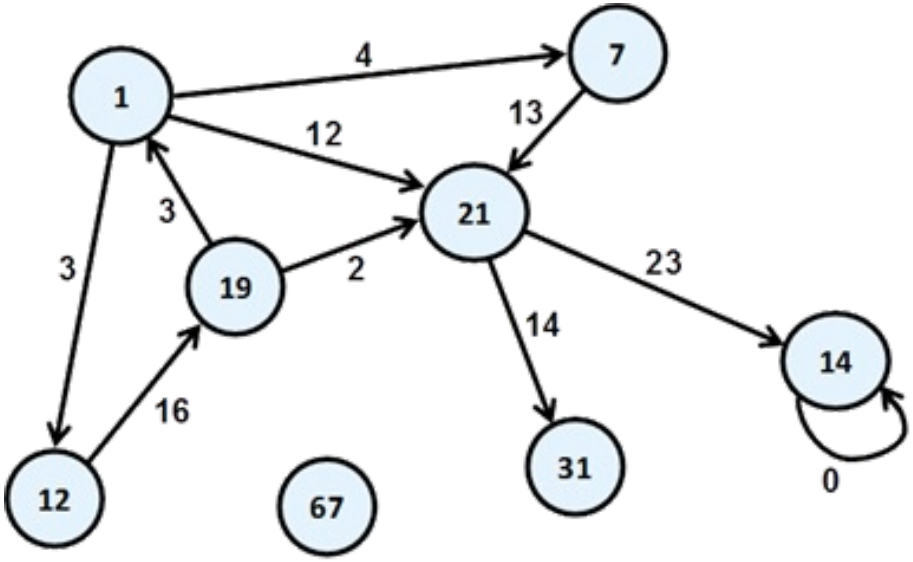
Graph Algorithms 1



NYIT CSCI-651

Graphs and Graph Theory

- Graphs are a really powerful formalism to model the *relationships* between *things*
  - Things can be whatever you want.
  - Relationships can be any relation that makes sense on those things

Graphs and Applications

- Examples:
  - Cities, highways
  - Intersections, streets
  - Airports, flights
  - Webpages, links
  - Facebook ids, friends
  - Computer, network connections
  - Task scheduling in OS
  - etc.

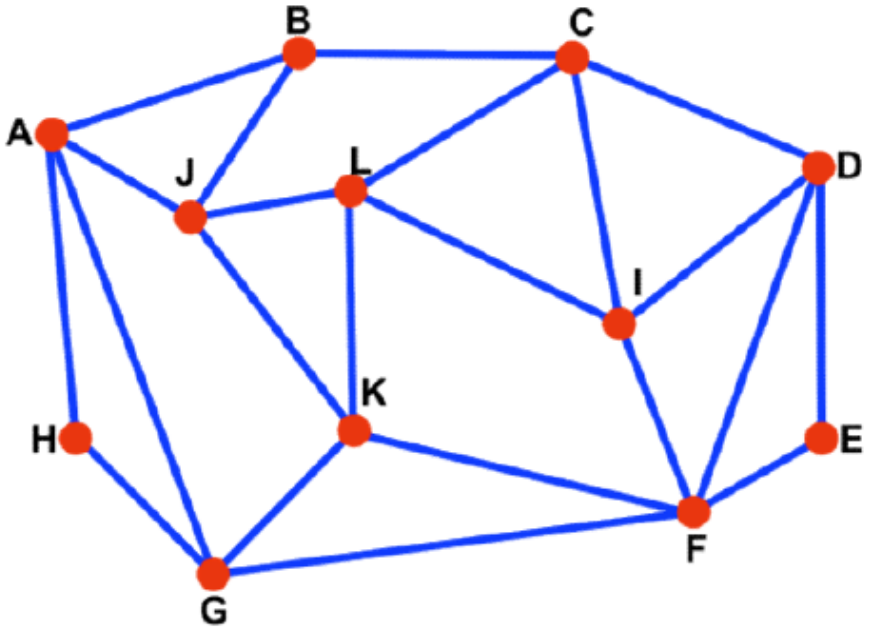
Drawing Graphs

- We draw graphs with circles or dots for the things (called vertices, or nodes) and lines or arrows for the relationships (called edges, or arcs).
  - Vertices usually labeled.
  - Edges can be labeled, too.

Graph Abstract Data Type (ADT)

- Graphs are a formalism, useful for representing relationships between things
  - a graph G is represented as G = (V, E)
    - V is a set of vertices: $\{v_1, v_2, \ldots, v_n\}$
    - E is a set of edges: $\{e_1, e_2, \ldots, e_m\}$ where each e_i connects two vertices (v_{i1}, v_{i2})
      - a set $E \subseteq V \times V$ of ***edges***.

Example



G = <**V**,**E**>

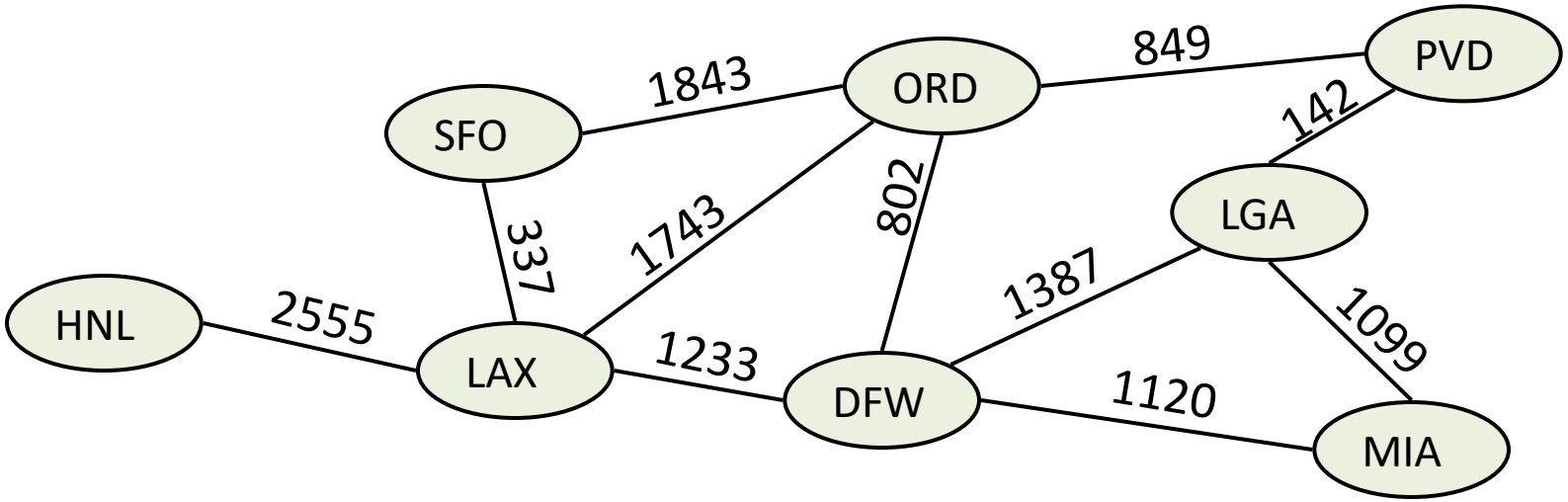
V = vertices =
{A,B,C,D,E,F,G,H,I,J,K,L}

E = edges =
{(A,B),(B,C),(C,D),(D,E),(E,F),
(F,G),(G,H),(H,A),(A,J),(A,G),
(B,J),(K,F),(C,L),(C,I),(D,I),
(D,F),(F,I),(G,K),(J,L),(J,K),
(K,L),(L,I)}

Graphs

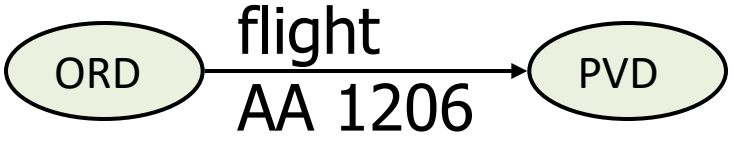
- A graph *G* is a pair (***V, E***), where ***V*** is a set of nodes, called vertices, and ***E*** is a collection of pairs of vertices, called edges. Vertices and edges can store elements

- Example: A vertex represents an airport and stores the three-letter airport code. An edge represents a flight route between two airports and stores the mileage of the route



Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., flight network
- Undirected graph
  - all the edges are undirected
  - e.g., route network



Graph ADT

- Operations might include:
  - creation (with a certain number of vertices)
  - inserting/removing edges
  - iterating over vertices adjacent to a specific vertex
  - asking whether an edge exists connecting two vertices

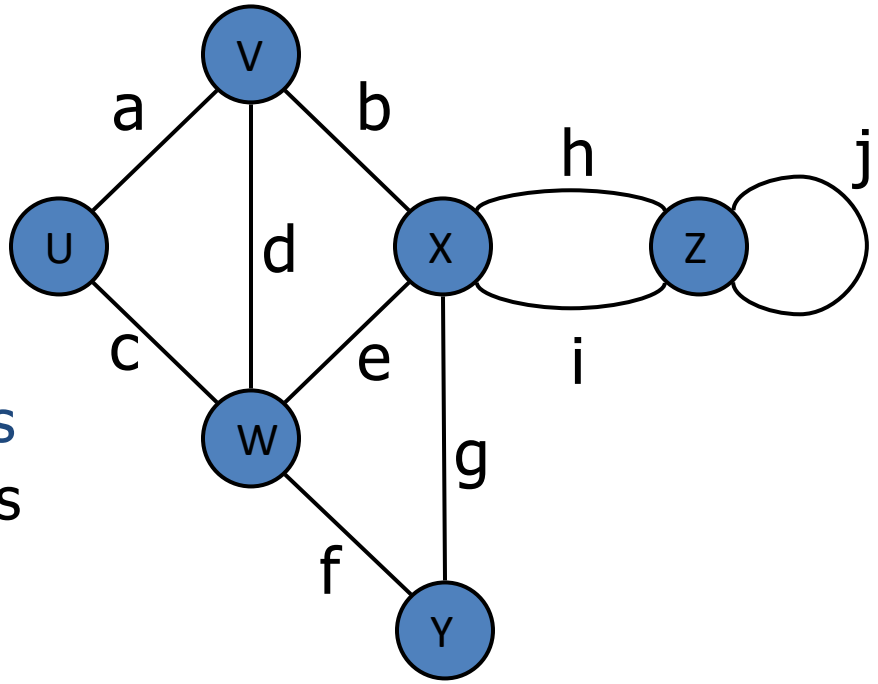
Graph Representation

- 2-D matrix of vertices
  (marking edges in the cells)
  - "adjacency matrix"

- List of vertices each with a list of adjacent vertices
  - "adjacency list"

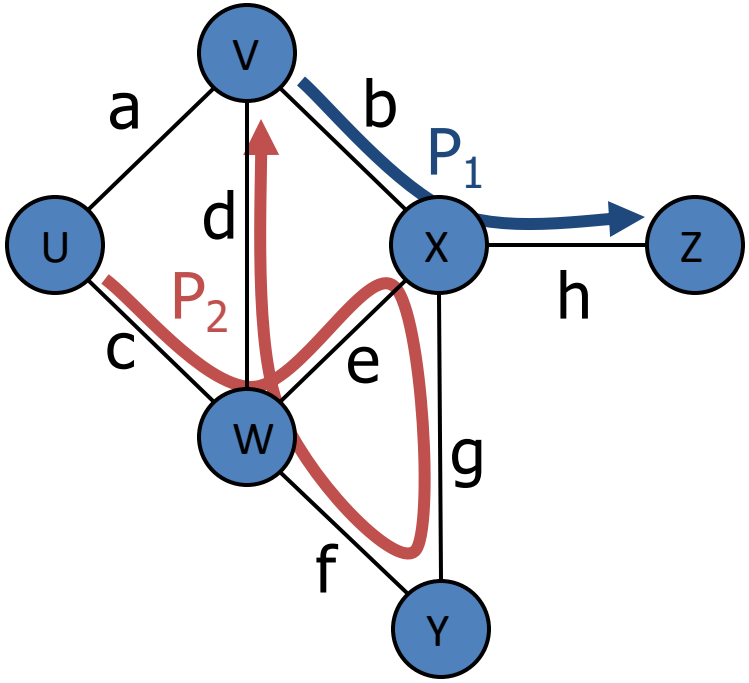
Terminology

- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges and self-loops
  - h and i are parallel edges
  - j is a self-loop



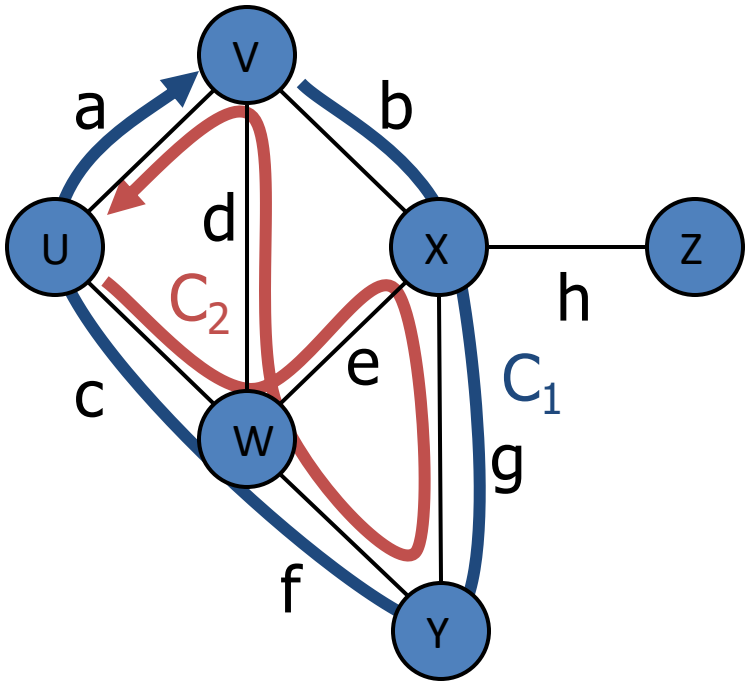
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



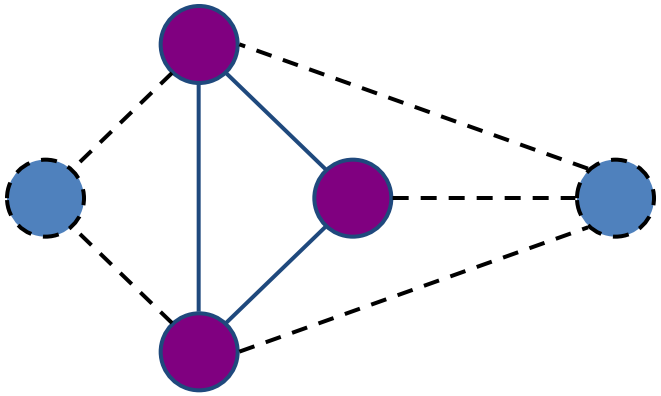
Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple

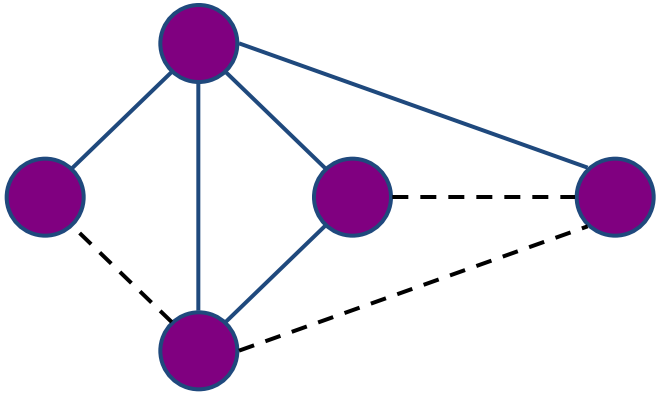


Subgraphs

- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



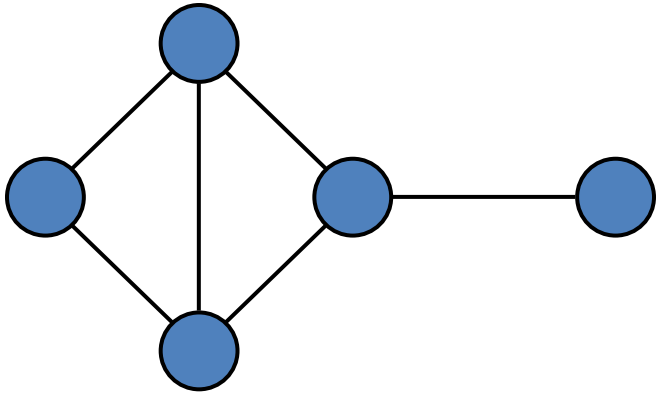
Subgraph



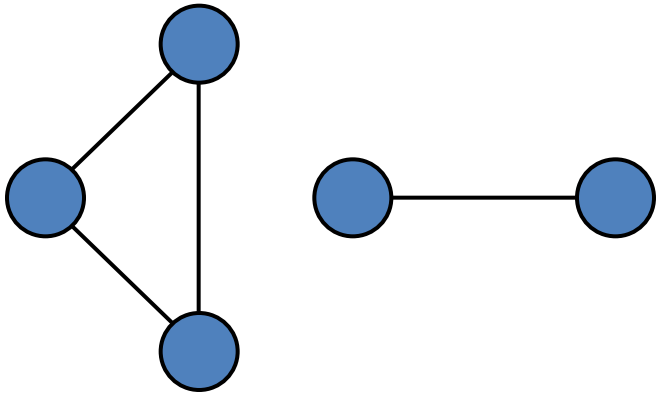
Spanning subgraph

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G



Connected graph

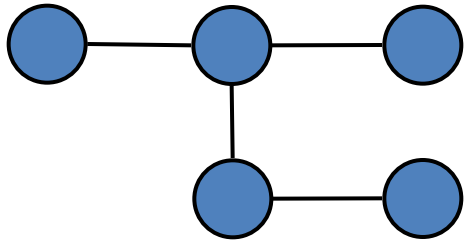


Non connected graph with two connected components

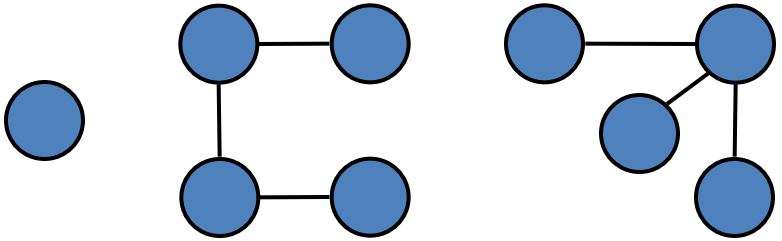
Trees and Forests

- A tree is an undirected graph T such that
  - T is connected
  - T has no cycles

  This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



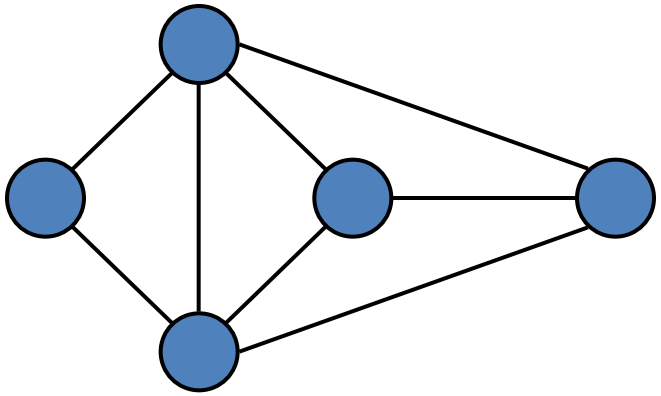
Tree



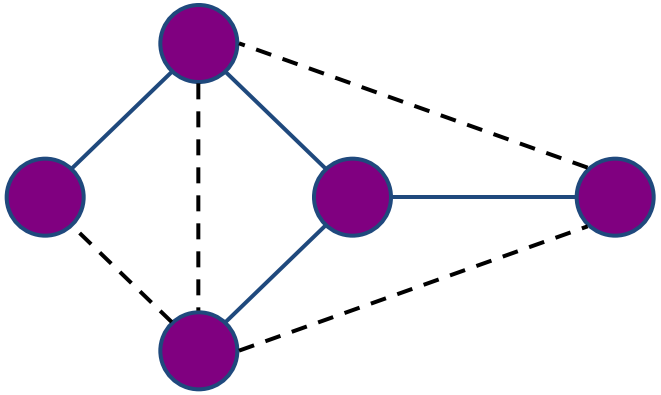
Forest

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

Properties

Property 1

$\sum_v \deg(v) = 2m$

Proof: each edge is counted twice

Property 2

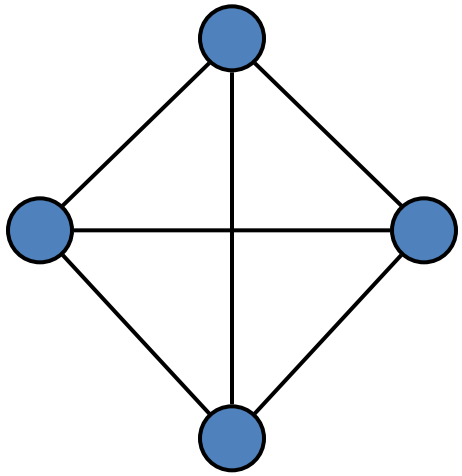
In an undirected graph with no self-loops and no parallel edges

