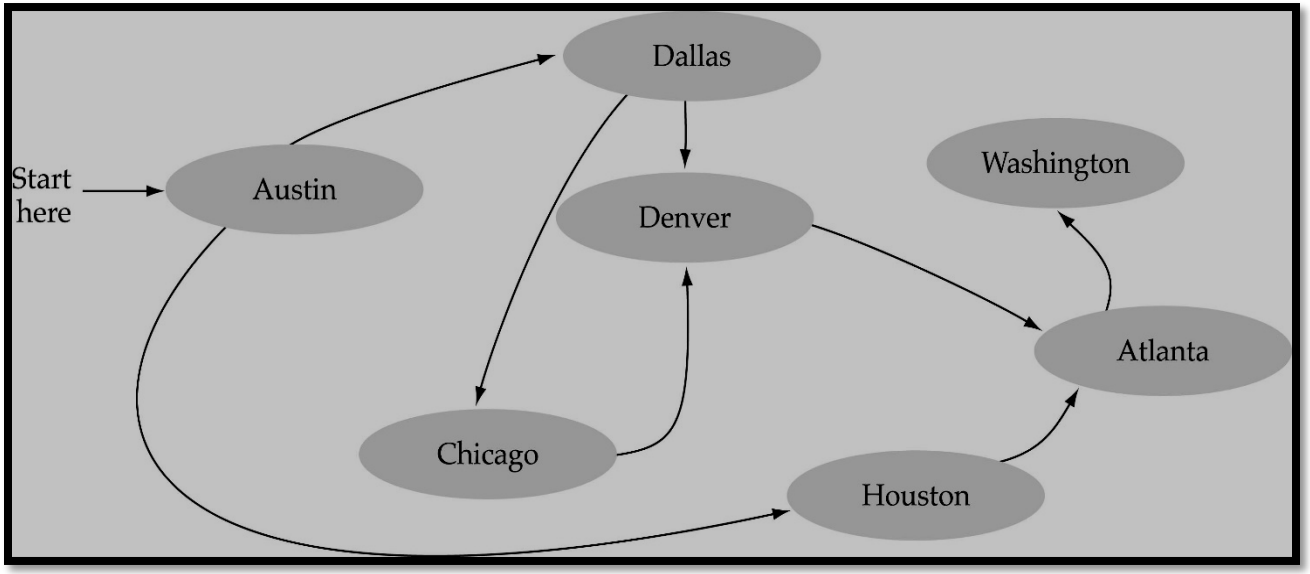
Data structures and Algorithms

Graph

What is a graph?

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices



Formal definition of graphs

- A graph *G* is defined as follows:

  $G=(V,E)$

  V(G): a finite, nonempty set of vertices

  E(G): a set of edges (pairs of vertices)

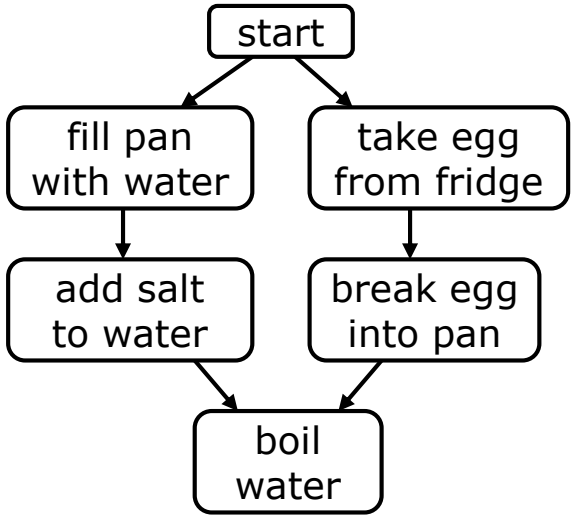
Graph terminology

- A graph **G = (V,E)** consists of an ordered pair consisting of a set of vertices V and a set of edges **E**, such that each edge in **E** is a connection between a pair of vertices in **V**.

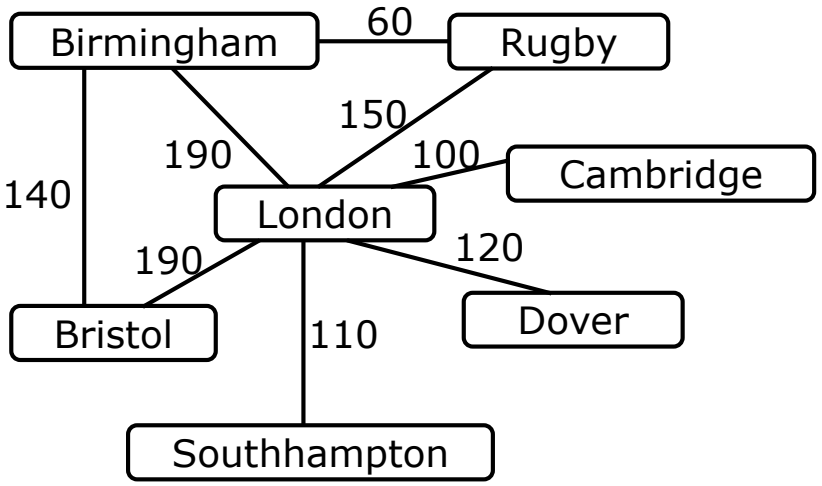
- The number of vertices is written **|V|**, and the number of edges is written **|E|**. **|E|** can range from zero to a maximum of **$|V|^2 - |V|$**.
- A graph with relatively few edges is called **sparse**, while a graph with many edges is called **dense**.
- A graph containing all possible edges is said to be **complete**.

Graph terminology (cont.)

- There are two kinds of graphs: directed graphs (sometimes called digraphs) and undirected graphs



A directed graph



An undirected graph

Graph terminology (cont.)

- A graph with edges directed from one vertex to another is called a **directed graph** or **digraph**.
- A graph whose edges are not directed is called an **undirected graph**.
- A graph with labels associated with its vertices is called a **labeled graph**.
- Two vertices are **adjacent** if they are joined by an edge. Such vertices are also called **neighbors**.

Graph terminology (cont.)

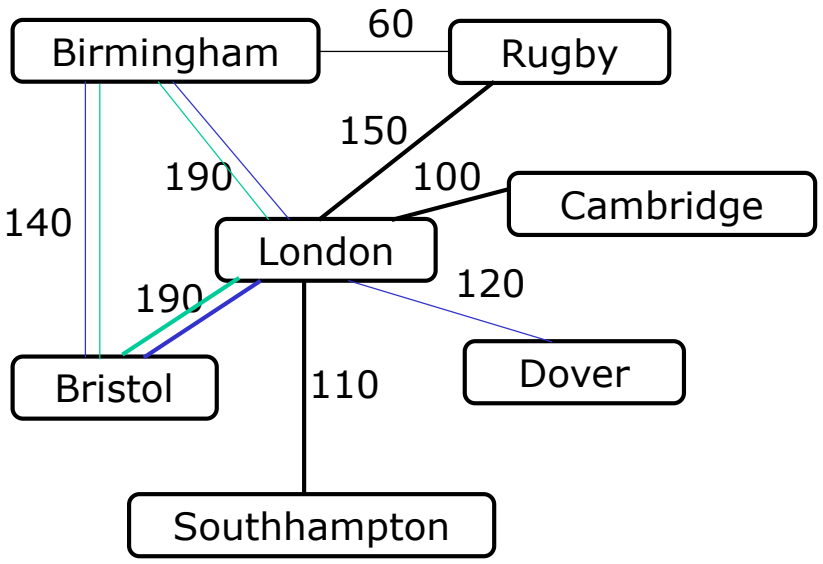
- The **size** of a graph is the number of *nodes or vertices* in it
- The **length** of a path is the number of edges it contains.
- The **empty graph** has size zero (no nodes)
- An edge connecting Vertices U and V is written (U, V). Such an edge is said to be **incident** on Vertices U and V.
- Associated with each edge may be a cost or **weight**. Graphs whose edges have weights are said to be **weighted**.

Graph terminology (cont.)

- The degree of a node is the number of edges it has
- For directed graphs,
  - If a directed edge goes from node S to node D, we call S the source and D the destination of the edge
    - The edge is an out-edge of S and an in-edge of D
    - S is a predecessor of D, and D is a successor of S
  - The in-degree of a node is the number of in-edges it has
  - The out-degree of a node is the number of out-edges it has

Graph terminology (cont.)

- A path is a list of edges such that each node (but the last) is the predecessor of the next node in the list
- A cycle is a path whose first and last nodes are the same



  - Example: (London, Bristol, Birmingham, London, Dover) is a path
  - Example: (London, Bristol, Birmingham, London) is a cycle
  - A cyclic graph contains at least one cycle
  - An acyclic graph does not contain any cycles

Graph terminology (cont.)

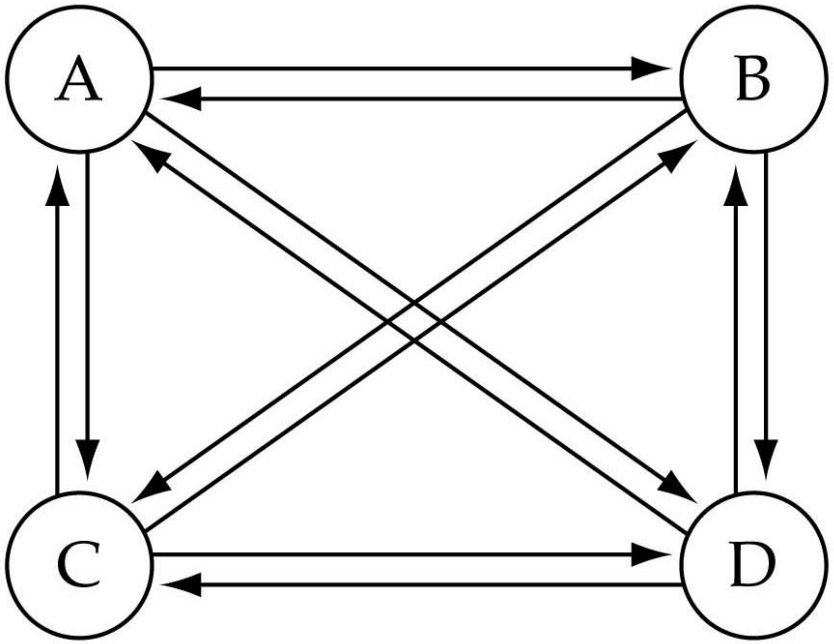
- An undirected graph is connected if there is a path from every node to every other node
- A *directed graph* is strongly connected if there is a path from every node to every other node
- A directed graph is weakly connected if the underlying undirected graph is connected
- Node X is reachable from node Y if there is a path from Y to X
- A subset of the nodes of the graph is a connected component (or just a component) if there is a path from every node in the subset to every other node in the subset

Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

*N * (N-1)*

$O(N^2)$



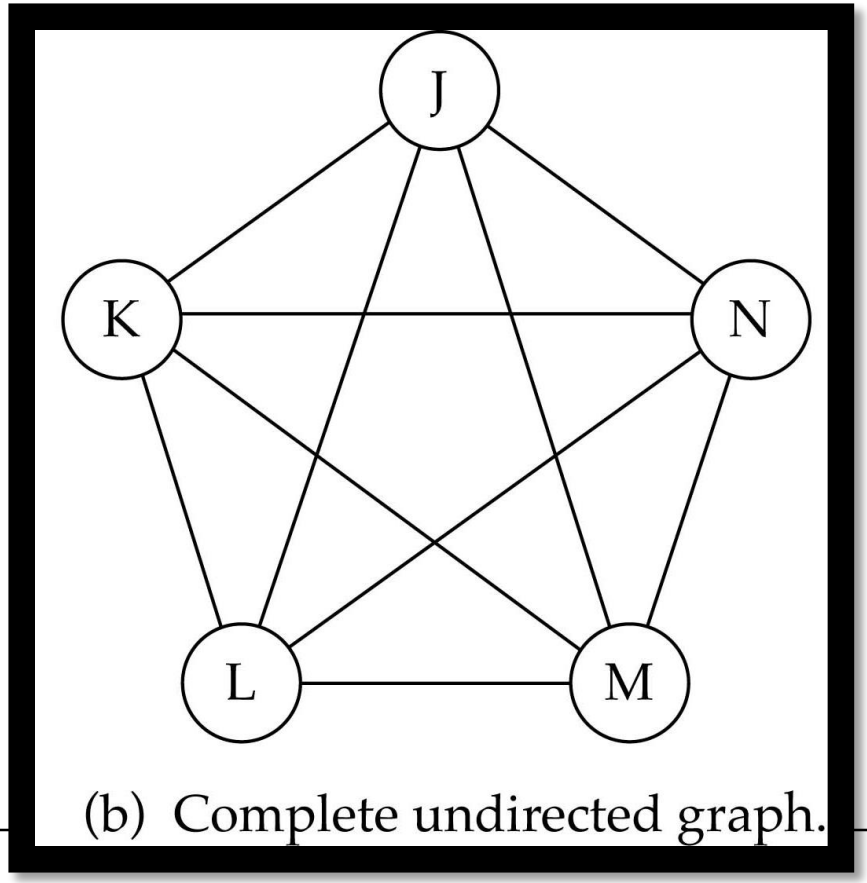
(a) Complete directed graph.

Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

*N * (N-1) / 2*

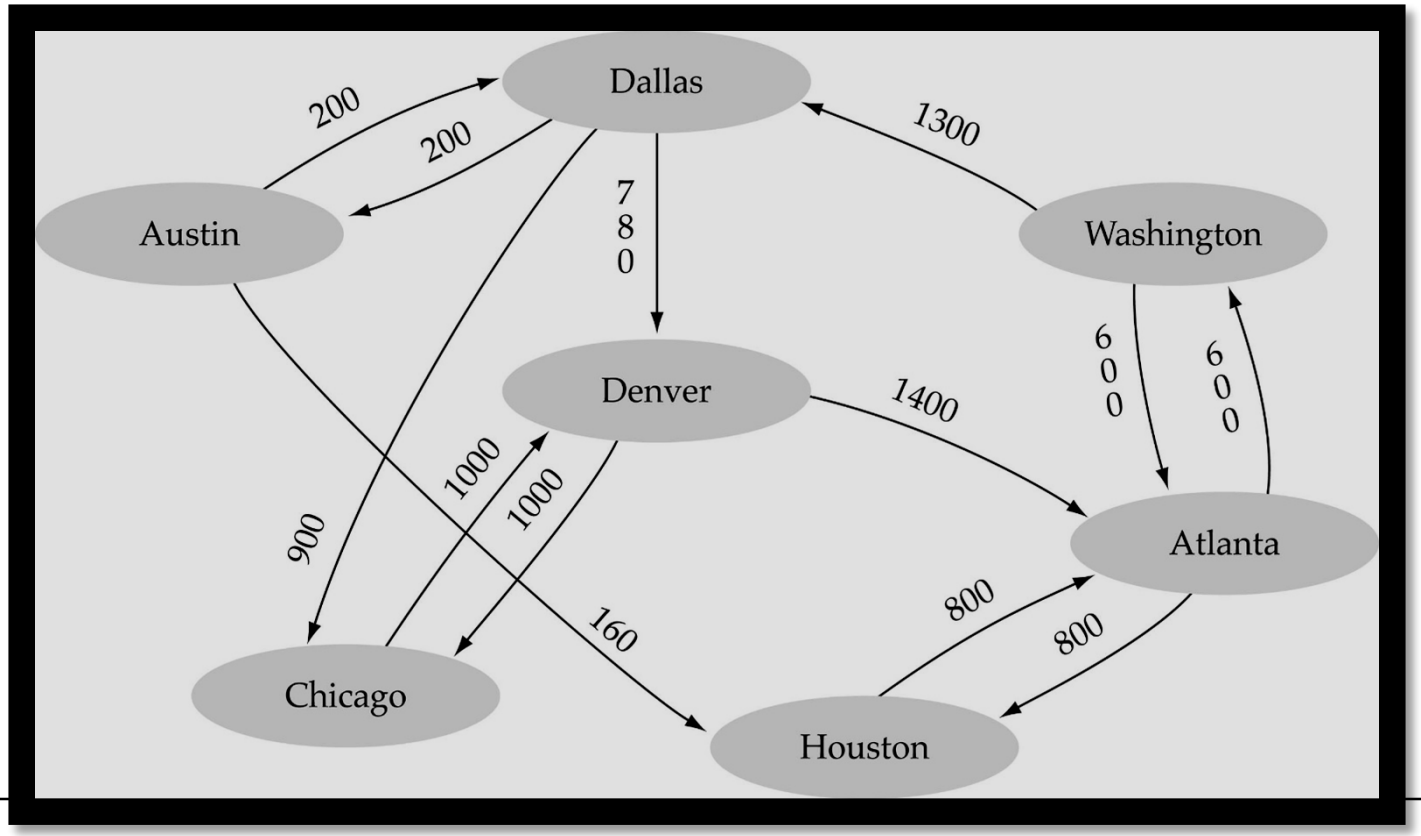
$O(N^2)$



(b) Complete undirected graph.

Graph terminology (cont.)

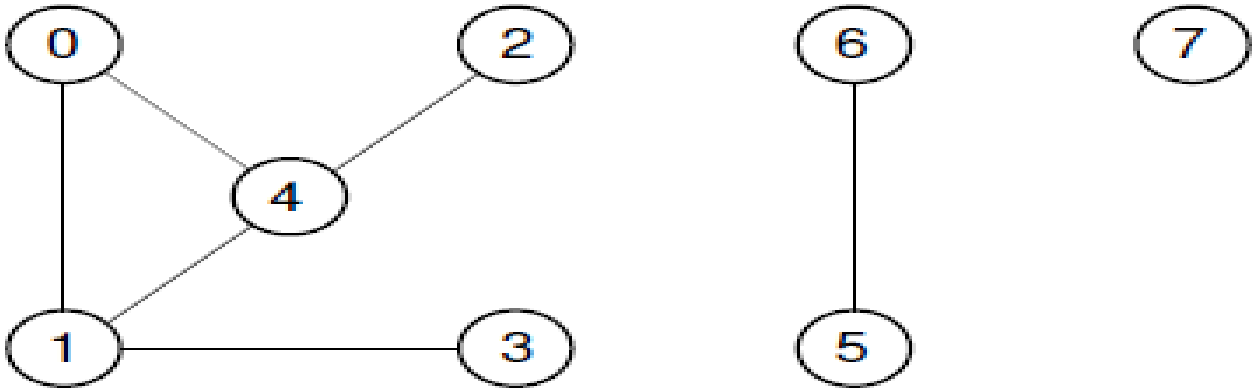
- Weighted graph: a graph in which each edge carries a value



Graph terminology (cont.)

- An undirected graph with three connected components. Vertices 0, 1, 2, 3, and 4 form one connected component. Vertices 5 and 6 form a second connected component. Vertex 7 by itself forms a third connected component.

Figure 1



Graph terminology (cont.)

- A **subgraph S** is formed from graph **G** by selecting a subset V_s of **G's** vertices and a subset E_s of G's edges such that for every edge **E** in E_s, both of its vertices are in V_s.

Graph terminology (cont.)

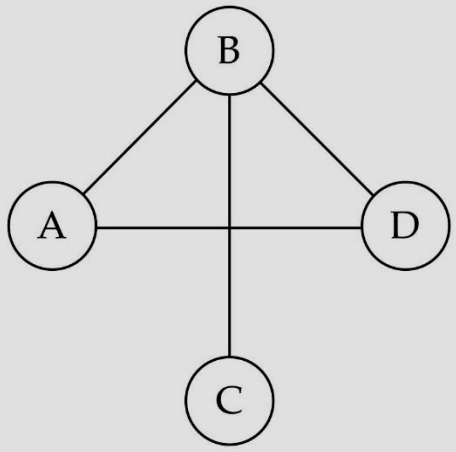
- A graph without cycles is called **acyclic**. Thus, a directed graph without cycles is called a **directed acyclic graph** or DAG.

- A **free tree** is a connected, undirected graph with no simple cycles. An equivalent definition is that a free tree is connected and has |**V**| – 1 edges.

Directed vs. undirected graphs

- When the edges in a graph have no direction, the graph is called *undirected*

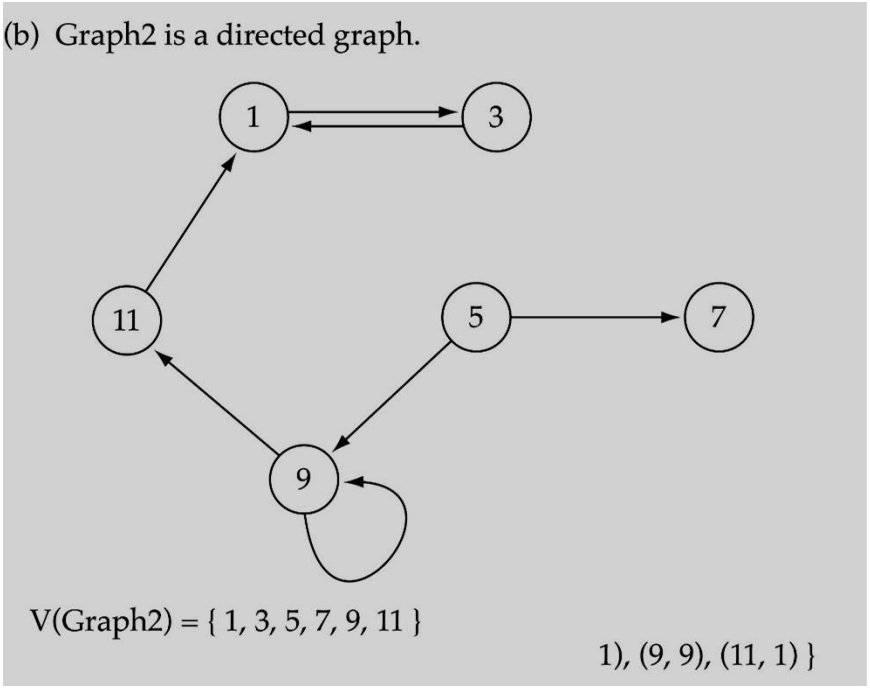
ected graph.



V(Graph1) = { A, B, C, D }
E(Graph1) = { (A, B), (A, D), (B, C), (B, D) }

Directed vs. undirected graphs (cont.)

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)



E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7)

Warning: if the graph is directed, the order of the vertices in each edge is important !!

