handshaking Theorem

- Let G=(V,E) be an undirected graph with |E|=e edges. Then

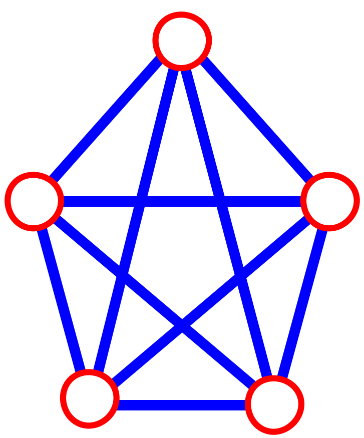
$$2e = \sum_{v \in V} \deg(v)$$

Add up the degrees of all vertices.

- Every edge contributes +1 to the degree of each of the two vertices it is incident with
  - number of edges is exactly half the sum of deg(v)
  - the sum of the deg(v) values must be even

Connectivity

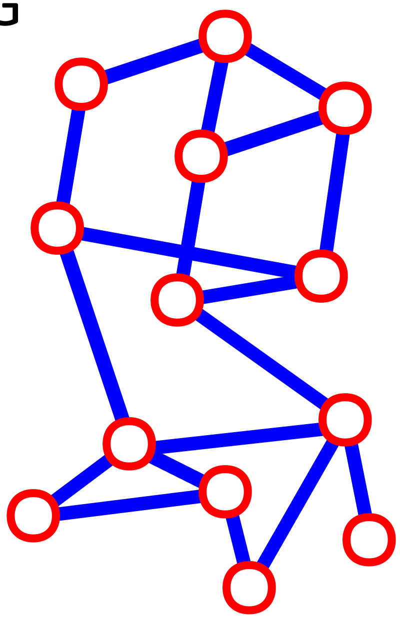
- Let **n** = #vertices, and **m** = #edges
- **A complete graph**: one in which all pairs of vertices are adjacent
- *How many total edges in a complete graph?*
  - Each of the n vertices is incident to **n**-1 edges, however, we would have counted each edge twice!!! Therefore, intuitively, m = **n**(**n** -1)/2.
- Therefore, if a graph is not complete, m < **n**(**n** -1)/2



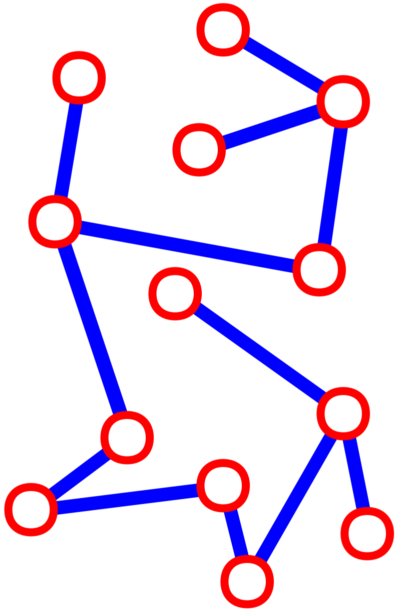
n = 5
m = (5 ∗ 4)/2 = 10

Spanning Tree

- A spanning tree of G is a subgraph which is a tree and which contains all vertices of G



G



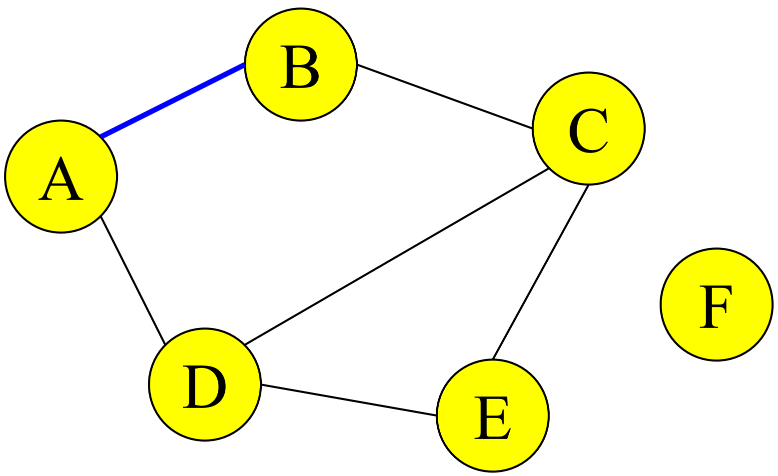
spanning tree of **G**

- Failure on any edge disconnects system (least fault tolerant)

Graph Representations

- Space and time are analyzed in terms of:
    - Number of vertices = $|V|$ and
    - Number of edges = $|E|$
- There are at least two ways of representing graphs:
    - The *adjacency matrix* representation
    - The *adjacency list* representation

Adjacency Matrix

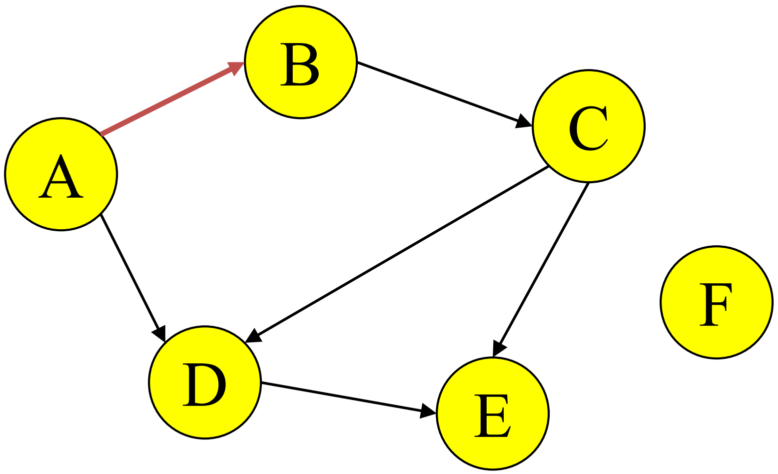


$$
\begin{array}{c|cccccc}
 & A & B & C & D & E & F \\
\hline
A & 0 & 1 & 0 & 1 & 0 & 0 \\
B & 1 & 0 & 1 & 0 & 0 & 0 \\
C & 0 & 1 & 0 & 1 & 1 & 0 \\
D & 1 & 0 & 1 & 0 & 1 & 0 \\
E & 0 & 0 & 1 & 1 & 0 & 0 \\
F & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

$$
M(v, w) = \begin{cases} 1 \text{ if } (v, w) \text{ is in E} \\ 0 \text{ otherwise} \end{cases}
$$

Space = $|V|^2$

Adjacency Matrix for a Digraph



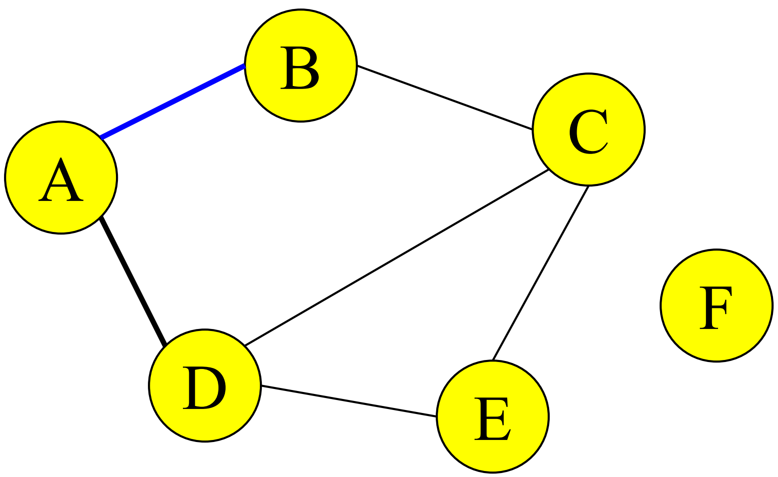
$$
\begin{array}{c|cccccc}
 & A & B & C & D & E & F \\
\hline
A & 0 & 1 & 0 & 1 & 0 & 0 \\
B & 0 & 0 & 1 & 0 & 0 & 0 \\
C & 0 & 0 & 0 & 1 & 1 & 0 \\
D & 0 & 0 & 0 & 0 & 1 & 0 \\
E & 0 & 0 & 0 & 0 & 0 & 0 \\
F & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