$m \leq n\,(n-1)/2$

Proof: $(n-1) + (n-2) + \ldots + 1$

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v

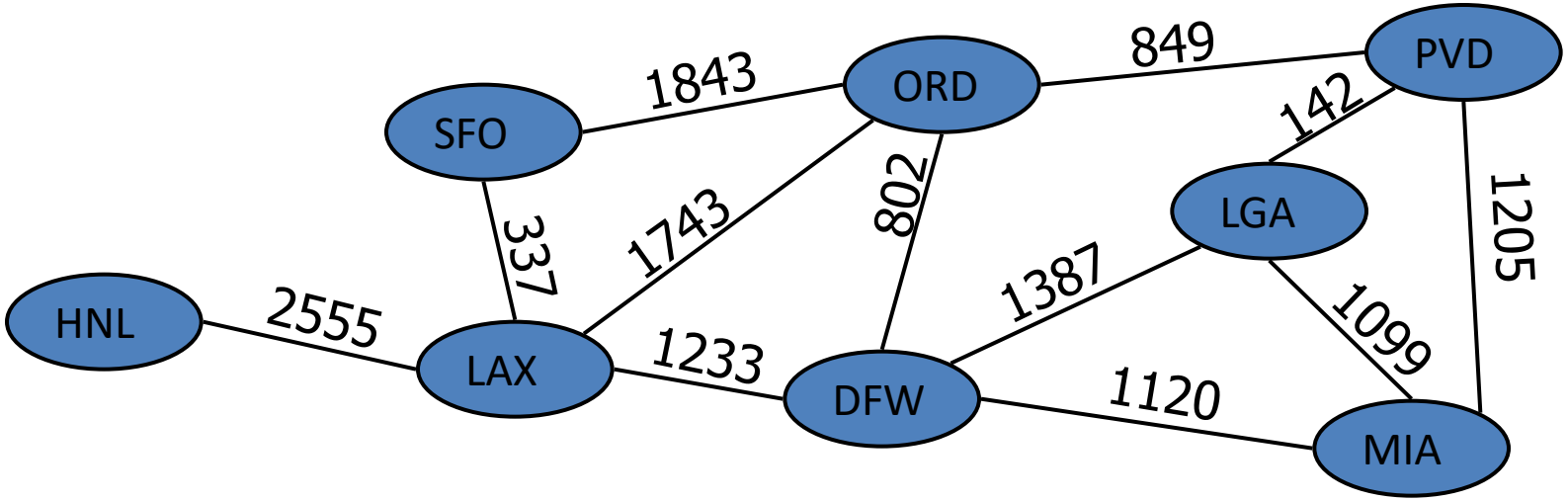


Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

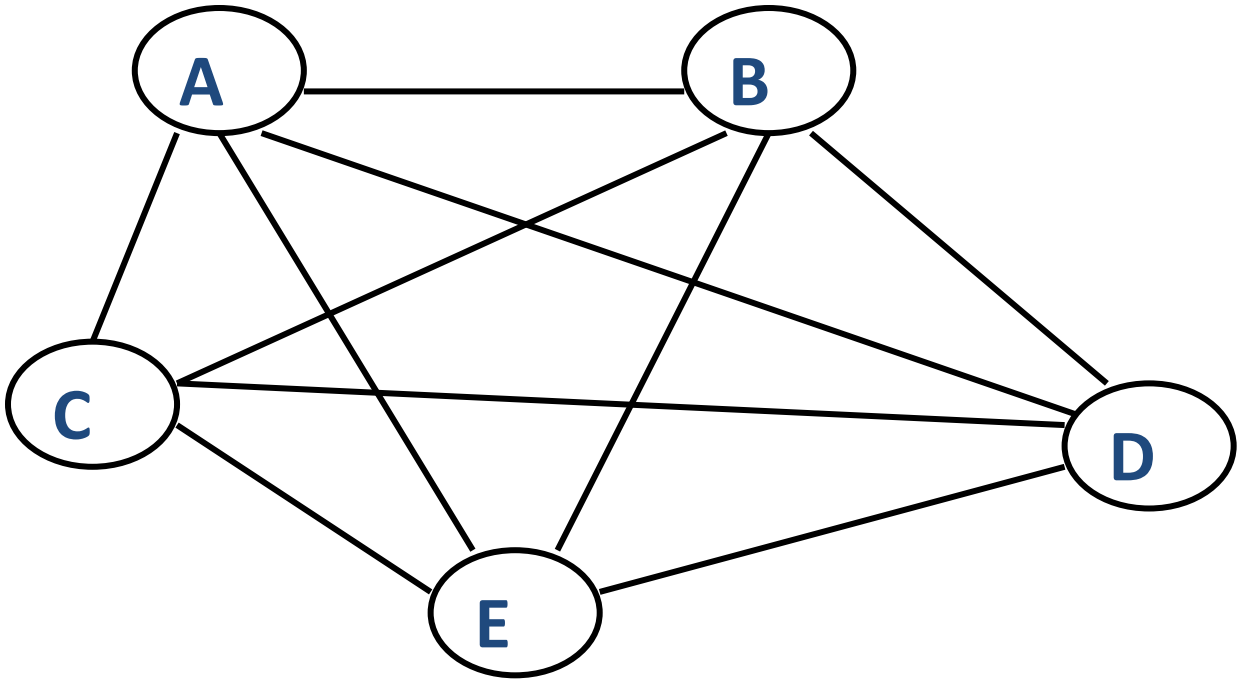
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Complete graphs

Graphs with all edges present



A complete graph

Dense graphs: relatively few of the possible edges are missing

Sparse graphs: relatively few of the possible edges are present

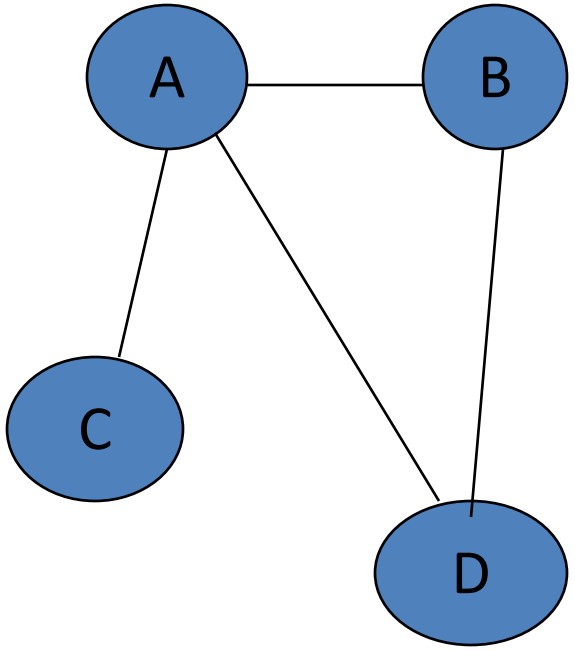
Adjacency matrix – undirected graphs

Vertices: A,B,C,D

Edges: AC, AB, AD, BD

The matrix is symmetrical

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

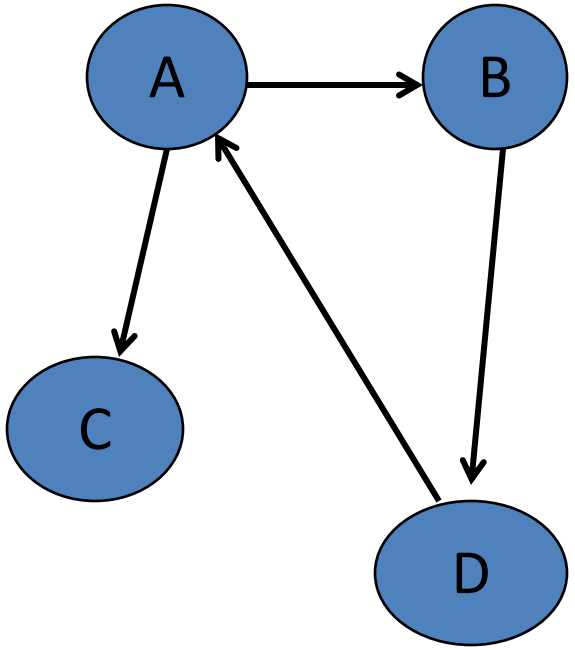


Adjacency matrix – directed graphs

Vertices: A,B,C,D

Edges: AC, AB, BD, DA

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



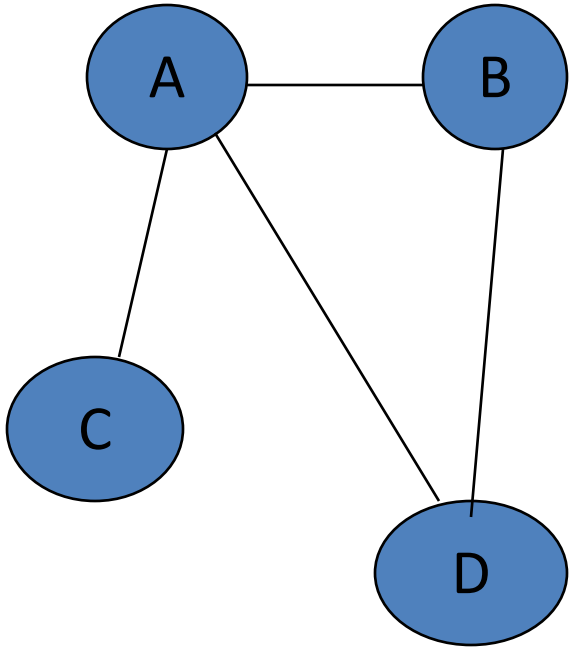
Adjacency lists – undirected graphs

Vertices: A,B,C,D

Edges: AC, AB, AD, BD

Heads lists

A	B C D
B	A D
C	A
D	A B

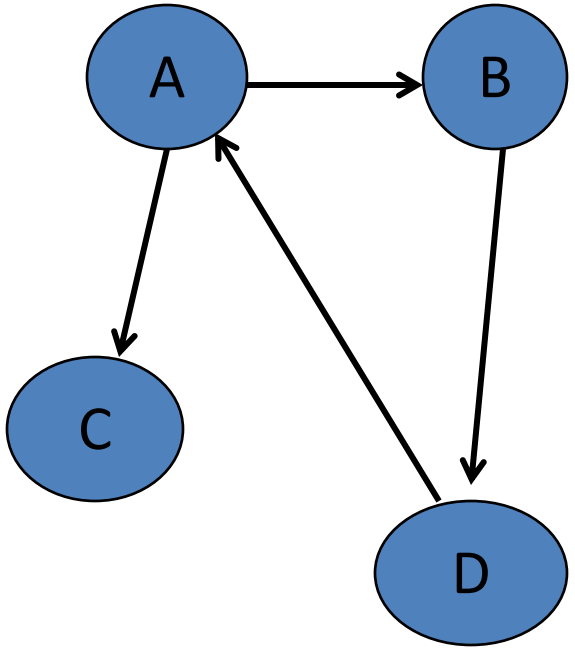


Adjacency lists – directed graphs

Vertices: A,B,C,D

Edges: AC, AB, BD, DA

Heads	lists
A	B C
B	D
C	=
D	A



Graph Representation

- If the graph is sparse, then number of edges is much less than n^2
- If the graph is dense, then the number of edges is close to (the complete) $n(n-1)/2$, or to n^2 if the graph is directed with self-loops.
  - Then there is no advantage of using adjacency list over matrix.
- In terms of space complexity
  Adjacency matrix: $O(n^2)$
  Adjacency list: $O(n+m)$
  where n is the number nodes, m is the number of edges.

Graph Representation

- When the graph is **undirected tree** then
  Adjacency matrix: $O(n$^2)
  Adjacency list: $O(n+n)$ is $O(n)$ (better than n^2)

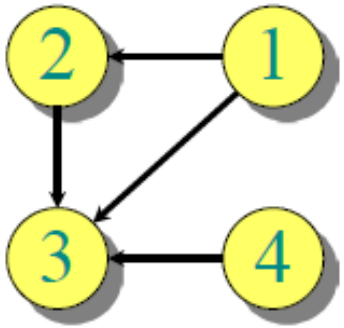
- When the graph is **directed, complete, with self-loops** then
  Adjacency matrix: $O(n$^2)
  Adjacency list: $O(n+n$^2) is $O(n$^2) (no difference)

- And finally, when you implement using matrix, checking if there is an edge between two nodes takes O**(1) times**, while with an adjacency list, it may take linear time in n.

Adjacency-matrix representation

The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \ldots n, 1 \ldots n]$ given by

$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in \mathrm{E}, \\ 0 & \text{if } (i,j) \notin \mathrm{E}. \end{cases}$$



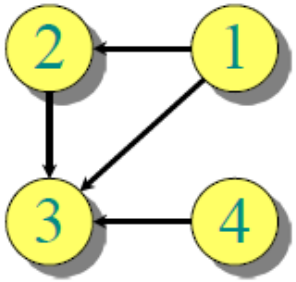
A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

$\Theta(V^2)$ storage

$\Rightarrow$ ***dense*** representation.

Adjacency-list representation

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



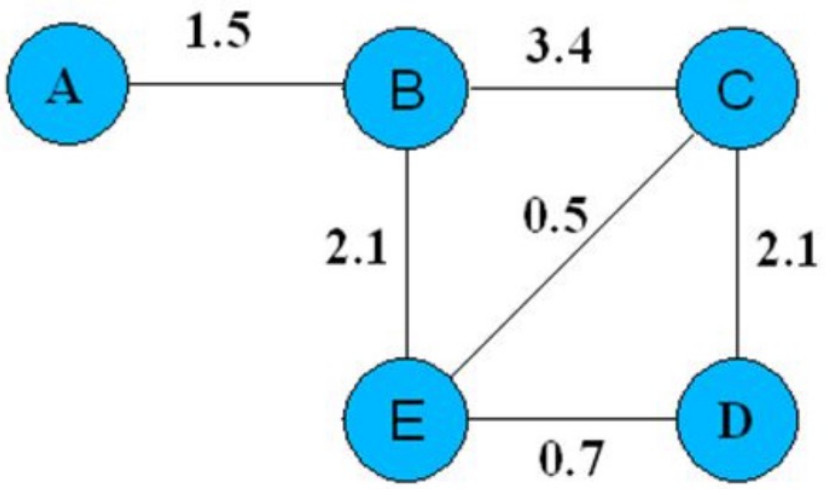
$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

Handshaking Lemma: $\sum_{v \in V} degree(v) = 2|E|$ for undirected graphs $\Rightarrow$ adjacency lists use $\Theta(V + E)$ storage — a ***sparse*** representation.

Weighted Graphs

- How can we store weights in adjacency matrix



	A	B	C	D	E
A	0	1.5	0	0	0
B	1.5	0	3.4	0	0
C	0	3.4	0	2.1	0.5
D	0	0	2.1	0	0.7
E	0	2.1	0.5	0.7	0

Graph Density

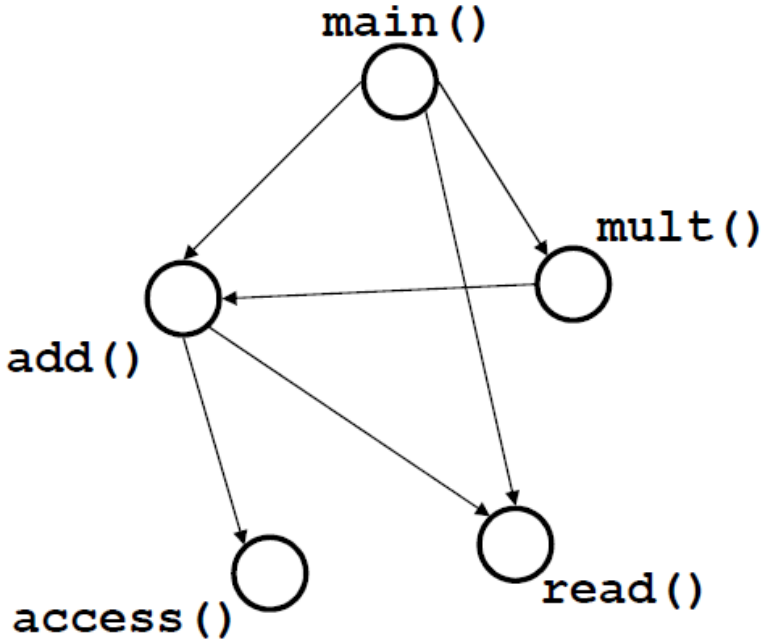
- A *sparse* graph has O(|V|) edges
  - Why is the adjacency list likely to be a better representation than the adjacency matrix for sparse graphs?

- A *dense* graph has $O(|V|^2)$ edges

Anything in between is either on the sparse side or on the dense side, depending critically on context!

Directed Acyclic Graphs (DAGs)

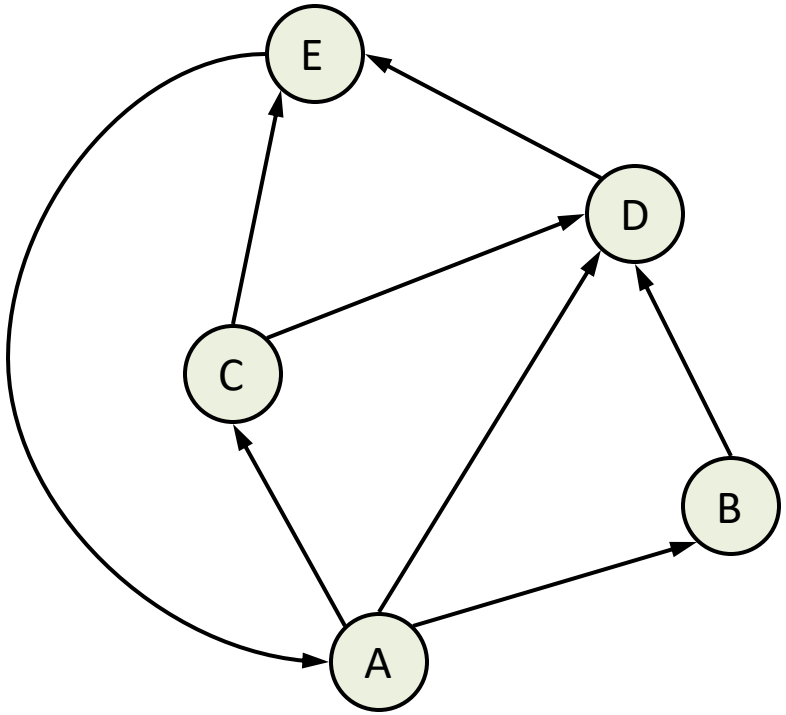
- DAGs are directed graphs with no (directed) cycles.



Trees ⊂ DAGs ⊂ Graphs

Directed Graphs (Digraphs)

- A digraph is a graph whose edges are all directed
  - Short for "directed graph"
- Applications
  - one-way streets
  - flights
  - task scheduling



DAG and Topological Sort

- Directed acyclic graph is a partially ordered set
- A topological sort is the process of sorting items over which a partial order is defined.
- **Partial order** is a relation on a set A that is reflexive, transitive and antisymmetric.
- **Total order** is a partial order that has one additional property - any two elements in the set should be related.

DAGs and Topological Ordering

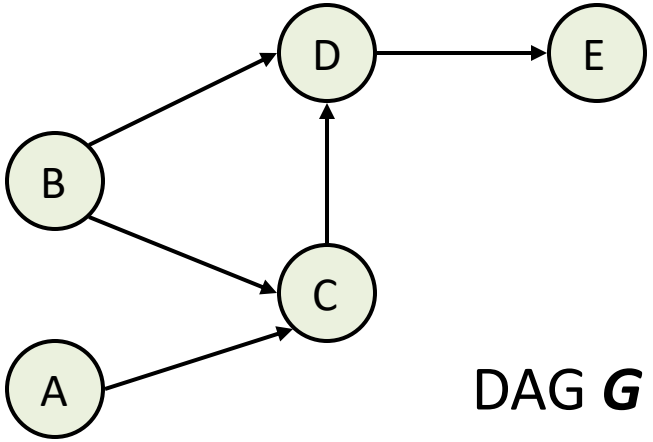
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

  $v_1, \ldots, v_n$

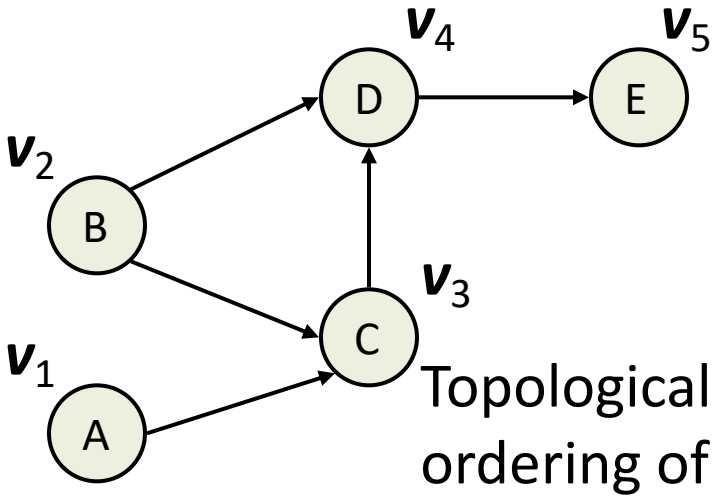
  of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

Theorem

A digraph admits a topological ordering if and only if it is a DAG



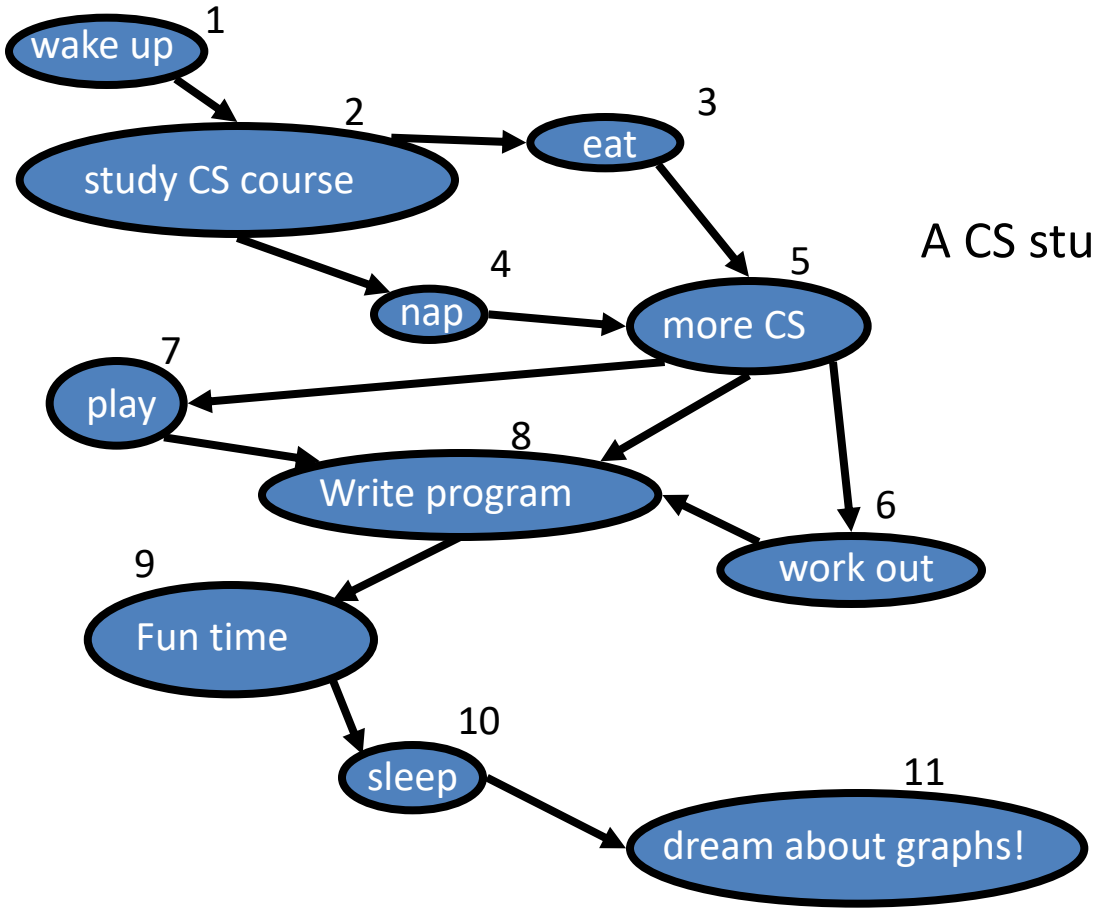
DAG ***G***



Topological ordering of ***G***

Topological Sorting

- Number vertices, so that (u,v) in E implies u < v



A CS student day

Topological Sort

RULE: If there is a path from ***u*** to ***v***, then ***v*** appears after ***u*** in the ordering.