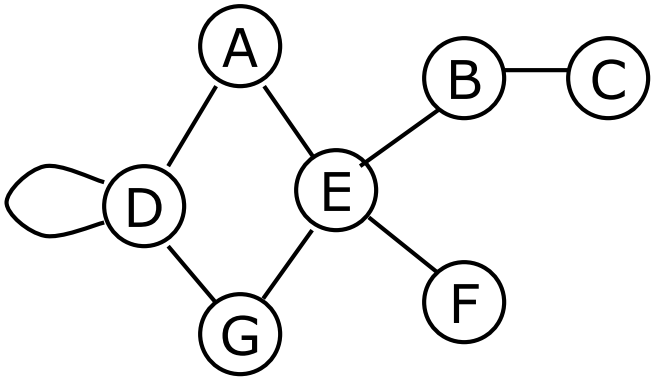
Graph applications

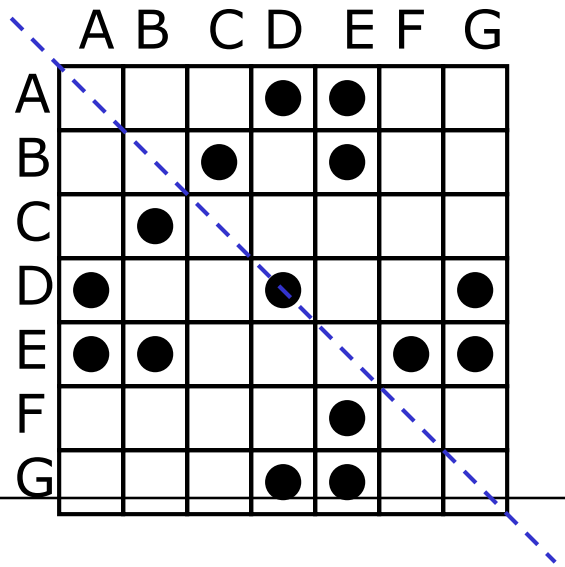
- Graphs can be used for:
  - Finding a route to drive from one city to another
  - Finding connecting flights from one city to another
  - Determining least-cost highway connections
  - Designing optimal connections on a computer chip
  - Implementing automata
  - Implementing compilers
  - Doing garbage collection
  - Representing family histories
  - Doing similarity testing (e.g. for a dating service)
  - Pert charts
  - Playing games

Graph implementation(Adjacent Matrix)

- Represent a graph with a two-dimensional array *G*
  - *G[i][j] = 1* if there is an edge from node *i* to node *j*
  - *G[i][j] = 0* if there is no edge from node *i* to node *j*
- *If the* graph is undirected, matrix is symmetric
- *Can* represent edges labeled with a cost as well:
  - *G[i][j] =* cost of link between *i* and *j*
  - if there is no direct link, *G[i][j] =* ∞

Graph implementation

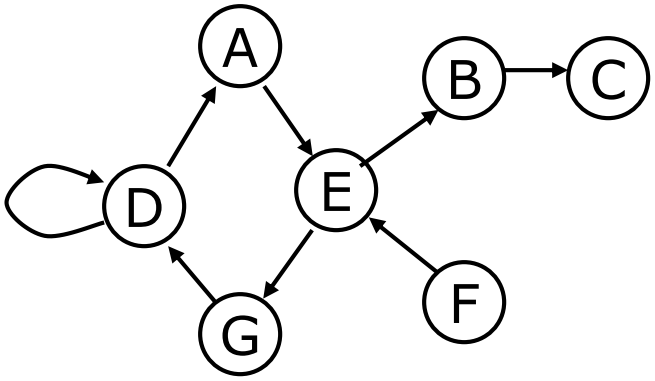




- One simple way of representing a graph is the adjacency matrix
- A 2-D array has a mark at [i][j] if there is an edge from node i to node j
- The adjacency matrix is symmetric about the main diagonal

- This representation is only suitable for *small* graphs! (Why?)

Graph implementation (cont.)

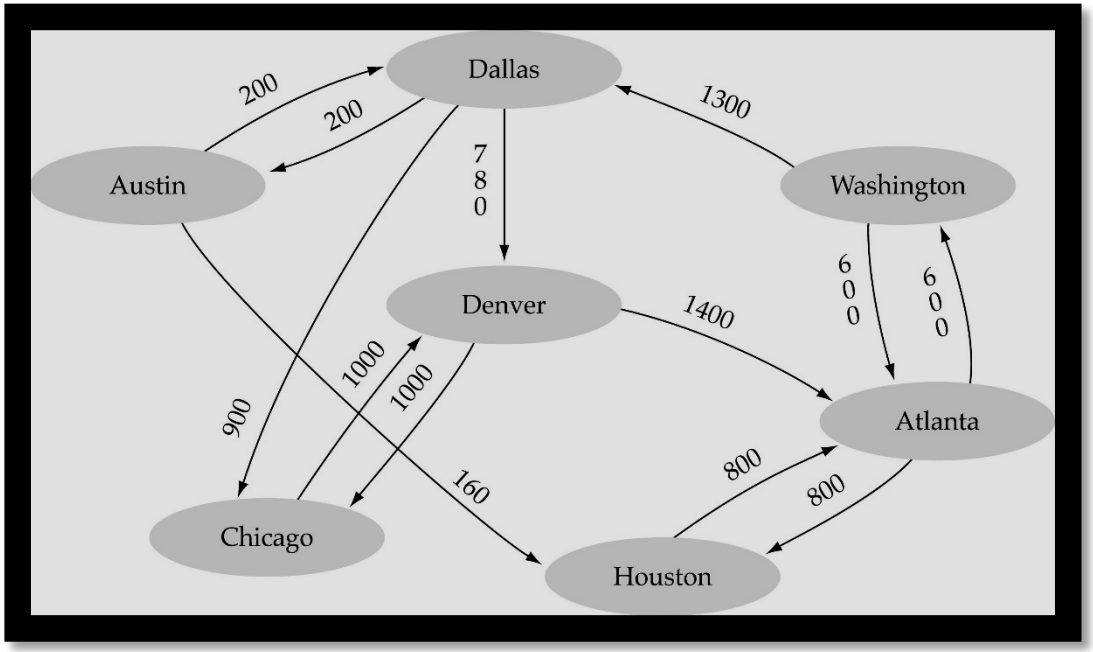


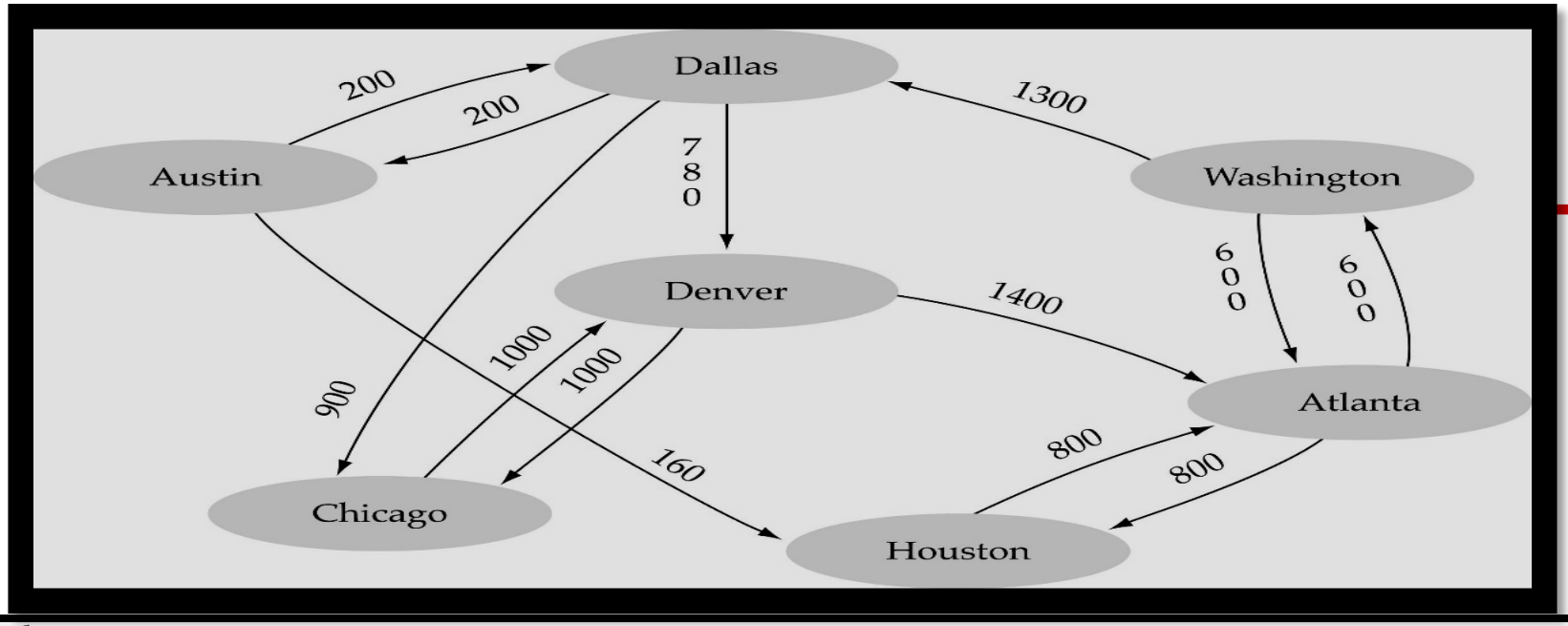
	A	B	C	D	E	F	G
A					●		
B			●				
C							
D	●			●			
E		●					●
F					●		
G				●			

- An adjacency matrix can equally well be used for digraphs (directed graphs)
- A 2-D array has a mark at [i][j] if there is an edge from node i to node j
- Again, this is only suitable for *small* graphs!

Graph implementation (cont.)

- Array-based implementation
  - A 1D array is used to represent the vertices
  - A 2D array (adjacency matrix) is used to represent the edges





graph

.numVertices 7

.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

(Array positions marked '•' are undefined)

Graph implementation (Adjacency List)

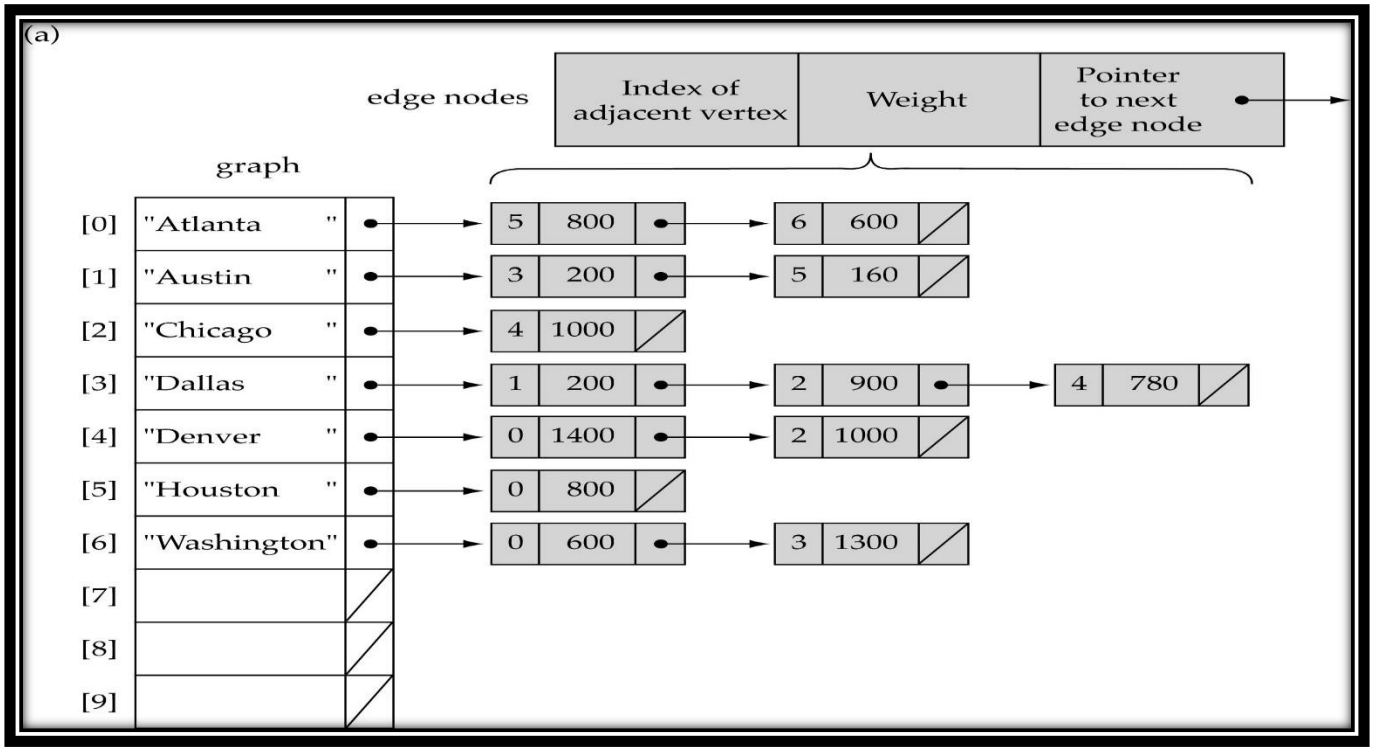
- Maintain a linked-list of the neighbors of every vertex.
  - n vertices
  - Array of n lists, one per vertex
  - Each list i contains a list of all vertices adjacent to i

Graph implementation (Adjacency List)

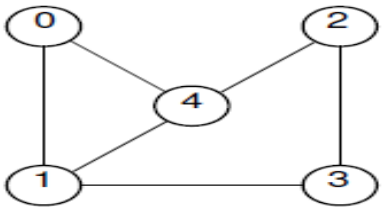
- The adjacency list is an array of linked lists. The array is |V| items long, with position *i* storing a pointer to the linked list of edges for Vertex *vi*.
- This linked list represents the edges by the vertices that are adjacent to Vertex *vi*.

Graph implementation (cont.)

- Linked-list implementation
  - A 1D array is used to represent the vertices
  - A list is used for each vertex *v* which contains the vertices which are adjacent from *v* (adjacency list)



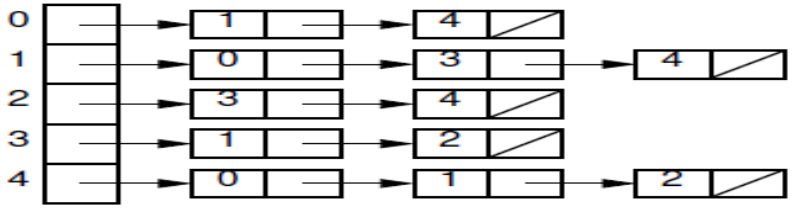
Graph implementation (cont.)



(a)

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

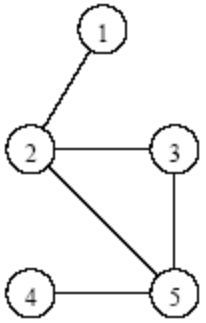
(b)



(c)

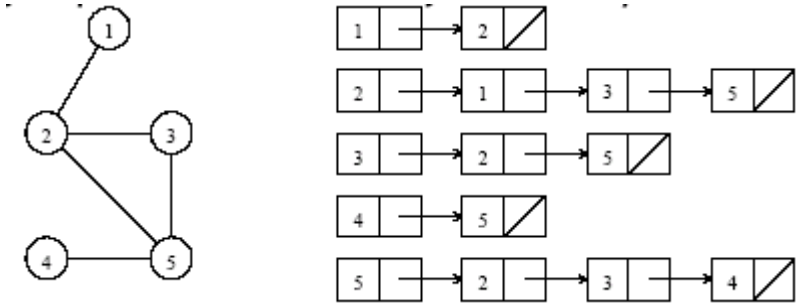
- Using the graph representations for undirected graphs. (a) An undirected graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

Graph implementation (cont.)



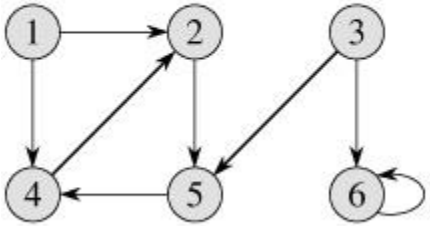
$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

An undirected graph and its adjacency matrix representation.



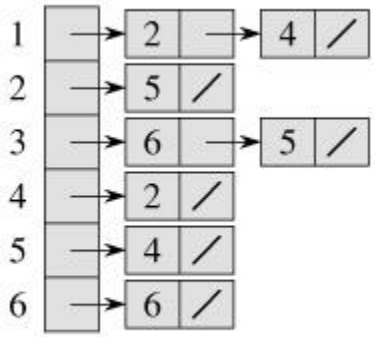
An undirected graph and its adjacency list representation.

Graph representation – directed



(a)

graph



(b)

Adjacency list

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

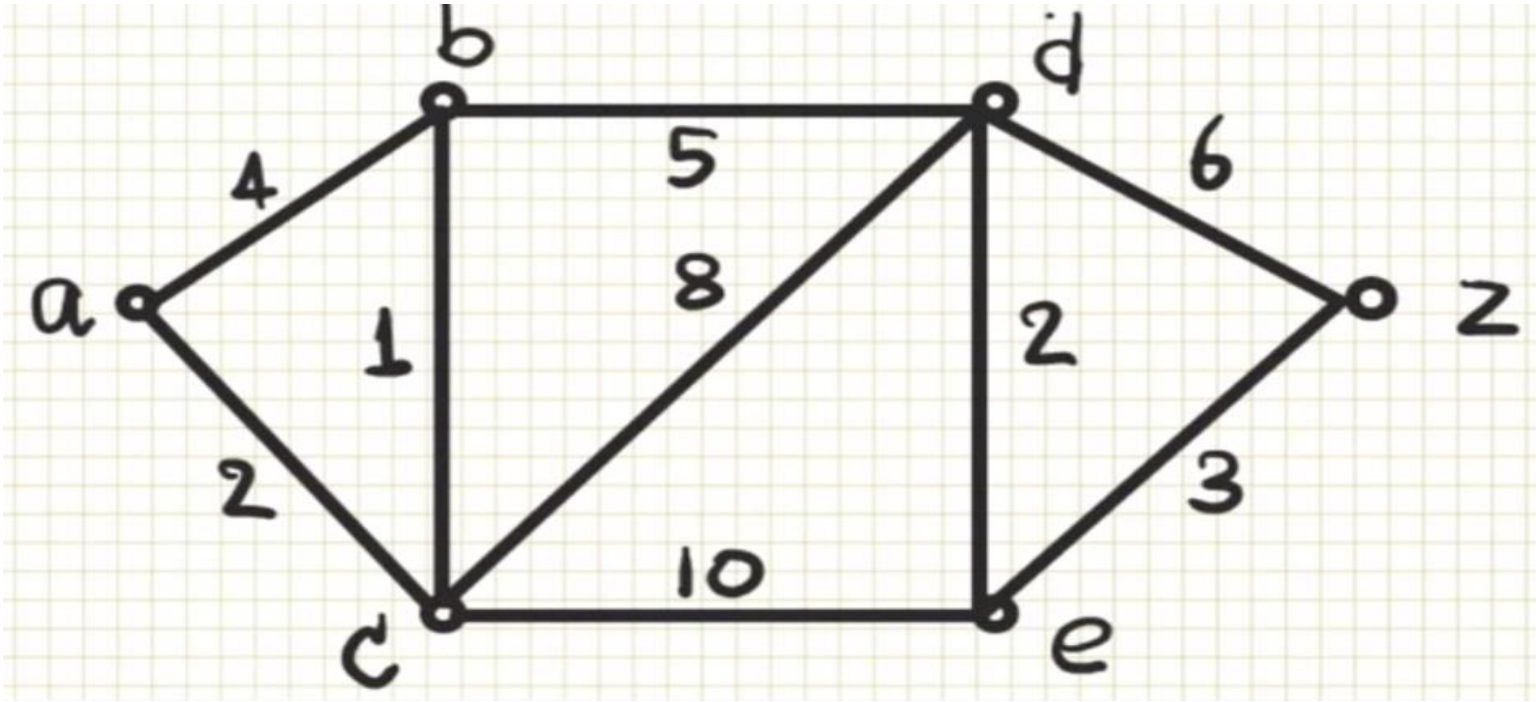
Adjacency matrix

Adjacency matrix vs. adjacency list representation

- Adjacency matrix
  - Good for dense graphs --$|E| \sim O(|V|^2)$
  - Memory requirements: $O(|V| + |E|) = O(|V|^2)$
  - Connectivity between two vertices can be tested quickly
- Adjacency list
  - Good for sparse graphs -- $|E| \sim O(|V|)$
  - Memory requirements: $O(|V| + |E|) = O(|V|)$
  - Vertices adjacent to another vertex can be found quickly

Question for Practice

- Implement the graph below using the edge list, adjacency matrix and adjacency list



Graph Traversal Techniques

- The previous connectivity problem, as well as many other graph problems, can be solved using graph traversal techniques
- There are two standard graph traversal techniques:
  - *Depth-First Search* (DFS)
  - *Breadth-First Search* (BFS)

Graph Traversal (Contd.)

- In both DFS and BFS, the nodes of the undirected graph are visited in a systematic manner so that every node is visited exactly once.
- Both BFS and DFS give rise to a tree:
  - When a node x is visited, it is labeled as visited, and it is added to the tree
  - If the traversal got to node x from node y, y is viewed as the parent of x, and x a child of y

Graph Search (traversal)

- How do we search a graph?
  - At a particular vertices, where shall we go next?
- Two common framework:
    - the depth-first search (DFS)
    - the breadth-first search (BFS) and
  - In DFS, go as far as possible along a single path until reach a dead end (a vertex with no edge out or no neighbor unexplored) then backtrack
  - In BFS, one explore a graph level by level away (explore all neighbors first and then move on)

Searching a graph

- With certain modifications, any tree search technique can be applied to a graph
  - This includes depth-first, breadth-first, depth-first iterative deepening, and other types of searches
- The difference is that a graph may have cycles
  - We don't want to search around and around in a cycle
- To avoid getting caught in a cycle, we must keep track of which nodes we have already explored
- There are two basic techniques for this:
  - Keep a set of already explored nodes, or
  - Mark the node itself as having been explored
    - Marking nodes is not always possible (may not be allowed)

Example: Depth-first search

- **Here is how to do DFS on a tree:**

```
Put the root node on a stack;
while (stack is not empty) {
    remove a node from the stack;
    if (node is a goal node) return success;
    put all children of the node onto the stack;
}
return failure;
```

- **Here is how to do DFS on a graph:**

```
Put the starting node on a stack;
while (stack is not empty) {
    remove a node from the stack;
    if (node has already been visited) continue;
    if (node is a goal node) return success;
    put all adjacent nodes of the node onto the stack;
}
return failure;
```

Finding connected components

- A depth-first search can be used to find connected components of a graph
  - A connected component is a set of nodes; therefore,
  - A set of connected components is a set of sets of nodes
- To find the connected components of a graph:

```
while there is a node not assigned to a component {
    put that node in a new component
    do a DFS from the node, and put every node
        reached into the same component
                                    }
```

Backtracking Algorithm
Depth-First Search

Text

Read Weiss, § 9.6 Depth-First Search and § 10.5 Backtracking Algorithms

Depth-First Search

- DFS follows the following rules:
  1. Select an unvisited node x, visit it, and treat as the current node
  2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
  3. If the current node has no unvisited neighbors, backtrack to the its parent, and make that parent the new current node;
  4. Repeat steps 2 and 3 until no more nodes can be visited.
  5. If there are still unvisited nodes, repeat from step 1.

Implementation of DFS

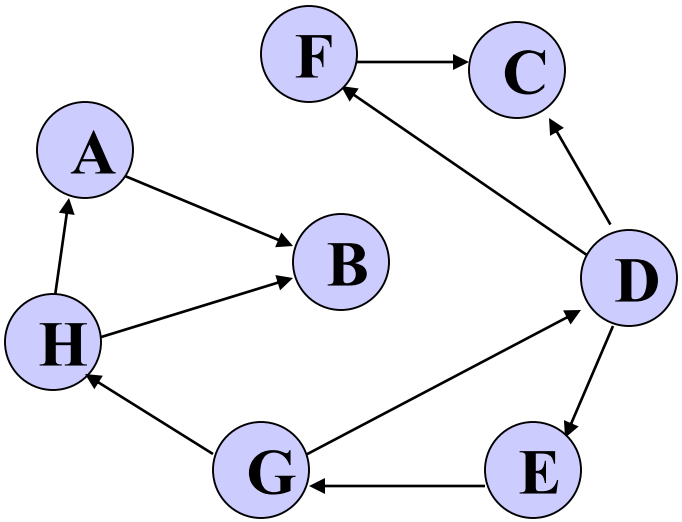
- Observations:
  - the last node visited is the first node from which to proceed.
  - Also, the backtracking proceeds on the basis of "last visited, first to backtrack too".
  - This suggests that a stack is the proper data structure to remember the current node and how to backtrack.