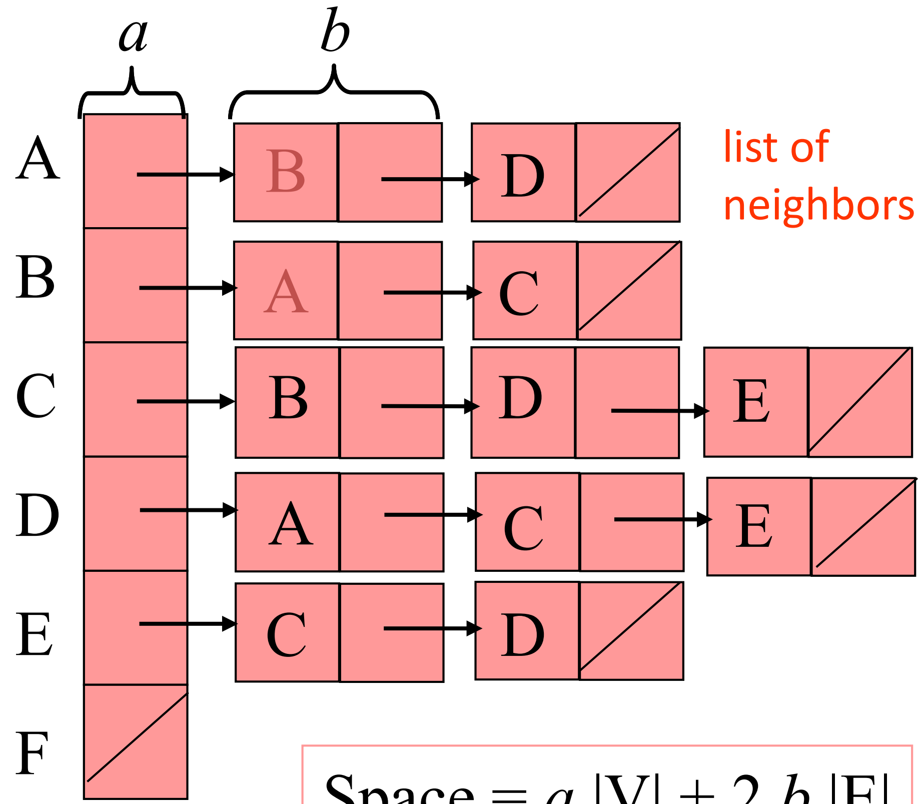
$$M(v, w) = \begin{cases} 1 \text{ if } (v, w) \text{ is in E} \\ 0 \text{ otherwise} \end{cases}$$

Space = $|V|^2$

Adjacency List

For each *v* in *V*, *L*(*v*) = list of *w* such that (*v*, *w*) is in *E*

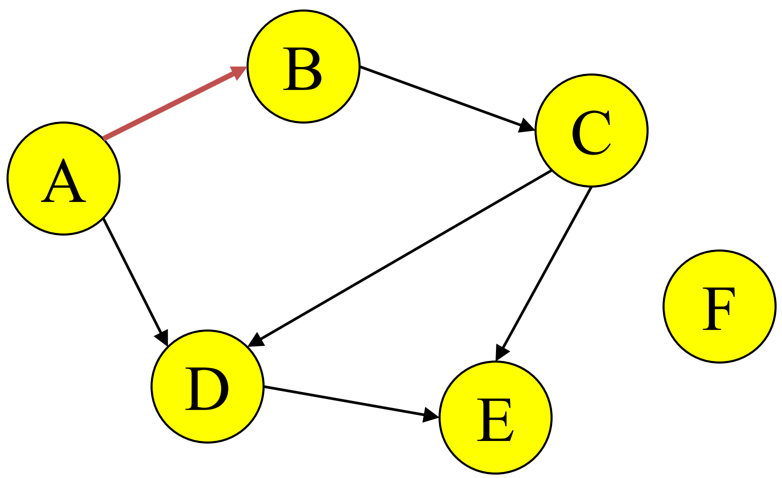


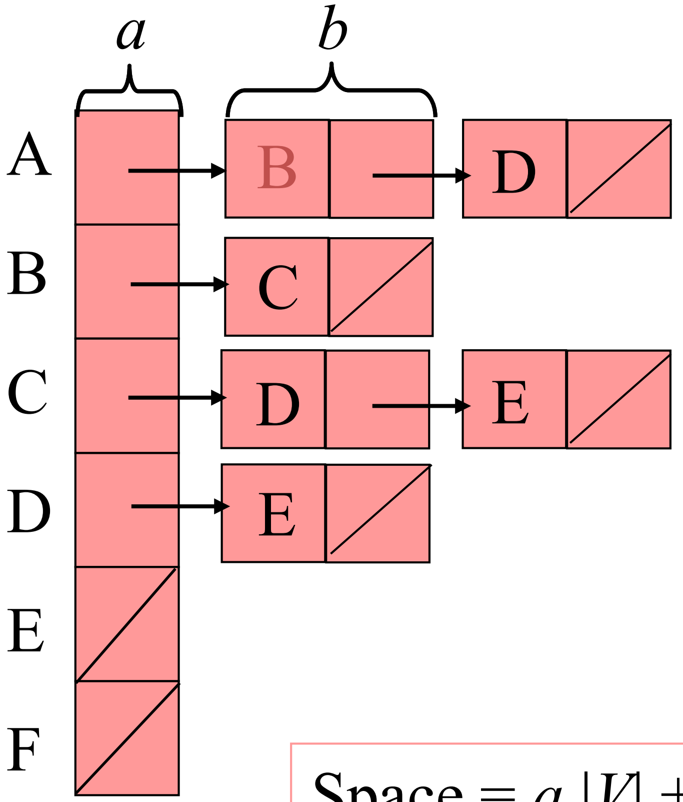


Space = $a\,|V| + 2\,b\,|E|$

Adjacency List for a Digraph

For each v in V, $L(v)$ = list of w such that (v, w) is in E





Space = $a\ |V| + b\ |E|$

Searching in graphs

- Find Properties of Graphs
  - Spanning trees
  - Connected components
  - Bipartite structure
  - Biconnected components
- Applications
  - Finding the web graph – used by Google and others
  - Garbage collection – used in Java run time system

Graph Searching Methodology Breadth-First Search (BFS)

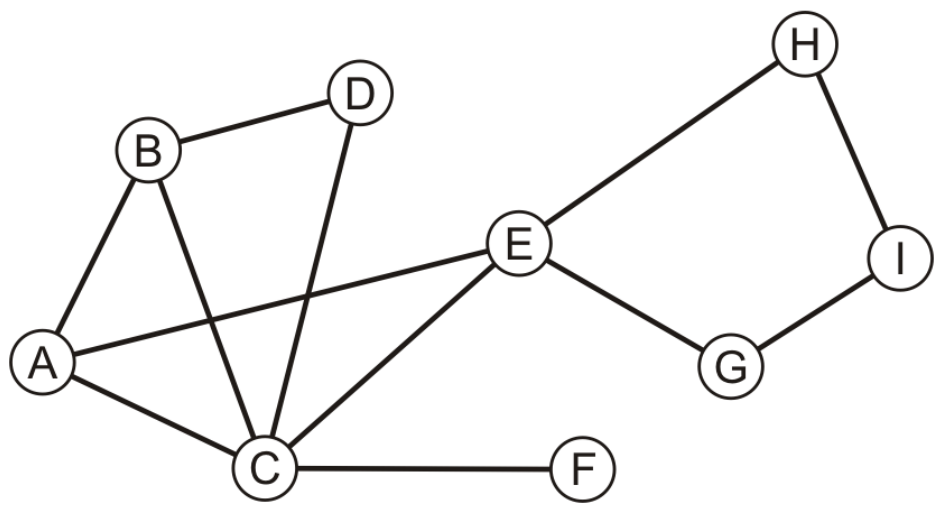
- Breadth-First Search (BFS)
  - Use a queue to explore neighbors of source vertex, then neighbors of neighbors etc.
  - All nodes at a given distance (in number of edges) are explored before we go further

Breadth-First Search

- A Breadth-First Search (BFS) traverses a connected component of a graph, and in doing so defines a spanning tree with several useful properties.
- The starting vertex *s* has level 0, and defines that point as an "anchor."
- In the first round, all of the nodes that are only one edge away from the anchor are visited.
- These nodes are placed into level 1
- In the second round, all the new nodes that can be reached by one edge from level 1 nodes are visited and placed in level 2.
- This continues until every vertex has been assigned a level.
- The label of any vertex ***v*** corresponds to the length of the shortest path from ***s*** to ***v***.

Example

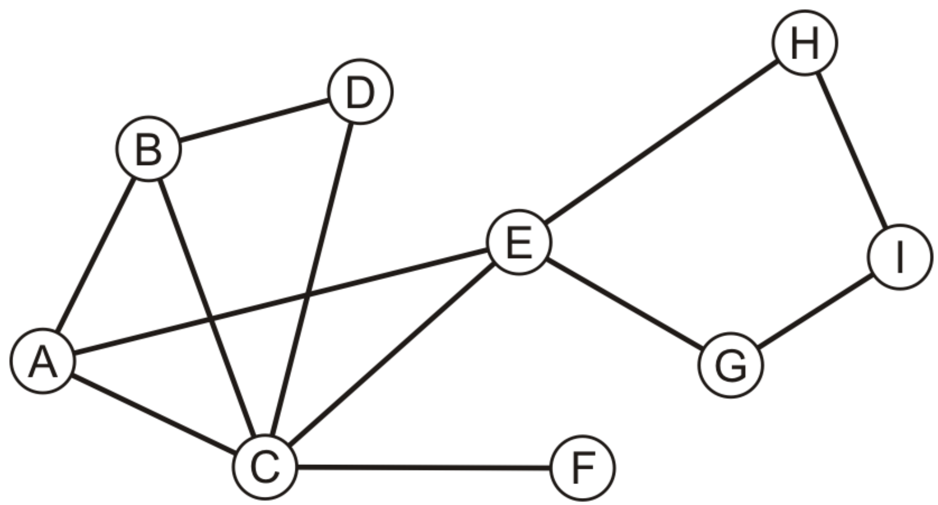
Consider the graph from previous example