Graphs: directed, acyclic (DAG)

OutDegree of a vertex **U**: the number of **outgoing** edges
Indegree of a vertex **U**: the number of **incoming** edges

The algorithm for topological sort uses "indegrees" of vertices.
Implementation: with a queue

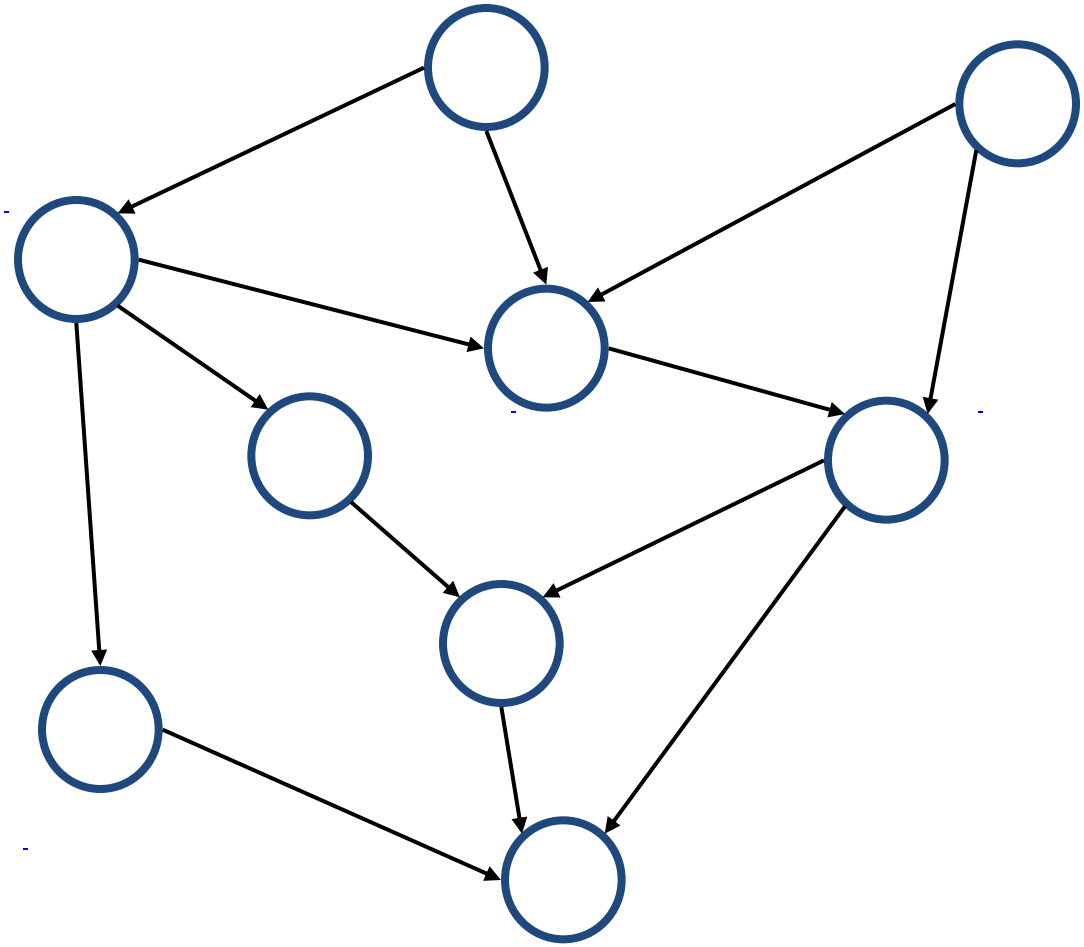
Complexity: **Adjacency lists: O(|E| + |V|),**
Matrix representation: O(|V|²)

Algorithm for Topological Sorting

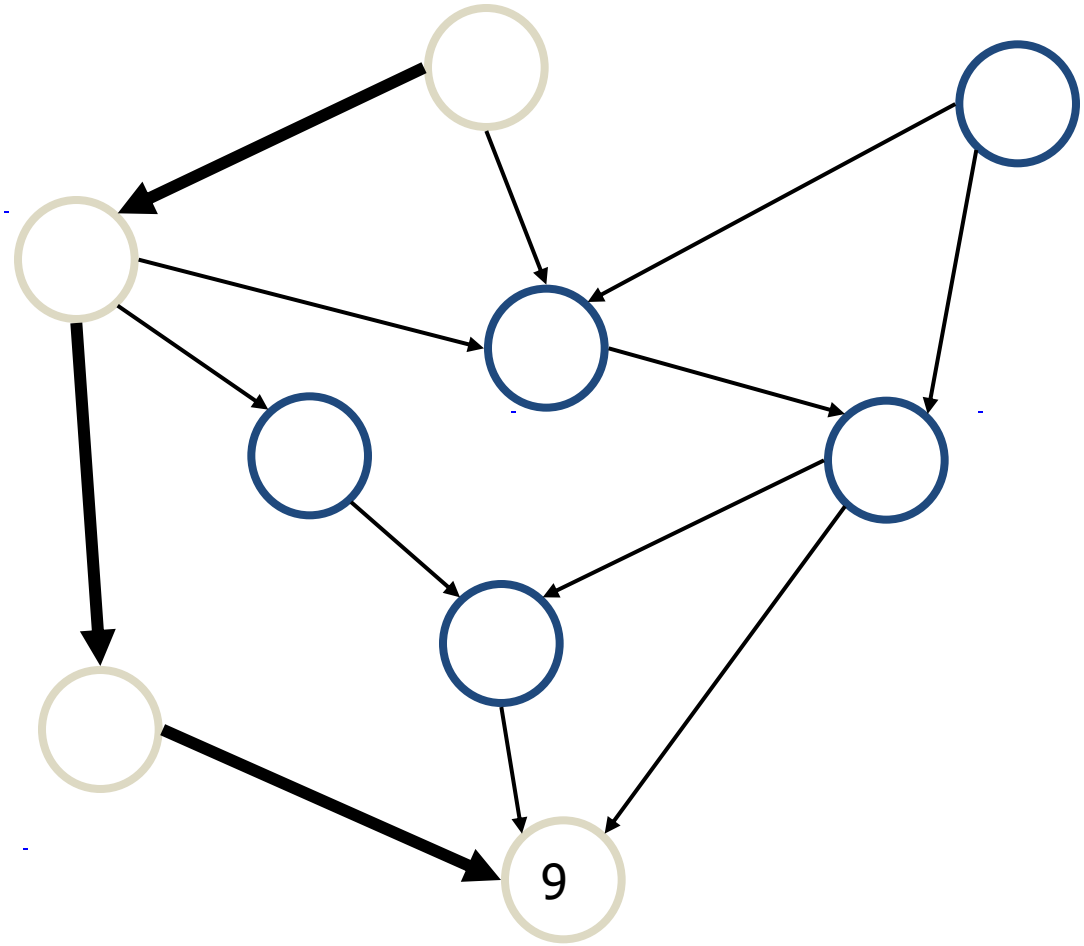
```
Algorithm TopologicalSort(G)
    H ← G    // Temporary copy of G
    n ← G.numVertices()
    while H is not empty do
     Let v be a vertex with no outgoing edges
     Label v ← n
     n ← n – 1
     Remove v from H
```

- Running time: O(n + m)