DFS (Pseudo Code)

depth-first-search

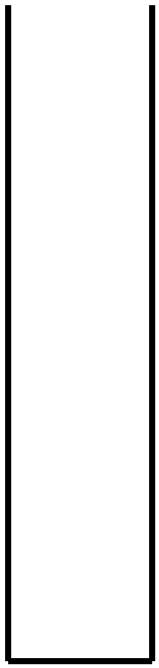
```
    mark vertex as visited
    for each adjacent vertex
        if unvisited
            do a depth-first search on adjacent
              vertex
```

Walk-Through



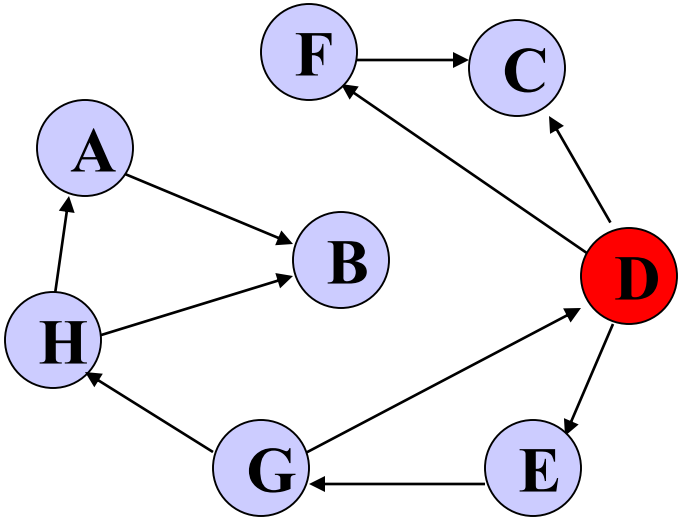
Visited Array

A	
B	
C	
D	
E	
F	
G	
H	



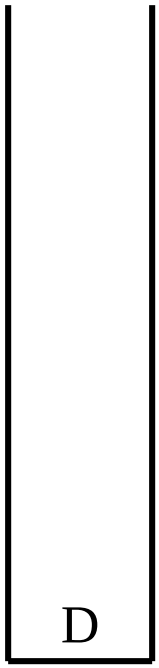
Task: Conduct a depth-first search of the graph starting with node D

Walk-Through



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

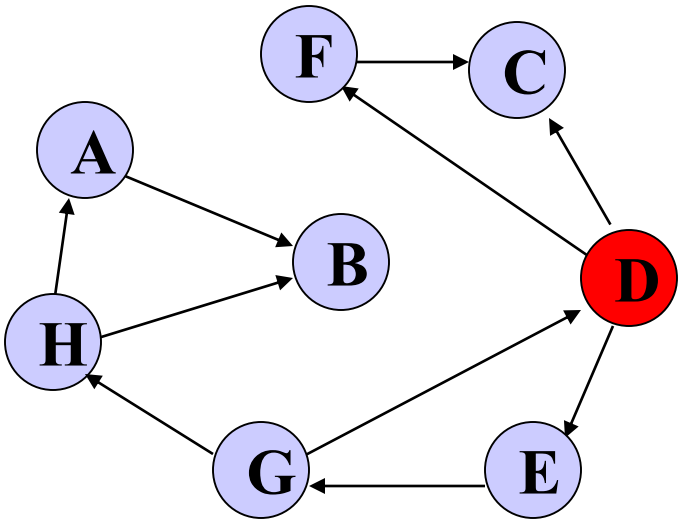


Visit D

The order nodes are visited:

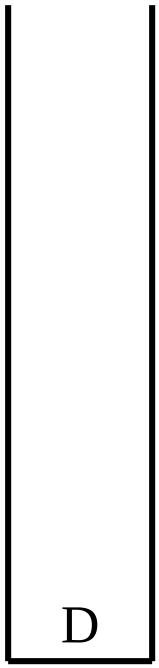
D

Walk-Through



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

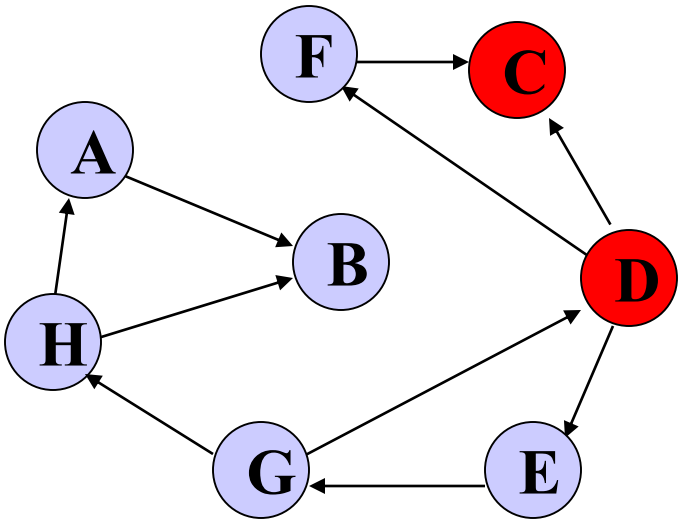


The order nodes are visited:

D

Consider nodes adjacent to D, decide to visit C first (Rule: visit adjacent nodes in alphabetical order)

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

C

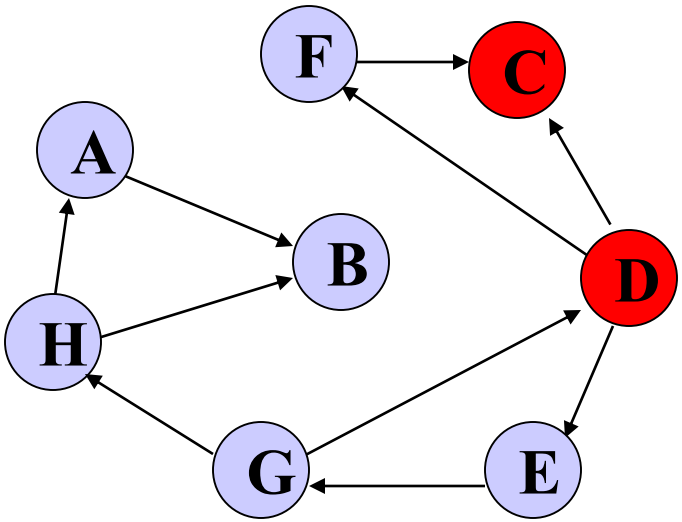
D

Visit C

The order nodes are visited:

D, C

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

C

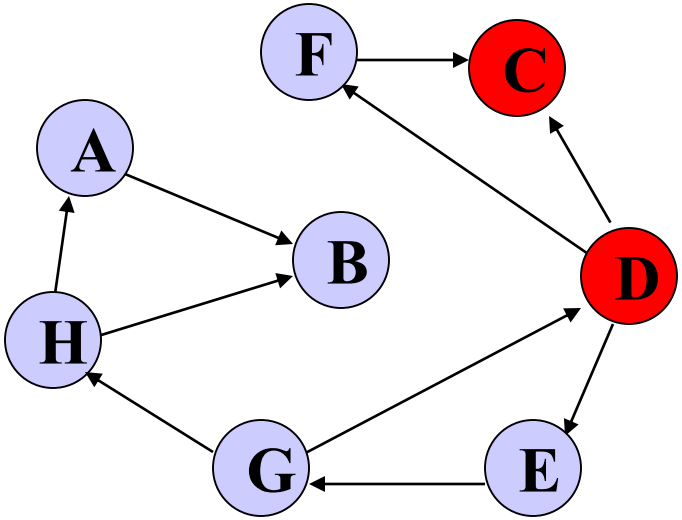
D

The order nodes are visited:

D, C

No nodes adjacent to C; cannot continue ➔ *backtrack*, i.e., pop stack and restore previous state

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

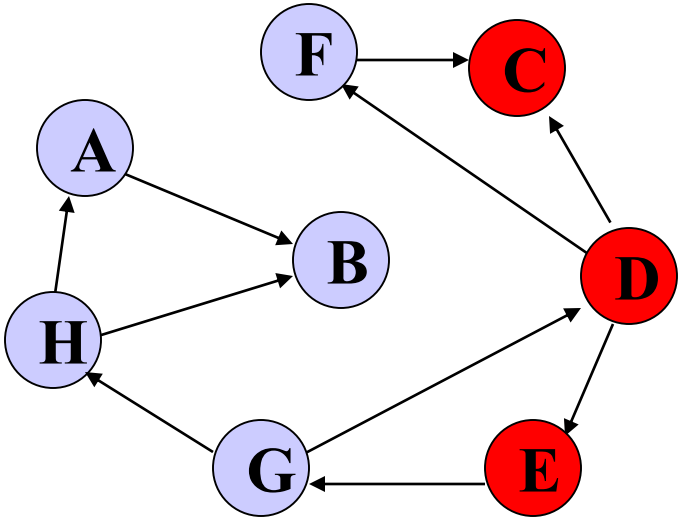
D

The order nodes are visited:

D, C

Back to D – C has been visited, decide to visit E next

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	

E

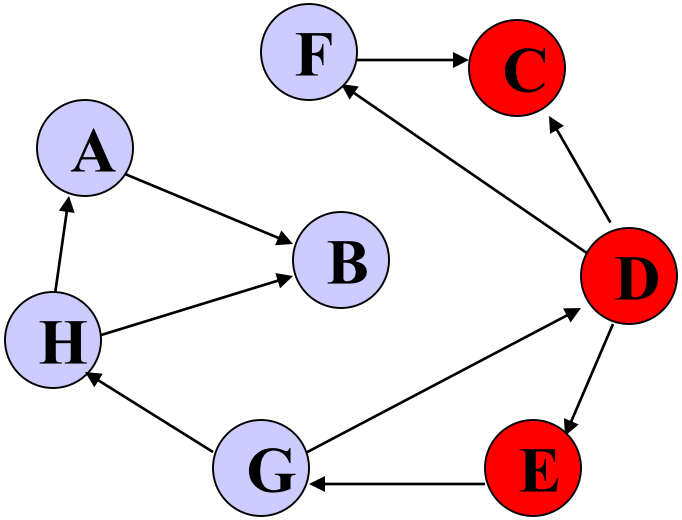
D

The order nodes are visited:

D, C, E

Back to D – C has been visited, decide to visit E next

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	

E

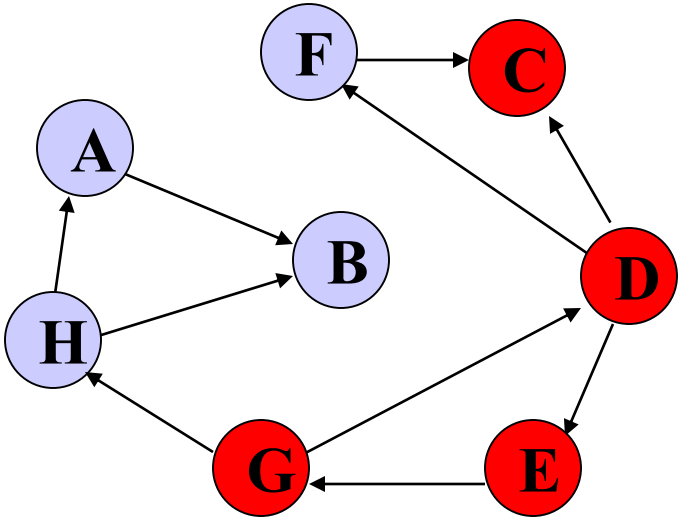
D

The order nodes are visited:

D, C, E

Only G is adjacent to E

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	

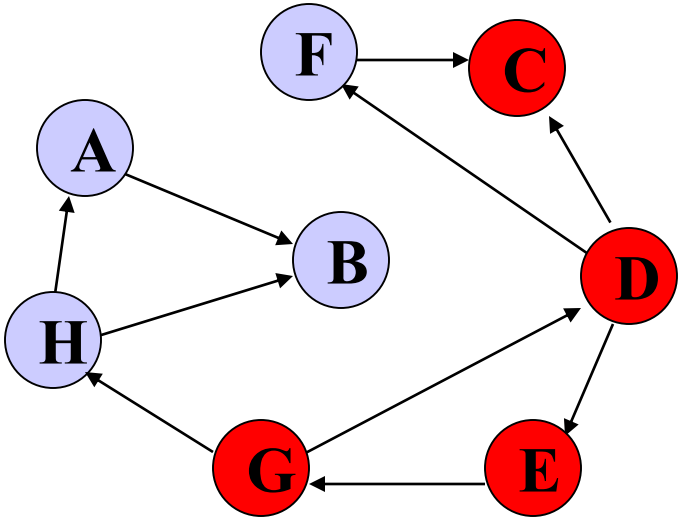
G
E
D

Visit G

The order nodes are visited:

D, C, E, G

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	

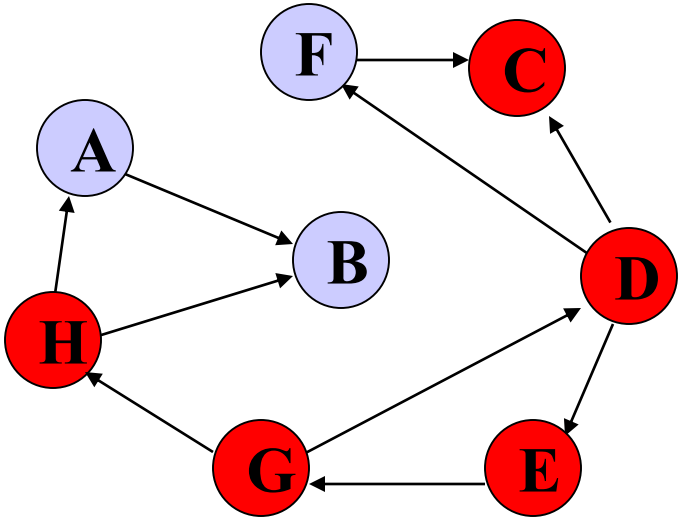
G
E
D

The order nodes are visited:

D, C, E, G

Nodes D and H are adjacent to G. D has already been visited. Decide to visit H.

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	√

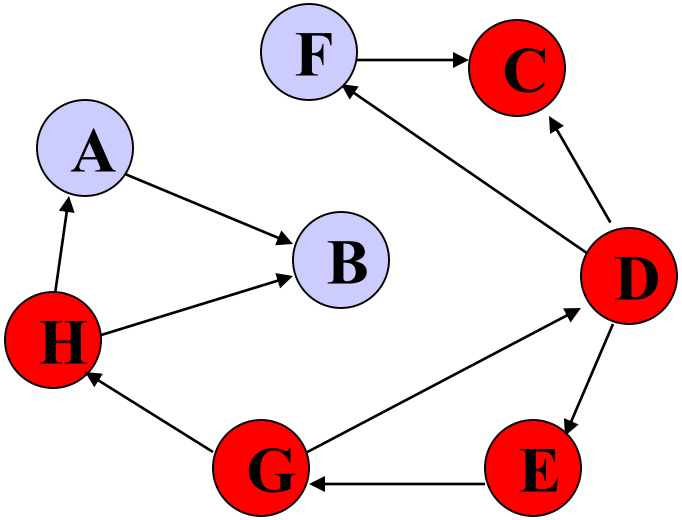
H
G
E
D

Visit H

The order nodes are visited:

D, C, E, G, H

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	√

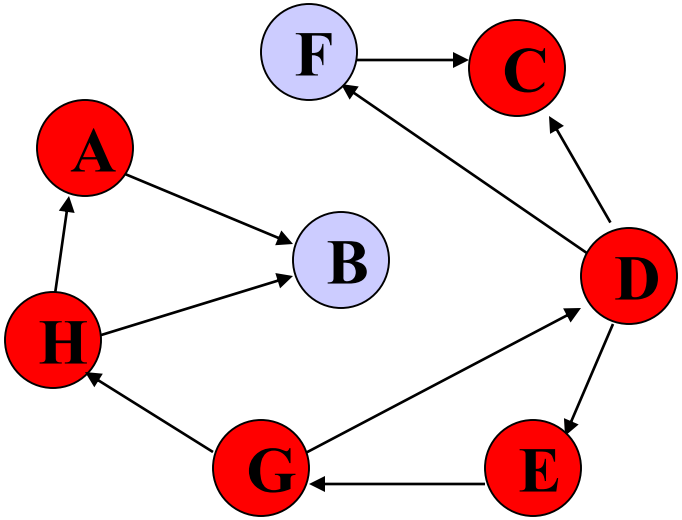
H
G
E
D

The order nodes are visited:

D, C, E, G, H

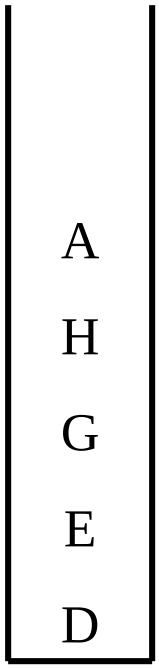
Nodes A and B are adjacent to F.
Decide to visit A next.

Walk-Through



Visited Array

A	√
B	
C	√
D	√
E	√
F	
G	√
H	√

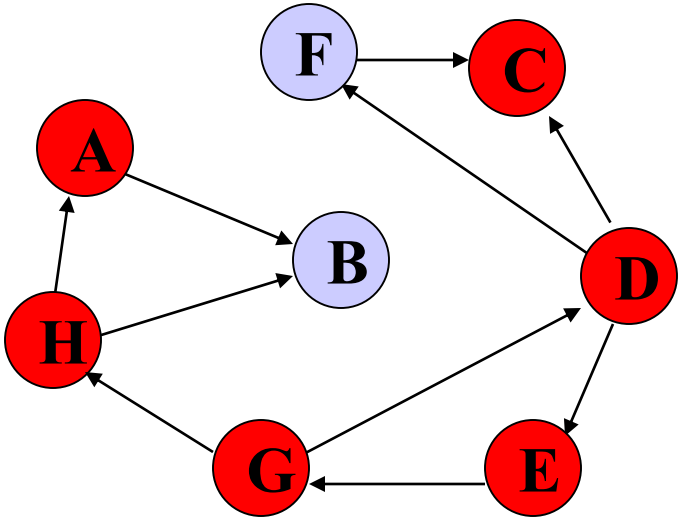


Visit A

The order nodes are visited:

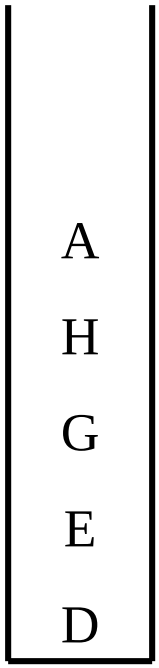
D, C, E, G, H, A

Walk-Through



Visited Array

A	√
B	
C	√
D	√
E	√
F	
G	√
H	√

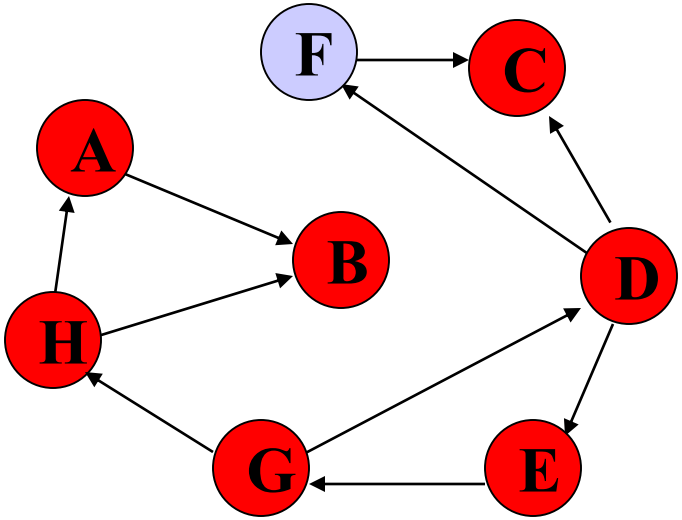


The order nodes are visited:

D, C, E, G, H, A

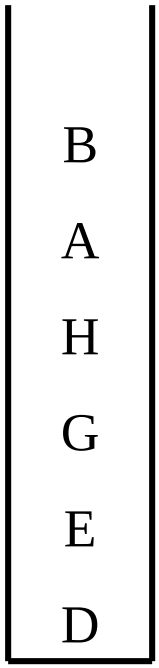
Only Node B is adjacent to A.
Decide to visit B next.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

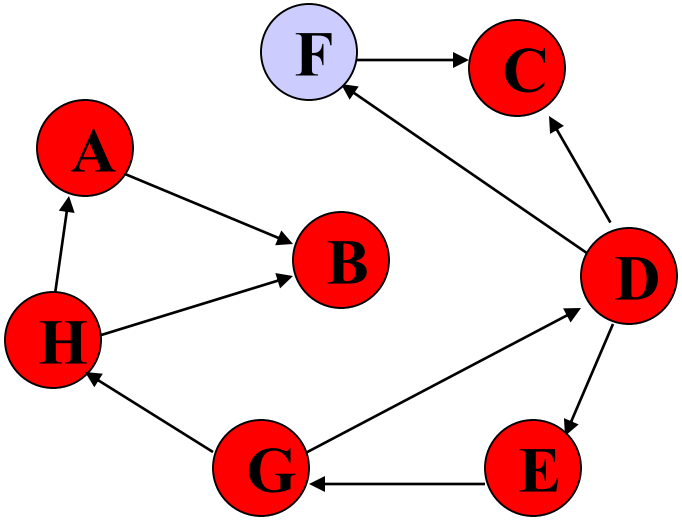


Visit B

The order nodes are visited:

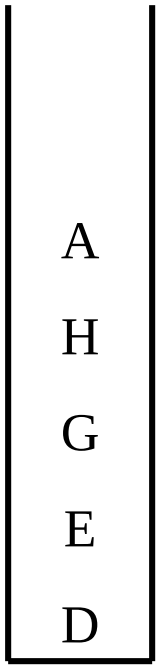
D, C, E, G, H, A, B

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

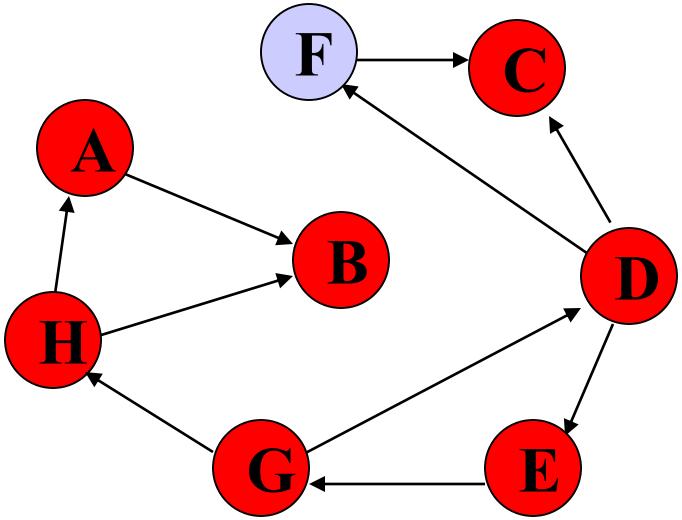


The order nodes are visited:

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

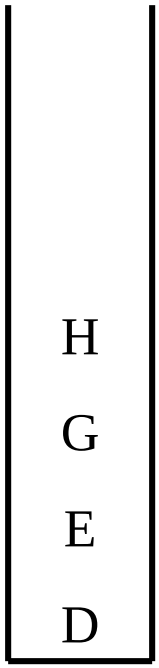
No unvisited nodes adjacent to B. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

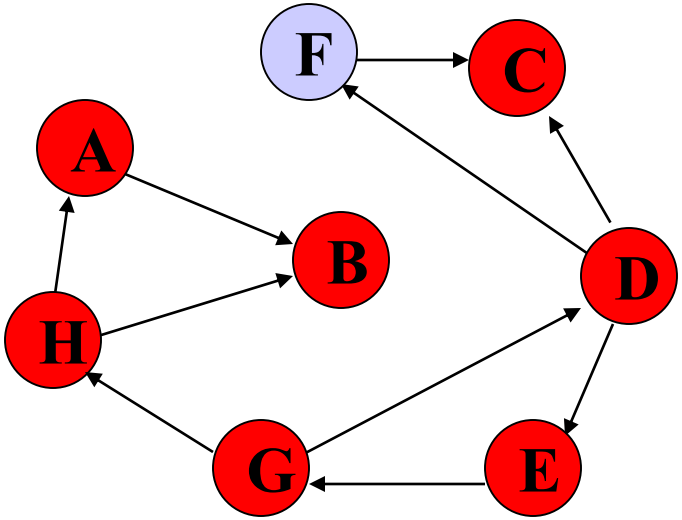


The order nodes are visited:

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

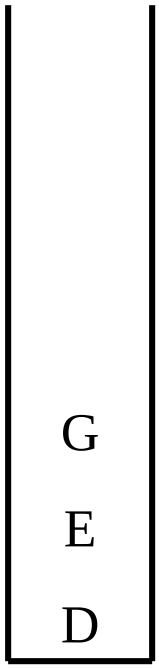
No unvisited nodes adjacent to A. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

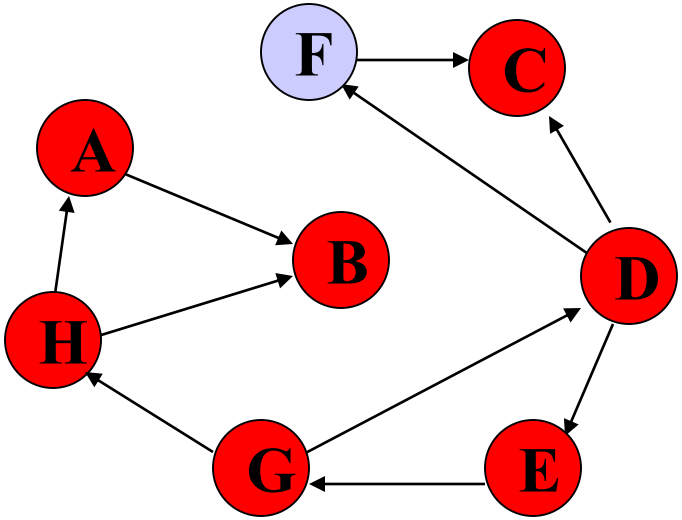


The order nodes are visited:

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

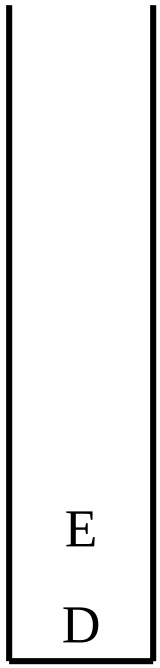
No unvisited nodes adjacent to H. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

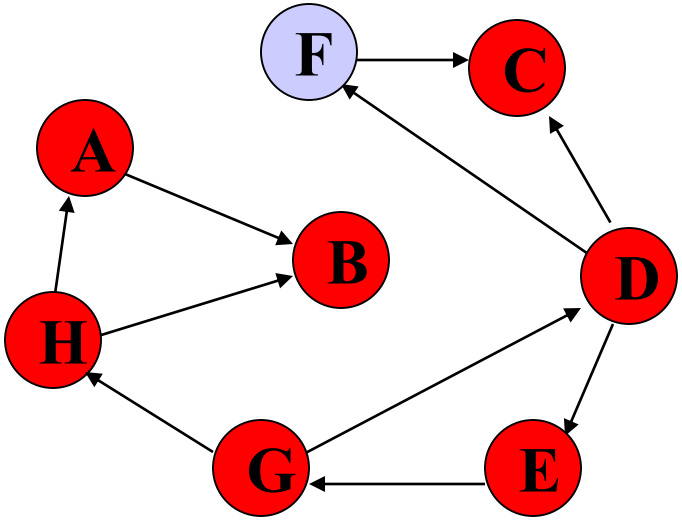


The order nodes are visited:

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

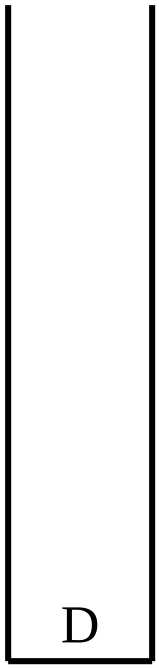
No unvisited nodes adjacent to G. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

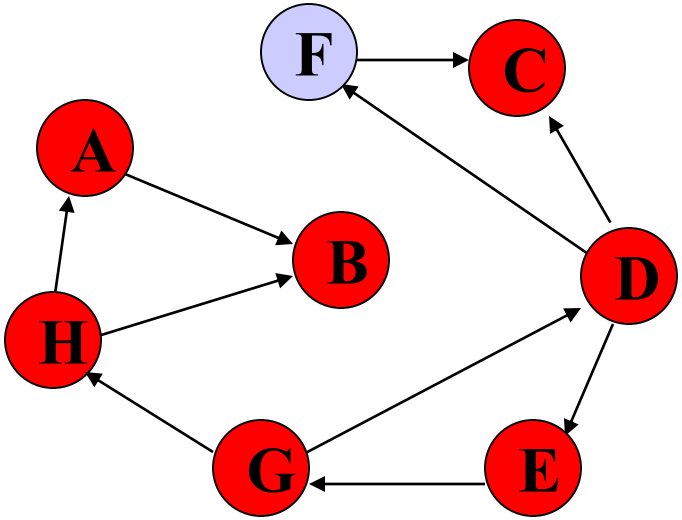


The order nodes are visited:

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

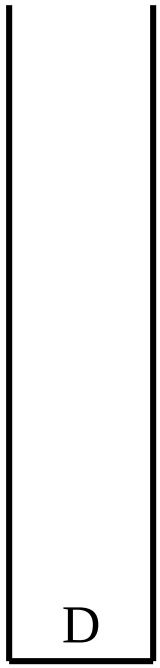
No unvisited nodes adjacent to E. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

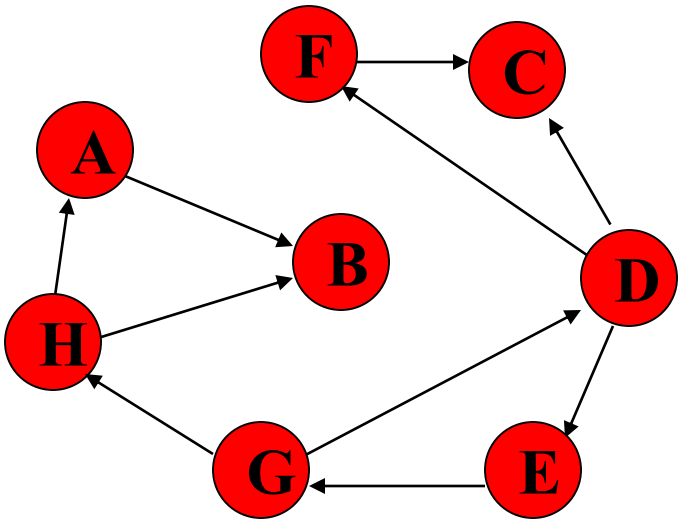


The order nodes are visited:

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

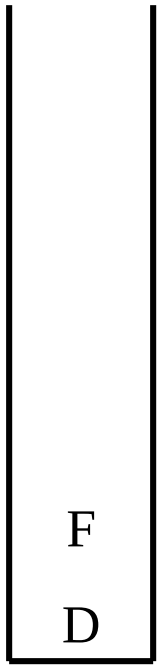
F is unvisited and is adjacent to D. Decide to visit F next.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

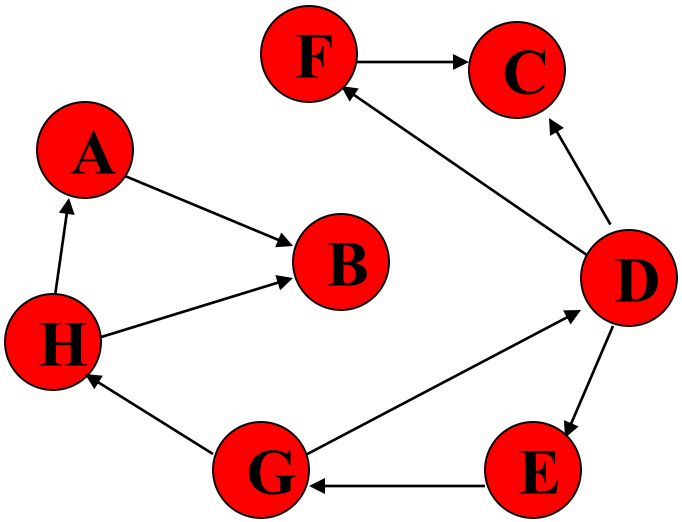


Visit F

The order nodes are visited:

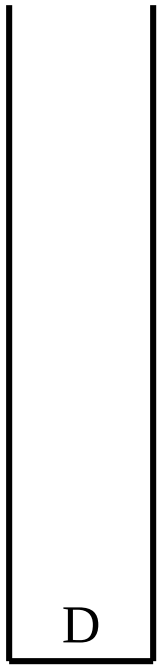
D, C, E, G, H, A, B, F

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

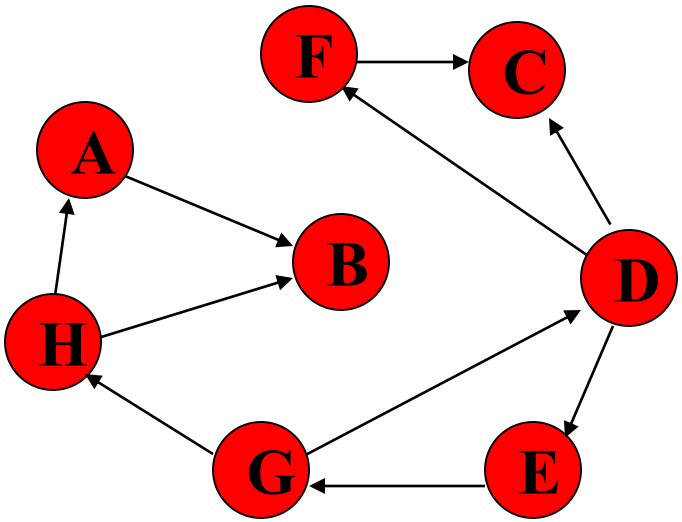


The order nodes are visited:

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

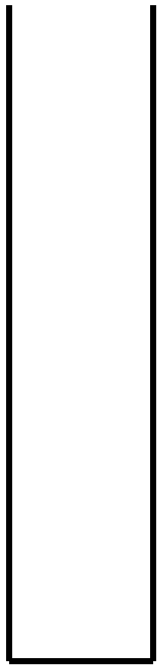
No unvisited nodes adjacent to F. Backtrack.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

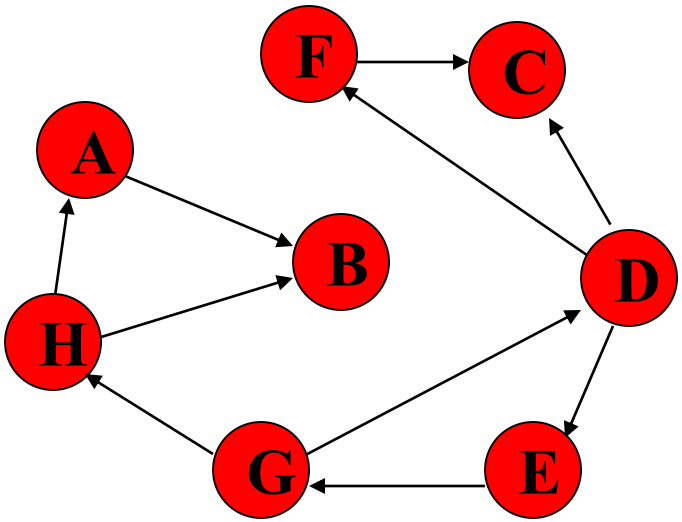


The order nodes are visited:

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

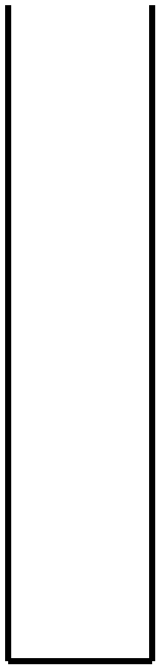
No unvisited nodes adjacent to D. Backtrack.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√



The order nodes are visited:

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

Stack is empty. Depth-first traversal is done.

- Was this a true search?
  - How would we make it a true search?
- Was this a true traversal?
  - How would we make it a true traversal?

Time and Space Complexity for Depth-First Search

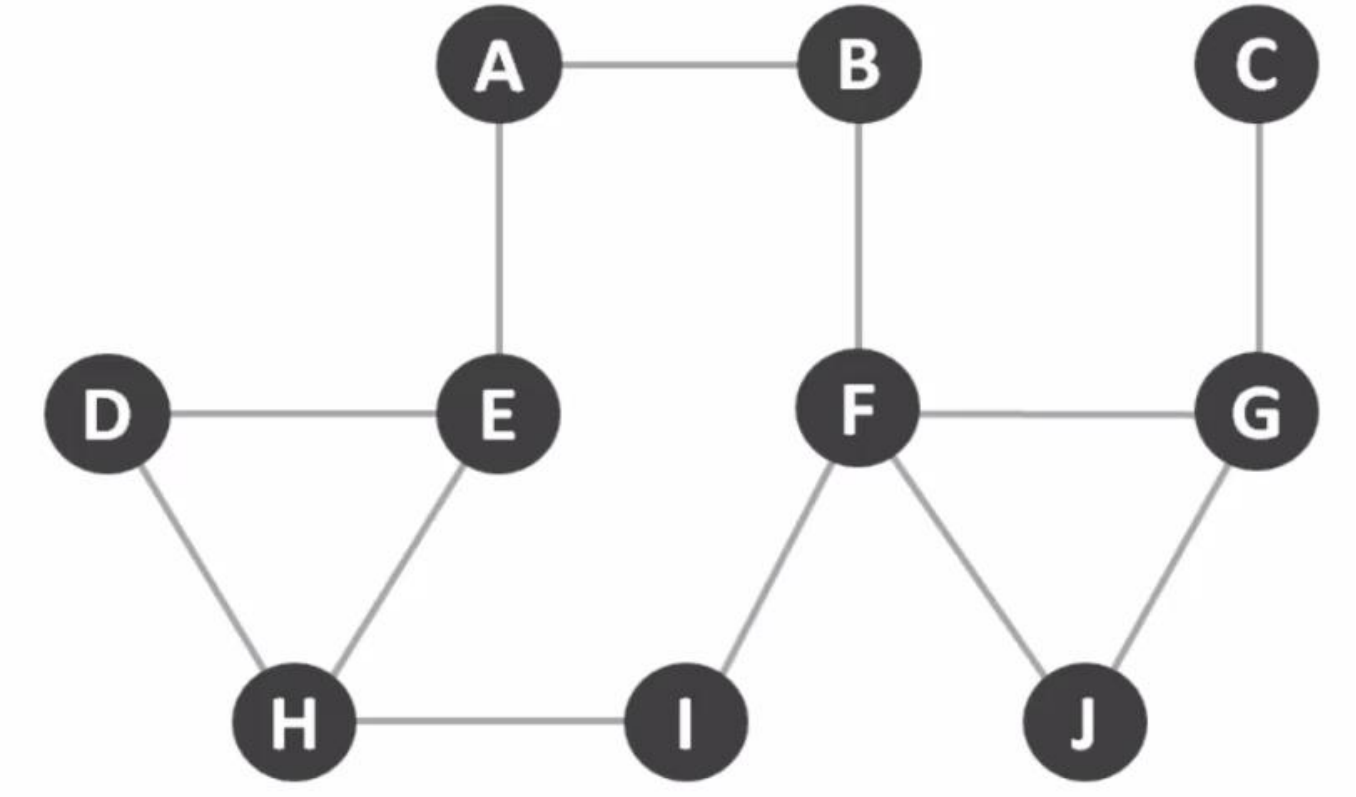
- Time Complexity
  - Adjacency Lists
    - Each node is marked visited once
    - Each node is checked for each incoming edge
    - O (v + e)
  - Adjacency Matrix
    - Have to check all entries in matrix: $O(n^2)$

Time and Space Complexity for Depth-First Search

- Space Complexity
  - Stack to handle nodes as they are explored
    - Worst case: all nodes put on stack (if graph is linear)
    - $O(n)$

Question for Practice

- Perform the Depth-First-Search starting with node D



Breadth-First Search

- BFS follows the following rules:
  1. Select an unvisited node x, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
  2. From each node z in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of z. The newly visited nodes from this level form a new level that becomes the next current level.
  3. Repeat step 2 until no more nodes can be visited.
  4. If there are still unvisited nodes, repeat from Step 1.

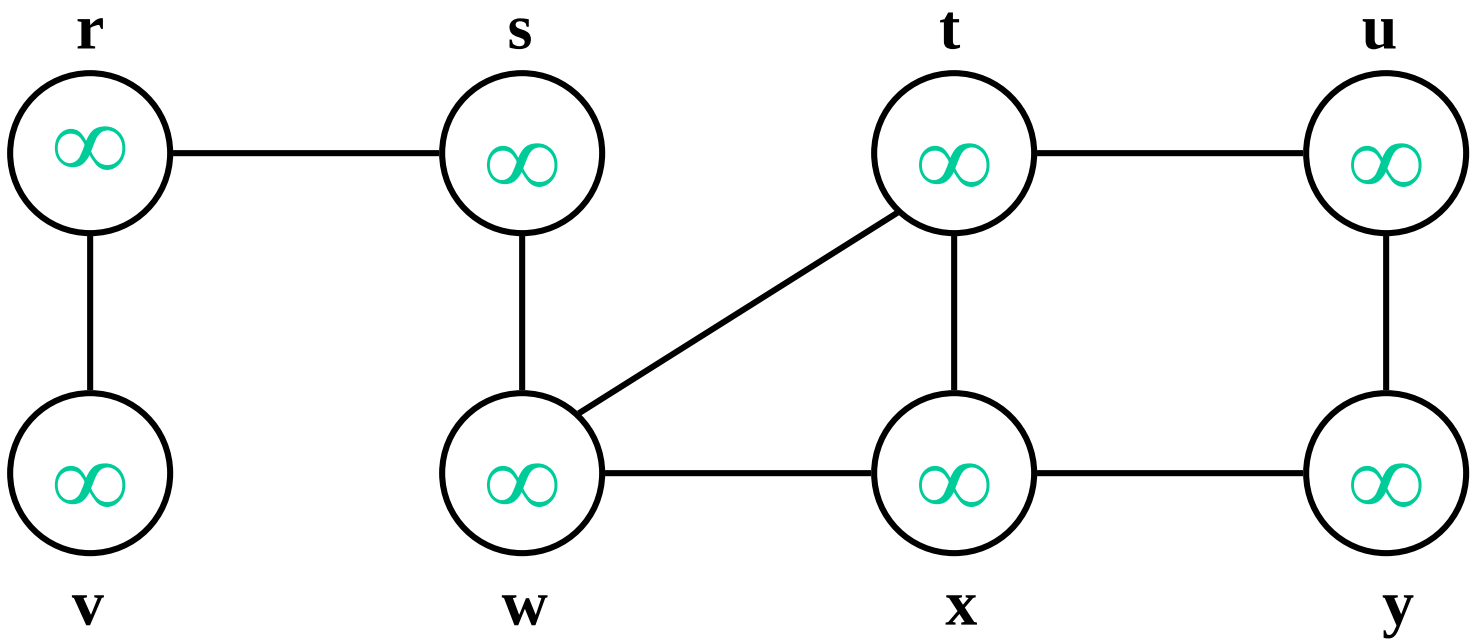
Review: Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.

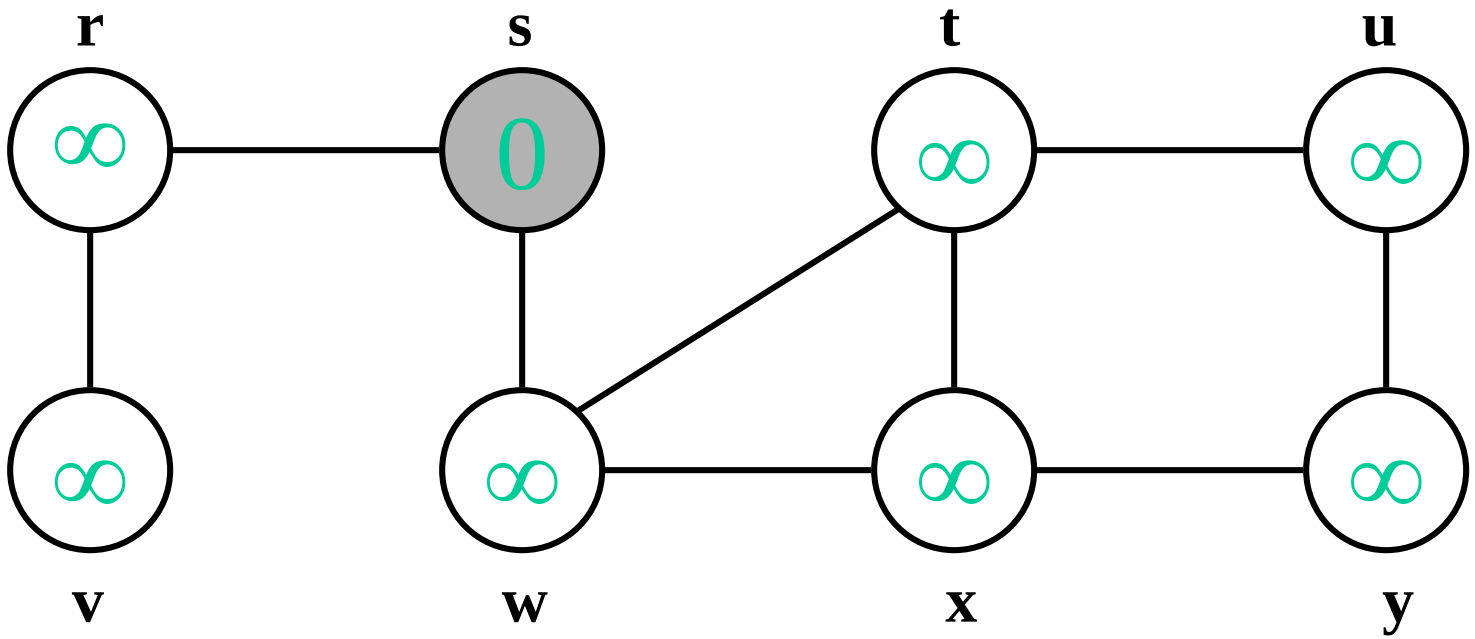
Review: Breadth-First Search

- Again will associate vertex "colors" to guide the algorithm
  - White vertices have not been discovered
    - All vertices start out white
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