Example

Performing a breadth-first traversal

– Push the first vertex onto the queue



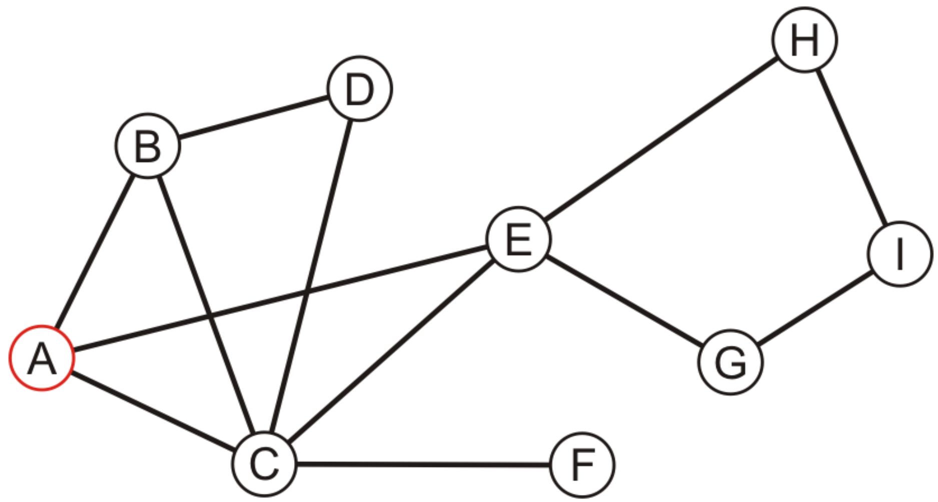
A					

Example

Performing a breadth-first traversal

– Pop A and push B, C and E

A



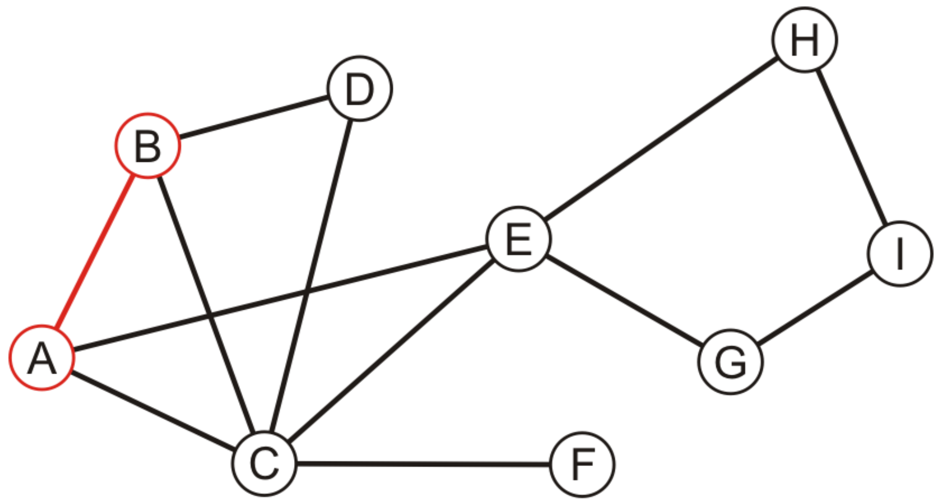
B	C	E			

Example

Performing a breadth-first traversal:

– Pop B and push D

A, B



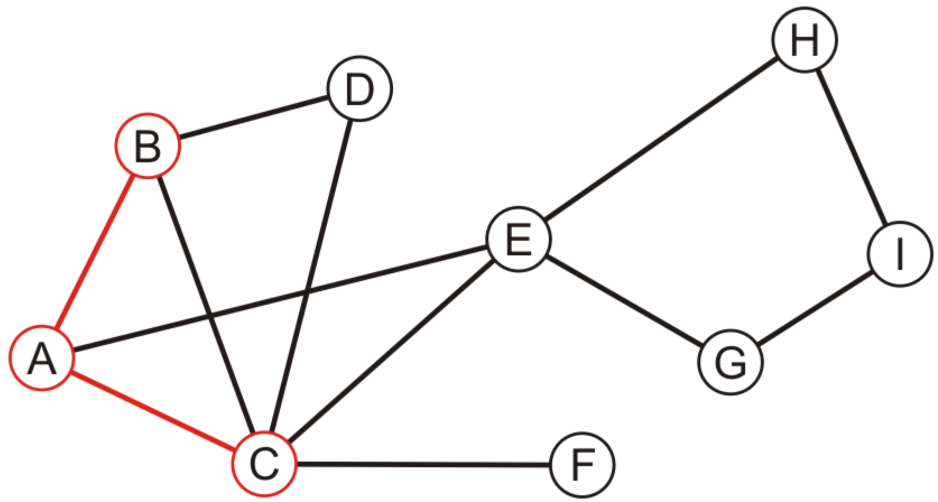
C	E	D			

Example

Performing a breadth-first traversal:

– Pop C and push F

A, B, C



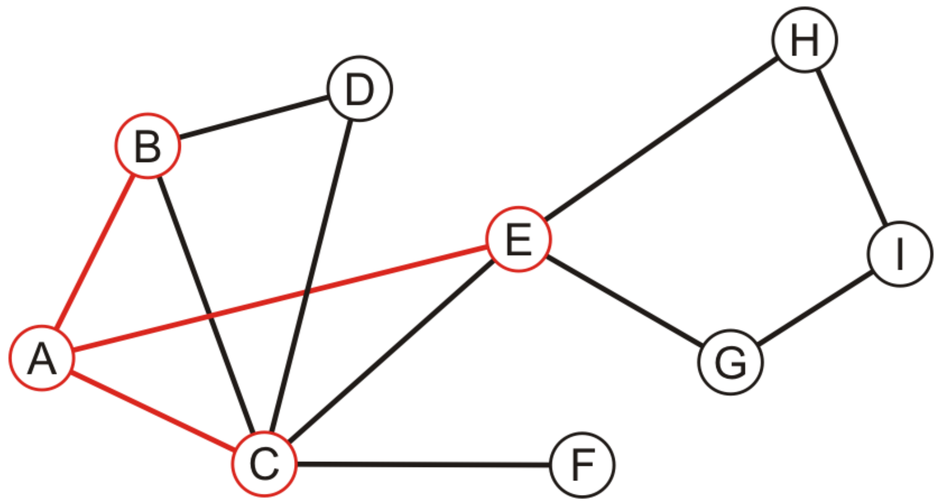
E	D	F			

Example

Performing a breadth-first traversal:

– Pop E and push G and H

A, B, C, E



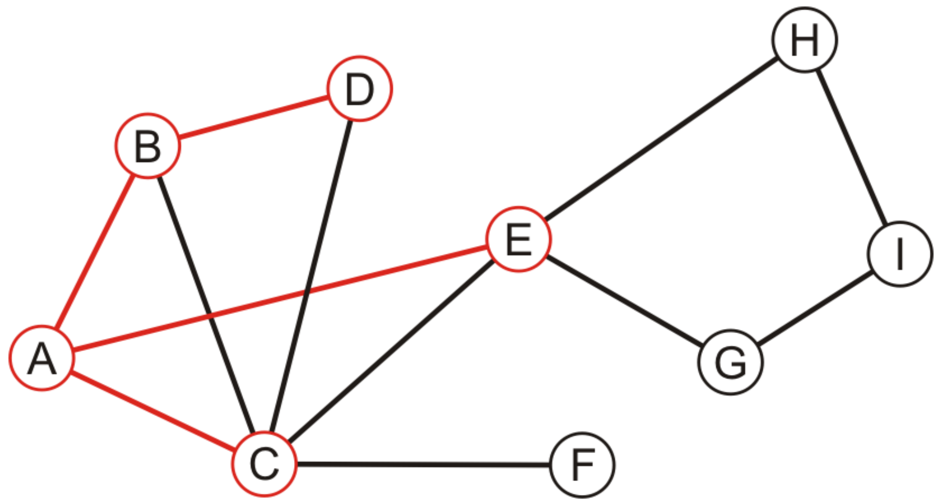
D	F	G	H		

Example

Performing a breadth-first traversal:

– Pop D

A, B, C, E, D



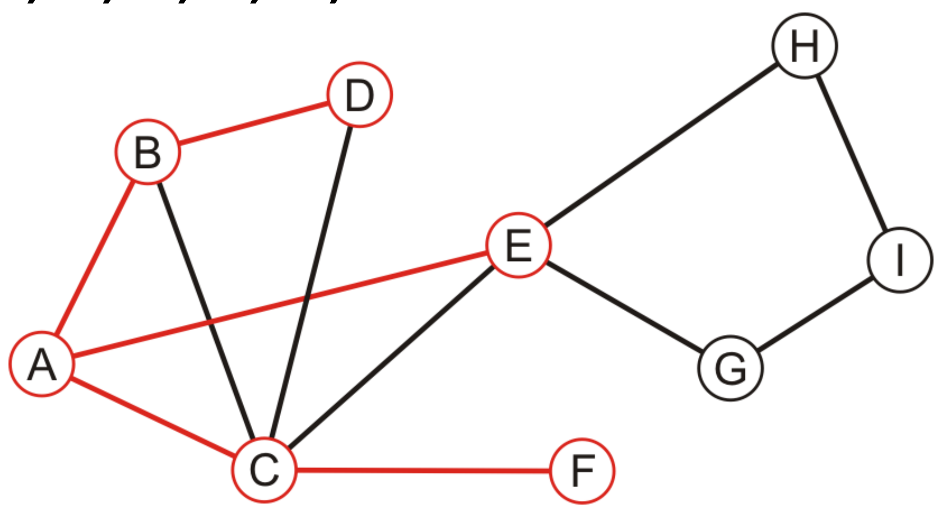
F	G	H			

Example

Performing a breadth-first traversal:

– Pop F

A, B, C, E, D, F



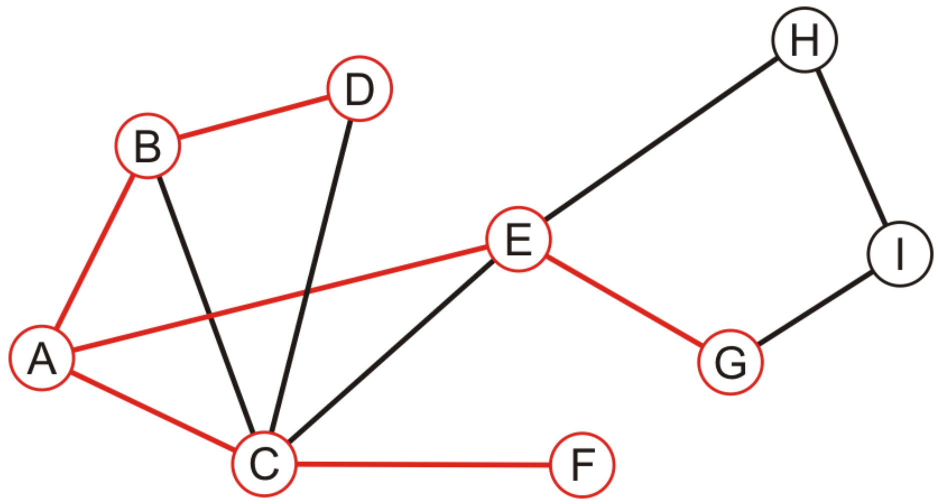
G	H				

Example

Performing a breadth-first traversal:

– Pop G and push I

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



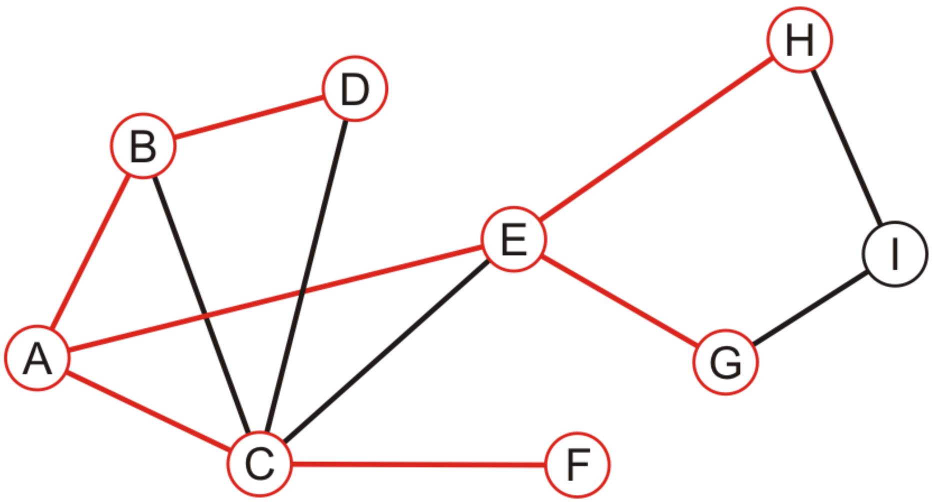
H	I				

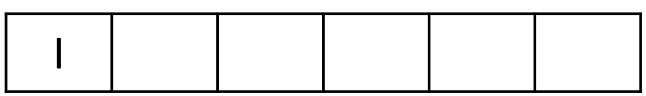
Example

Performing a breadth-first traversal:

– Pop H

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



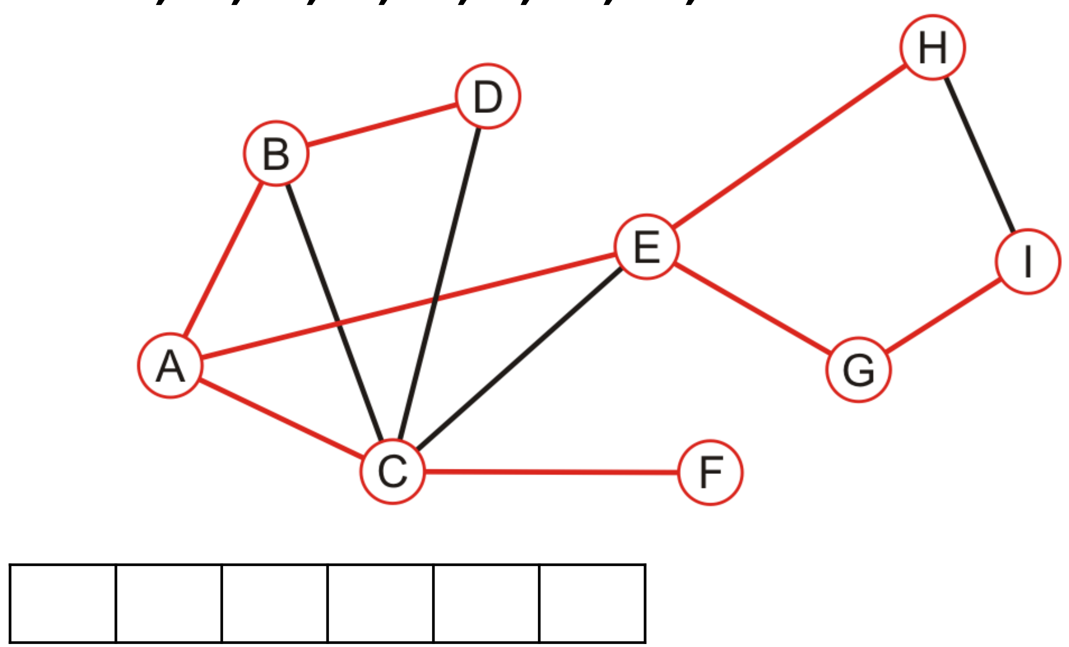


Example

Performing a breadth-first traversal:

– Pop I

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

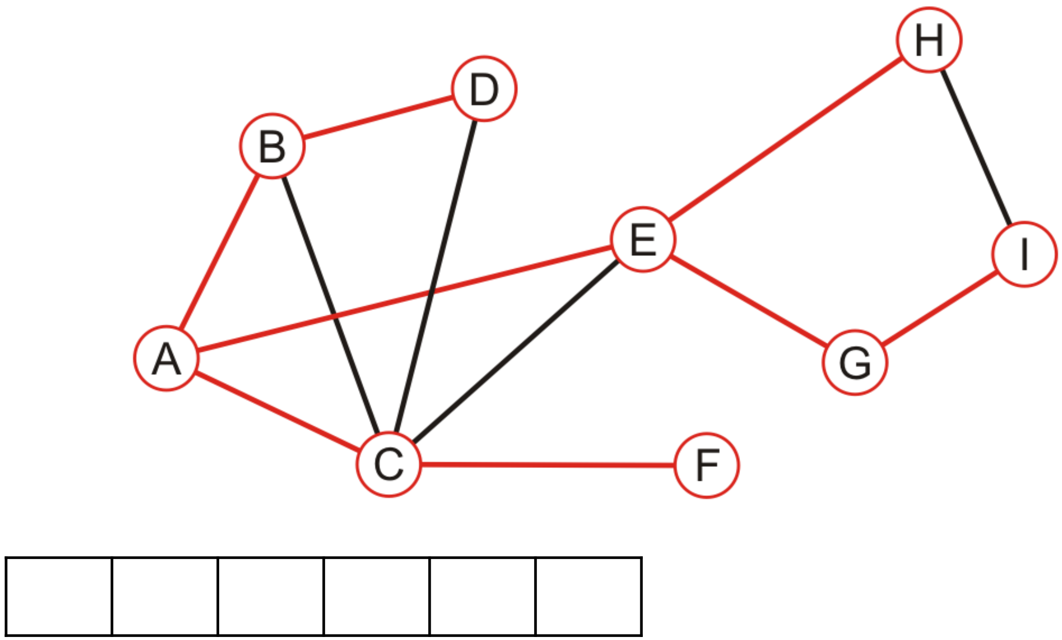


Example

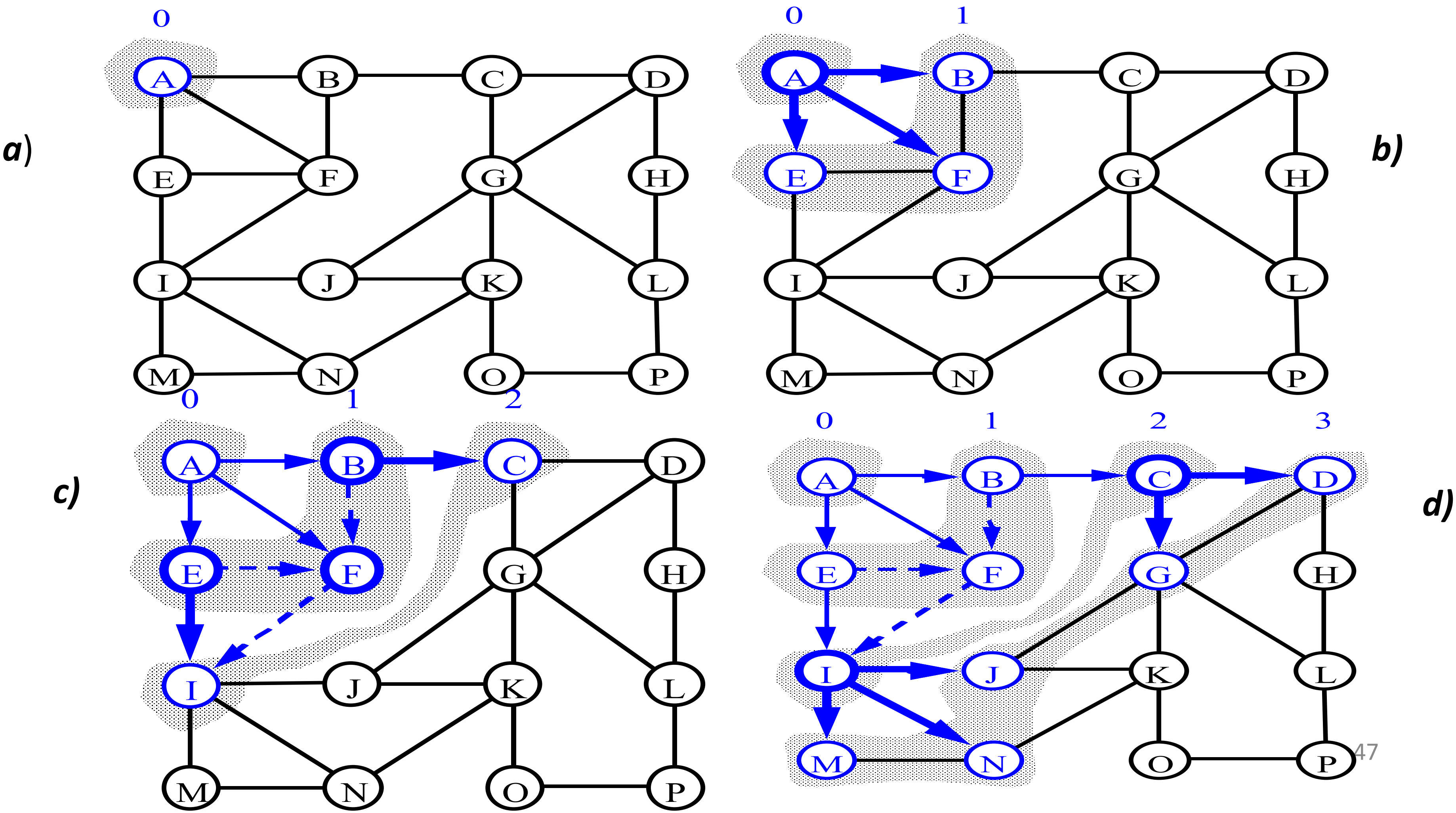
Performing a breadth-first traversal:

– The queue is empty: we are finished

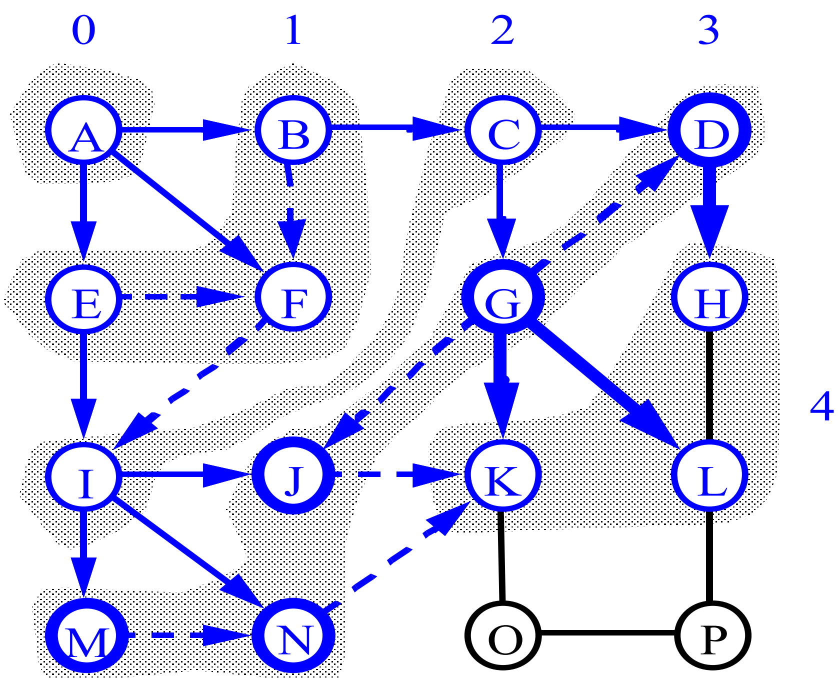
A, B, C, E, D, F, G, H, I

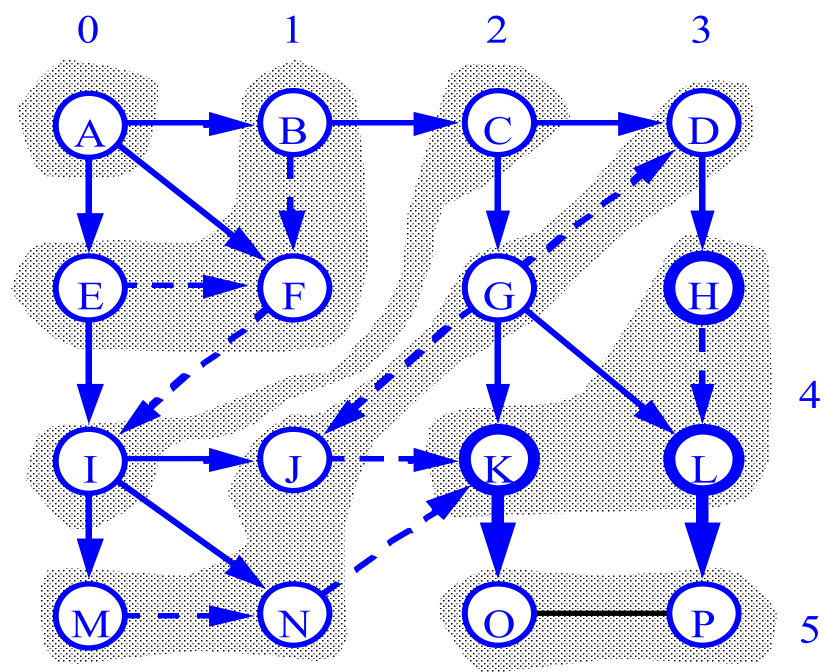


Another Example



Example Continued





Breadth-First Search

```
BFS
Initialize Q to be empty;
Enqueue(Q,1) and mark 1;
while Q is not empty do
   i := Dequeue(Q);
   for each j adjacent to i do
      if j is not marked then
         Enqueue(Q,j) and mark j;
end{BFS}
```

BFS Pseudo-Code

Algorithm BFS(s): Input: A vertex s in a graph

Output: A labeling of the edges as "discovery" edges and "cross edges"

```
initialize container L_0 to contain vertex s
i ← 0
while L_i is not empty do
    create container L_{i+1} to initially be empty
    for each vertex v in L_i do
      if edge e incident on v do
        let w be the other endpoint of e
        if vertex w is unexplored then
          label e as a discovery edge
            insert w into L_{i+1}
          else label e as a cross edge
            i ← i + 1
```

Properties of BFS

- Proposition: Let G be an undirected graph on which a a BFS traversal starting at vertex ***s*** has been performed. Then
  - The traversal visits all vertices in the connected component of ***s*** .
  - The discovery-edges form a spanning tree T, which we call the BFS tree, of the connected component of ***s***
  - For each vertex v at level i, the path of the BFS tree T between s and v has i edges, and any other path of G between ***s*** and v has at least i edges.
  - If (u, v) is an edge that is not in the BFS tree, then the level numbers of u and v differ by at most one.
- Proposition: Let G be a graph with n vertices and m edges. A BFS traversal of G takes time O(n + m). Also, there exist O(n + m) time algorithms based on BFS for the following problems:
  - Testing whether G is connected.
  - Computing a spanning tree of G
  - Computing the connected components of G
  - Computing, for every vertex v of G, the minimum number of edges of any path between ***s*** and v.

BFS Properties

- **Proposition**: Let G be an undirected graph on which a BFS traversal starting at a vertex *s* has been preformed. Then:
  1. The traversal visits all vertices in the connected component of *s*
  2. The discovery edges form a spanning tree of the connected component of *s*

- **Justification of 1**:
  - Let's use a contradiction argument: suppose there is at least on vertex v not visited and let w be the first unvisited vertex on some path from *s* to v.
  - Because w was the first unvisited vertex on the path, there is a neighbor u that has been visited.
  - But when we visited u we must have looked at edge(u, w). Therefore w must have been visited.