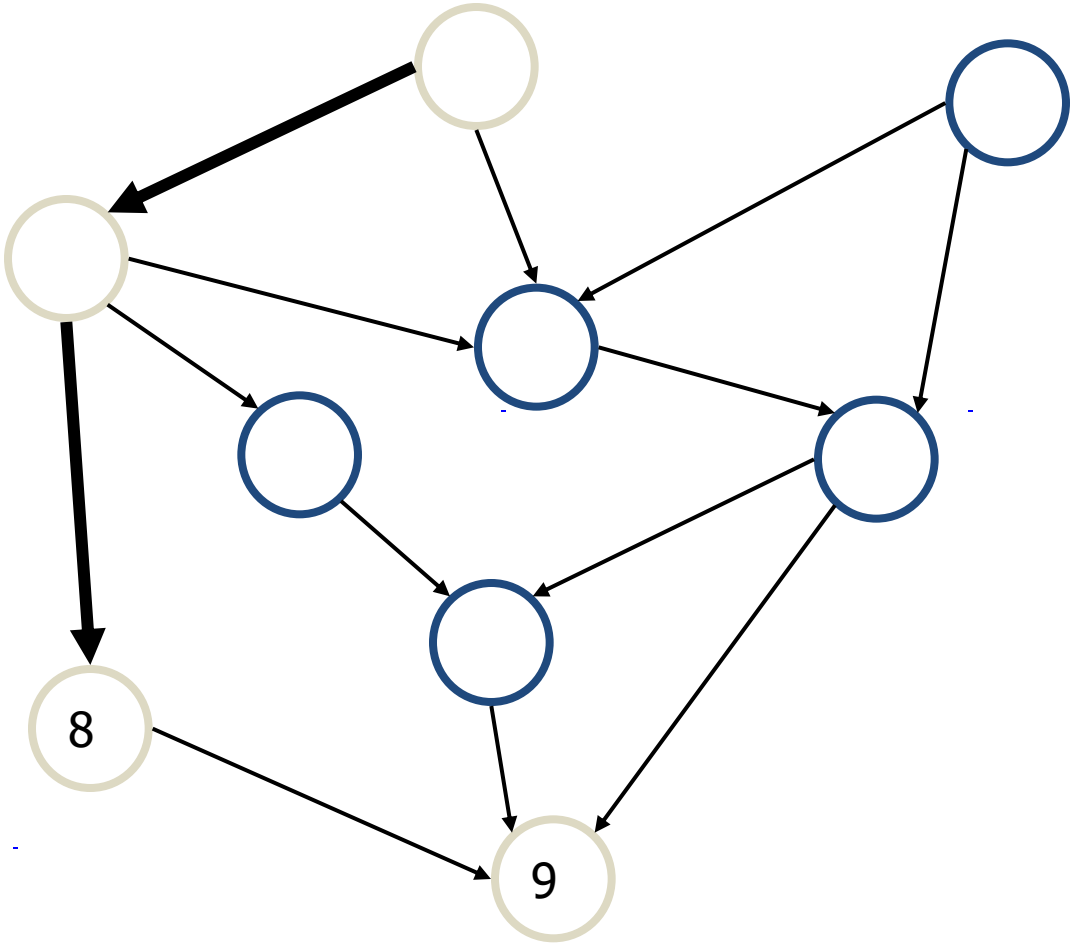
Topological Sorting Example



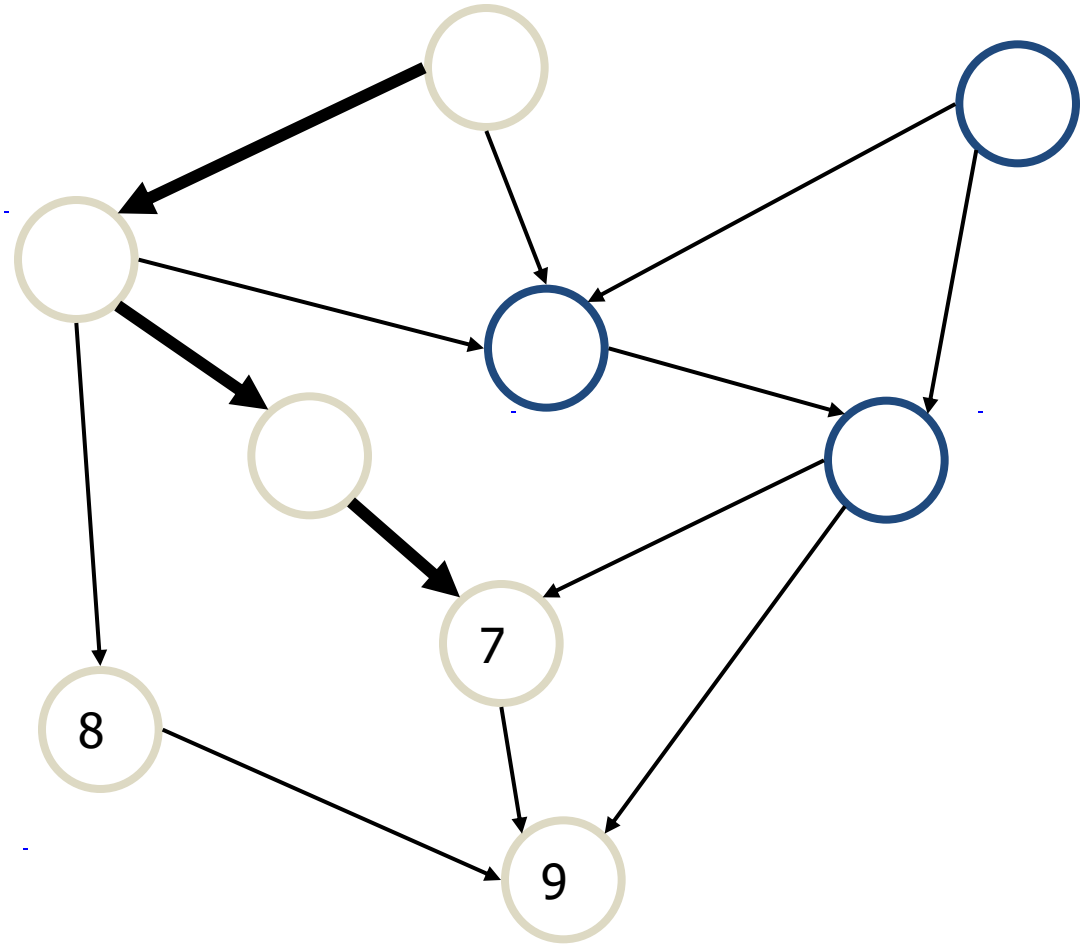
Topological Sorting Example



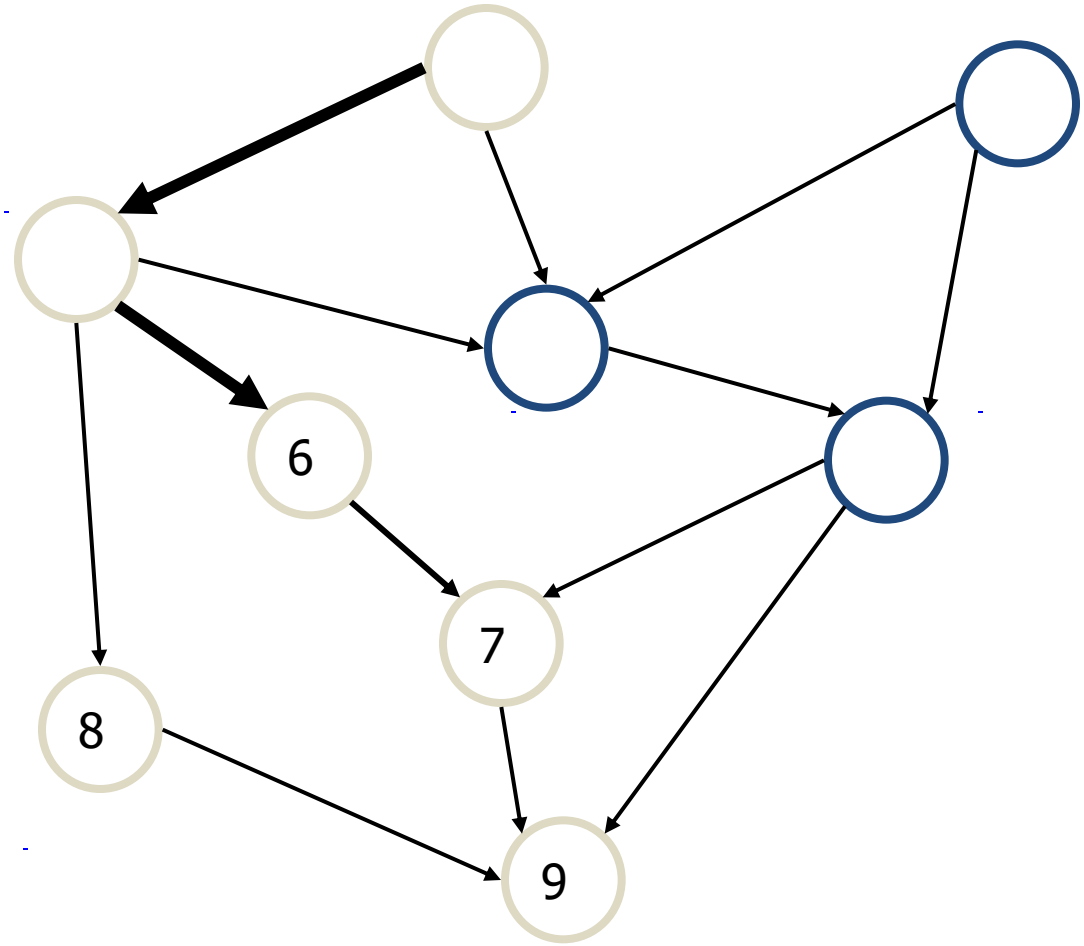
Topological Sorting Example



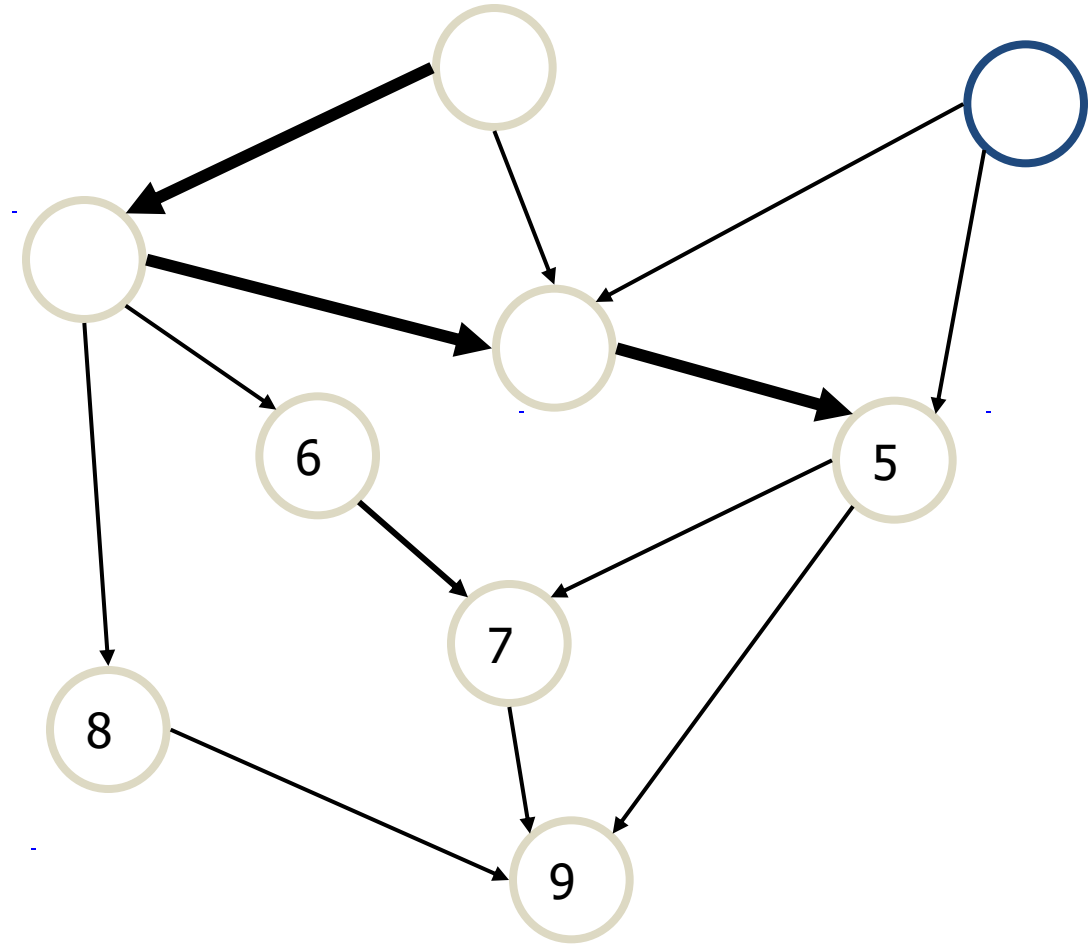
Topological Sorting Example



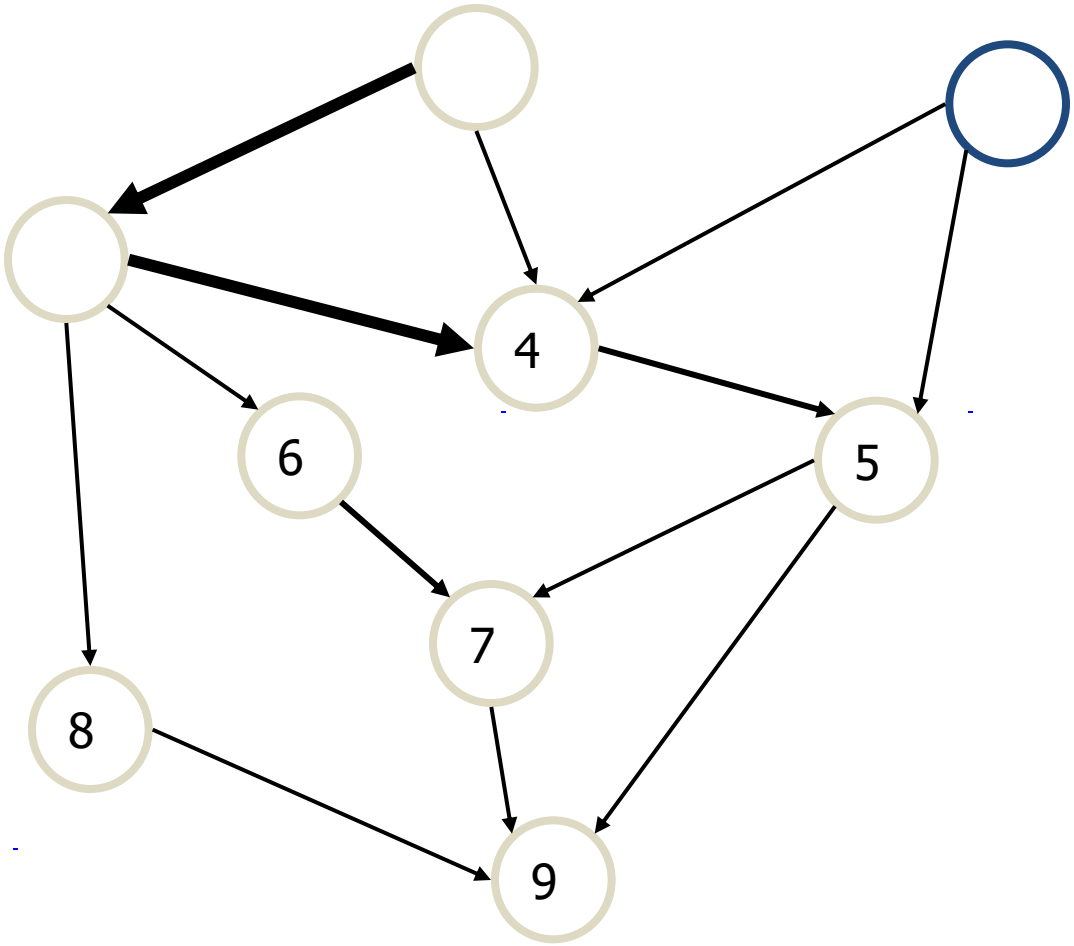
Topological Sorting Example



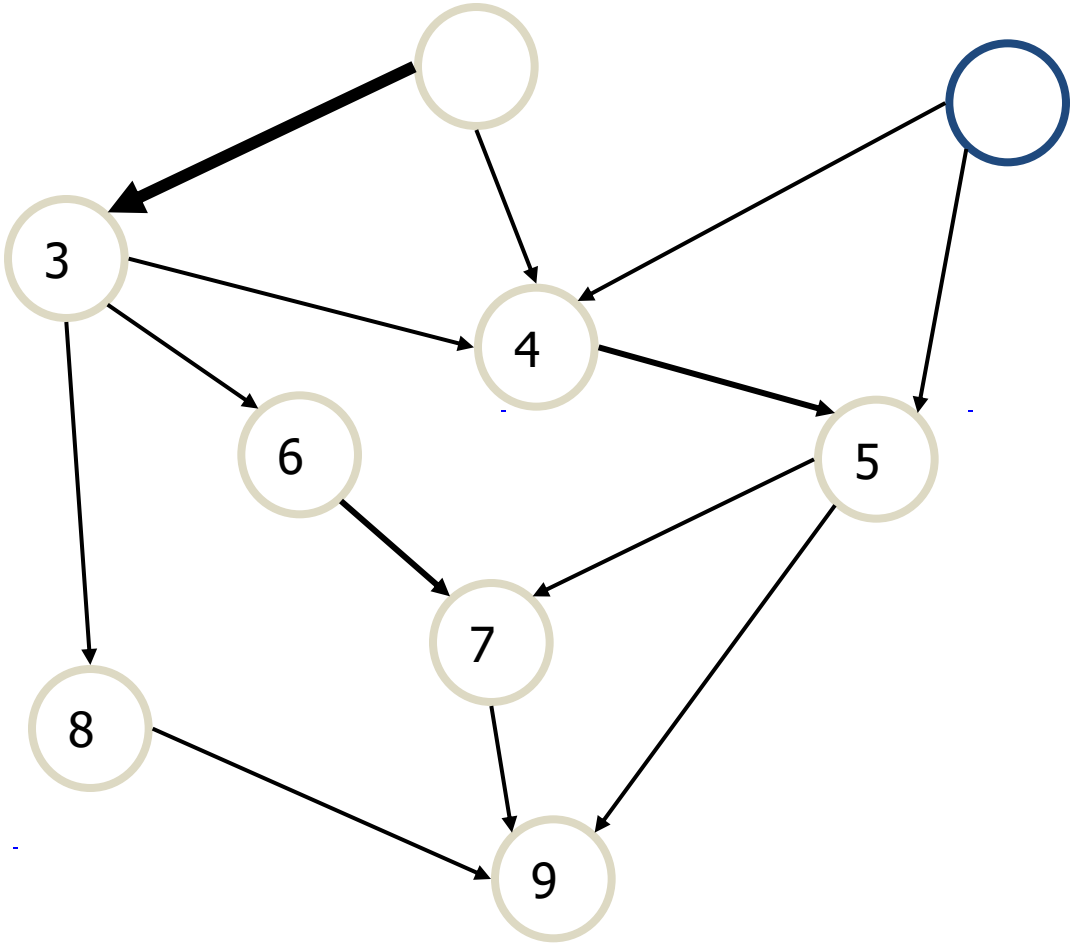
Topological Sorting Example



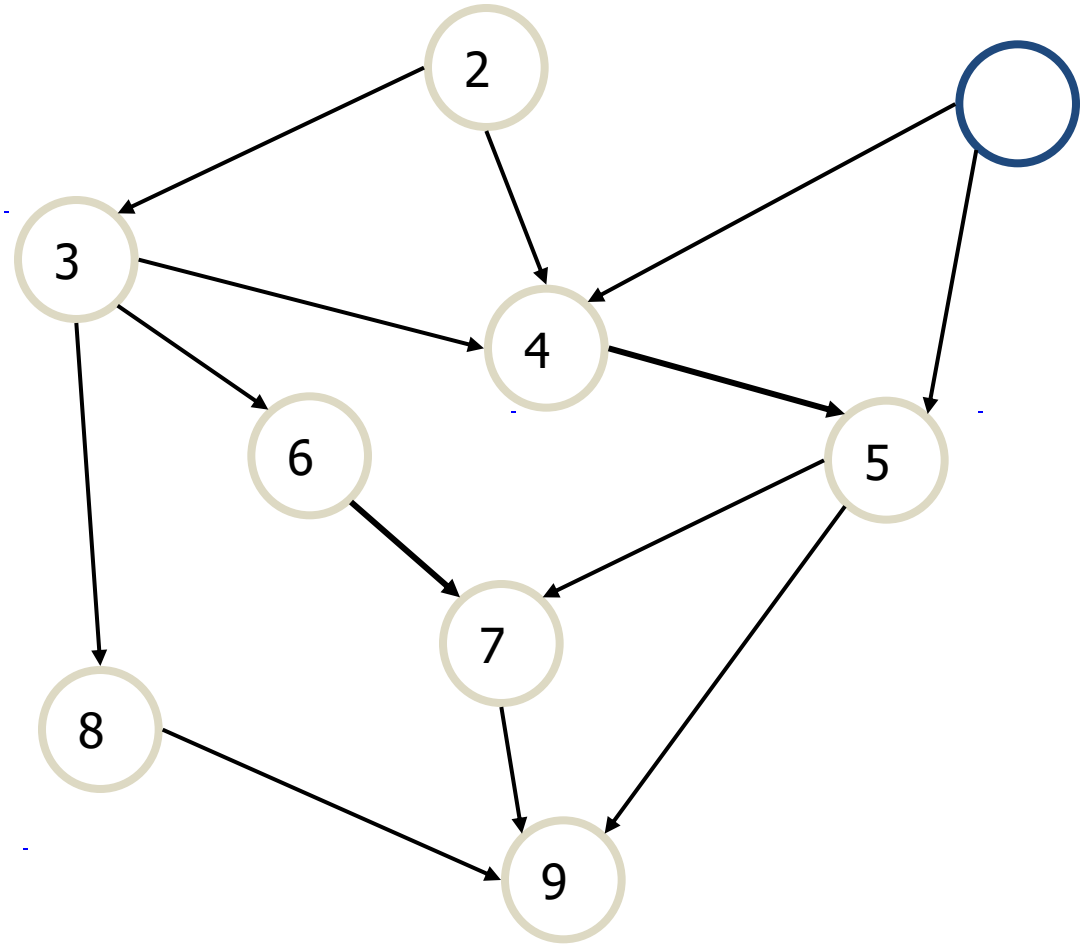
Topological Sorting Example



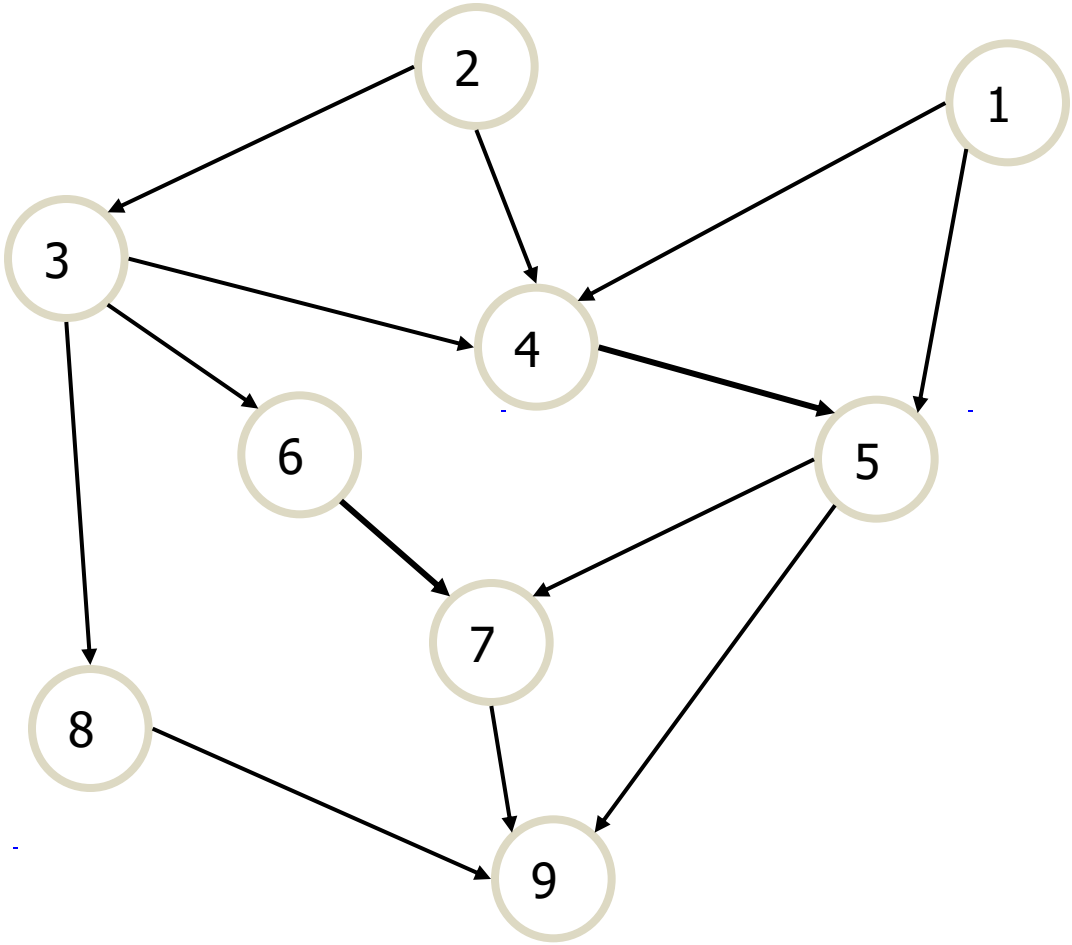
Topological Sorting Example



Topological Sorting Example



Topological Sorting Example



Minimum Spanning tree

Spanning Tree

- A spanning tree of a connected weighted undirected graph G 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.

Minimum Spanning Trees

Spanning subgraph

- Subgraph of a graph G containing all the vertices of G

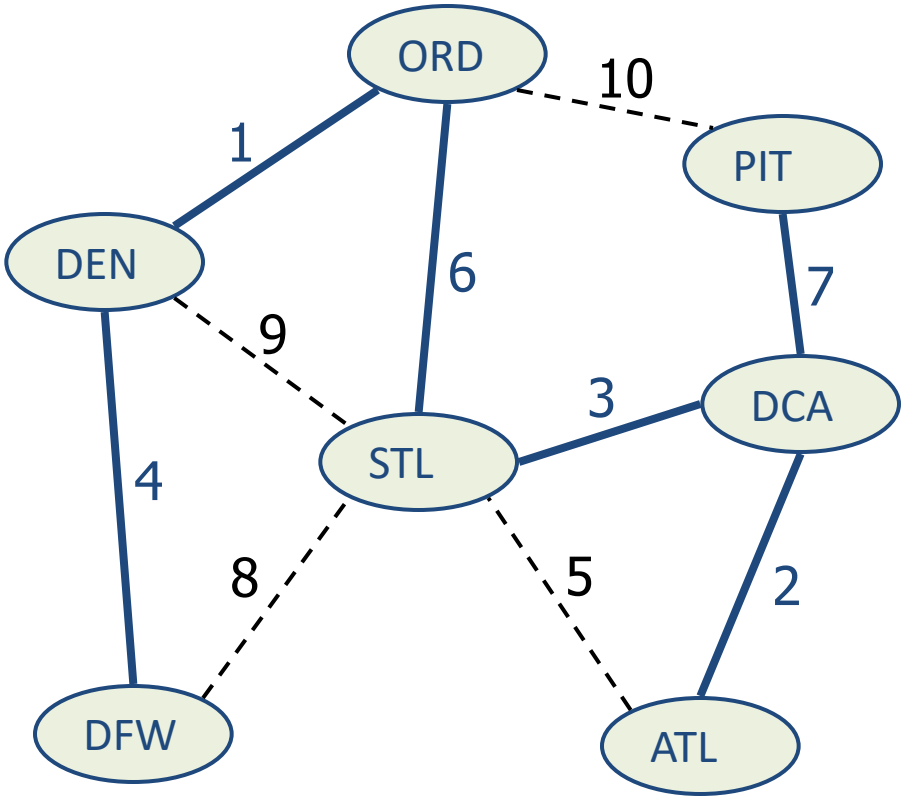
Spanning tree

- Spanning subgraph that is itself a (free) tree

Minimum spanning tree (MST)

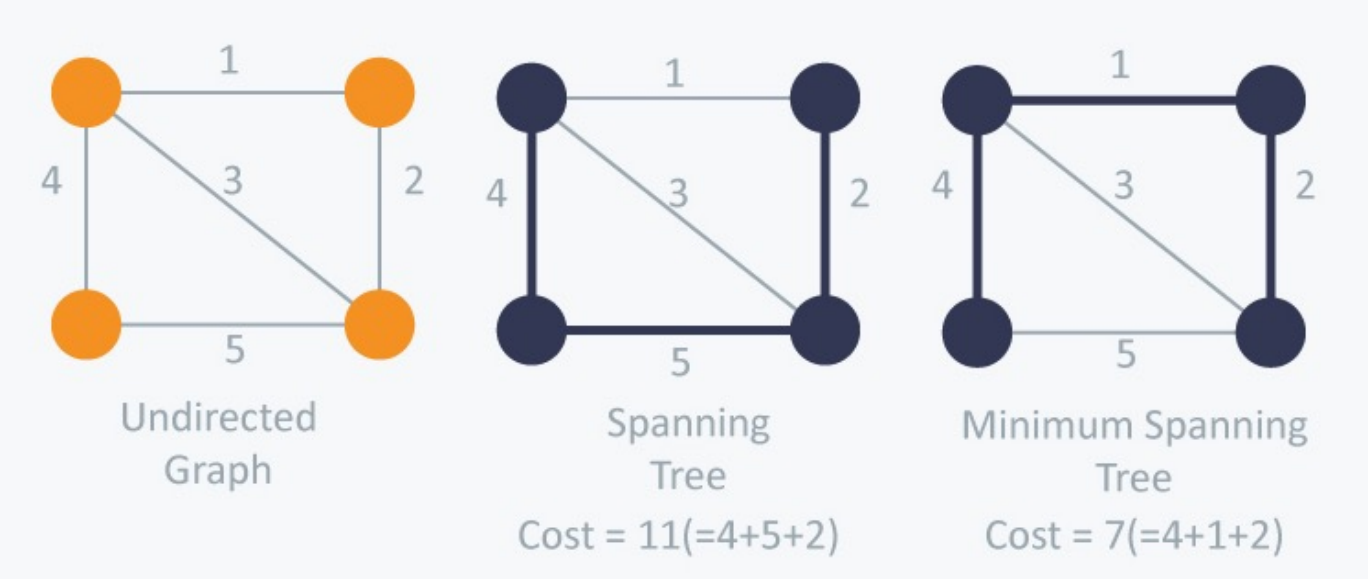
- Spanning tree of a weighted graph with minimum total edge weight

- Applications
  - Communications networks
  - Transportation networks

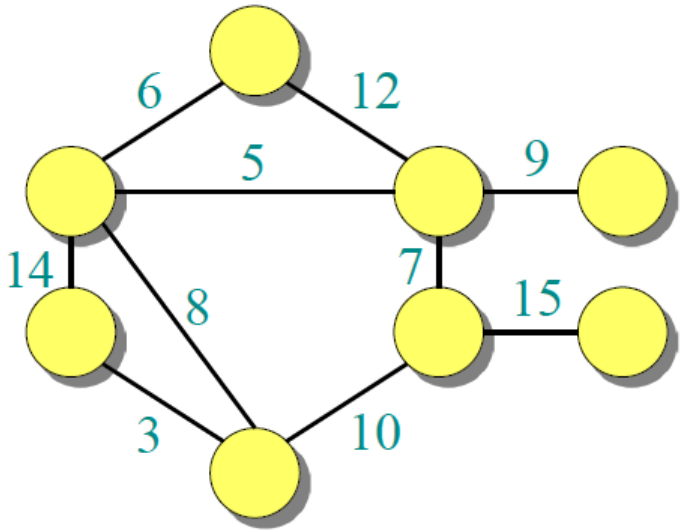


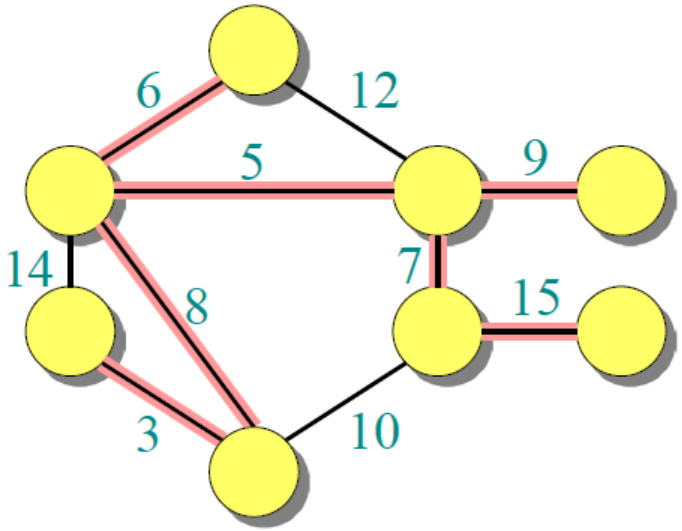
Minimum Spanning tree

- MST is a subset of edges of a connected edge-weighted undirected graph that connects all the vertices together without any cycles with the minimum possible edge weight



Minimum Spanning tree - Example





Minimum Spanning Tree

Input: A connected, undirected graph $G = (V, E)$ with weight function $w : E \to \mathsf{R}$.

Output: A ***spanning tree*** T — a tree that connects all vertices — of minimum weight:

$$w(T) = \sum_{(u,v) \in T} w(u,v) .$$

- The two algorithms we consider use a greedy approach to the problem.
  - **Prim's algorithm**
  - **Kruskal's algorithm**

Growing a minimum spanning tree

- This greedy strategy is captured by the following generic method, which **grows the minimum spanning tree one edge at a time**.
- The generic method manages a set of edges A, maintaining the following loop invariant: ***Prior to each iteration, A is a subset of some minimum spanning tree.***
- At each step, we determine an edge (u,v) that we can add to A without violating this invariant, in the sense that A Ụ {(u,v)} is also a subset of a minimum spanning tree
- We call such an **edge a safe edge for A**, since we can add it safely to A while maintaining the invariant

A loop invariant is a property of a program loop that is true before (and after) each iteration

Minimum Spanning Tree

- In Prim's algorithm, the set A forms a single tree. The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.

- In Kruskal's algorithm, the set A is a forest whose vertices are all those of the given graph. The safe edge added to A is always a least-weight edge in the graph that connects two distinct components.

Prim's algorithm

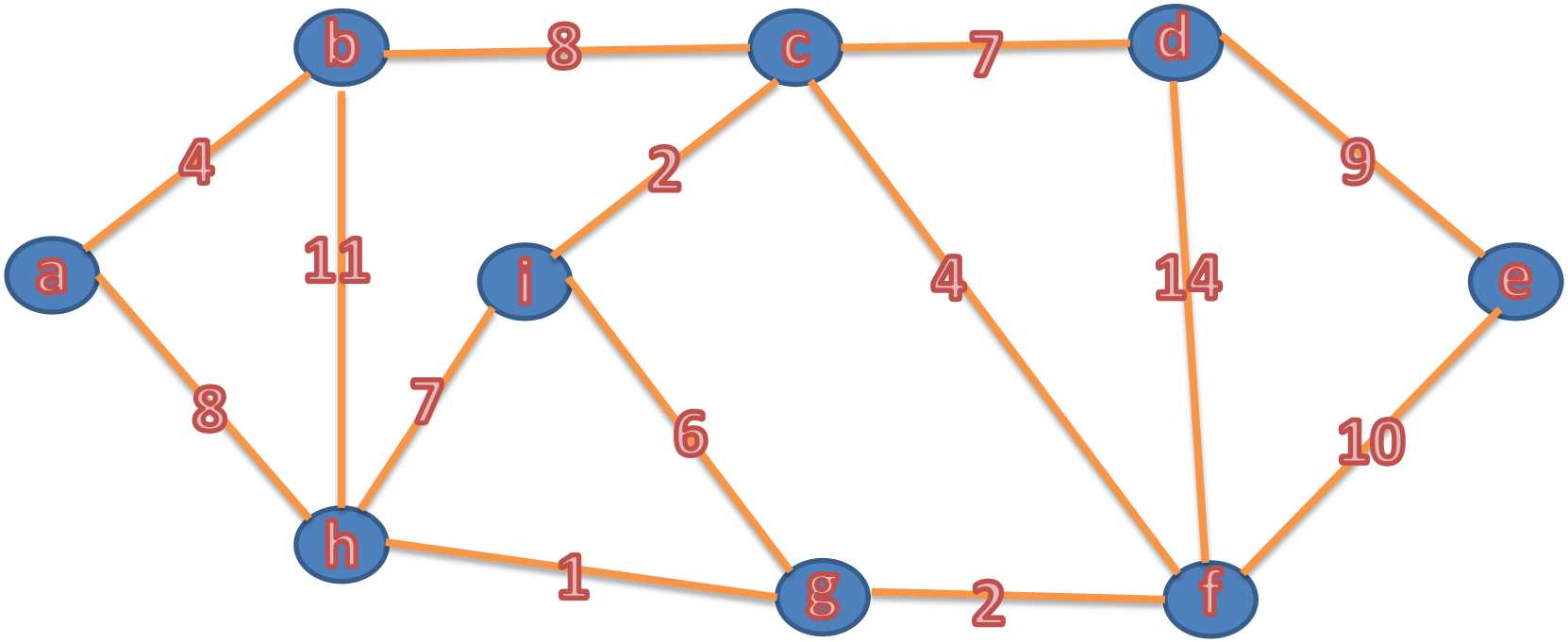
- Uses a greedy strategy
- You start with an arbitrary vertex v, choose the minimum edge that connects v to a new vertex w.
  - There's a extraction step and a relaxation step
- Apply the same process on {u,w} until you find the MST.