Breadth-First Search: Example

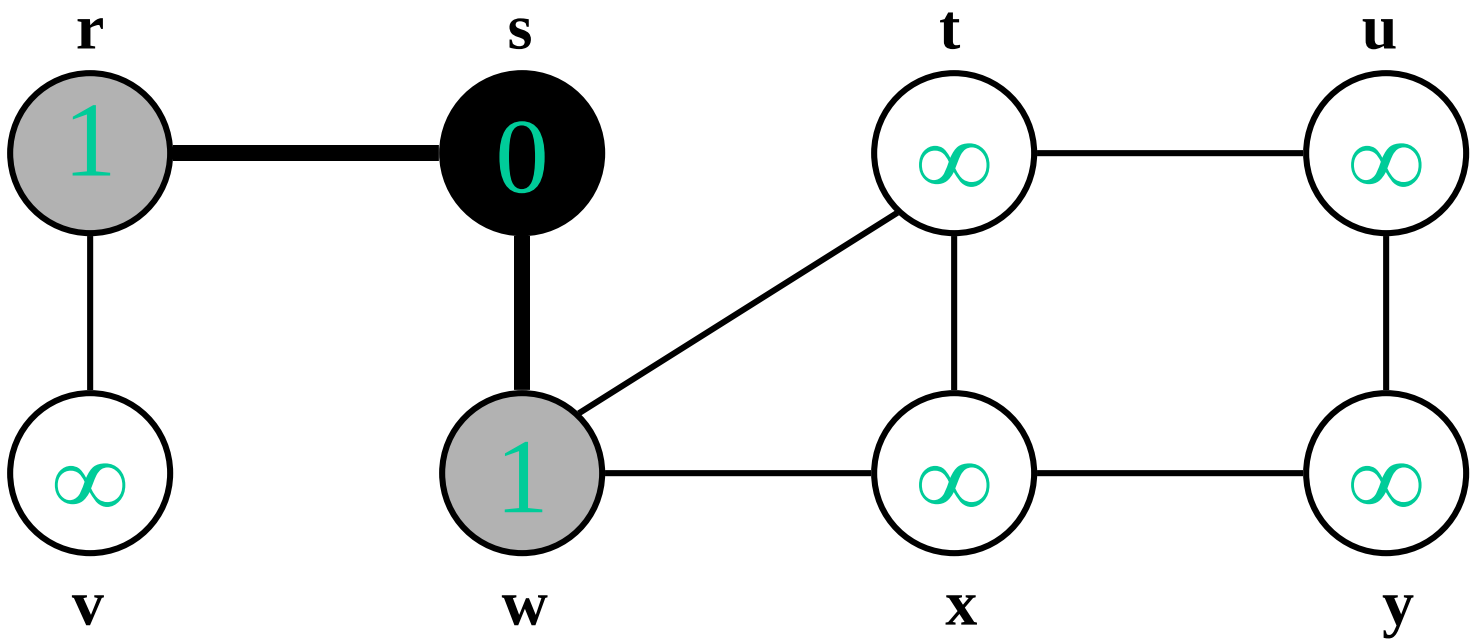


Breadth-First Search: Example



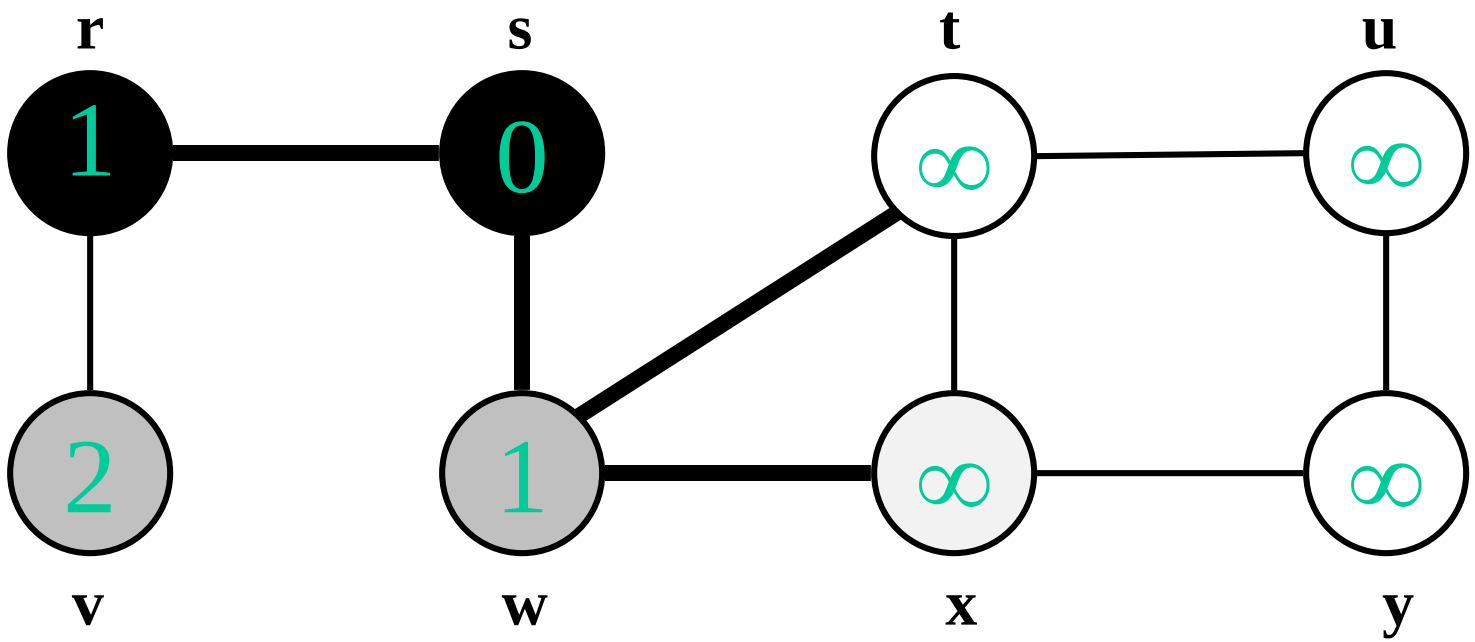
Q: s

Breadth-First Search: Example



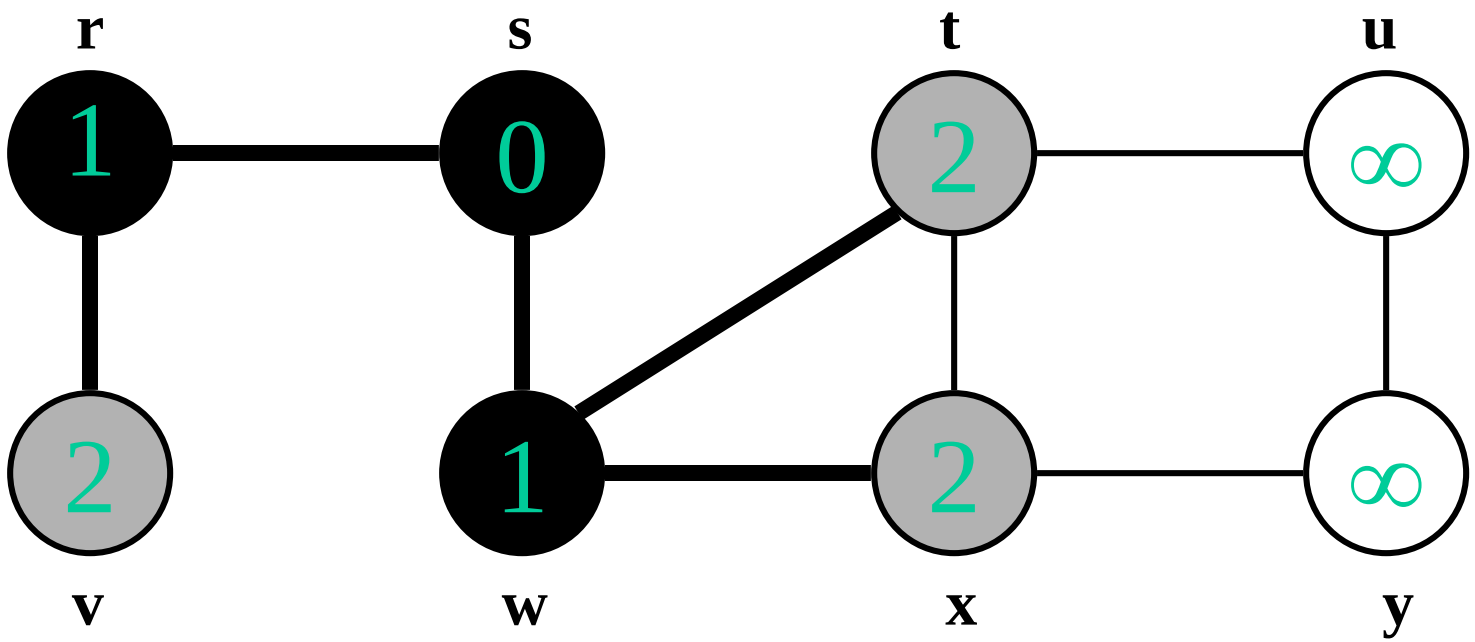
Q: | r | w |

Breadth-First Search: Example



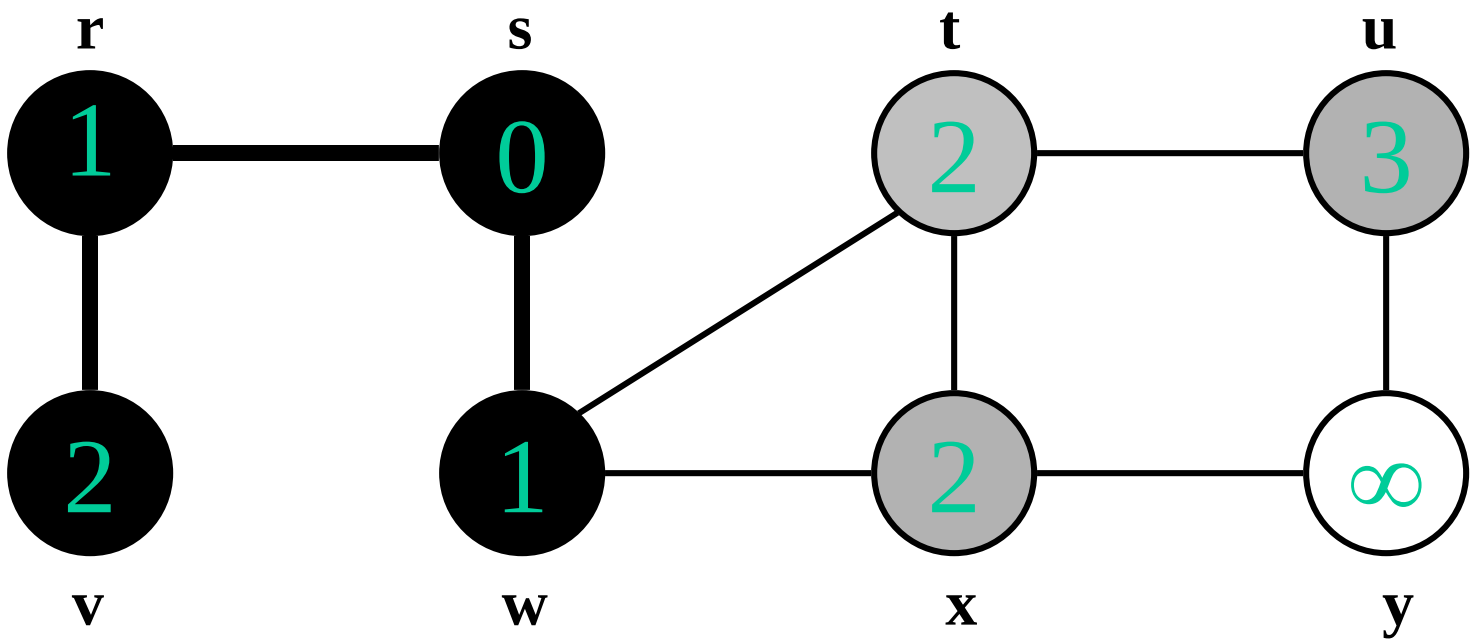
Q: | w | v |

Breadth-First Search: Example



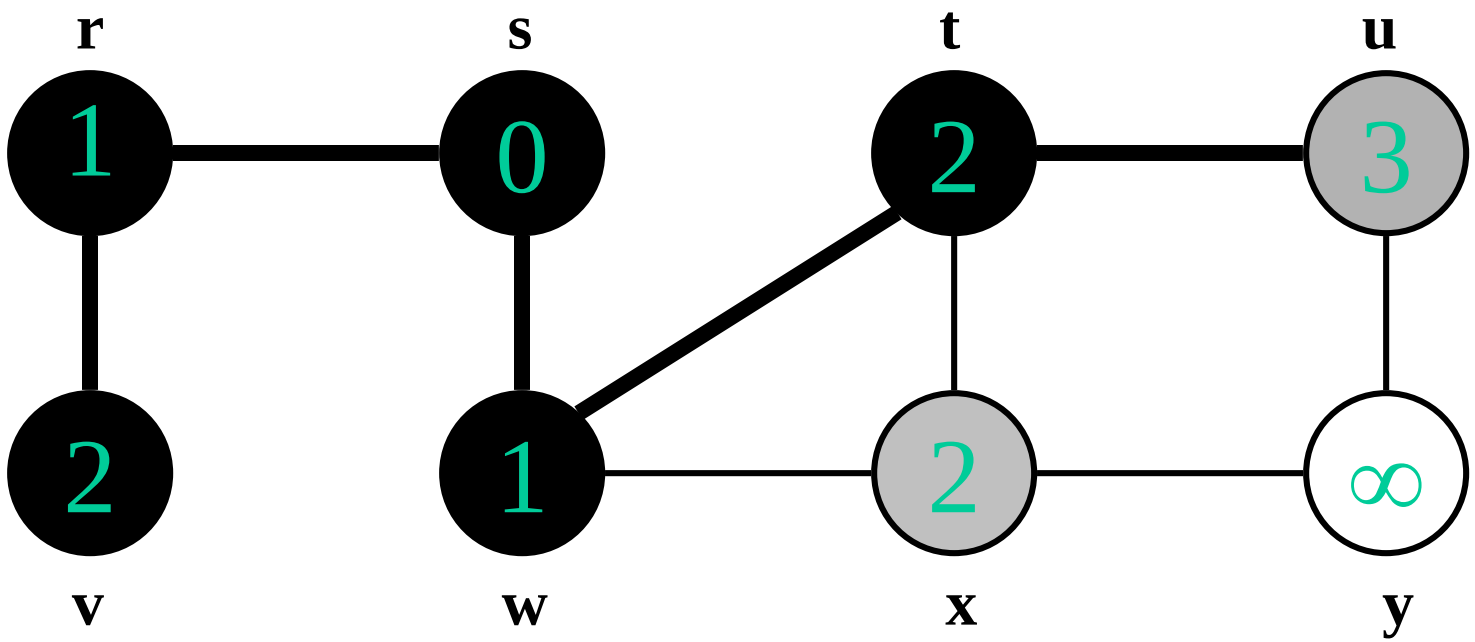
Q: | v | t | x |

Breadth-First Search: Example



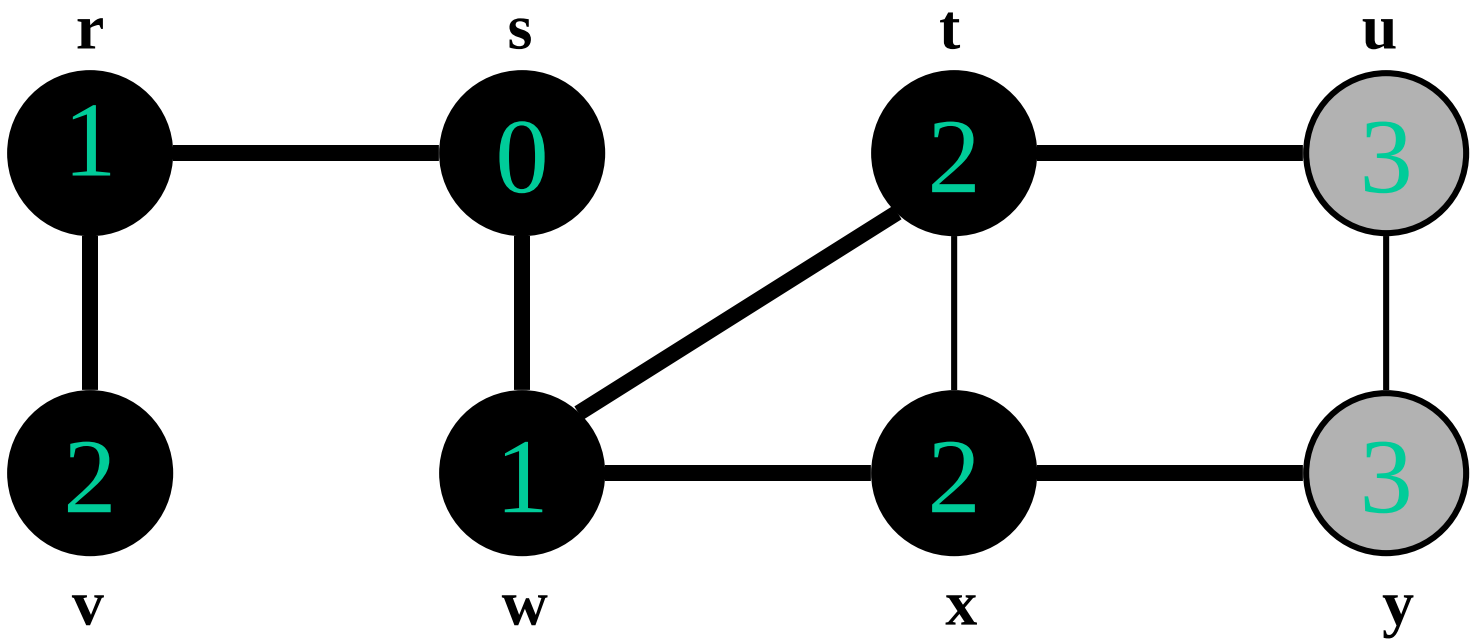
Q: | t | x |

Breadth-First Search: Example



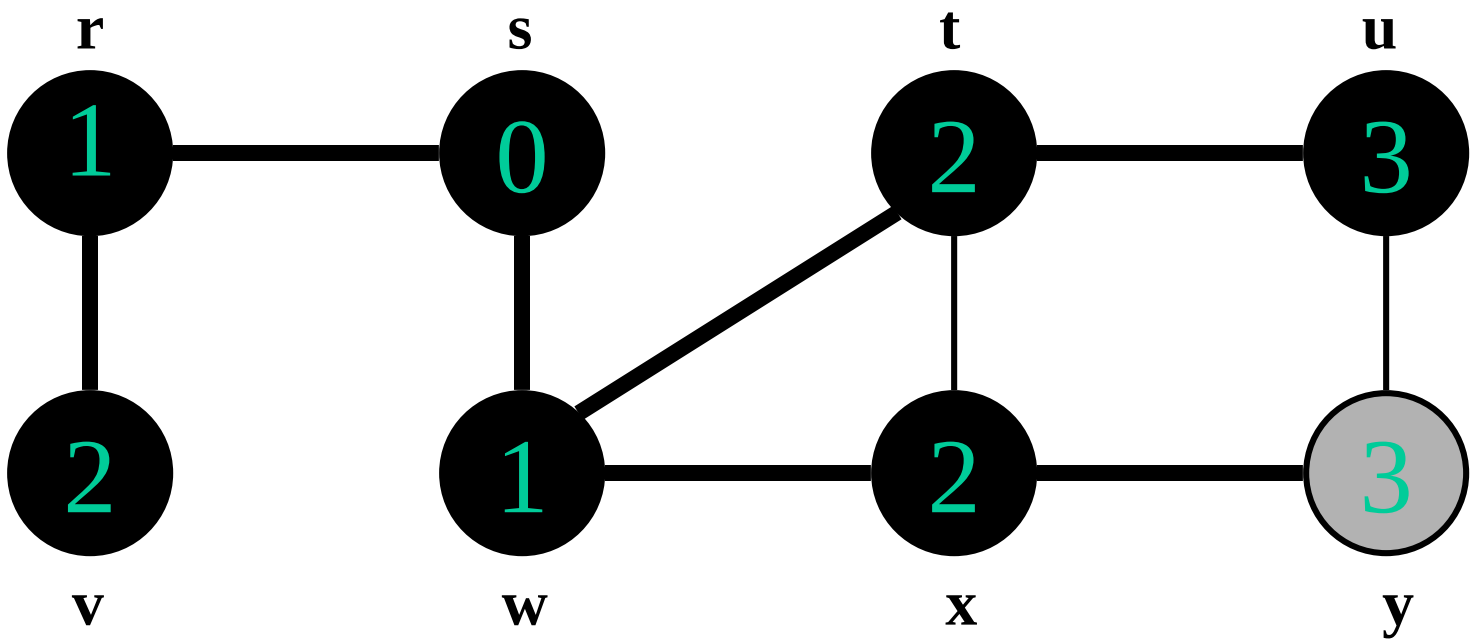
Q: | x | u |

Breadth-First Search: Example



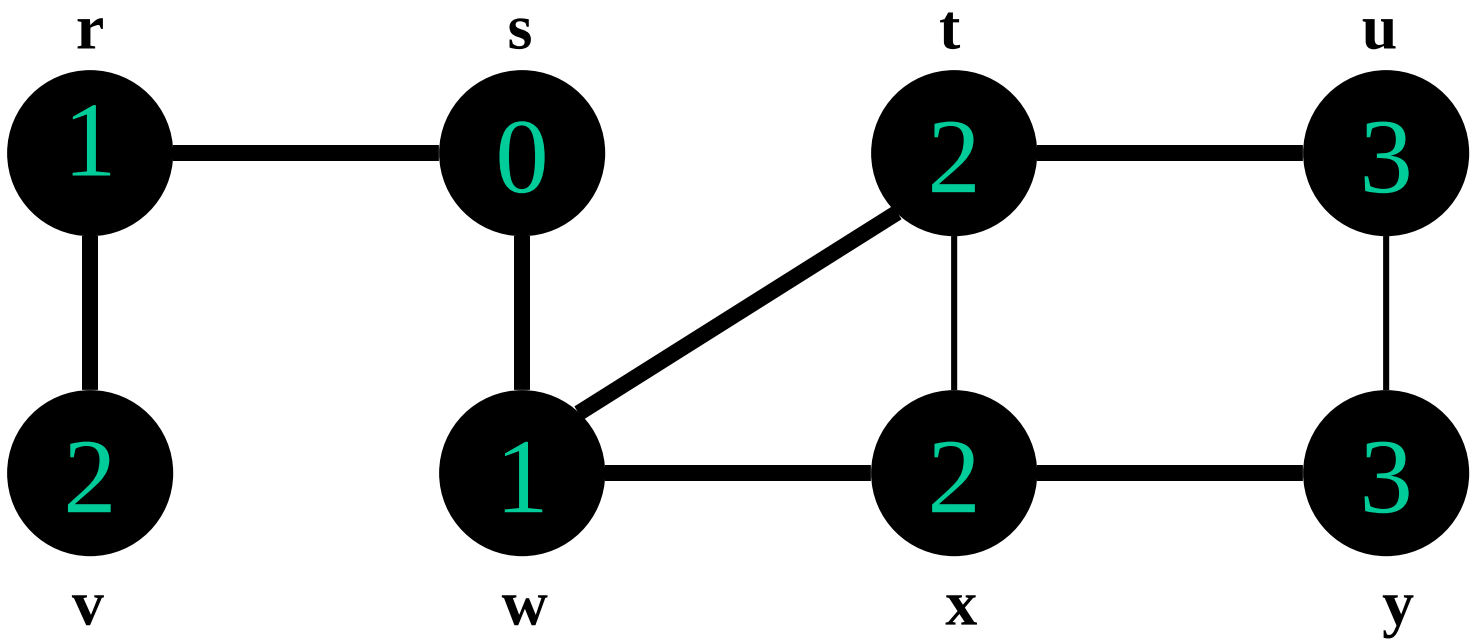
Q: | **u** | **y** |

Breadth-First Search: Example



Q: y

Breadth-First Search: Example



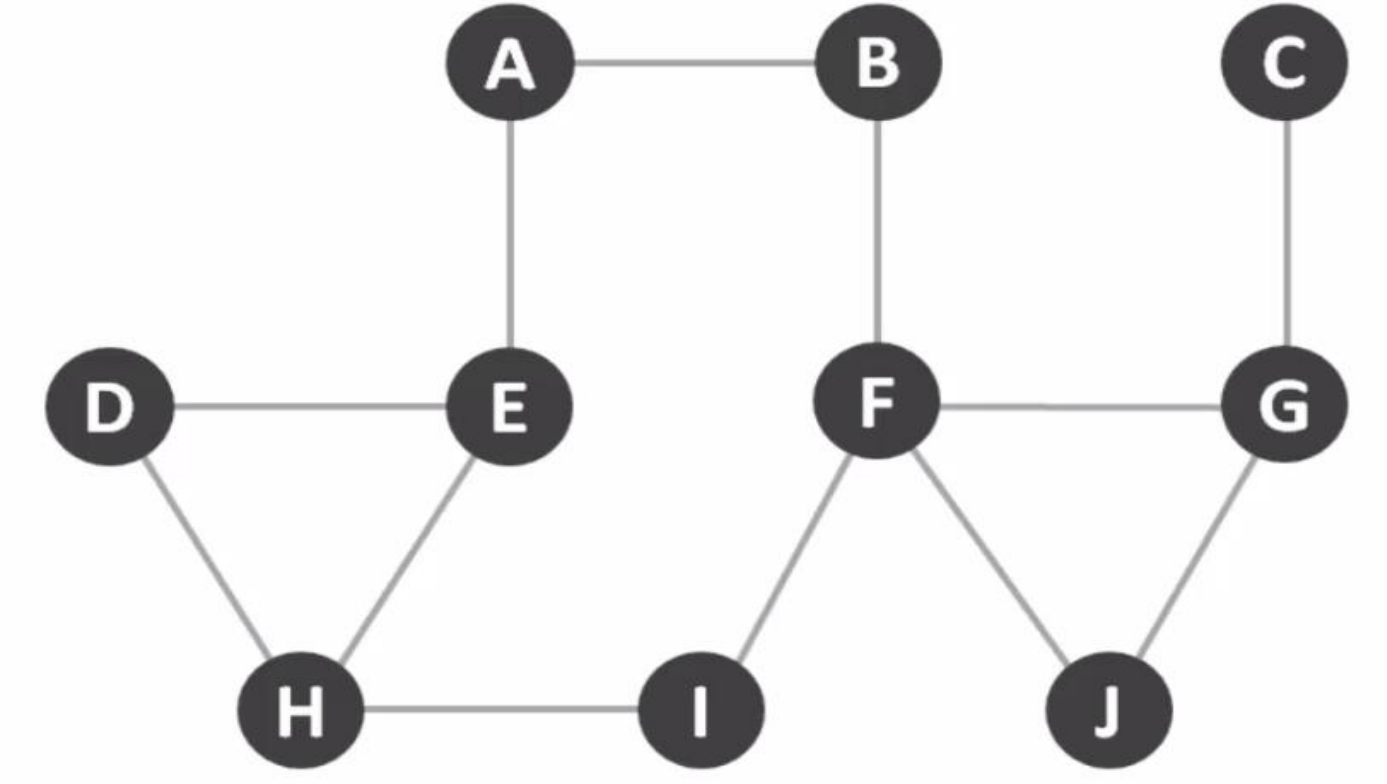
Q: Ø

BFS (Pseudo Code)

```
BFS(input: graph G) {
   Queue Q;    Integer x, z, y;
   while (G has an unvisited node x) {
         visit(x); Enqueue(x,Q);
         while (Q is not empty){
                  z := Dequeue(Q);
                  for all (unvisited neighbor y of z){
                                    visit(y); Enqueue(y,Q);
                  }
         }
   }
}
```