- **Justification of 2**:
  - We only mark edges from when we go to unvisited vertices. So we never form a cycle of discovery edges, i.e. discovery edges form a tree.
  - This is a spanning tree because BFS visits each vertex in the connected component of *s*

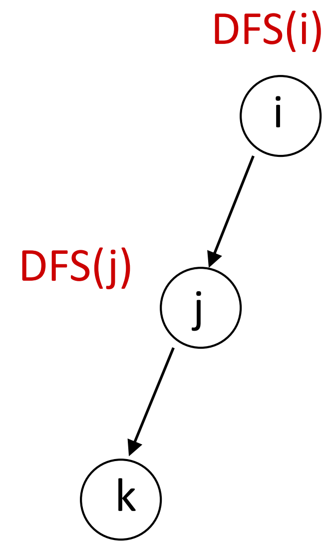
Graph Searching Methodology Depth-First Search (DFS)

- Depth-First Search (DFS)
  - Searches down one path as deep as possible
  - When no nodes available, it backtracks
  - When backtracking, it explores side-paths that were not taken
  - Uses a stack (instead of a queue in BFS)
  - Allows an easy recursive implementation

Depth First Search Algorithm

- Recursive marking algorithm
- Initially every vertex is unmarked

```
DFS(i: vertex)
  mark i;
  for each j adjacent to i do
    if j is unmarked then DFS(j)
end{DFS}
```



Marks all vertices reachable from i

Depth-First Search

Algorithm DFS(v); **Input**: A vertex v in a graph

Output: A labeling of the edges as "discovery" edges and "backedges"

for each edge *e* incident on *v* **do**

if edge *e* is unexplored **then** let *w* be the other endpoint of *e*

if vertex w is unexplored **then** label *e* as a discovery edge

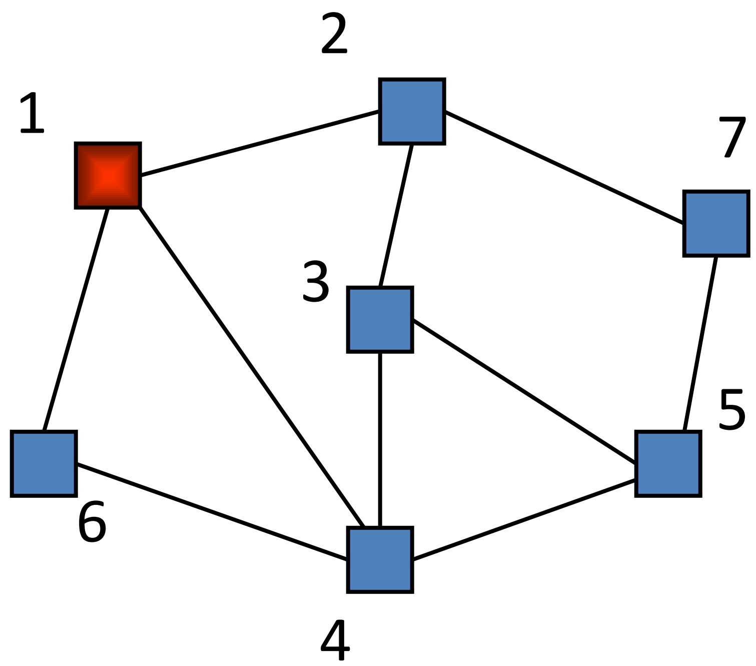
recursively call **DFS(*w*)**

else label *e* as a backedge

DFS Application: Spanning Tree

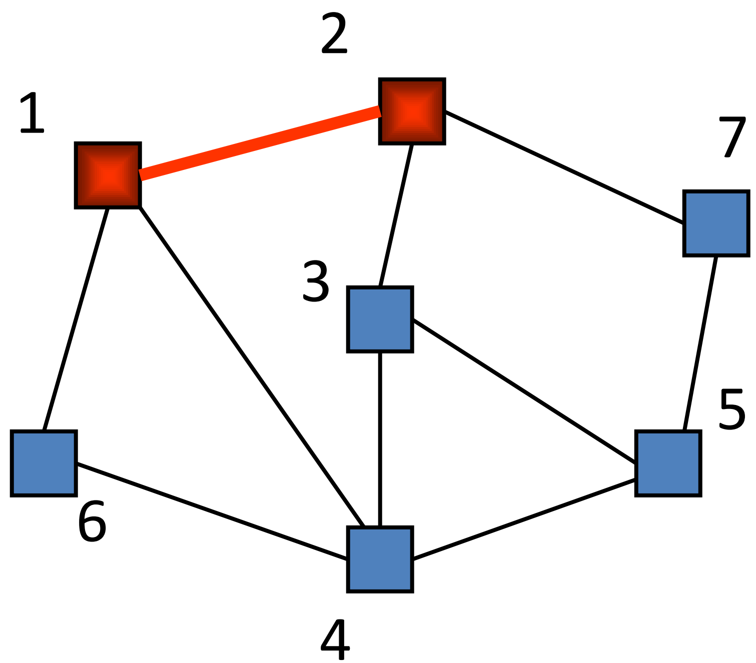
- Given a (undirected) connected graph G(V,E) a spanning tree of G is a graph G'(V',E')
  - V' = V, the tree touches all vertices (spans) the graph
  - E' is a subset of E such that G' is connected and there is no cycle in G'

Example of DFS: Graph connectivity and spanning tree



DFS(1)

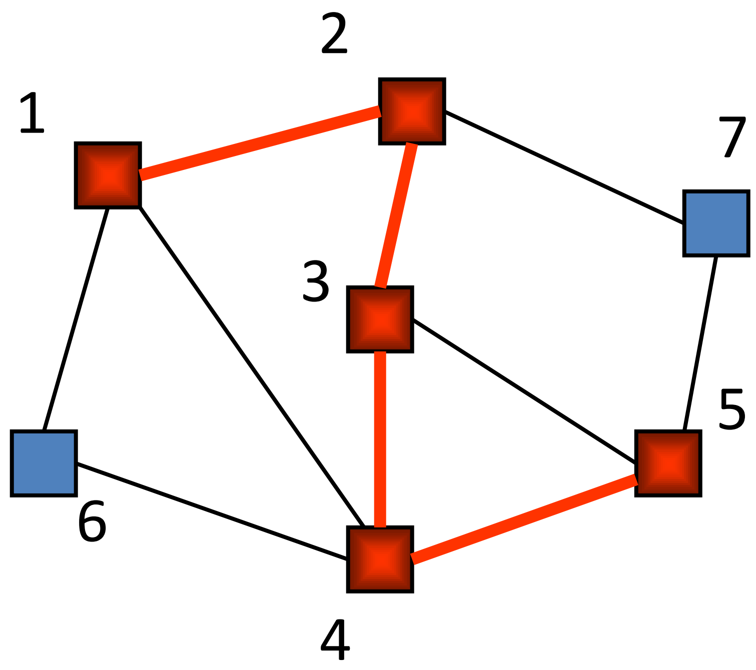
Example Step 2



DFS(1)
DFS(2)

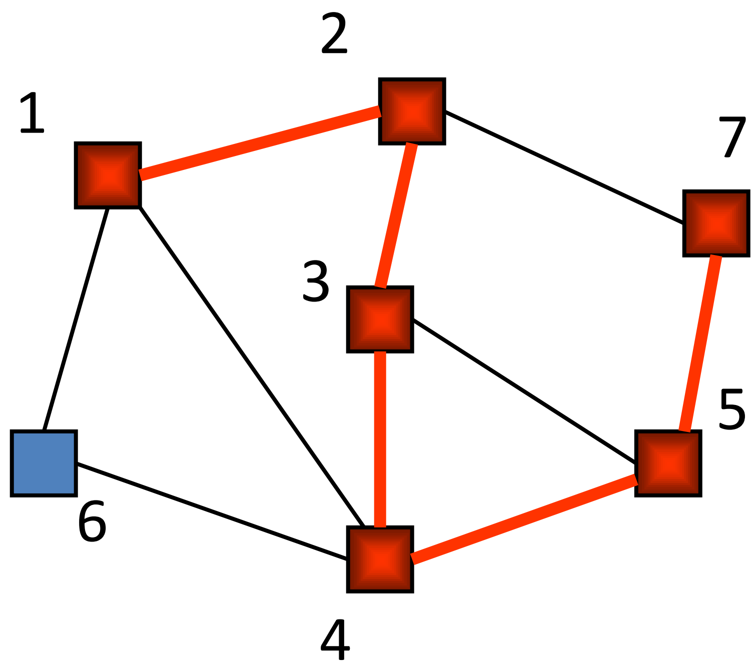
Red links will define the spanning tree if the graph is connected