Prim's algorithm

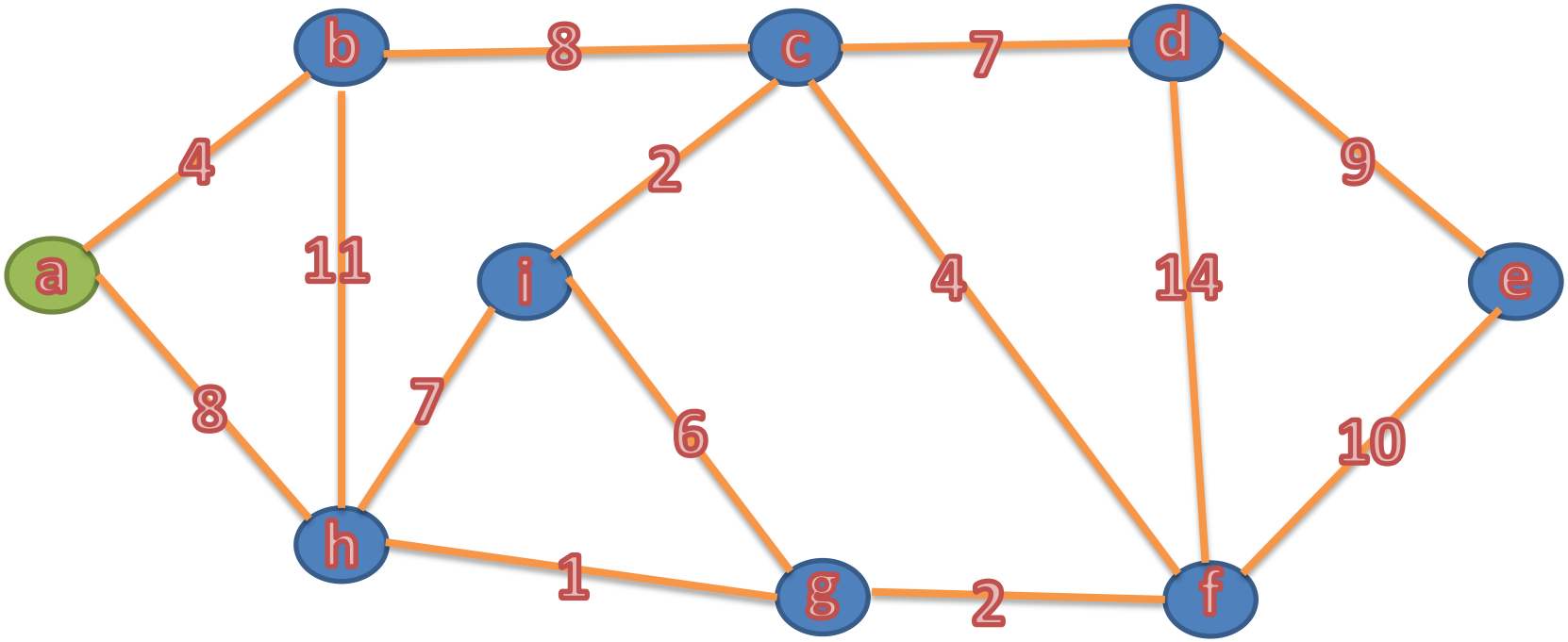
```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

- G – Graph
- *w – weight function*
- *r – arbitrary root node*
- *u.*π - parent of node u in the tree
- *v.key – minimum weight of edge connecting v to a vertex in the tree. This is infinity if no such edge exists*

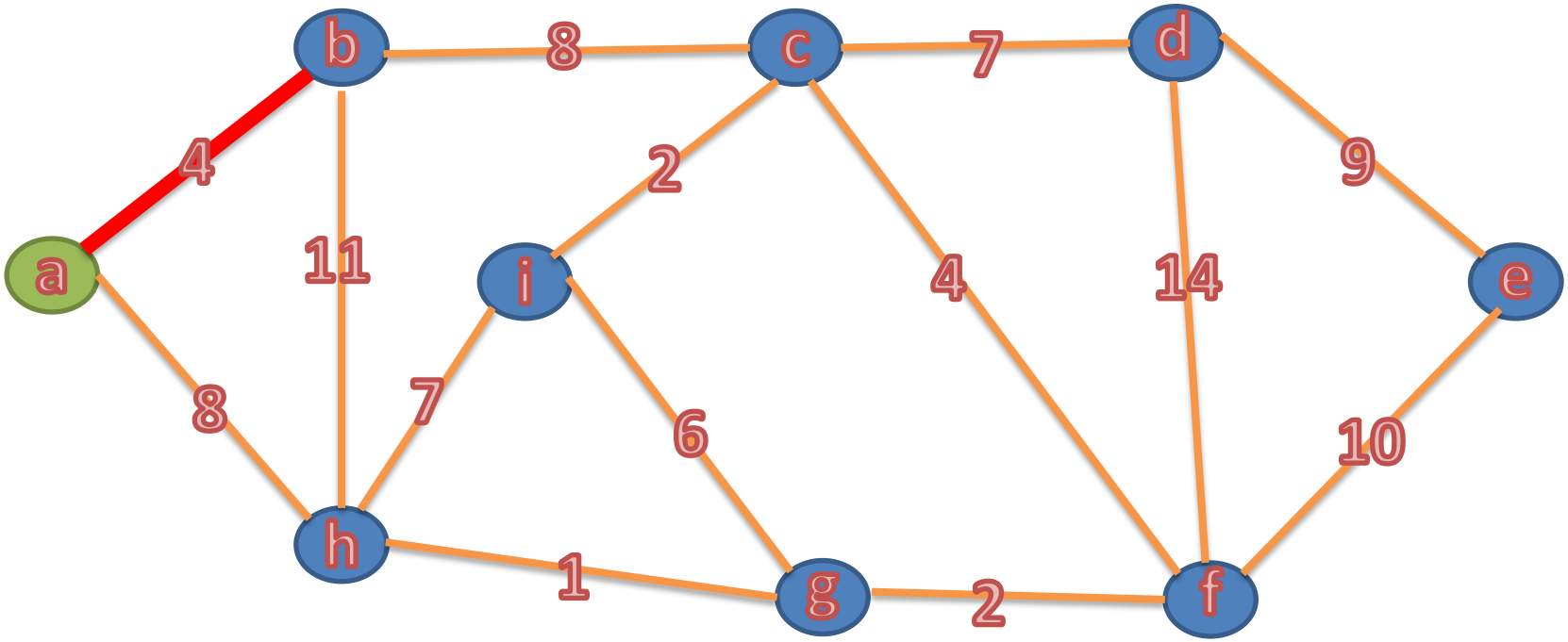
Prim’s algorithm



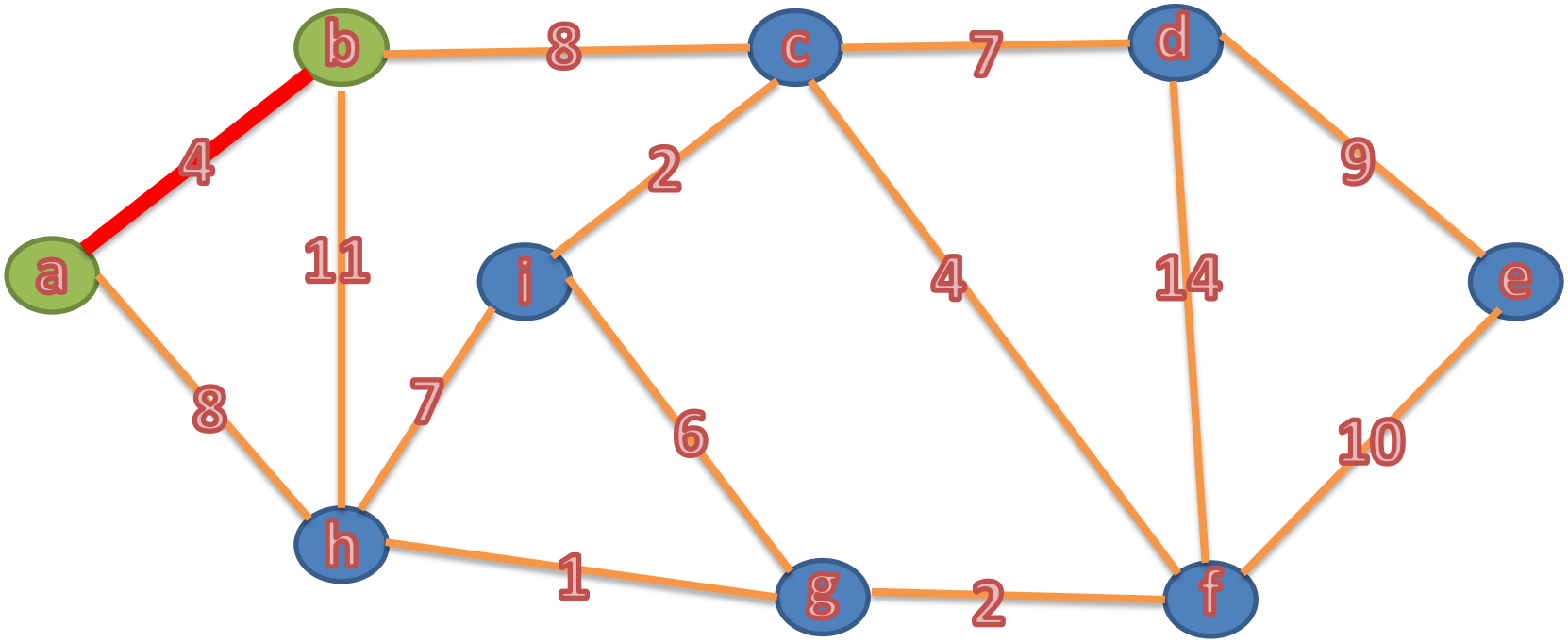
Prim's algorithm



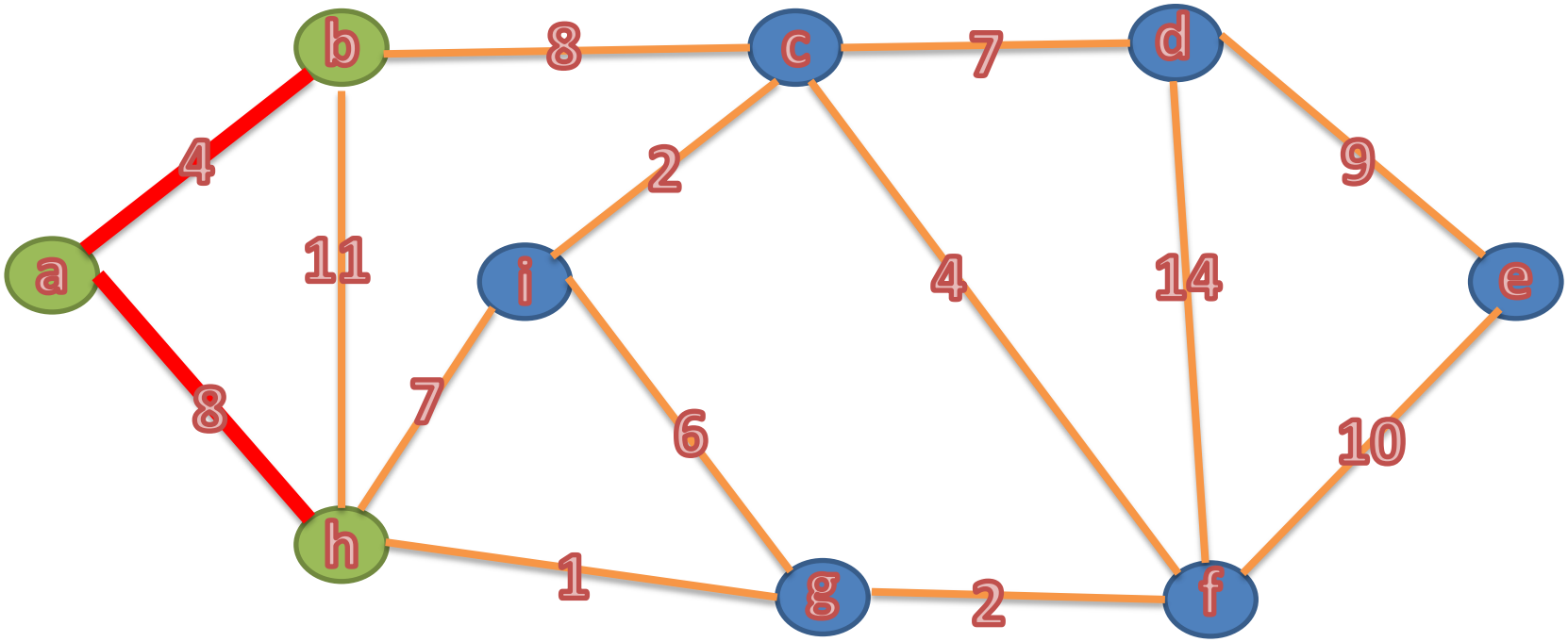
Prim’s algorithm



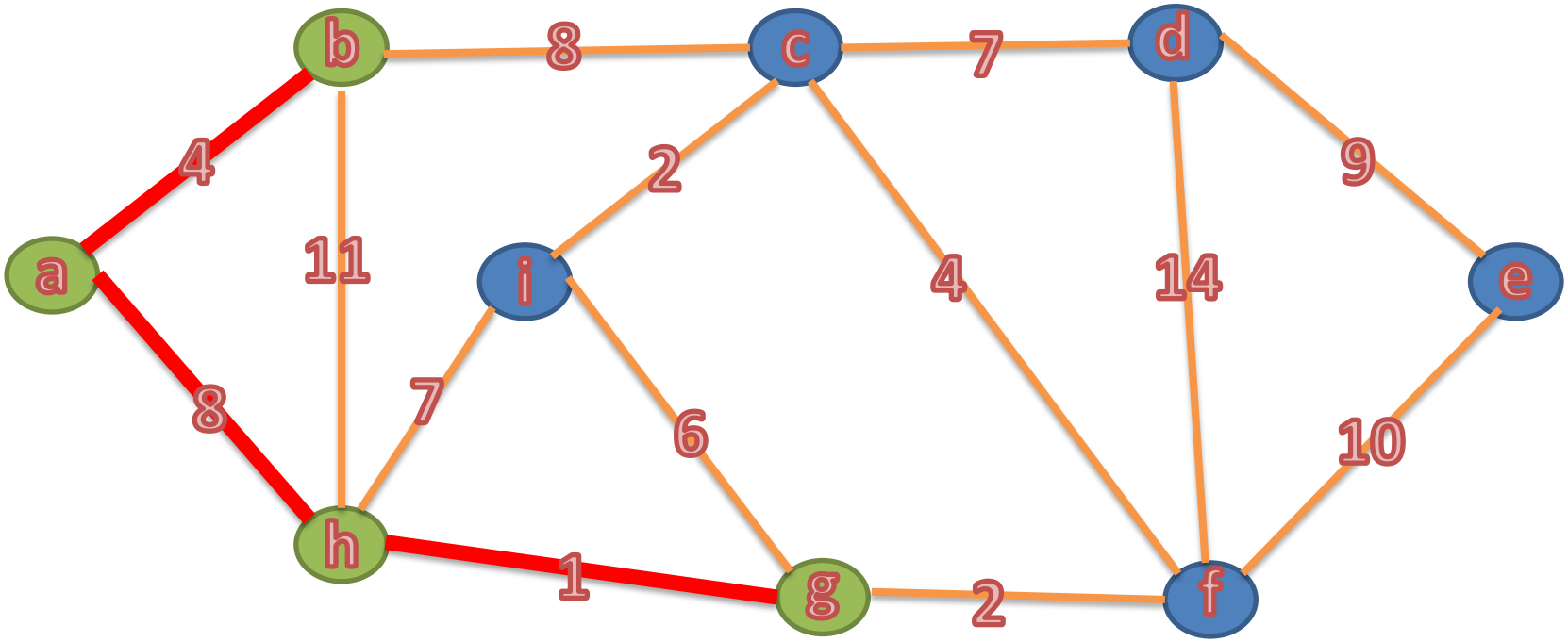
Prim’s algorithm



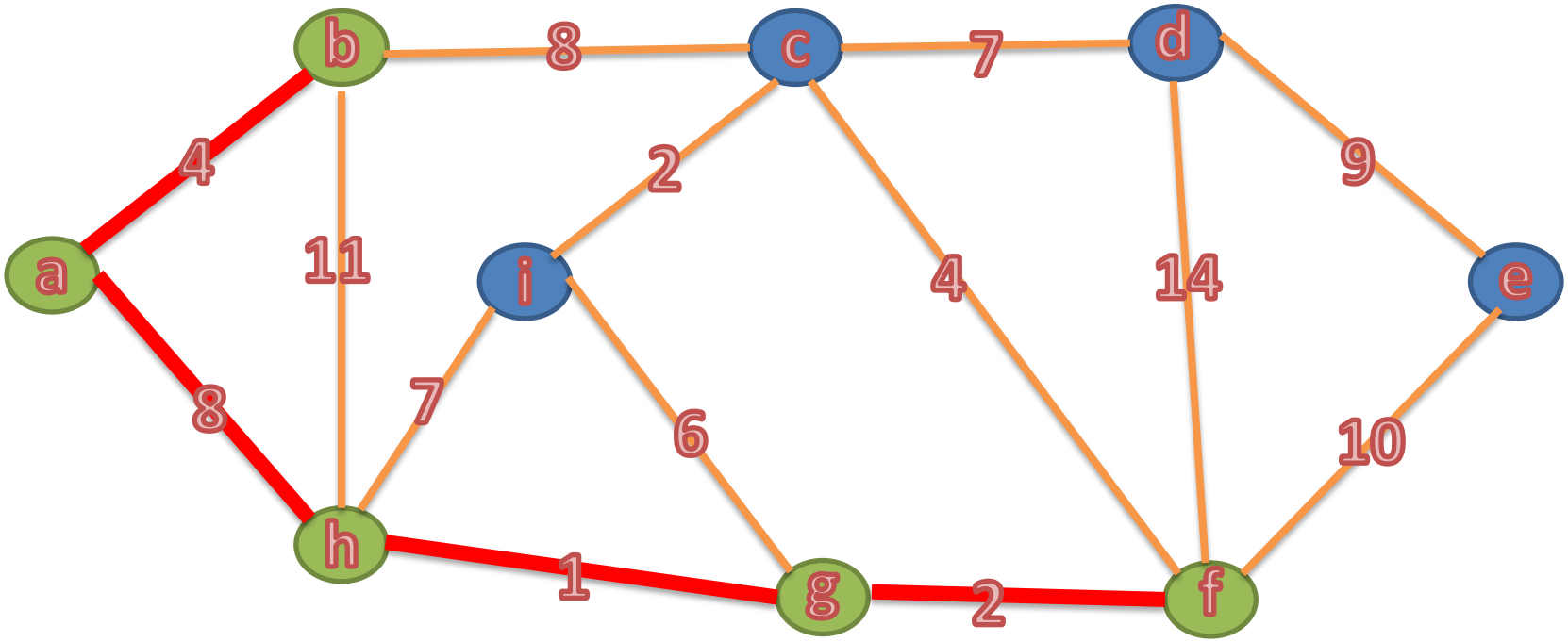
Prim's algorithm



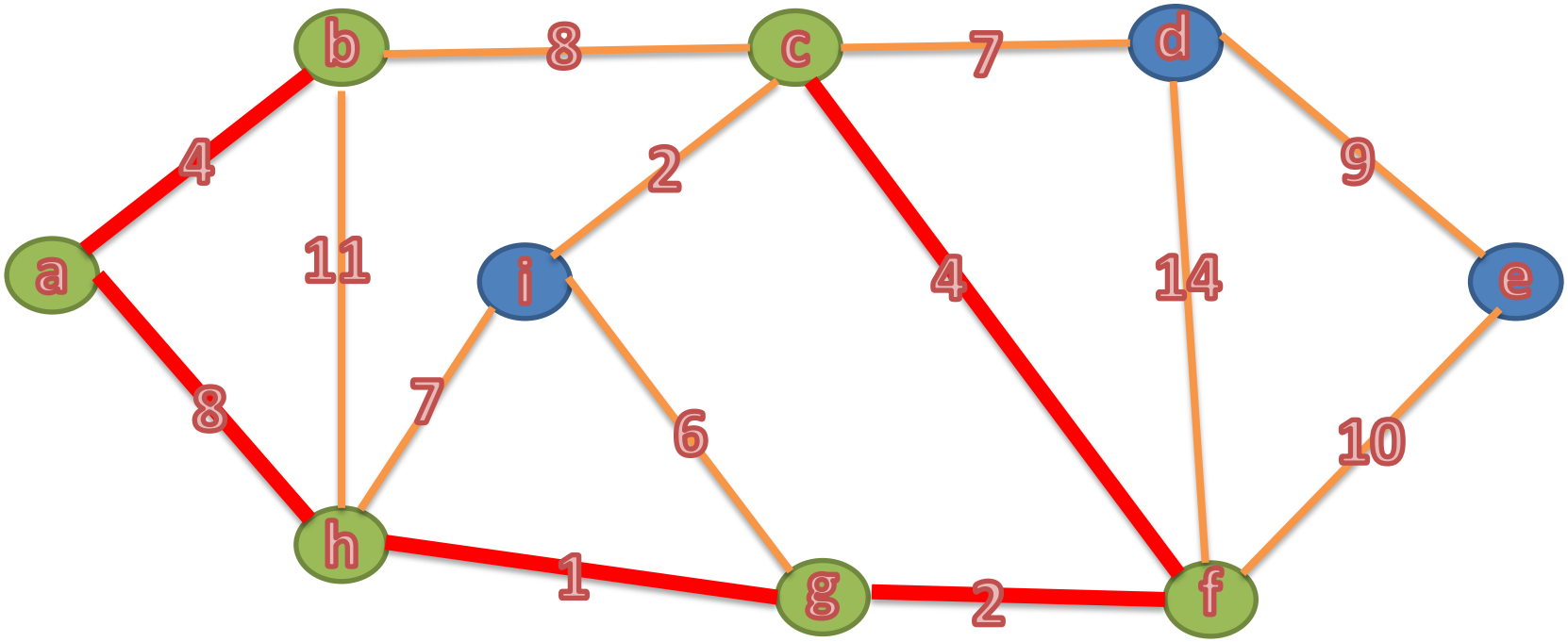
Prim’s algorithm



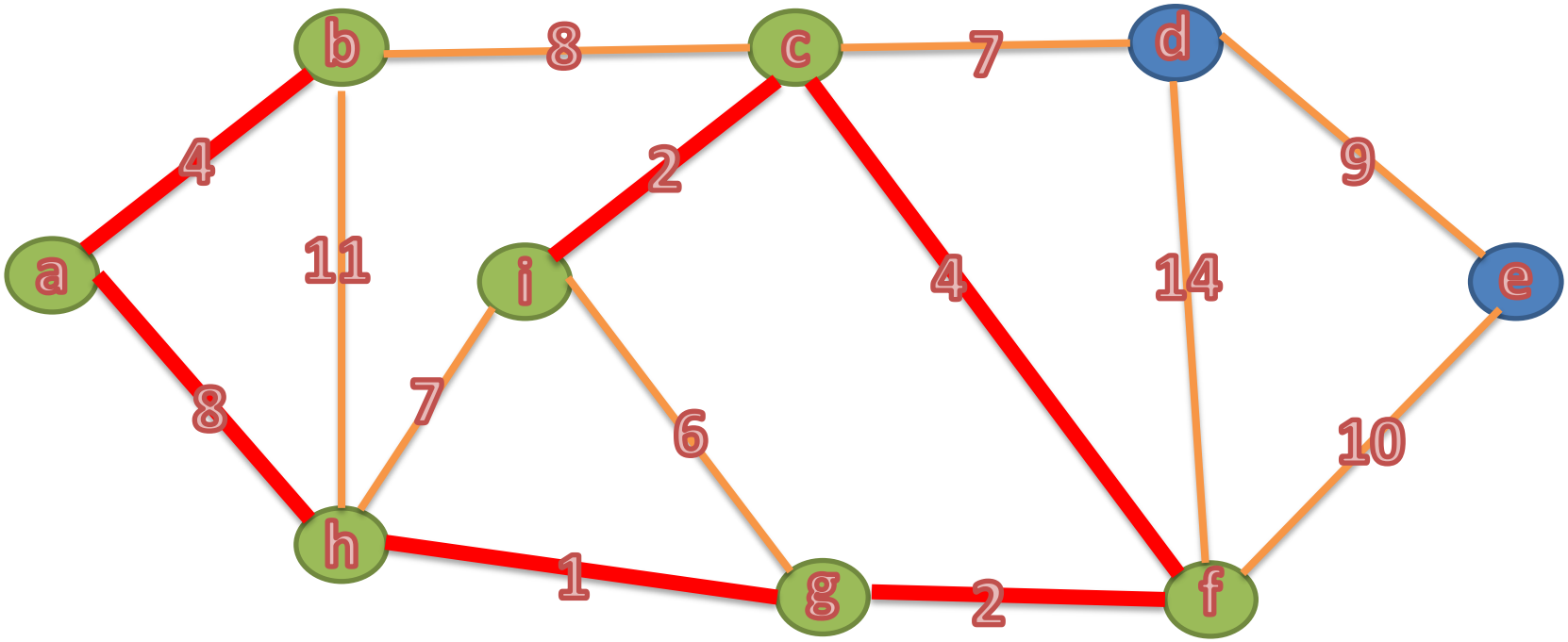
Prim’s algorithm



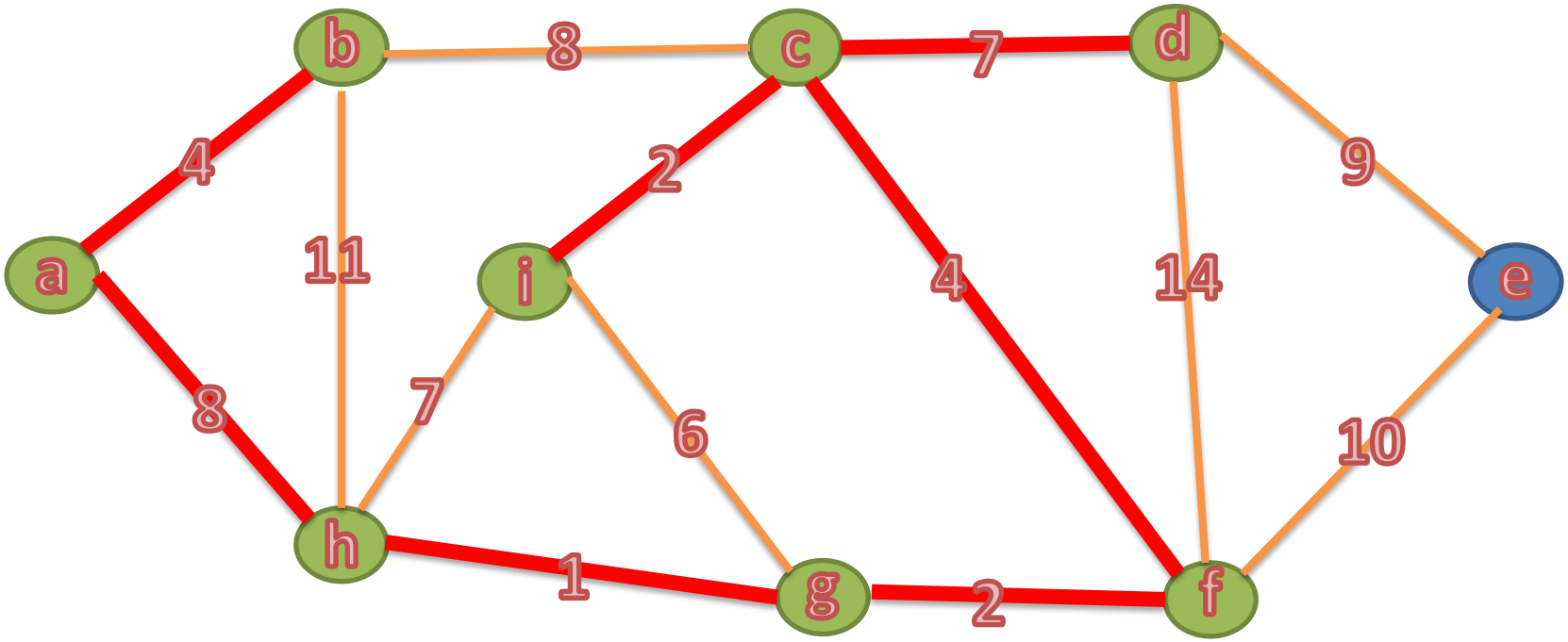
Prim’s algorithm



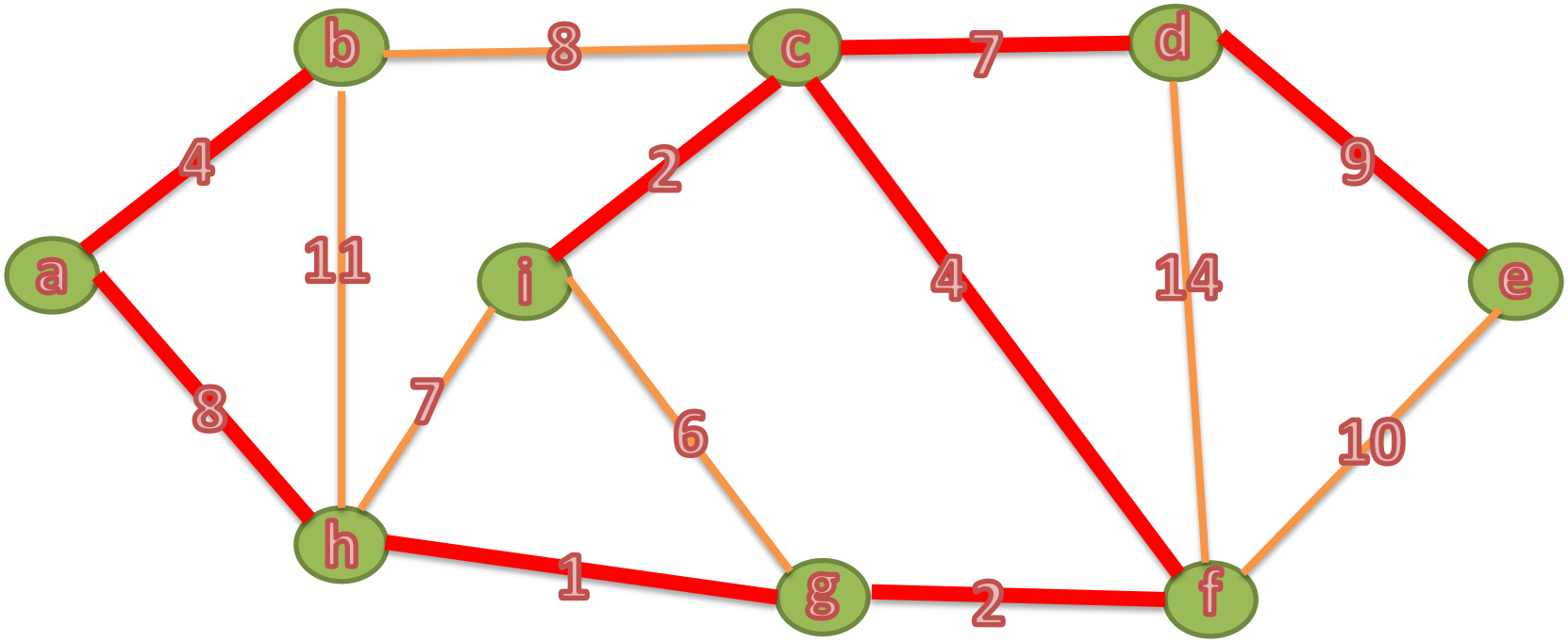
Prim’s algorithm



Prim’s algorithm



Prim’s algorithm



Prim's algorithm

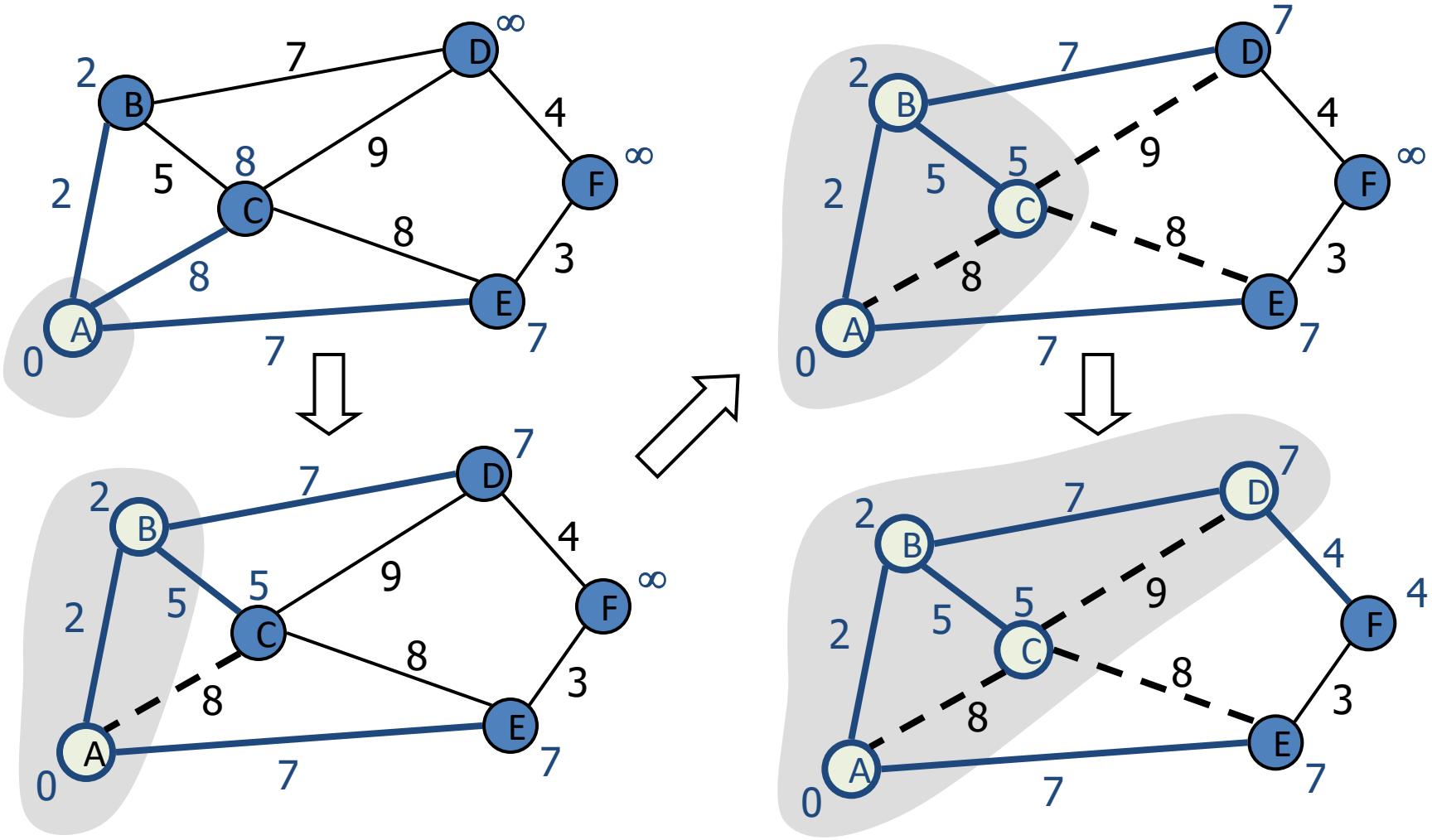
- Order:

Minimum edge weight data structure	Time complexity (total)
adjacency matrix, searching	$O(\|V\|^2)$
binary heap and adjacency list	$O((\|V\| + \|E\|) \log \|V\|) = O(\|E\| \log \|V\|)$
Fibonacci heap and adjacency list	$O(\|E\| + \|V\| \log \|V\|)$

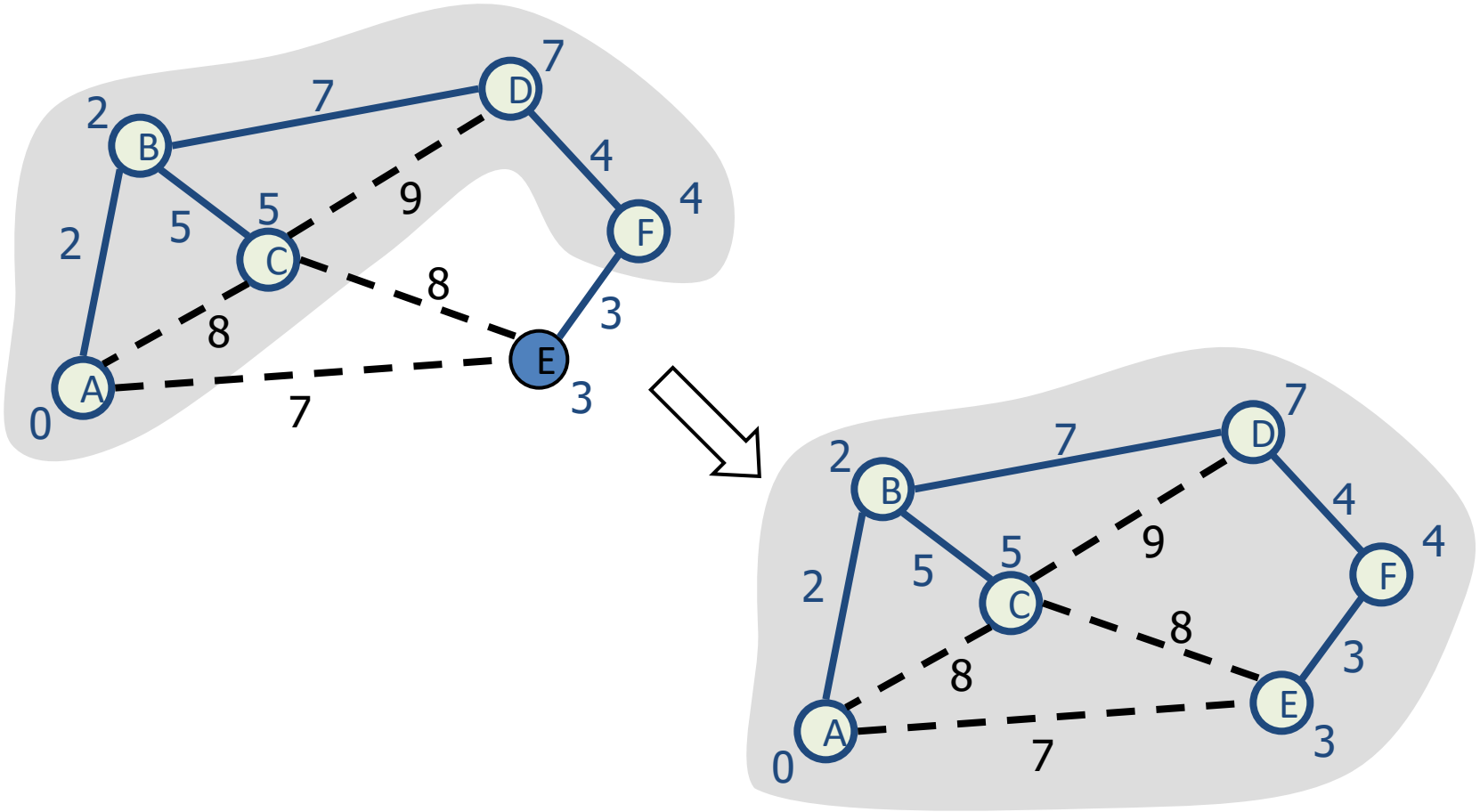
Prim's Algorithm Review

- Pick an arbitrary vertex ***s*** and grow the MST as a cloud of vertices, starting from ***s***