Practice question

- Perform breadth-first-search starting with the node D

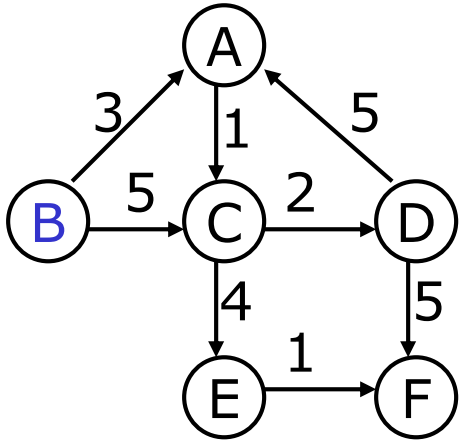


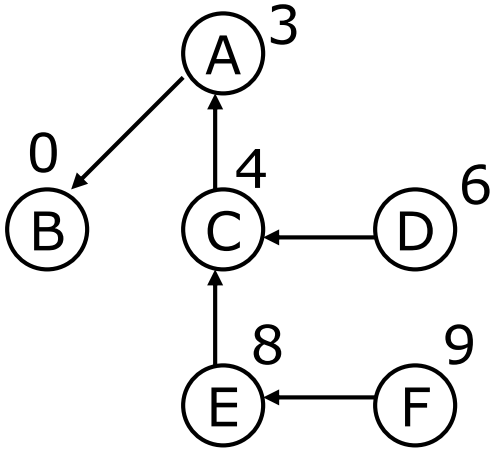
Shortest-path

- Suppose we want to find the shortest path from node X to node Y.
- It turns out that, in order to do this, we need to find the shortest path from X to *all* other nodes
  - Why?
  - If we don't know the shortest path from X to Z, we might overlook a shorter path from X to Y that contains Z
- Dijkstra's Algorithm finds the shortest path from a given node to *all* other reachable nodes

Dijkstra's algorithm I

- Dijkstra's algorithm builds up a *tree*: there is a path from each node back to the starting node
- For example, in the following graph, we want to find shortest paths from node B





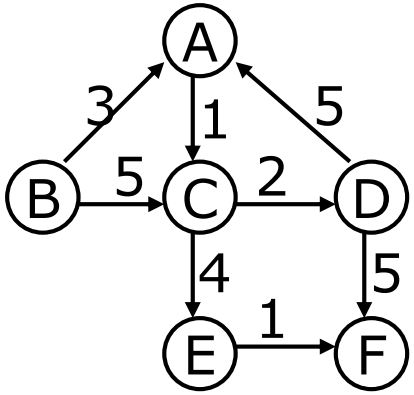
- Edge values in the graph are weights
- Node values in the tree are *total* weights
- The arrows point in the *right direction* for what we need (why?)

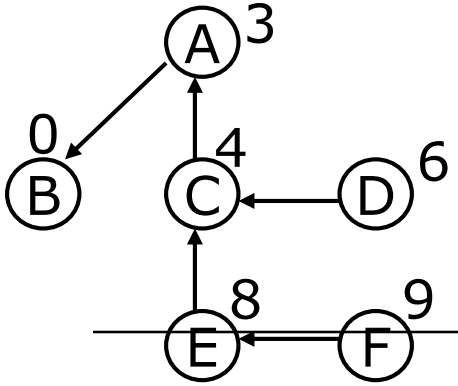
Dijkstra's algorithm II

- For each vertex v, Dijkstra's algorithm keeps track of three pieces of information:
  - A Boolean telling whether we *know* the shortest path to that node (initially true only for the starting node)
  - The length of the shortest path to that node known so far (0 for the starting node)
  - The predecessor of that node along the shortest known path (unknown for all nodes)
- Dijkstra's algorithm proceeds in phases—at each step:
  - From the vertices for which we don't know the shortest path, pick a vertex v with the smallest distance known so far
  - Set v's "known" field to true
  - For each vertex w adjacent to v, test whether its distance so far is greater than v's distance plus the distance from v to w; if so, set w's distance to the new distance and w's predecessor to v

Dijkstra's algorithm III

- Three pieces of information for each node (e.g. +3B):
  - + if the minimum distance is known *for sure*, blank otherwise
  - The best distance so far (3 in the example)
  - The node's predecessor (B in the example, - for the starting node)

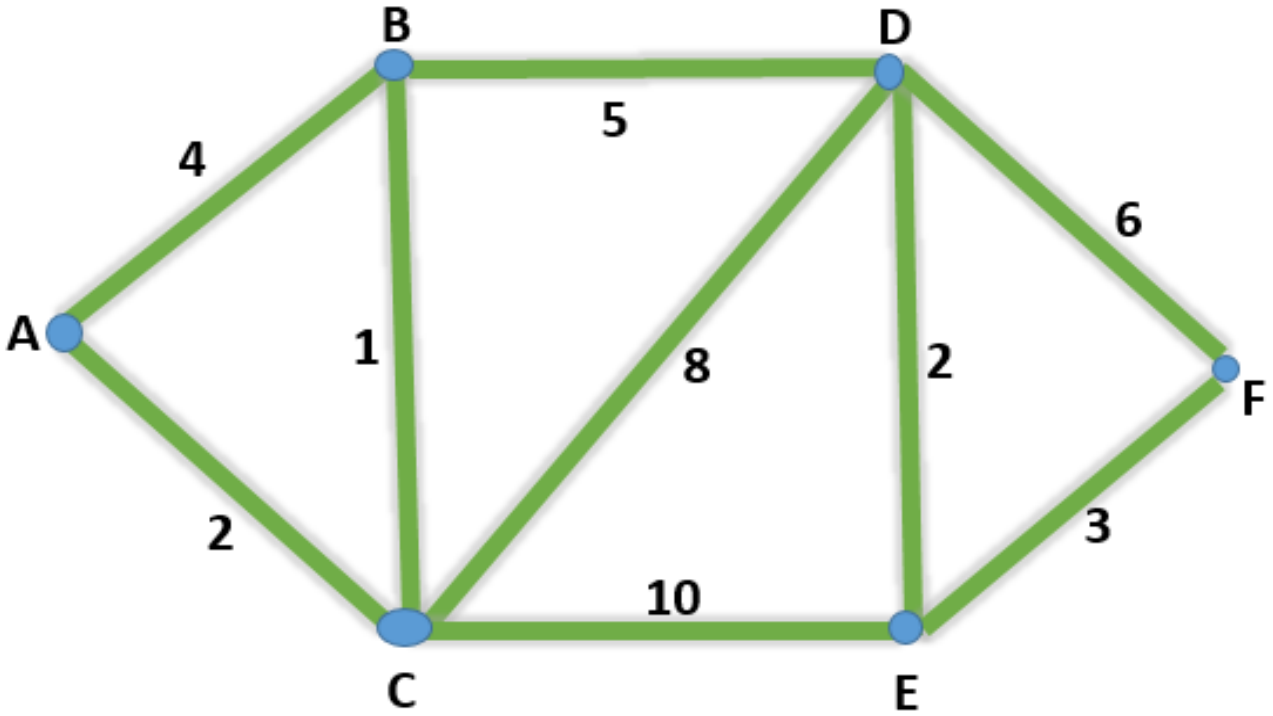




node	init'ly	1	2	3	4	5	6
A	inf	3B	+3B	+3B	+3B	+3B	+3B
B	0-	+0-	+0-	+0-	+0-	+0-	+0-
C	inf	5B	4A	+4A	+4A	+4A	+4A
D	inf	inf	inf	6C	+6C	+6C	+6C
E	inf	inf	inf	8C	8C	+8C	+8C
F	inf	inf	inf	inf	11D	9E	+9E

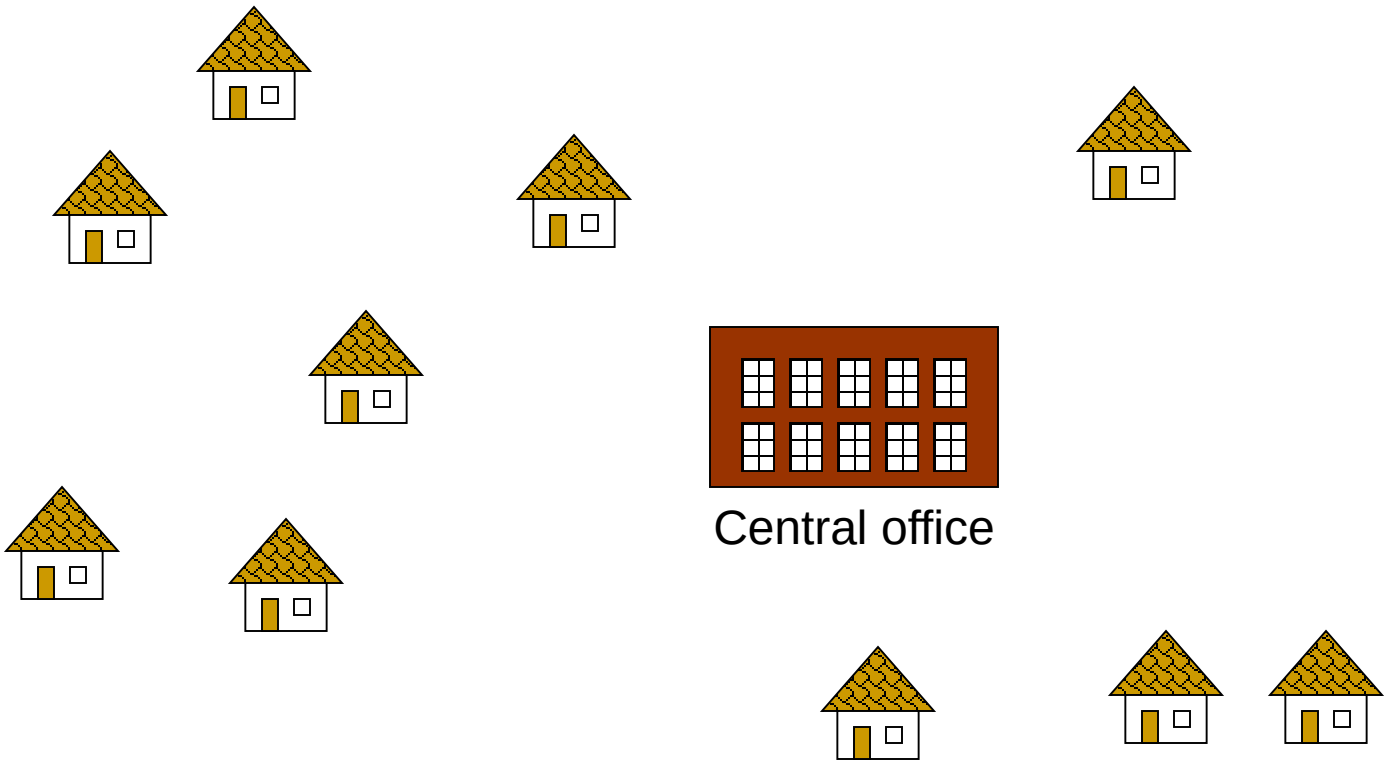
Dijkstra's algorithm Example

- Determine the shortest distance from node A to node F

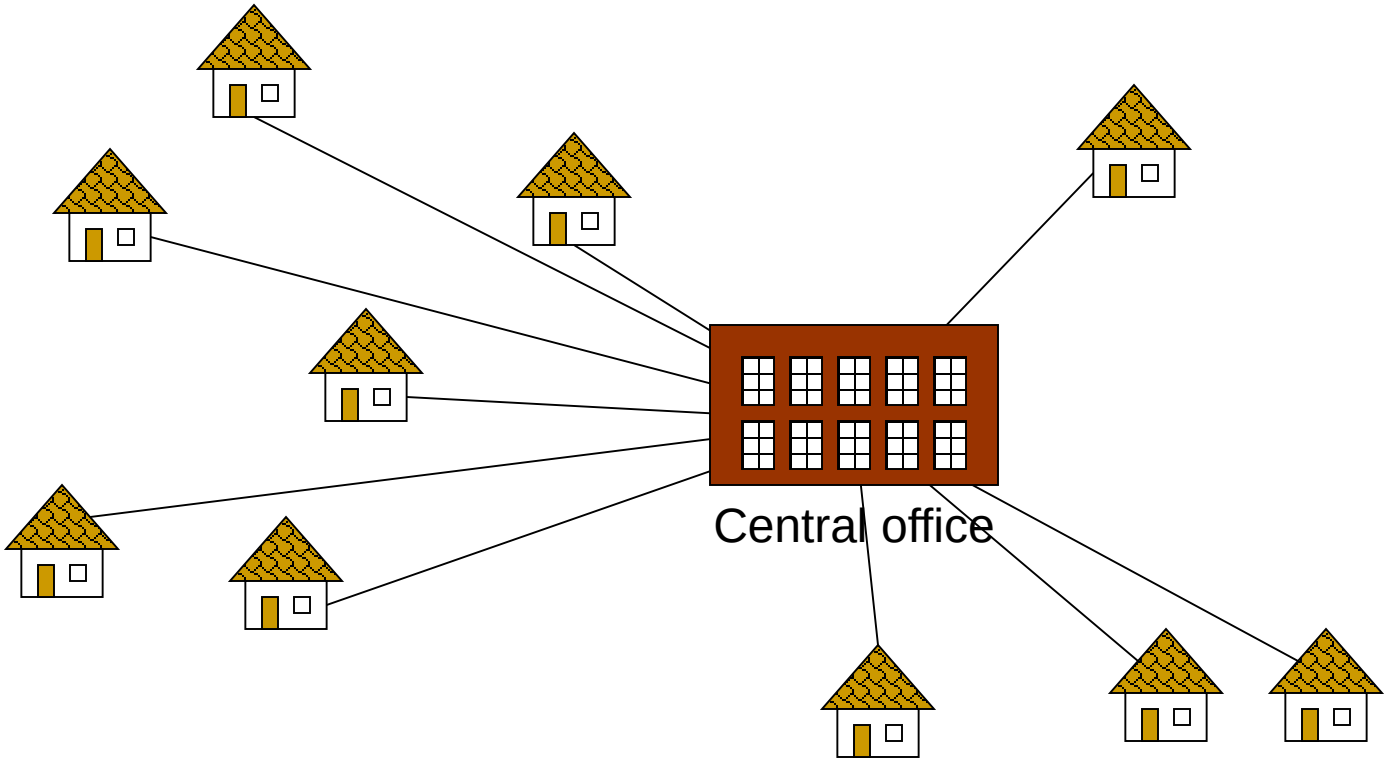


Minimum Spanning Trees

Problem: Laying Telephone Wire

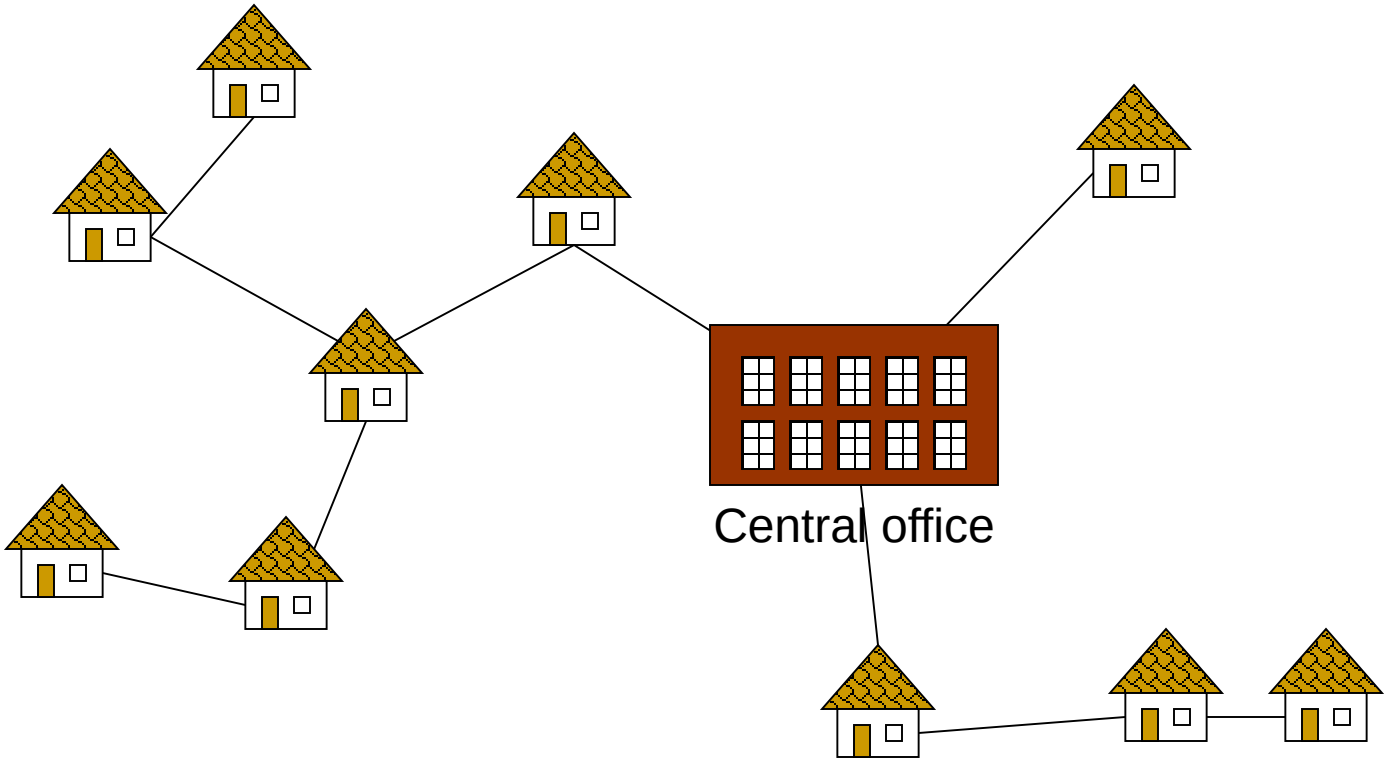


Wiring: Naïve Approach



Expensive!

Wiring: Better Approach



Minimize the total length of wire connecting the customers

Minimum Spanning Tree

- A spanning tree of a graph is a tree that has all the vertices of the graph connected by some edges.
- A graph can have one or more number of spanning trees.
- If the graph has N vertices then the spanning tree will have N-1 edges.
- A minimum spanning tree (MST) is a spanning tree that has the minimum weight than all other spanning trees of the graph.

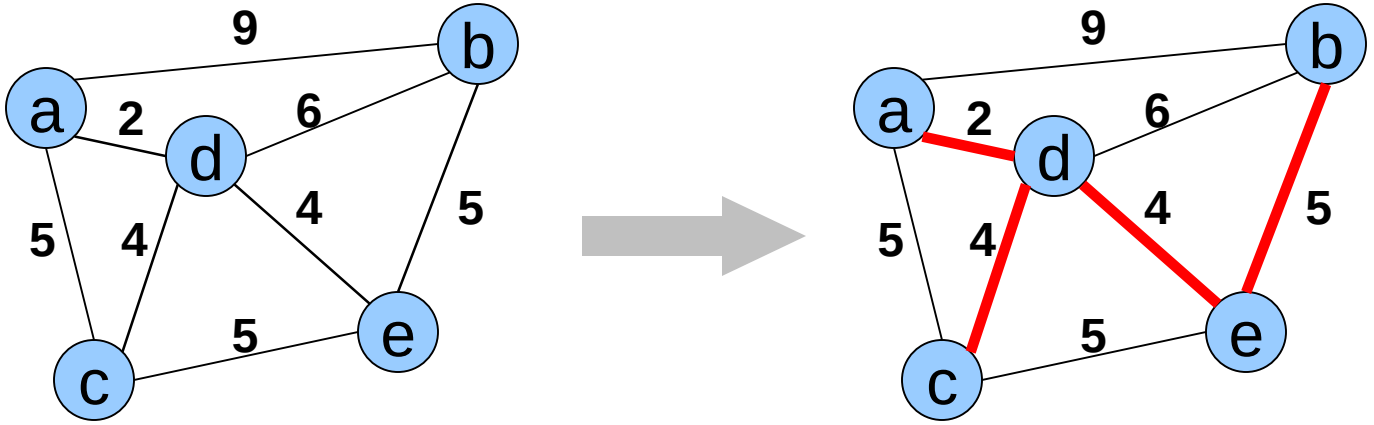
- A Minimum Spanning Tree (MST) is a subgraph of an undirected graph such that the subgraph spans (includes) all nodes, is connected, is acyclic, and has minimum total edge weight

Minimum Spanning Tree (MST) (see Weiss, Section 24.2.2)

A **minimum spanning tree** is a subgraph of an undirected weighted graph G, such that

- it is a tree (i.e., it is acyclic)
- it covers all the vertices V
  - contains $|V| - 1$ edges
- the total cost associated with tree edges is the minimum among all possible spanning trees
- not necessarily unique

How Can We Generate a MST?