Example Step 5



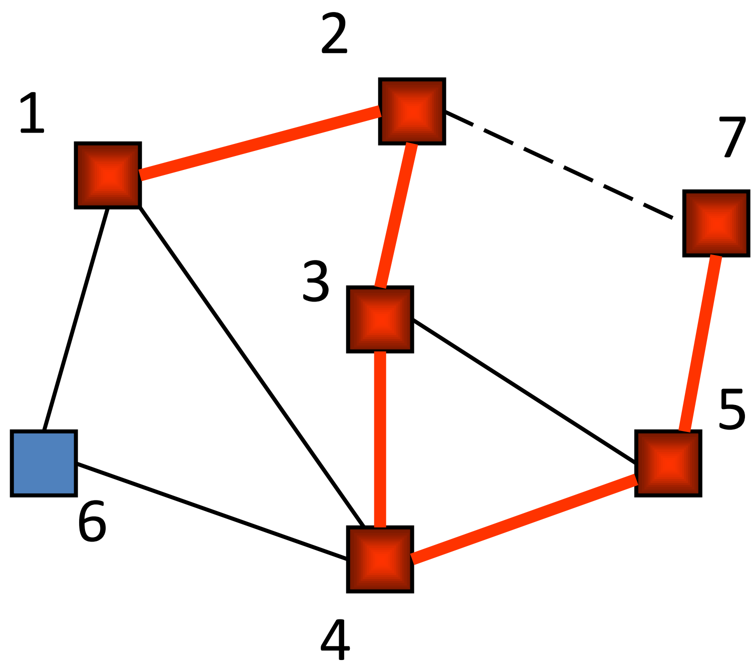
DFS(1)
DFS(2)
DFS(3)
DFS(4)
DFS(5)

Example Steps 6 and 7



DFS(1)
DFS(2)
DFS(3)
DFS(4)
DFS(5)
~~DFS(3)~~
DFS(7)

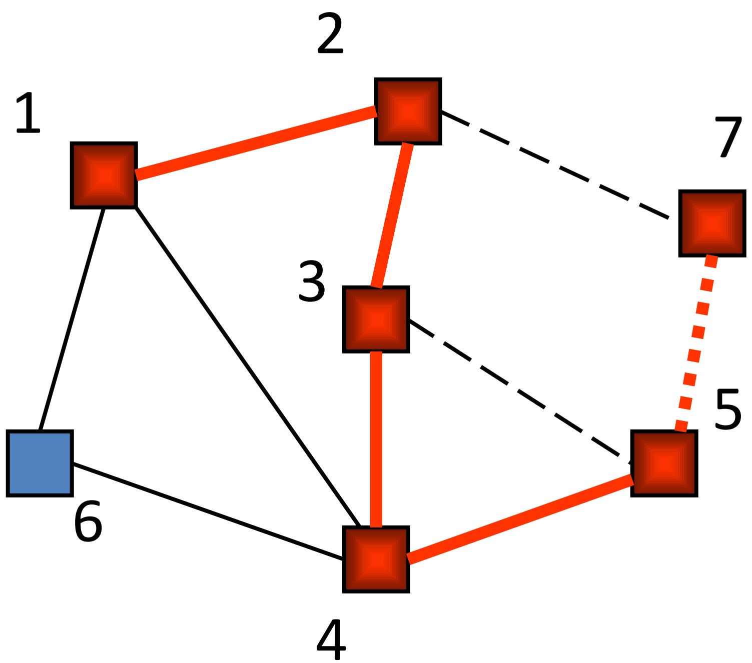
Example Steps 8 and 9



DFS(1)
 DFS(2)
  DFS(3)
   DFS(4)
    DFS(5)
     DFS(7)

Now back up.

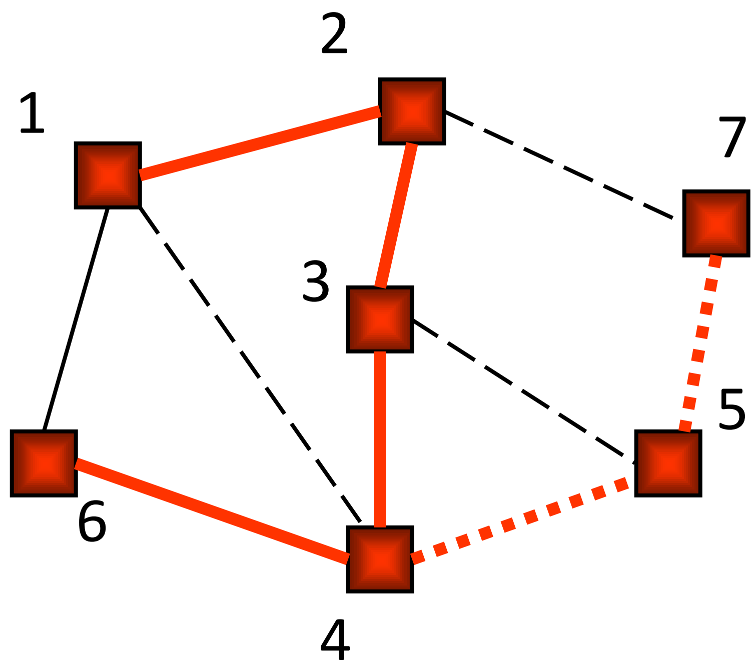
Example Step 10 (backtrack)



DFS(1)
 DFS(2)
  DFS(3)
   DFS(4)
    DFS(5)

Back to 5,
but it has no
more neighbors.

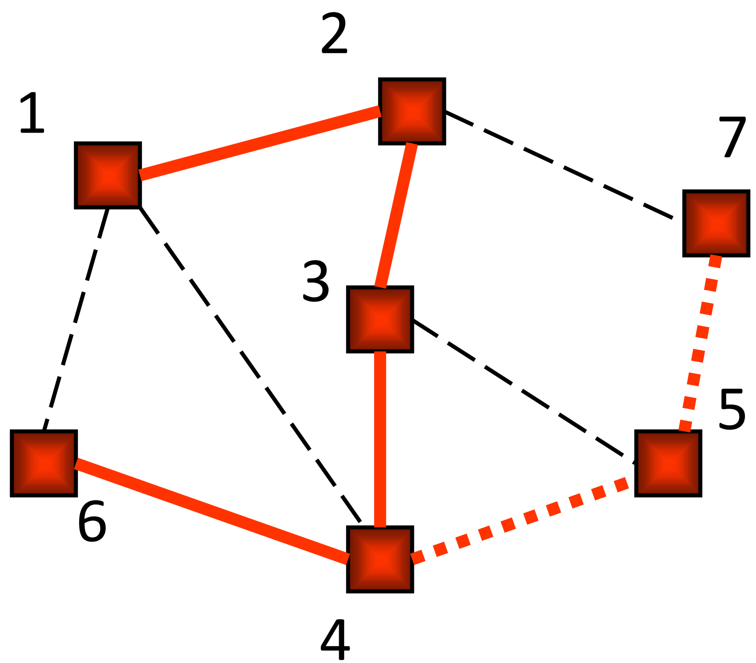
Example Step 12



DFS(1)
DFS(2)
DFS(3)
DFS(4)
DFS(6)

Back up to 4.
From 4 we can
get to 6.

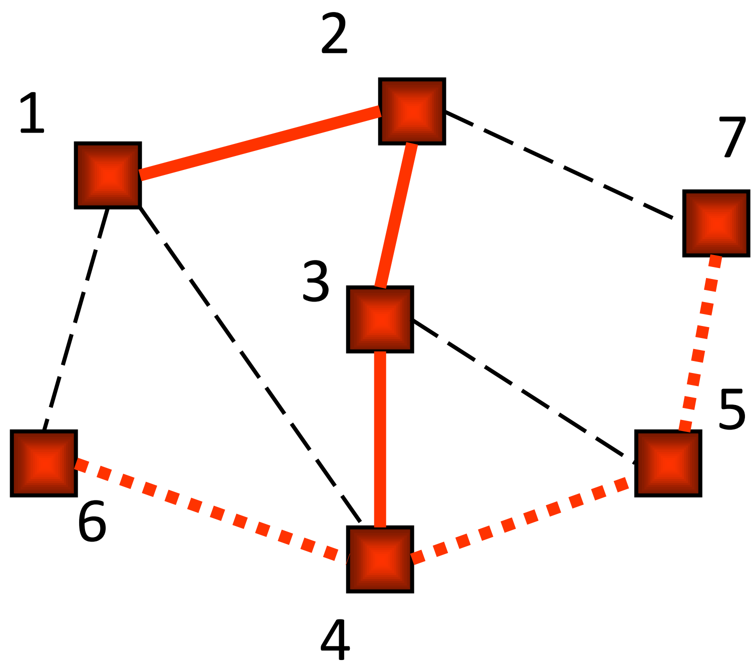
Example Step 13



DFS(1)
 DFS(2)
  DFS(3)
   DFS(4)
    DFS(6)

From 6 there is nowhere new to go. Back up.