- Store with each vertex ***v*** label ***d***(***v***) representing the smallest weight of an edge connecting ***v*** to a vertex in the cloud
- At each step:
  - We add to the cloud the vertex ***u*** outside the cloud with the smallest distance label
  - Make sure the result is a tree that is no cycle.
  - We update the labels of the vertices adjacent to ***u***

Prim's Algorithm for MST - Example



Example (contd.)



Analysis

- Graph operations
  - We cycle through the incident edges once for each vertex
- Label operations
  - We set/get the distance, parent and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex w in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time

- Prim's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure. Recall that sum of all degrees equals $2m$. The running time is $O(m \log n)$ since the graph is connected

Kruskal's Algorithm

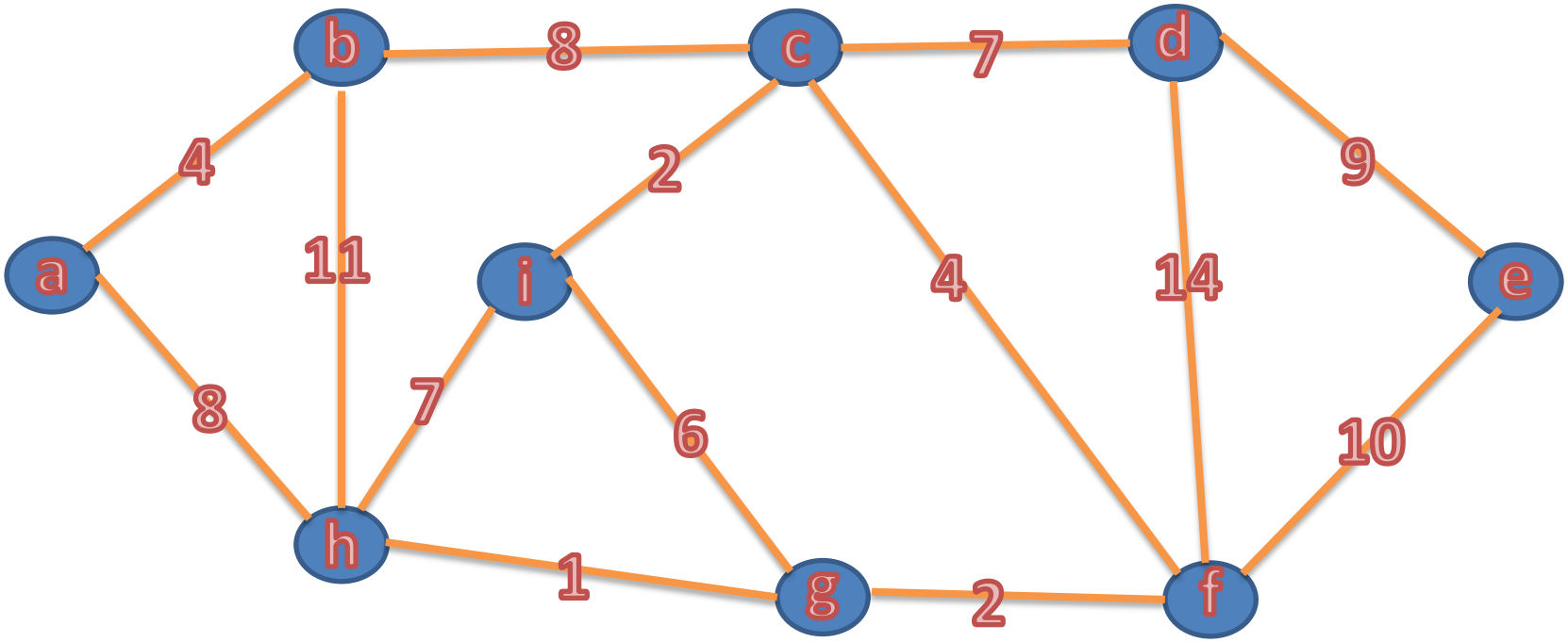
- In Kruskal's algorithm, edges are scanned once
- An edge with least weight is picked and added to vertices
- If an edge creates a cycle, it is deleted

Kruskal's Algorithm

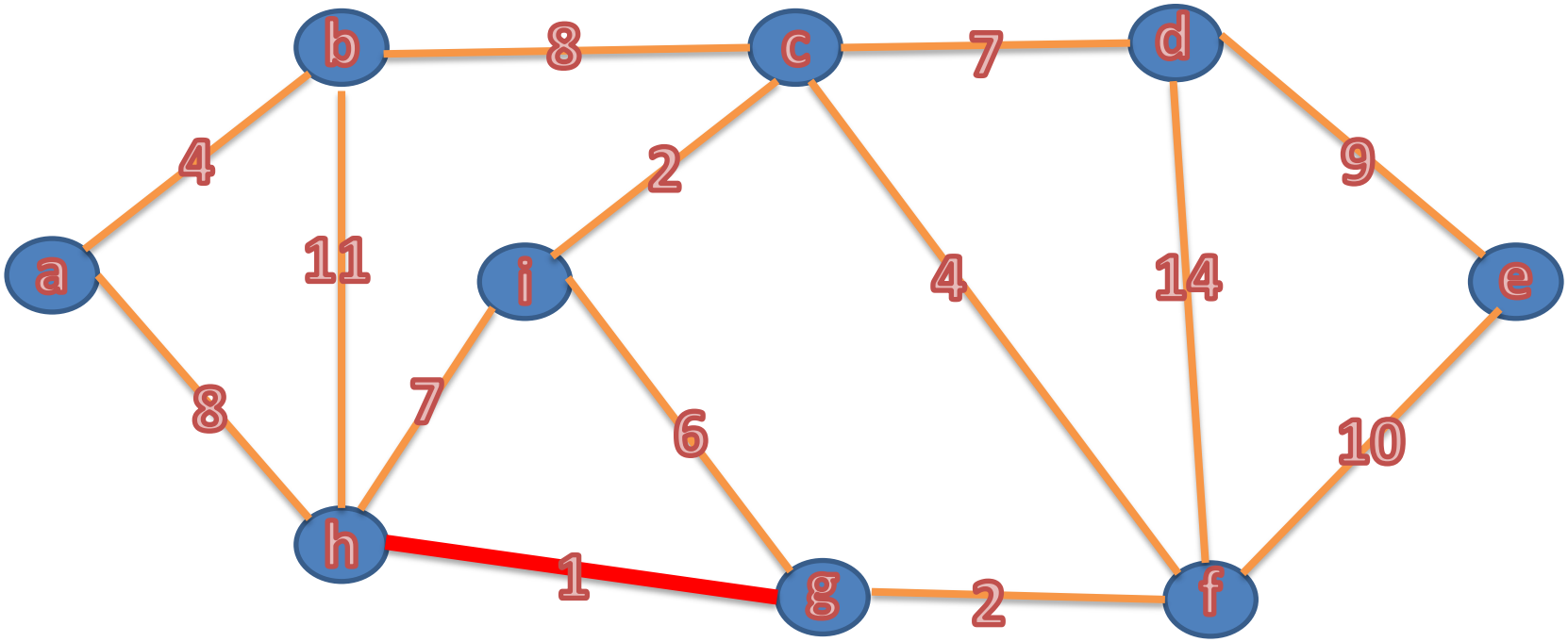
- G – Graph
- *w – weight function*
- *MAKE-SET(v) – Create a set containing node v*
- *FIND-SET(v) – Return a unique identifier for the set containing node v*
- *UNION(u,v) – merge the sets containing nodes u and v*

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

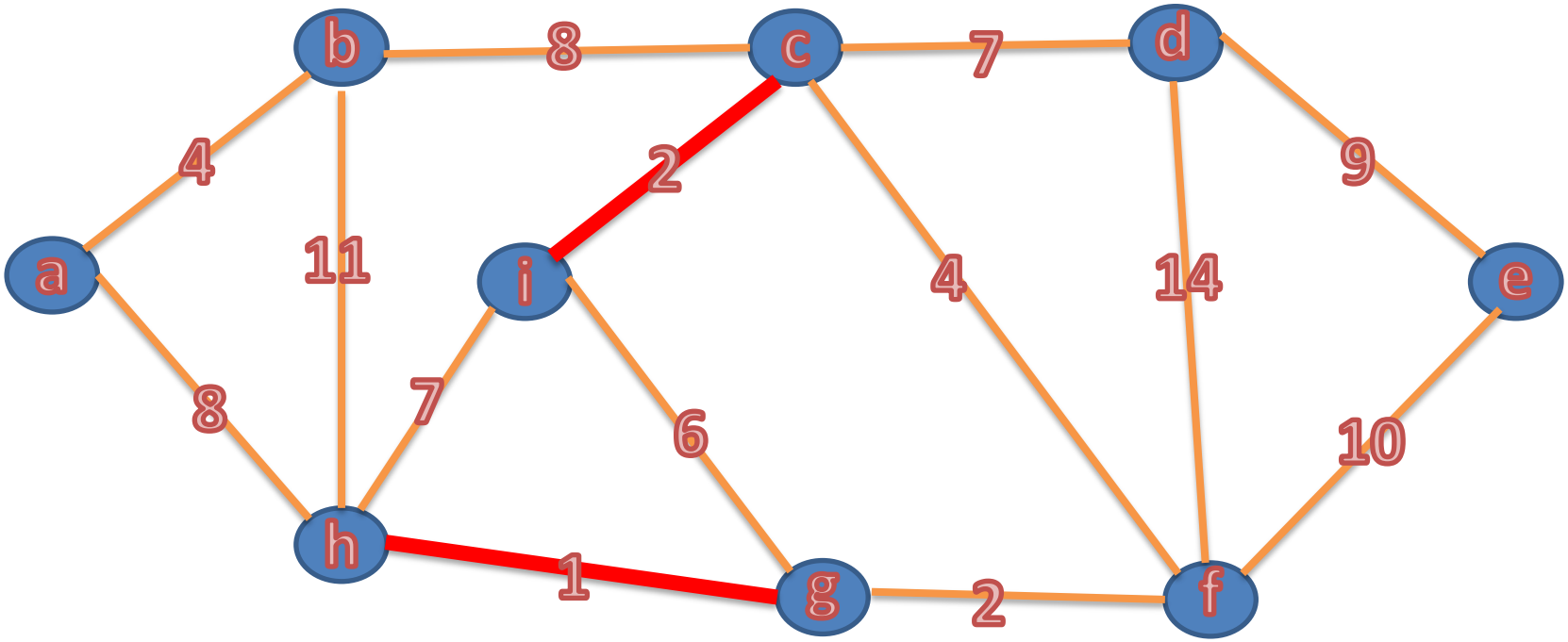
Kruskal's Algorithm



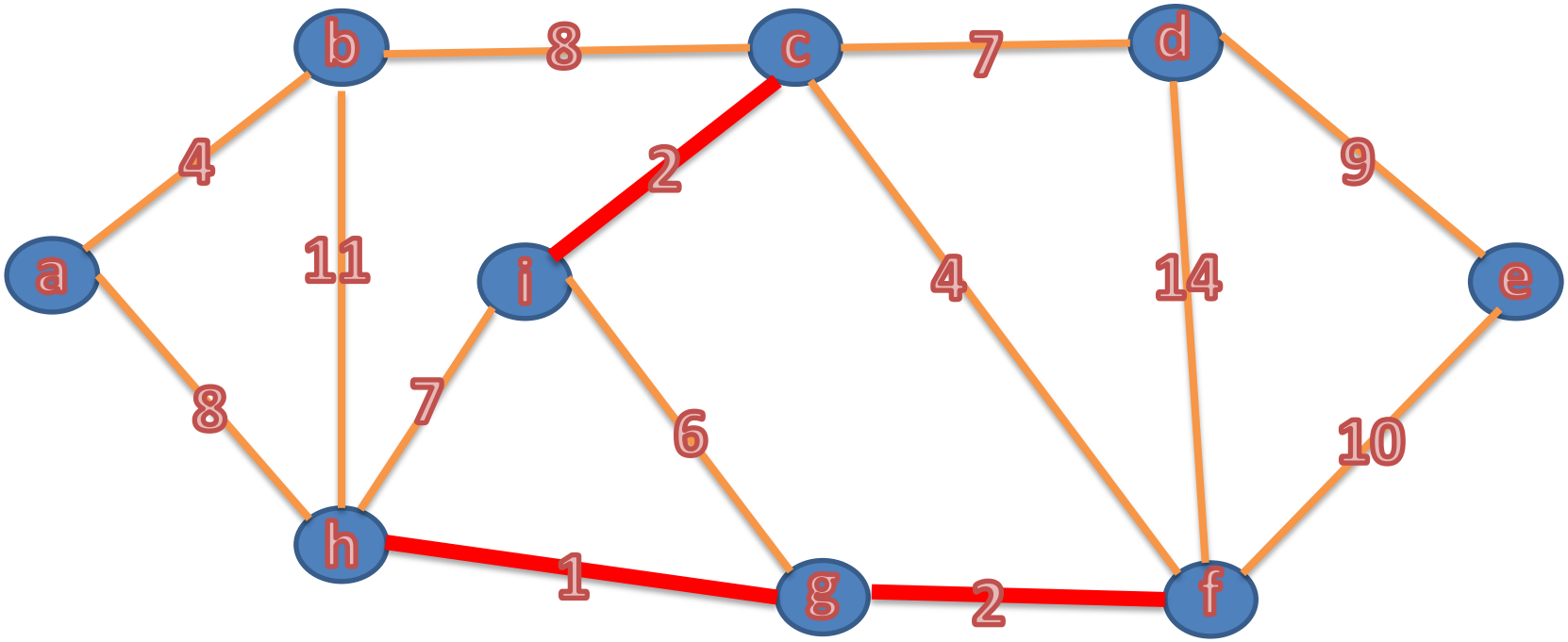