Algorithm Characteristics

- Both Prim's and Kruskal's Algorithms work with undirected graphs
- Both work with weighted and unweighted graphs but are more interesting when edges are weighted
- Both are greedy algorithms that produce optimal solutions

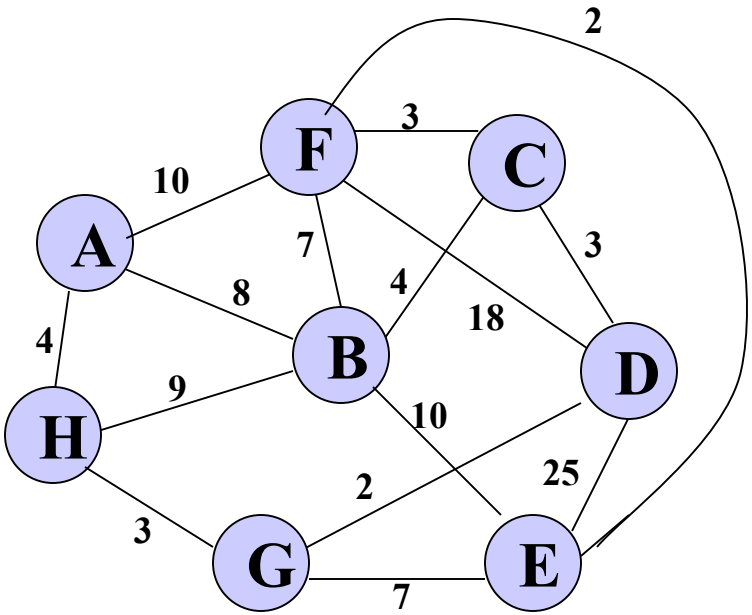
Prim's Algorithm

1. Select any vertex
2. Select the shortest edge connected to that vertex
3. Select the shortest edge connected to any vertex already connected
4. Repeat step 3 until all vertices have been connected

Prim's Algorithm

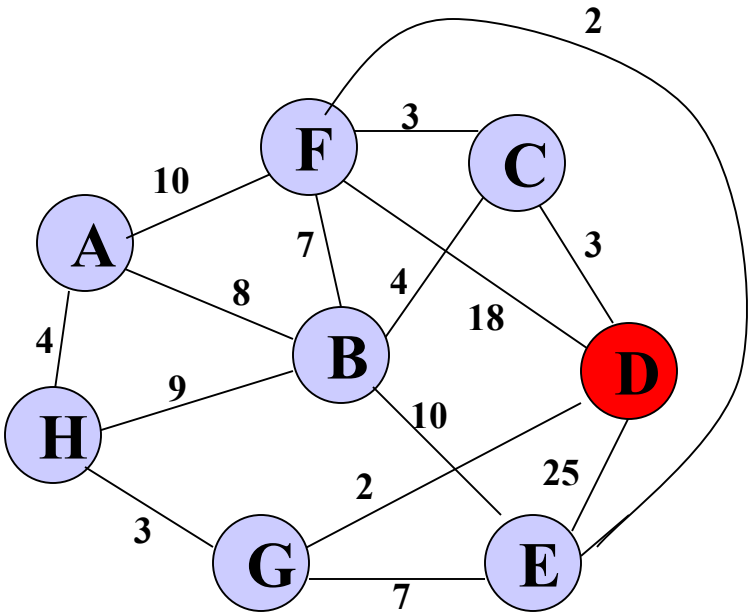
- Similar to Dijkstra's Algorithm except that d_v records edge weights, not path lengths

Walk-Through



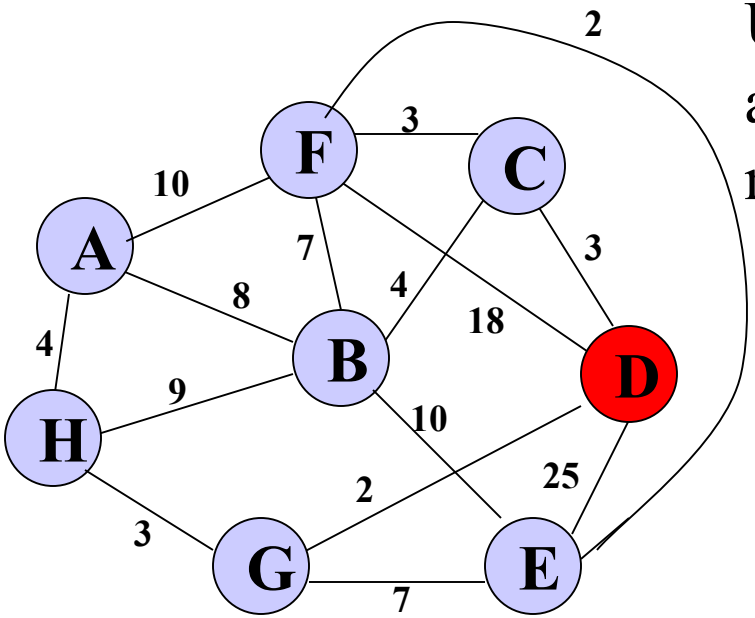
Initialize array

	K	d_v	p_v
A	F	∞	–
B	F	∞	–
C	F	∞	–
D	F	∞	–
E	F	∞	–
F	F	∞	–
G	F	∞	–
H	F	∞	–



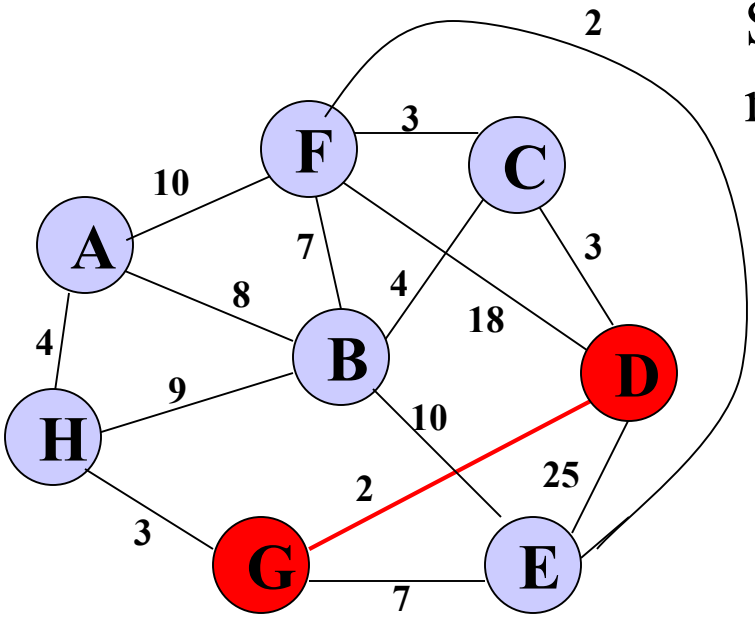
Start with any node, say D

	K	d_v	p_v
A			
B			
C			
D	T	0	–
E			
F			
G			
H			



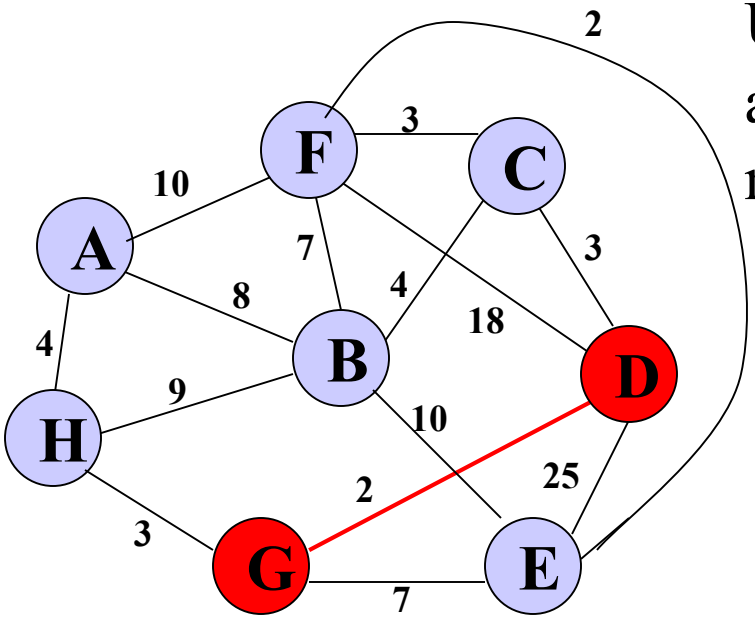
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G		2	D
H			



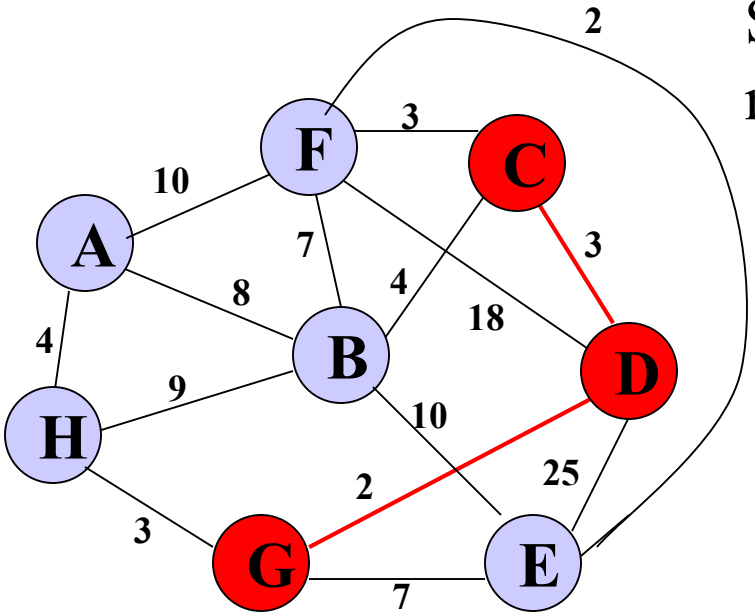
Select node with minimum distance

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G	T	2	D
H			



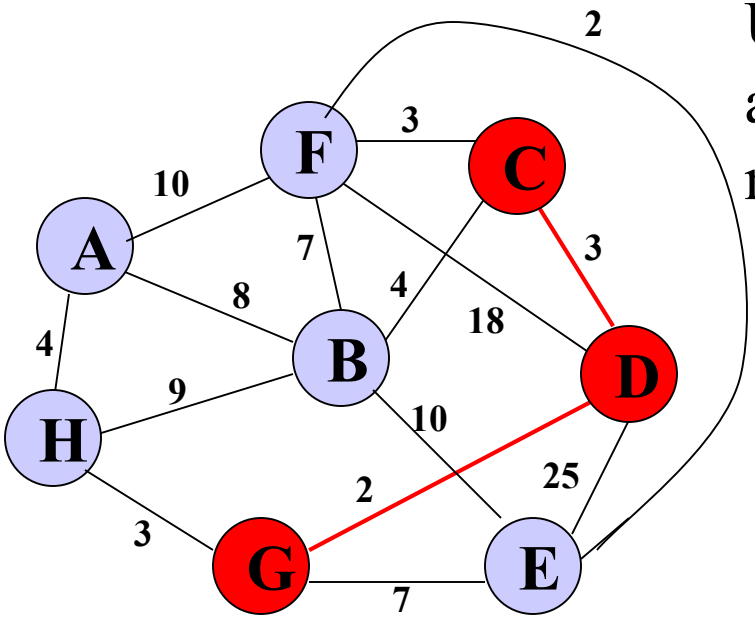
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G



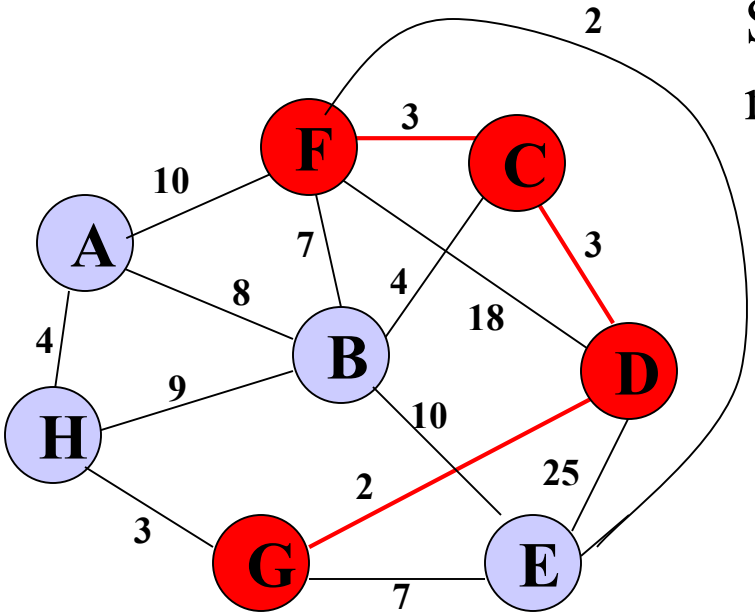
Select node with minimum distance

	K	d_v	p_v
A			
B			
C	T	3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G



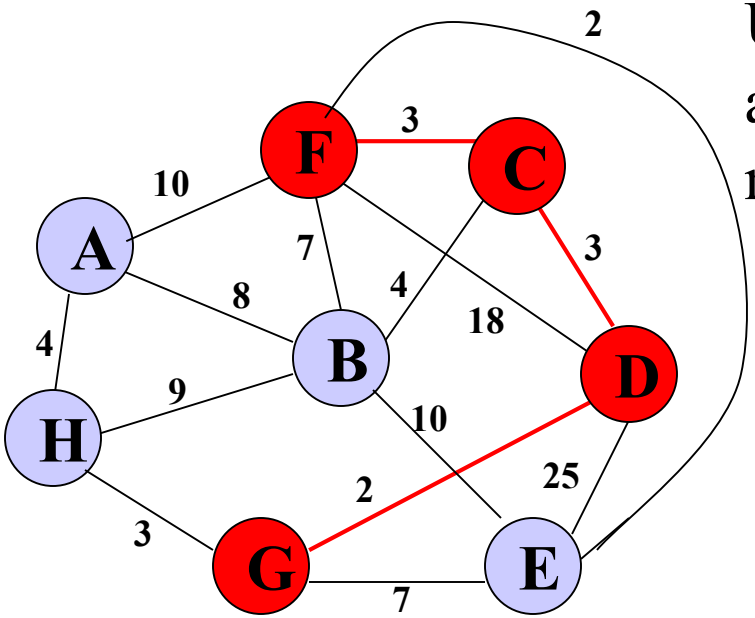
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F		3	C
G	T	2	D
H		3	G



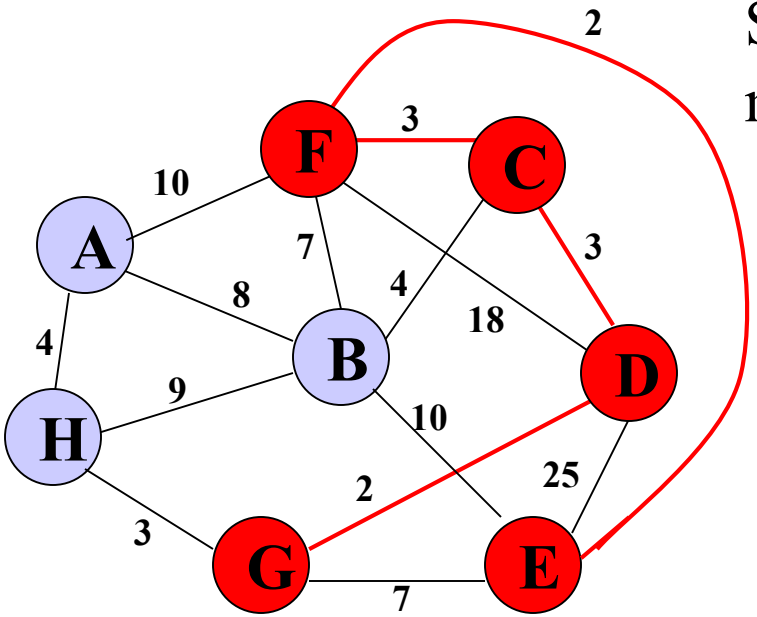
Select node with minimum distance

	K	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F	T	3	C
G	T	2	D
H		3	G



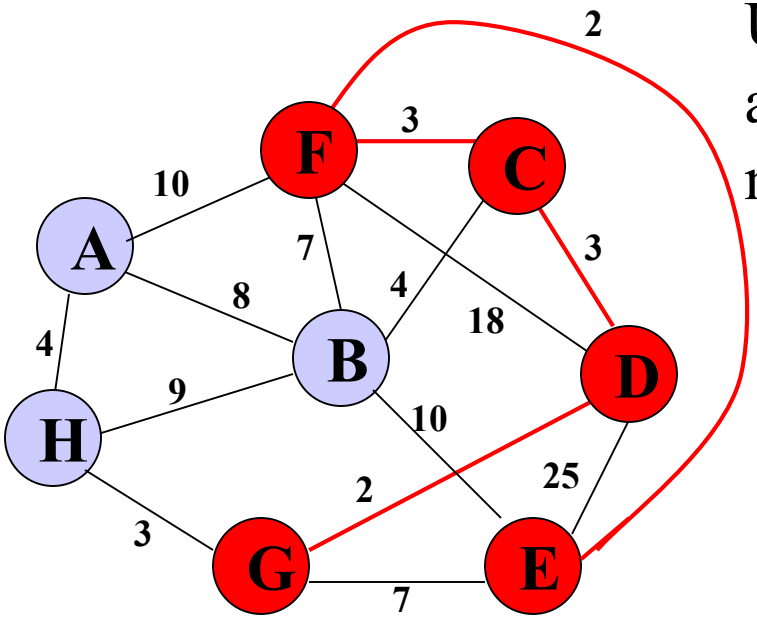
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E		2	F
F	T	3	C
G	T	2	D
H		3	G



Select node with minimum distance

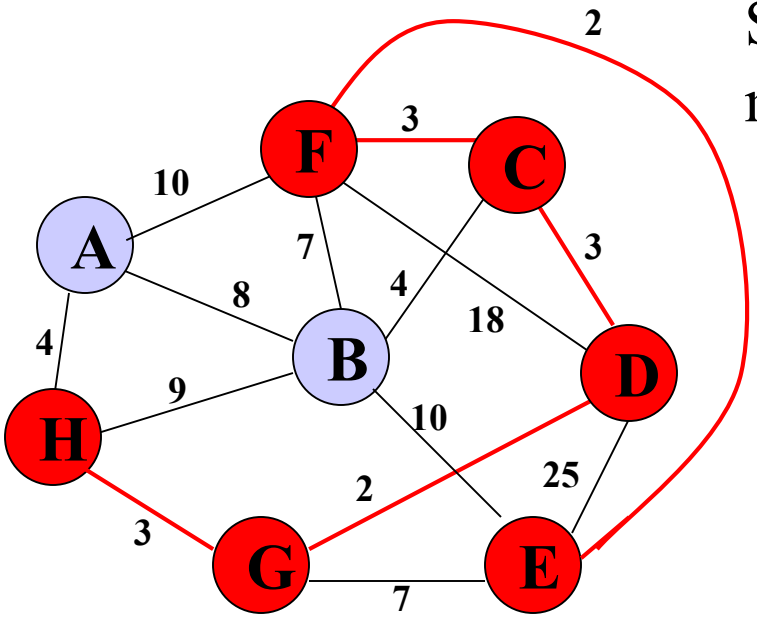
	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G



Update distances of adjacent, unselected nodes

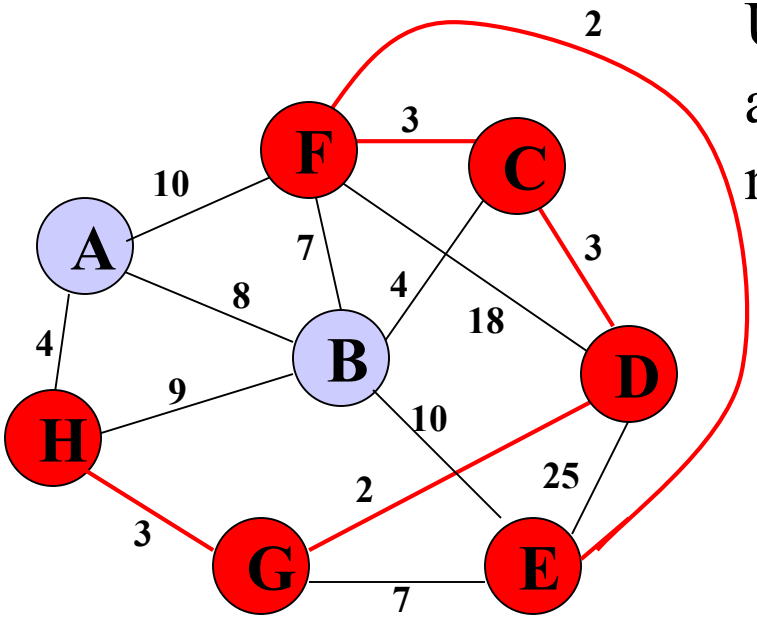
	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

Table entries unchanged



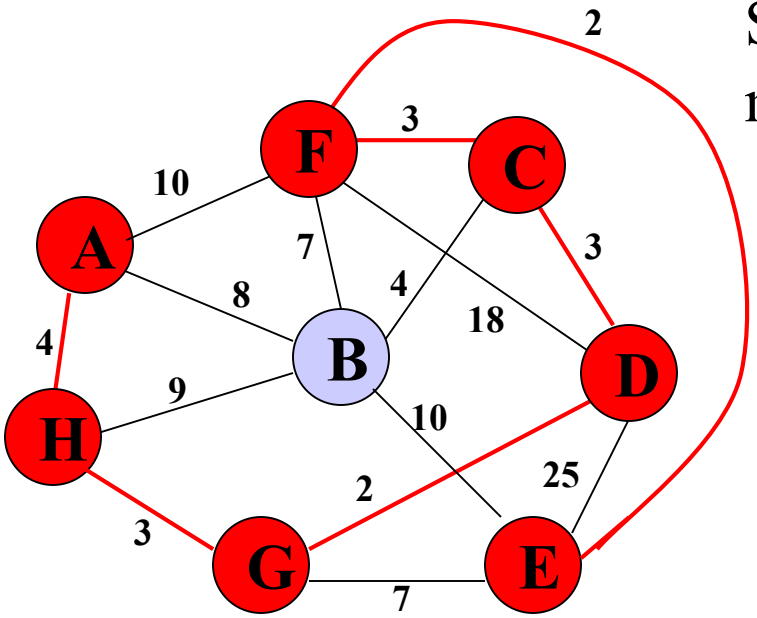
Select node with minimum distance

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



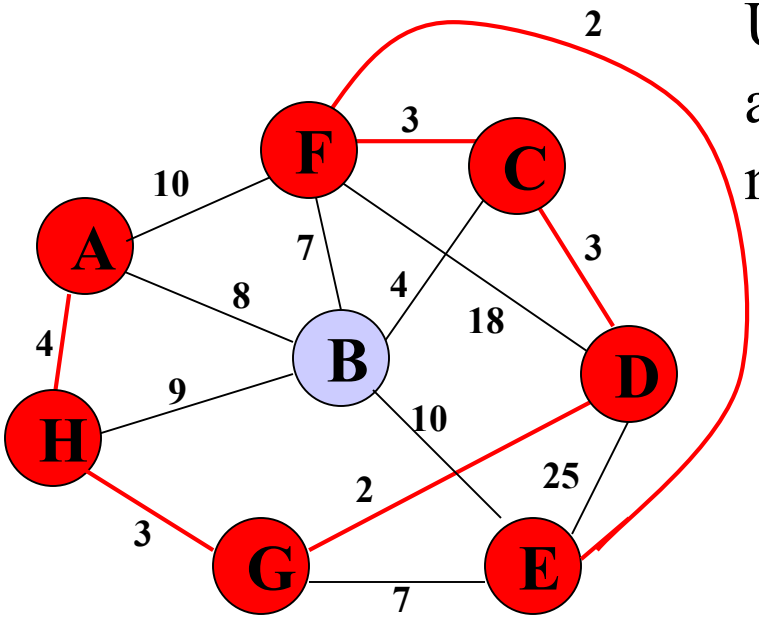
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A		4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Select node with minimum distance