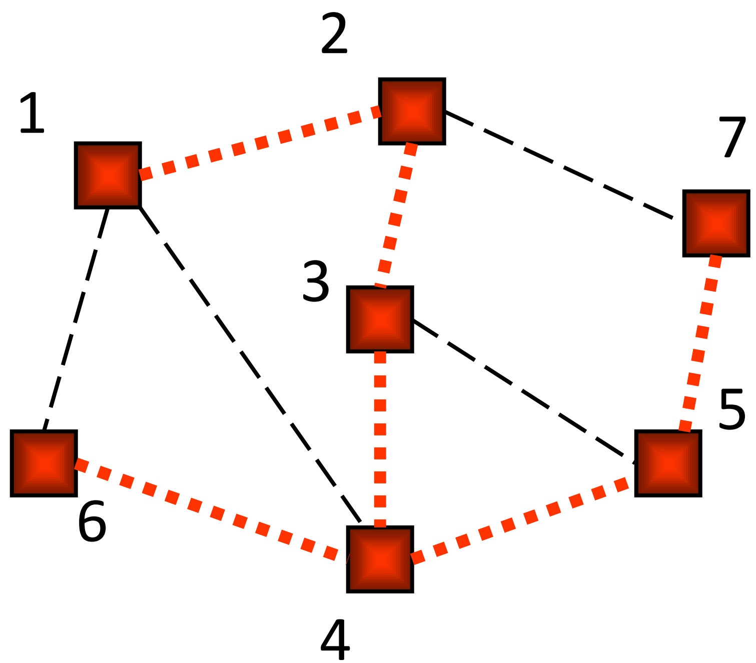
Example Step 14



DFS(1)
DFS(2)
DFS(3)
DFS(4)

Back to 4.
Keep backing up.

Example Step 17



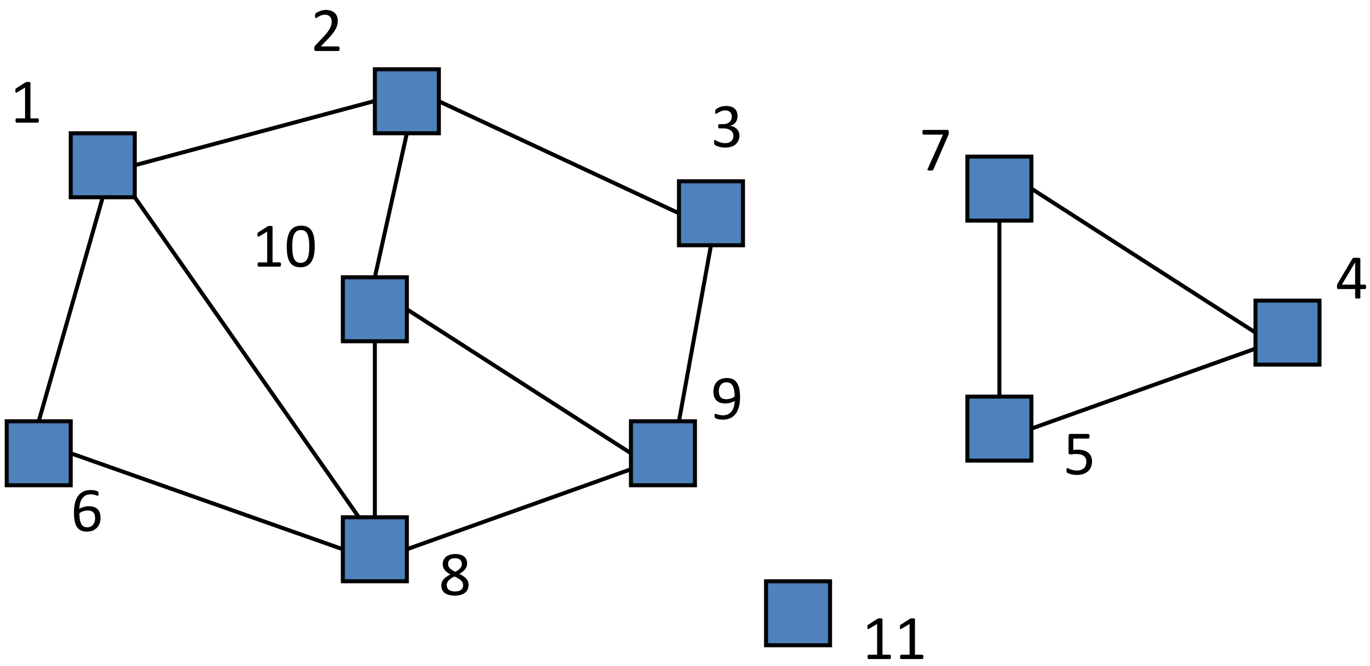
DFS(1)

All the way back to 1.

Done.

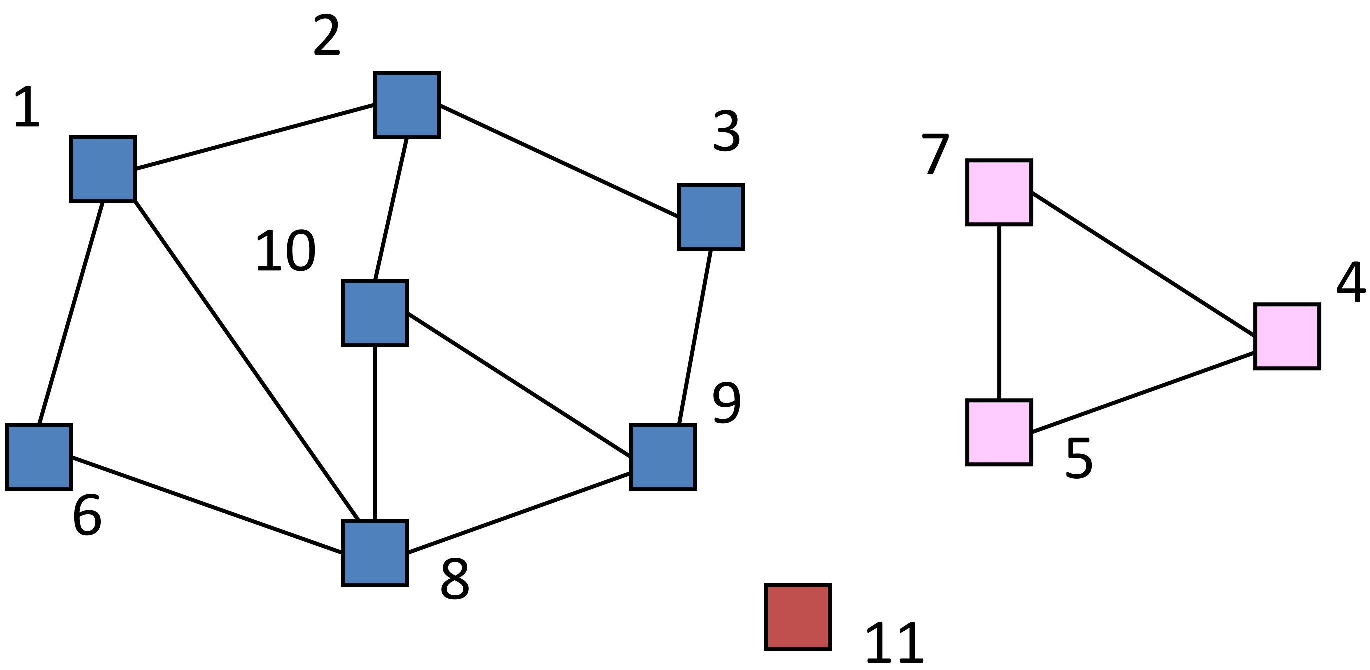
All nodes are marked so graph is connected; red links define a spanning tree

Finding Connected Components using DFS



3 connected components

Connected Components



3 connected components are labeled

Running Time Analysis

- Remember:

  -DFS is called on each vertex exactly once.

  -Every edge is examined exactly twice, once from each of its vertices
- For n_s vertices and m_s edges in the connected component of the vertex s, a DFS starting at s runs in $O(n_s + m_s)$ time if the graph is represented in a data structure, like the adjacency list, where vertex and edge methods take constant time.
- Marking a vertex as explored, and testing to see if a vertex has been explored, takes O(degree)
- By marking visited nodes, we can systematically consider the edges incident on the current vertex so we do not examine the same edge more than once.

Marking Vertices

- Let’s look at ways to mark vertices in a way that satisfies the above condition.
- Extend vertex positions to store a variable for marking

Before	After	
Position	Position	
Element	Element	isMarked

Use a hash table mechanism which satisfies the above condition is the probabilistic sense, because is supports the mark and test operations in O(1) expected time

Performance DFS

- n vertices and m edges
- Storage complexity $O(n + m)$
- Time complexity $O(n + m)$
- Linear Time!

DFS Properties

- **Proposition**: Let G be an undirected graph on which a DFS traversal starting at a vertex *s* has been preformed. Then:
  1. The traversal visits all vertices in the connected component of *s*
  2. The discovery edges form a spanning tree of the connected component of *s*

- **Justification of 1**:
  – Let's use a contradiction argument: suppose there is at least on vertex v not visited and let w be the first unvisited vertex on some path from *s* to v.
  – Because w was the first unvisited vertex on the path, there is a neighbor u that has been visited.
  – But when we visited u we must have looked at edge(u, w). Therefore w must have been visited.

- **Justification of 2**:
  – We only mark edges from when we go to unvisited vertices. So we never form a cycle of discovery edges, i.e. discovery edges form a tree.
  – This is a spanning tree because DFS visits each vertex in the connected component of *s*

Depth-First vs Breadth-First

- Depth-First
  - Stack or recursion
  - Many applications
- Breadth-First
  - Queue (recursion no help)
  - Can be used to find shortest paths from the start vertex