Kruskal's Algorithm



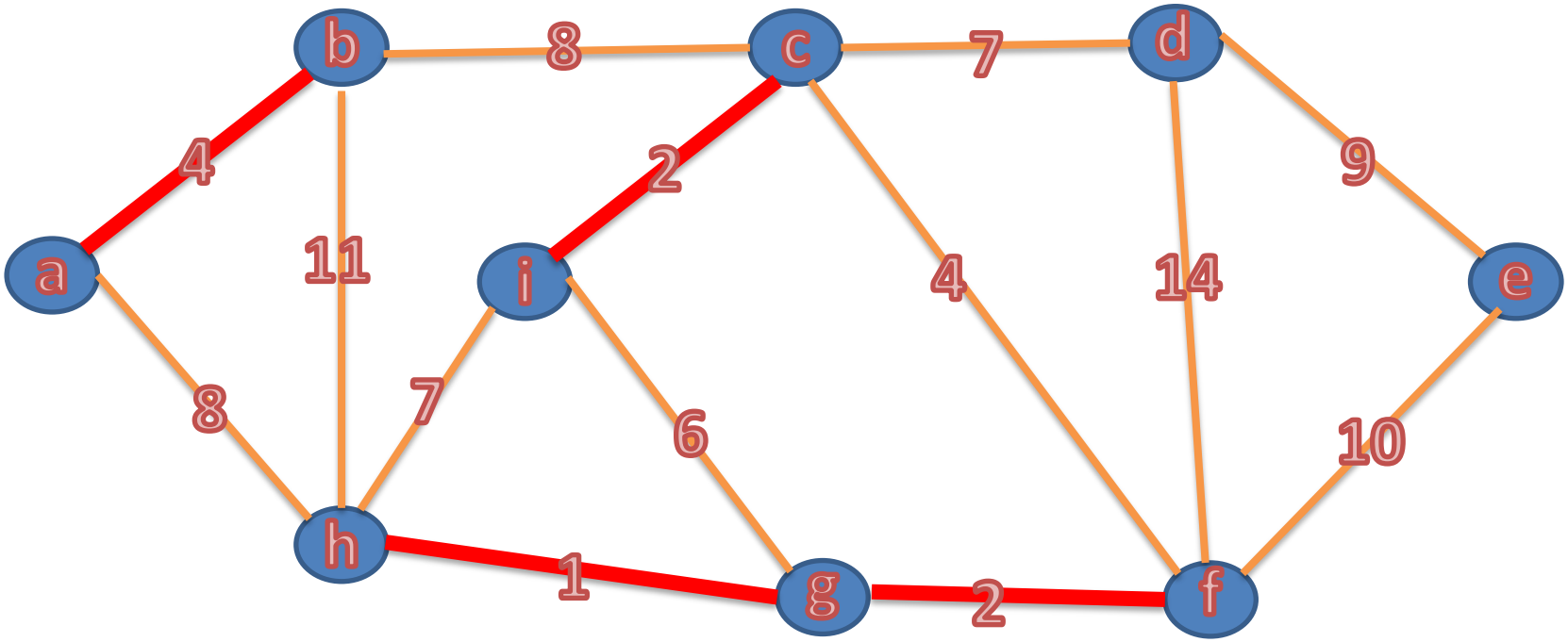
Kruskal's Algorithm



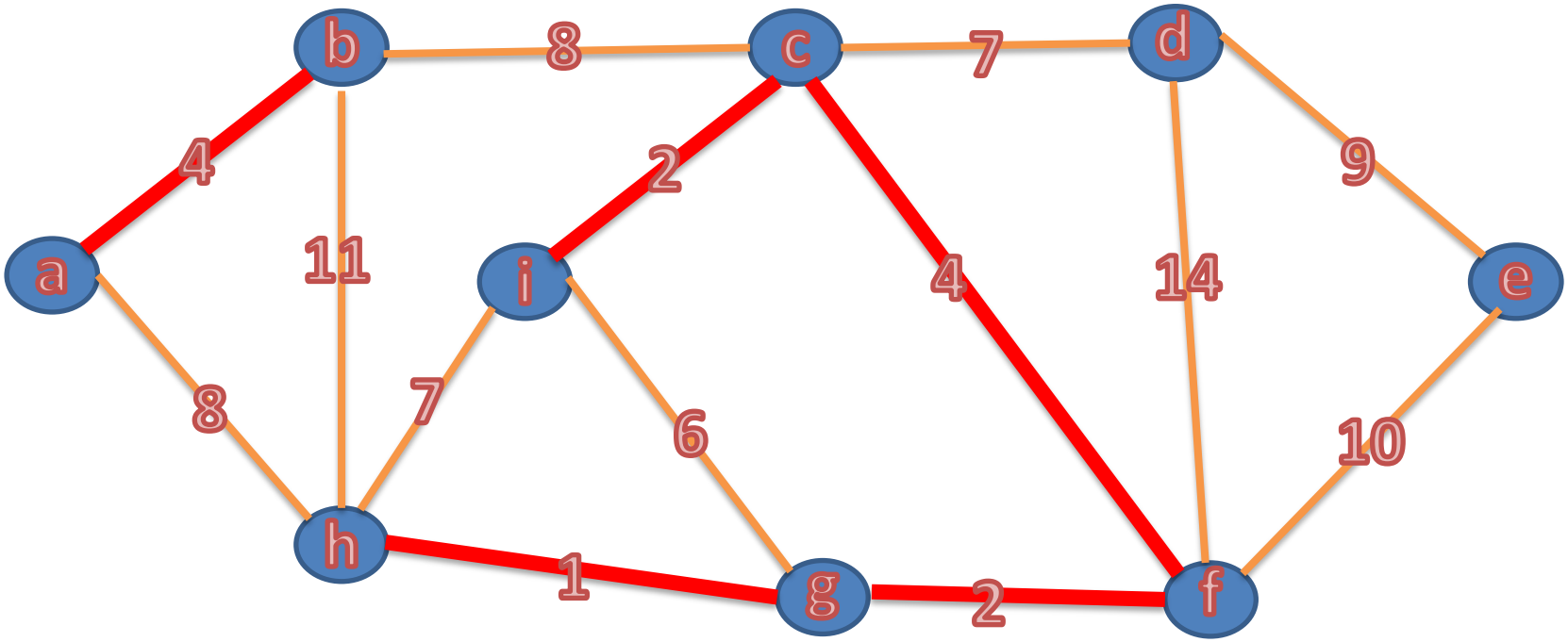
Kruskal's Algorithm



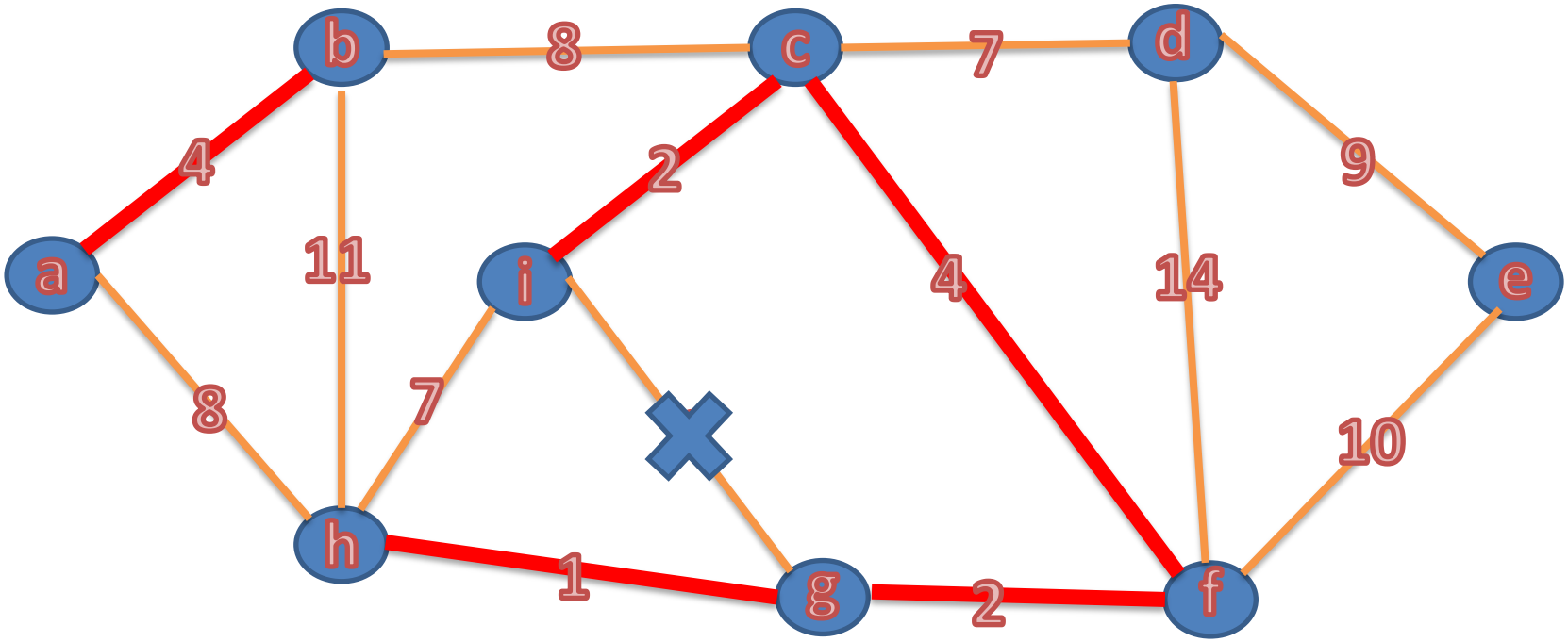
Kruskal's Algorithm



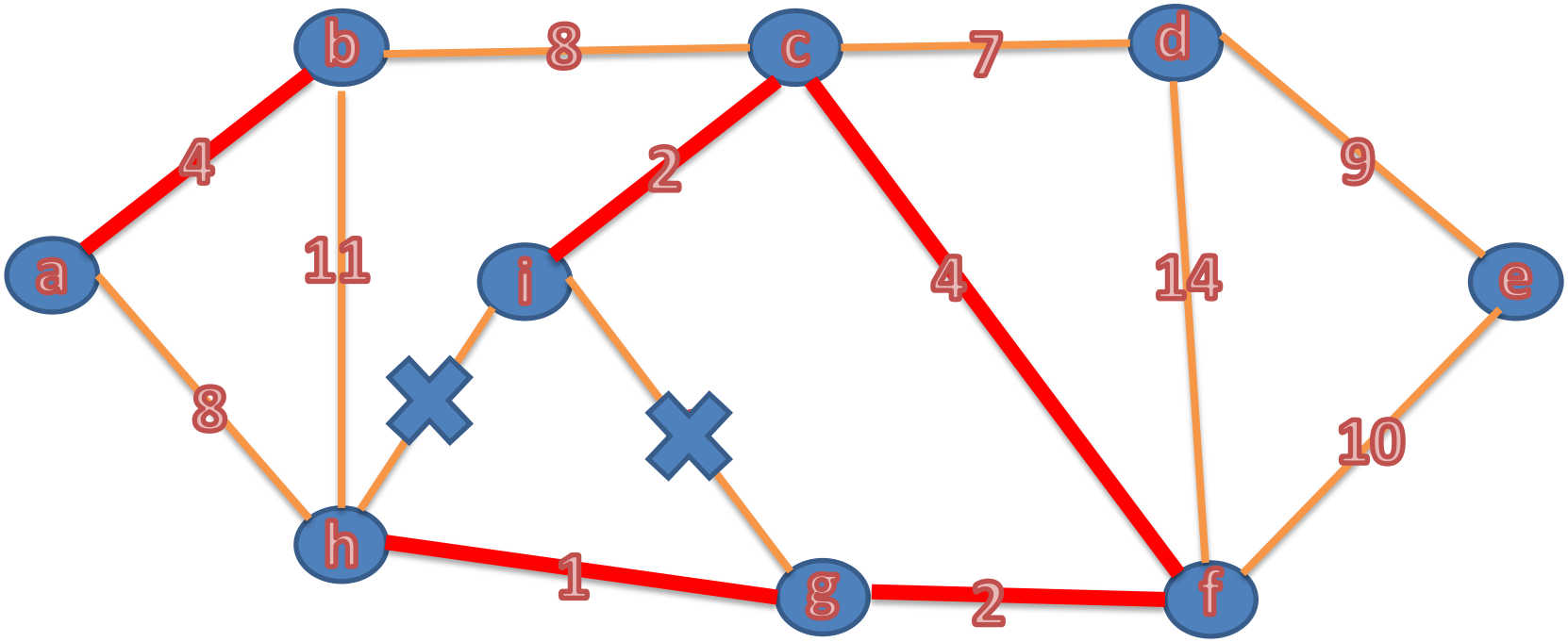
Kruskal's Algorithm



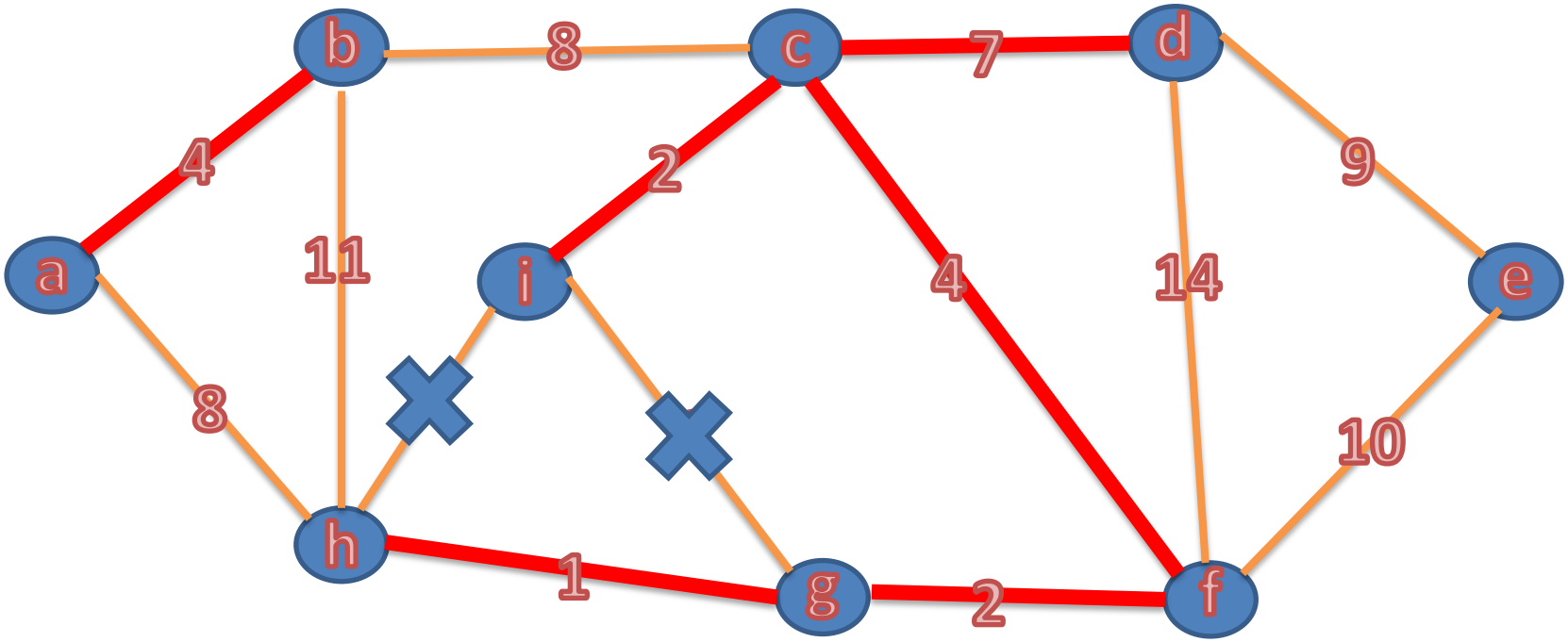
Kruskal's Algorithm



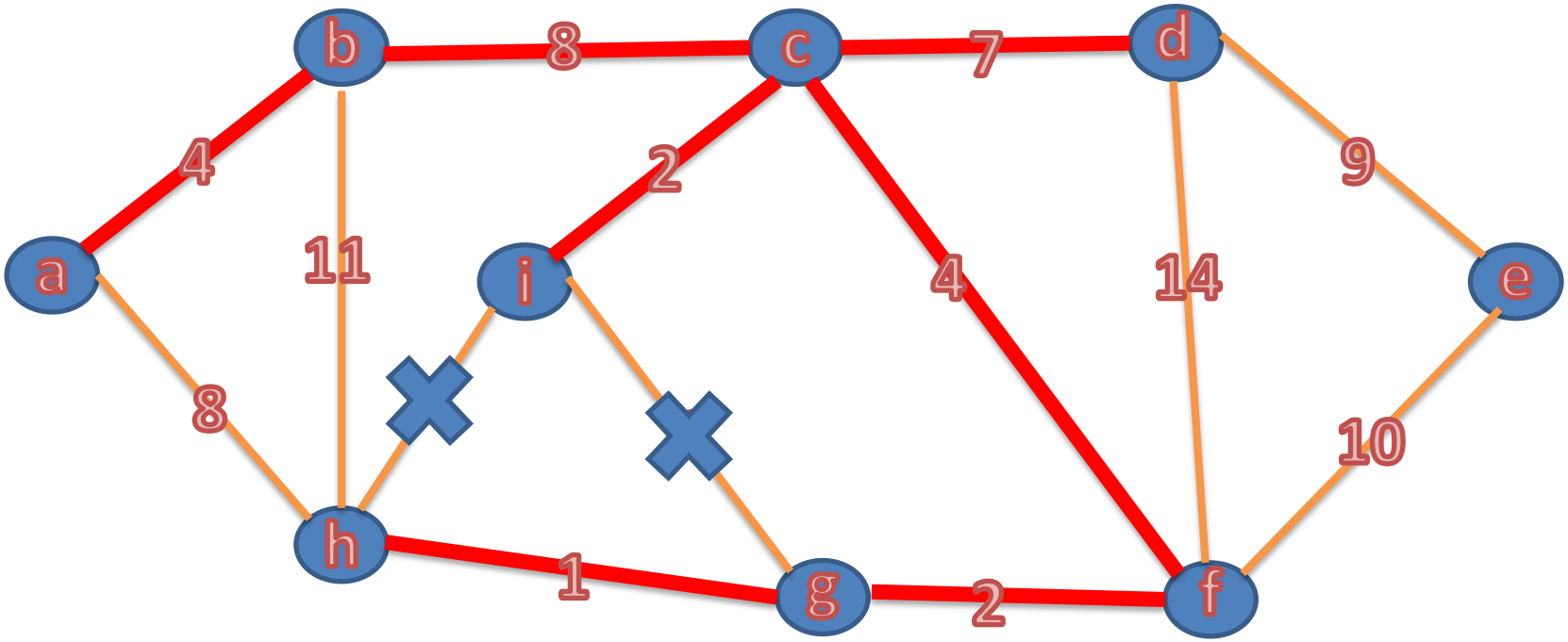
Kruskal's Algorithm



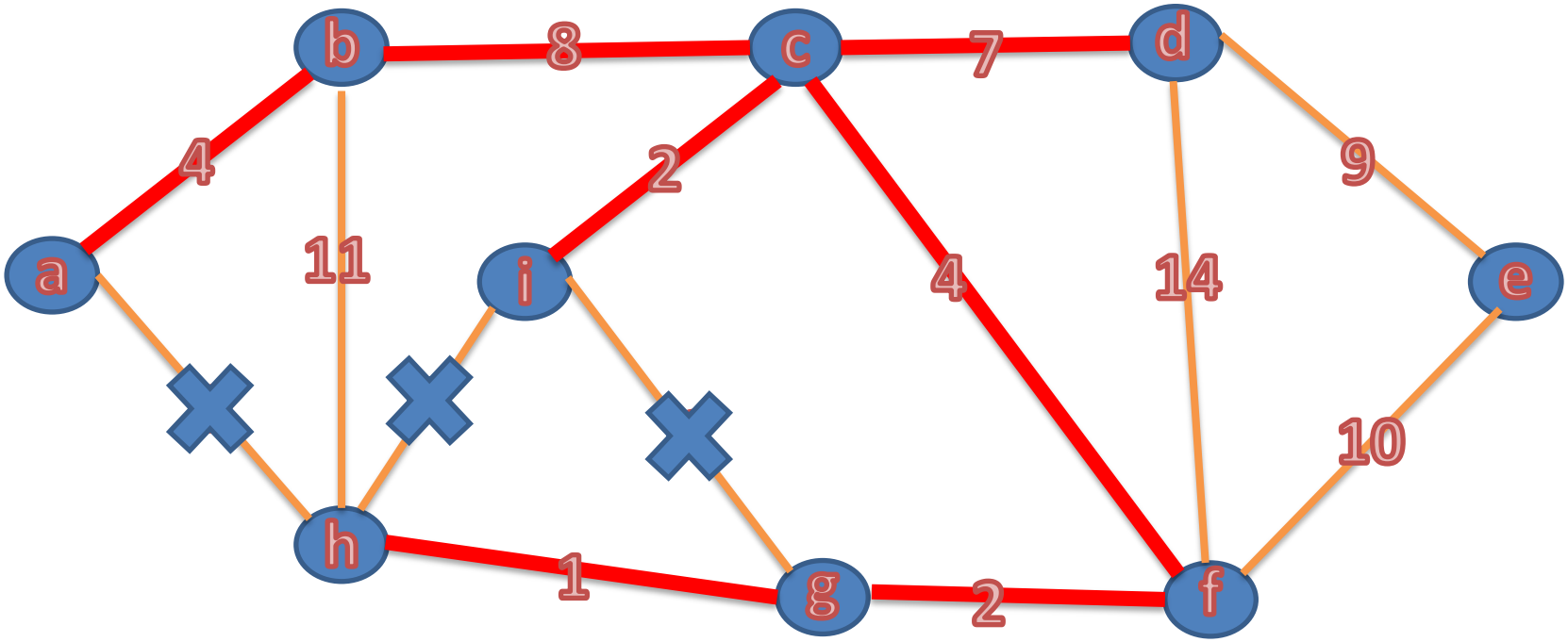
Kruskal's Algorithm



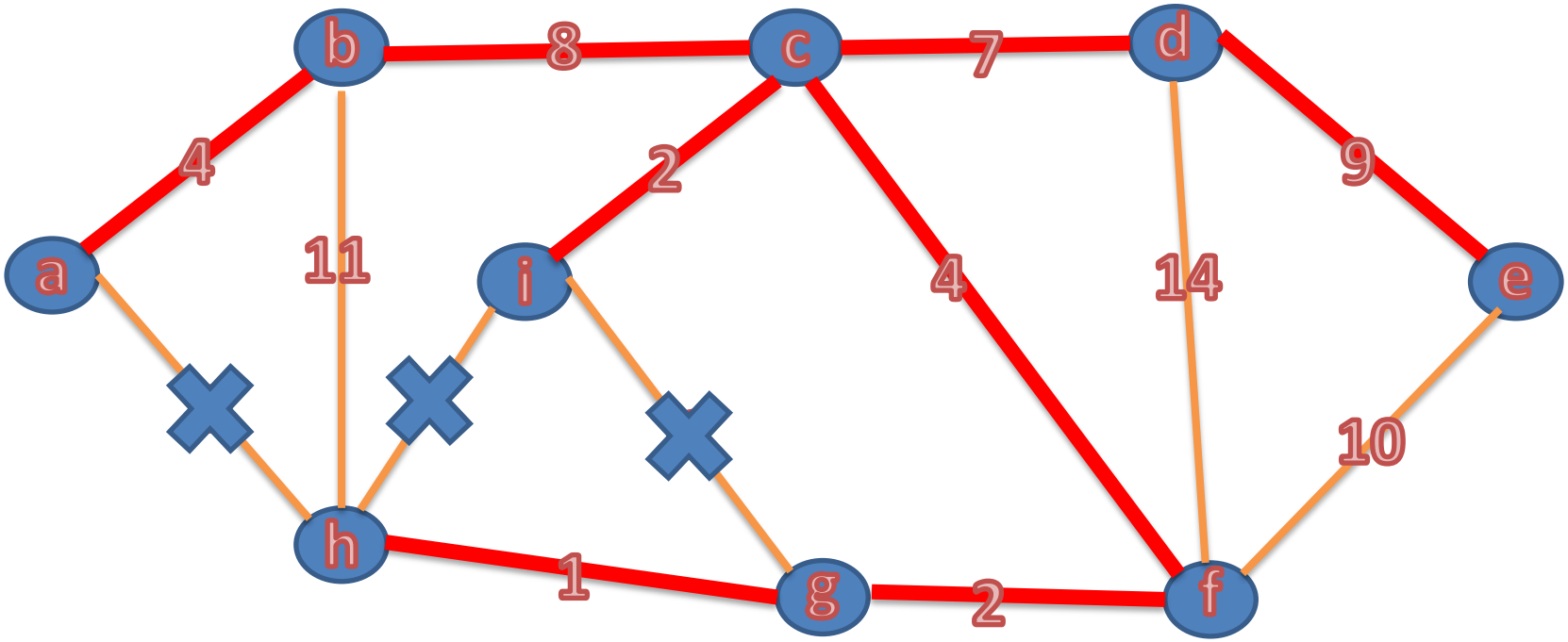
Kruskal's Algorithm



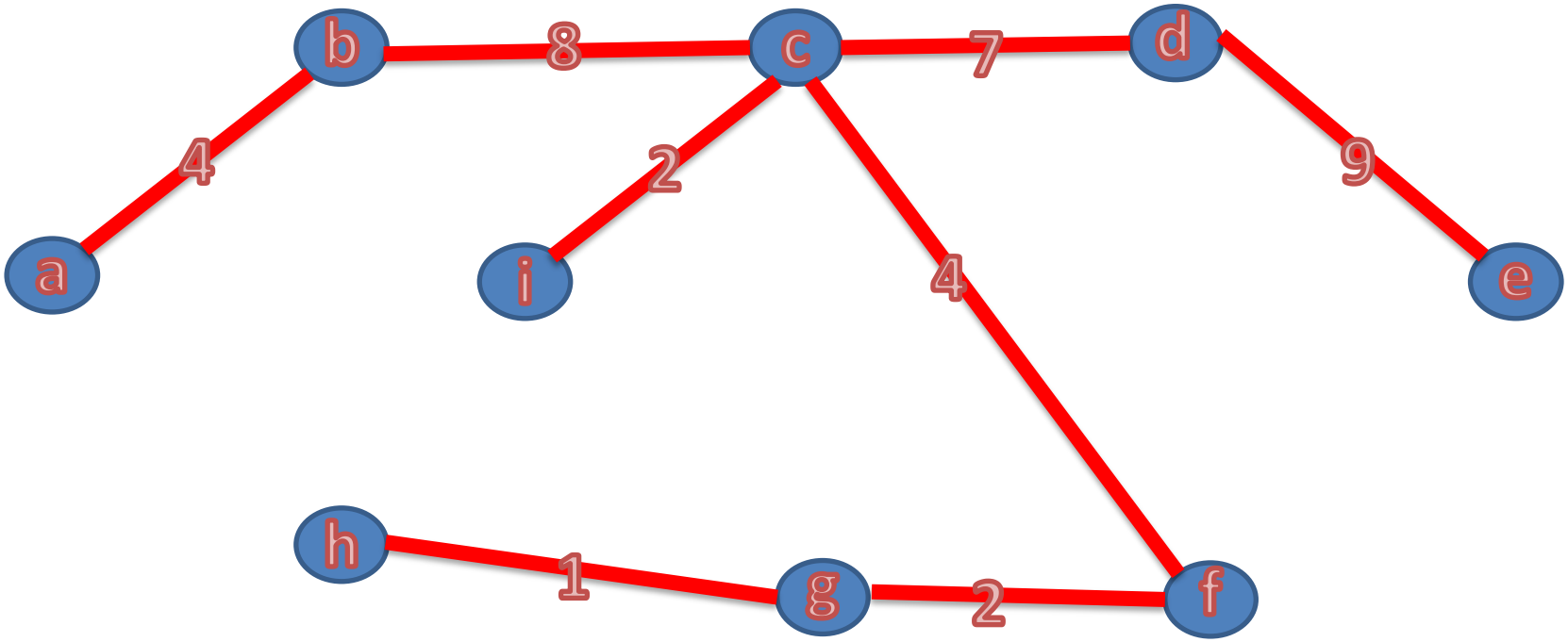
Kruskal's Algorithm



Kruskal's Algorithm



Kruskal's Algorithm



Kruskal's Algorithm

- Order: $\theta(ELogE)$

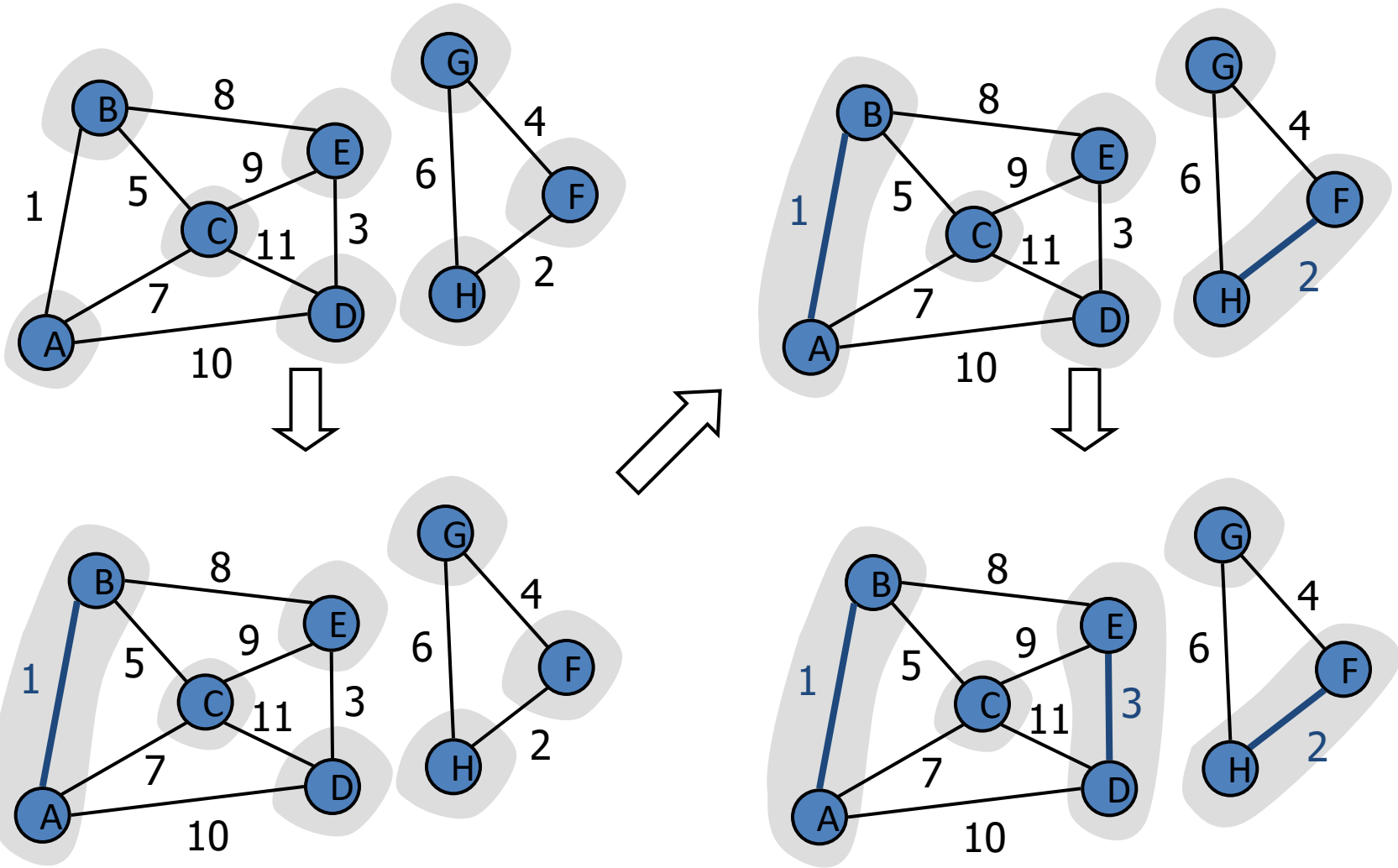
Kruskal's Algorithm Review

- Maintain a partition of the vertices into clusters
  - Initially, single-vertex clusters
  - Keep an MST for each cluster
  - Merge "closest" clusters and their MSTs
- A priority queue stores the edges outside clusters
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - One cluster and one MST

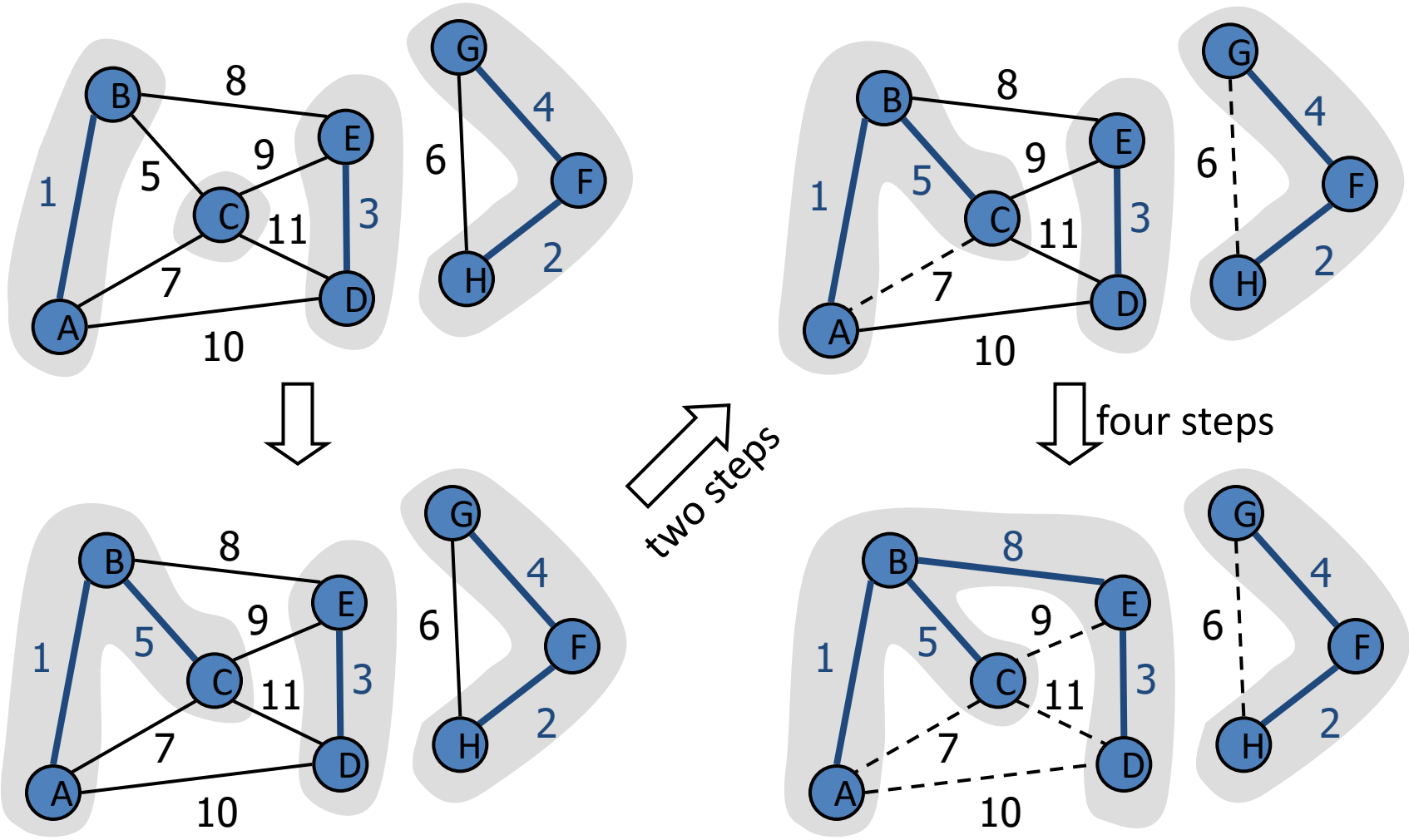
❑ The algorithm maintains a forest of trees

❑ A priority queue extracts the edges by increasing weight

Kruskal's Algorithm for MST - Example



Example (contd.)



Graph Traversal

- Depth First Search

- Breadth First Search

Graph Traversal - DFS

- Exploration of a vertex *v* is suspended as soon as a new vertex *u* is reached.
- Exploration of the new vertex *u* begins.