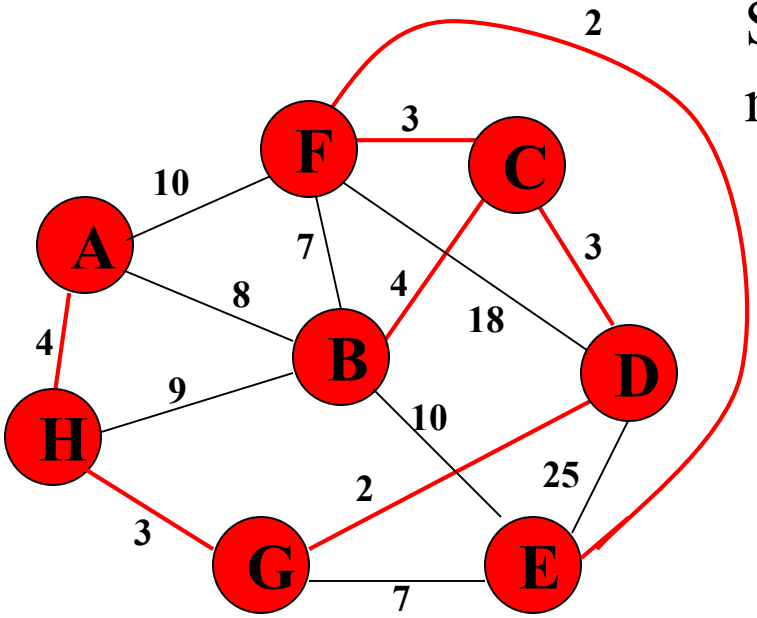
	K	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Update distances of adjacent, unselected nodes

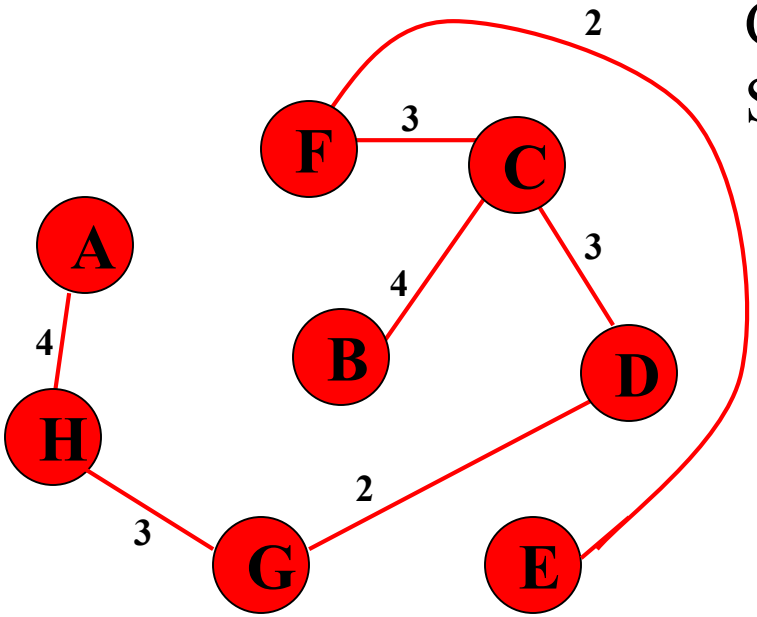
	K	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Table entries unchanged



Select node with minimum distance

	K	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Cost of Minimum Spanning Tree = $\Sigma\, d_v = 21$

	K	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Done

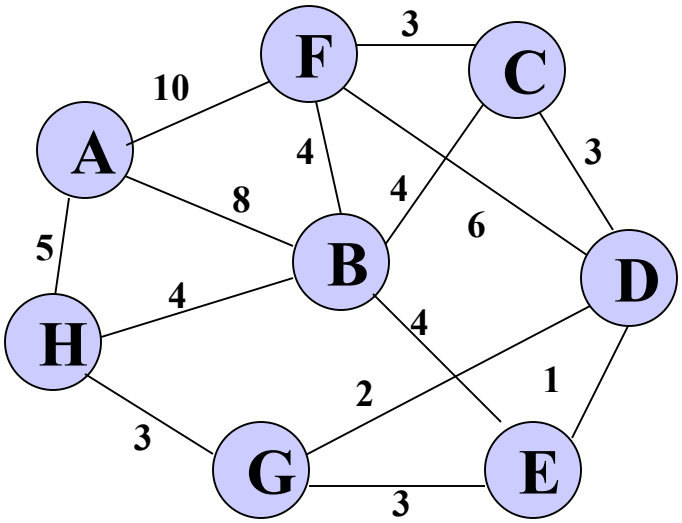
Kruskal's Algorithm

Work with edges, rather than nodes

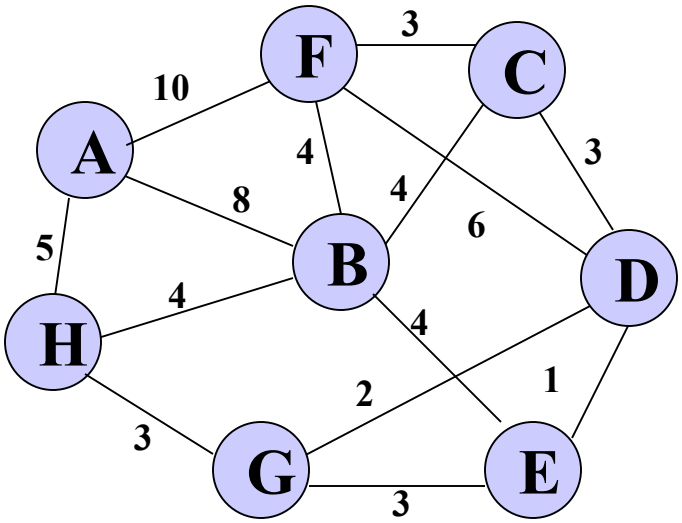
Two steps:

- Sort edges by increasing edge weight
- Select the first |V| – 1 edges that do not generate a cycle
- Select the next shortest edge which does not create a cycle
- Repeat step 2 until all vertices have been connected

Walk-Through



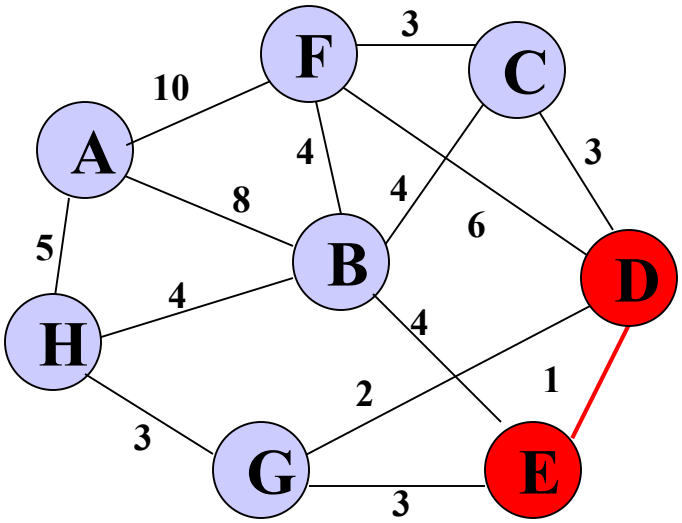
Consider an undirected, weight graph



Sort the edges by increasing edge weight

edge	d_v	
(D,E)	1	
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

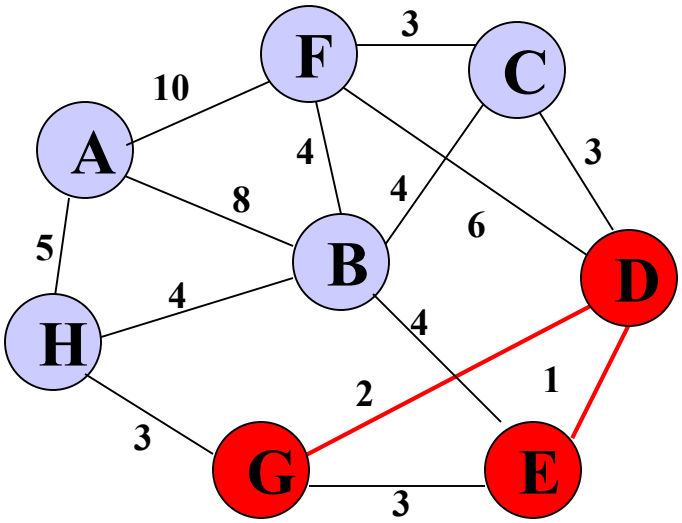
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

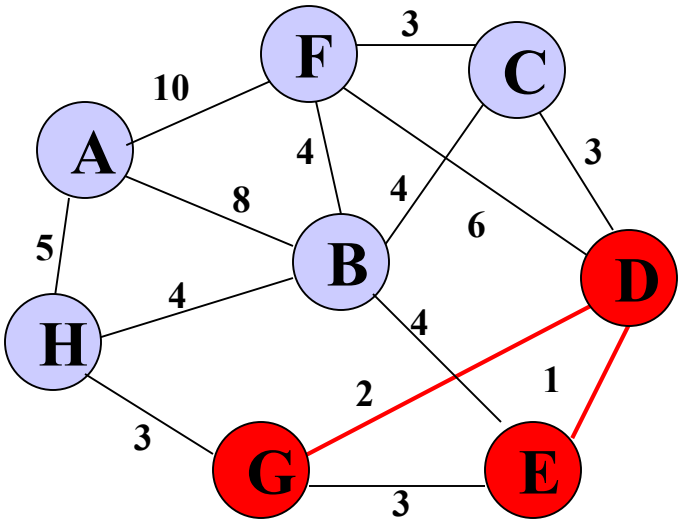
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

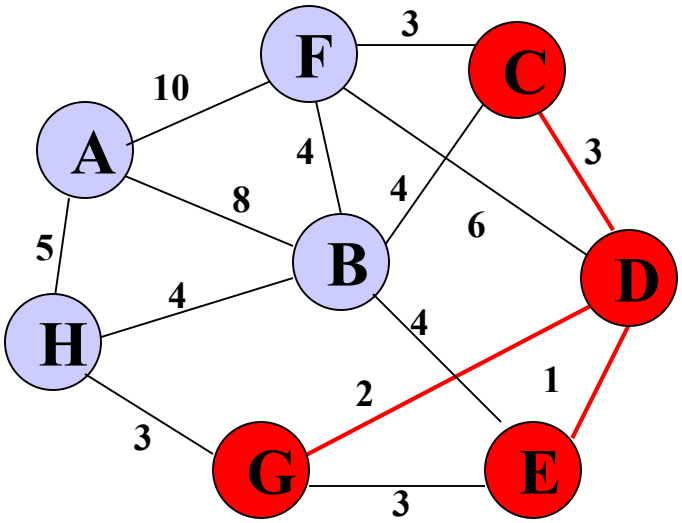


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

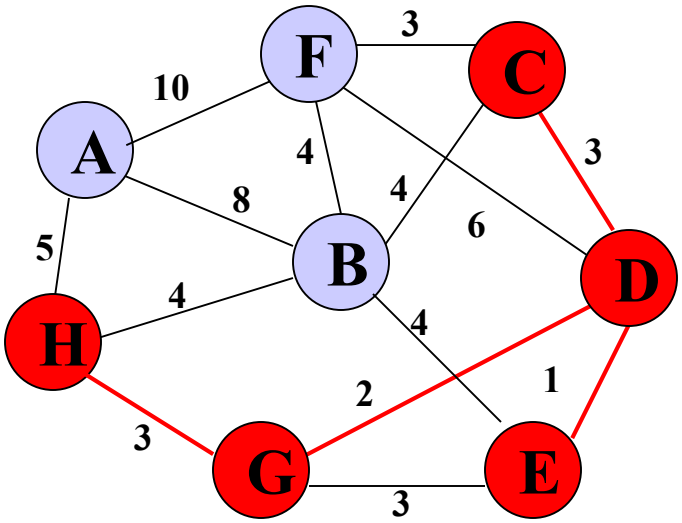
Accepting edge (E,G) would create a cycle



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	
(C,F)	3	
(B,C)	4	

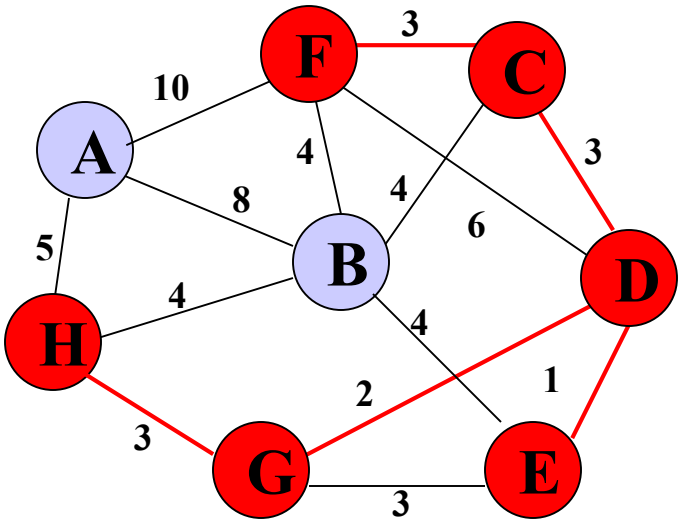
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	
(B,C)	4	

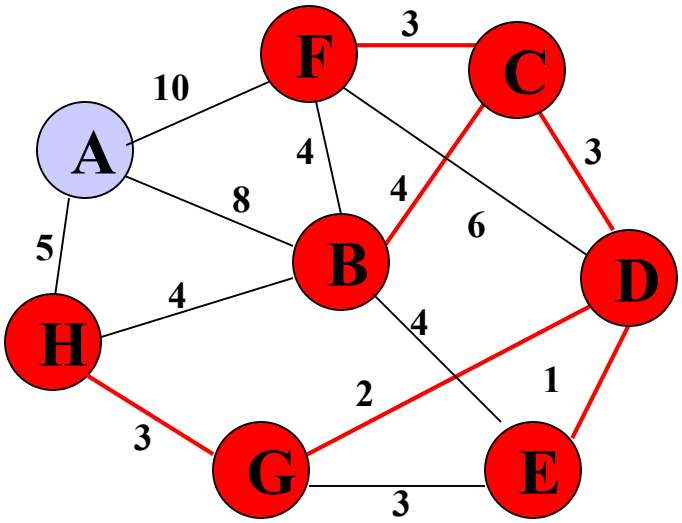
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	

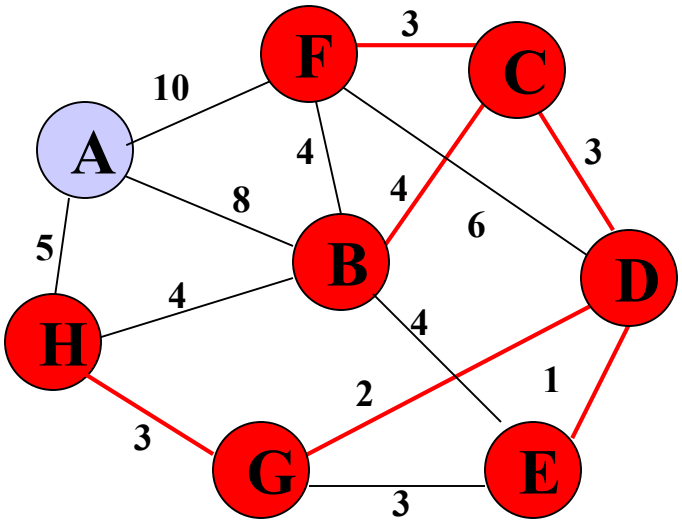
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

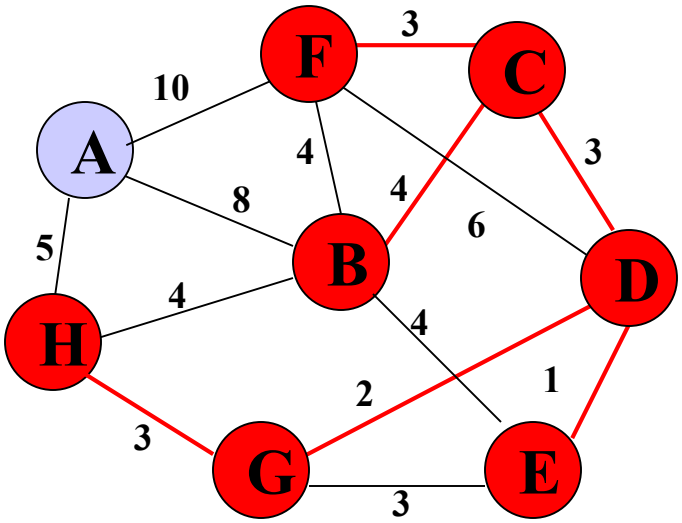
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

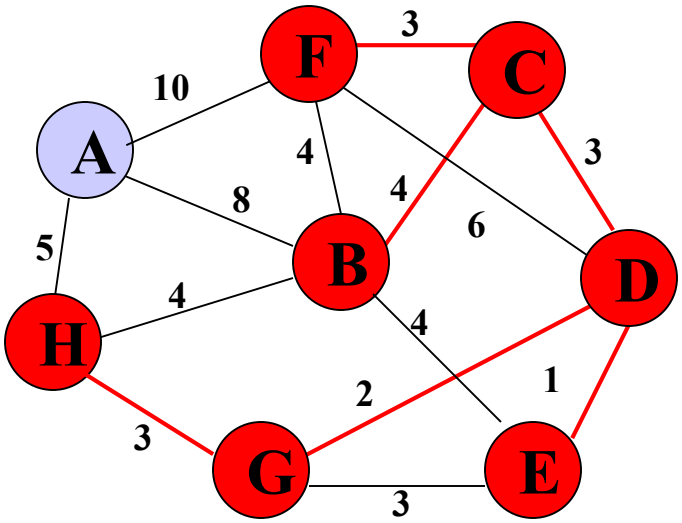
edge	d_v	
(B,E)	4	χ
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

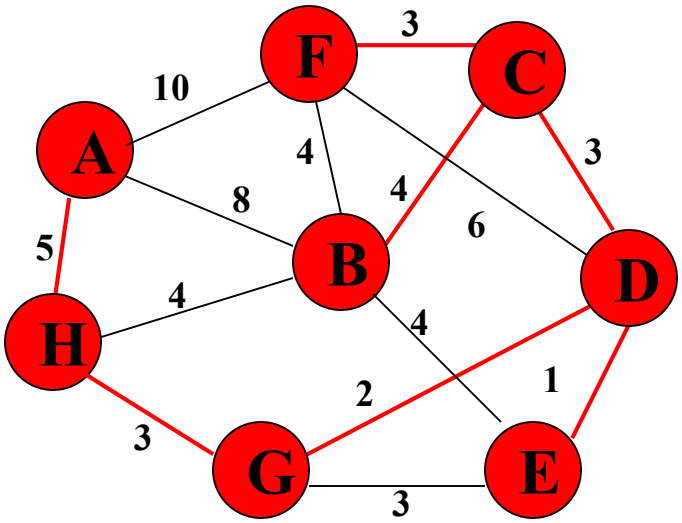
edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

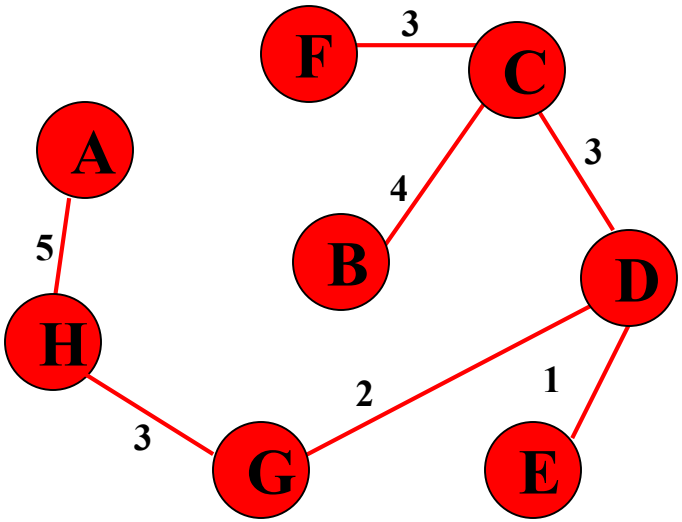
edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

} not considered

Done

Total Cost = Σ d_v = *21*

Some points to note

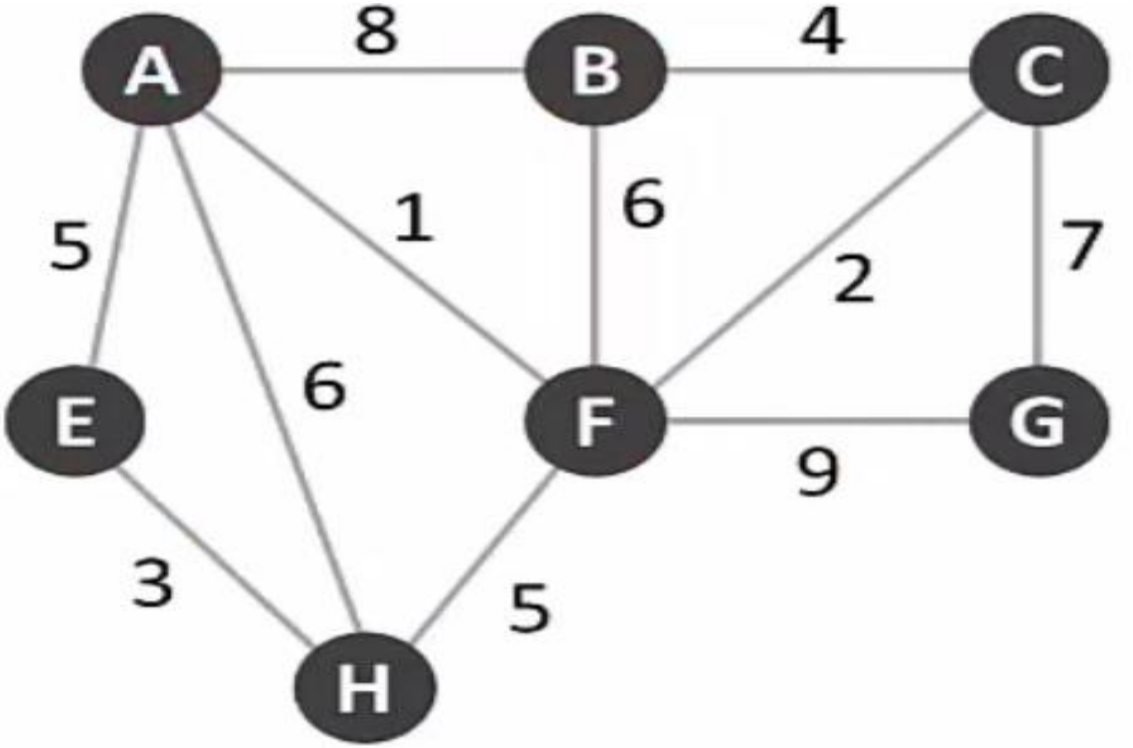
- Both algorithms will always give solutions with the same length.

- They will usually select edges in a different order – you must show this in your workings.

- Occasionally they will use different edges – this may happen when you have to choose between edges with the same length. In this case there is more than one minimum connector for the network.

Prim's and Kruskal's for Practice

- Using both Prim's and Kruskals algorithm, to determine the minimum spanning tree for each of the algorithms mentioned.



The End