- Exploration of *v* continues after *u* has been explored.
- This can be expressed as a recursive algorithm

Depth-First Search (DFS)

- A general technique for traversing a graph G that
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G

- For a graph with n nodes and m edges it takes $O(n + m)$

- DFS can be extended to solve other graph problems
  - Find a path between two given vertices
  - Find a cycle in the graph

DFS Algorithm from a Vertex

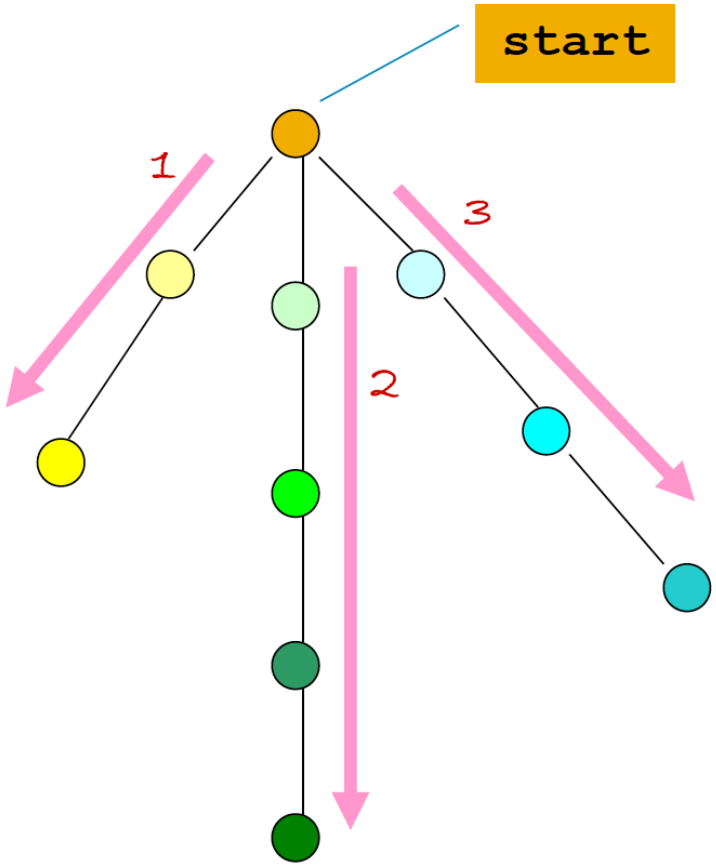
Algorithm DFS(G, u):

Input: A graph G and a vertex u of G

Output: A collection of vertices reachable from u, with their discovery edges

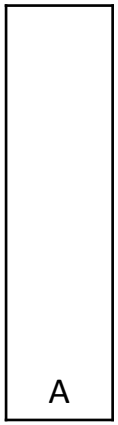
Mark vertex u as visited.
for each of u's outgoing edges, $e = (u, v)$ **do**
  if vertex v has not been visited **then**
    Record edge e as the discovery edge for vertex v.
    Recursively call DFS(G, v).

- Visit a vertex, *v*, move from *v* as deeply as possible
- Use a stack to store nodes
  - Stacks are LIFO
- DFS:
  - visit and push start
  - while (*s* not empty)
  - peek at node, *nd*, at top of *s*
  - if *nd* has an unvisited neighbour visit it and push it onto *s*
  - else pop *nd* from *s*

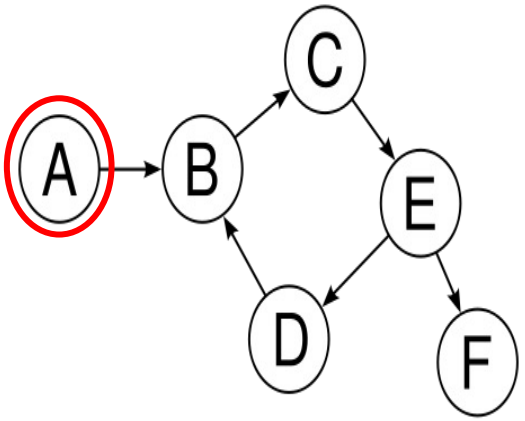


Graph Traversal - DFS

Stack

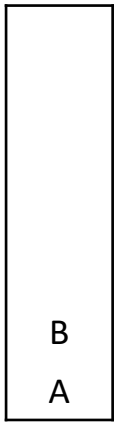


Output: A

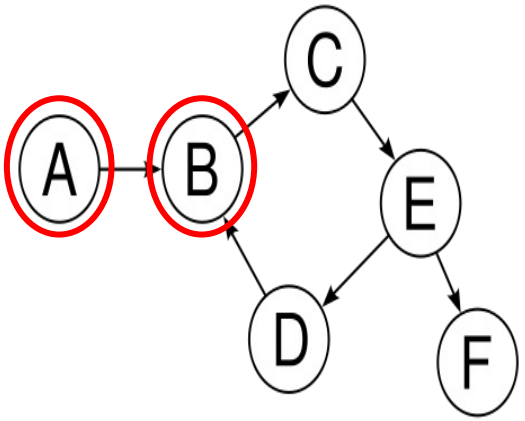


Graph Traversal - DFS

Stack

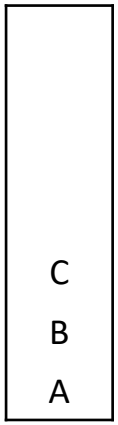


Output: A, B

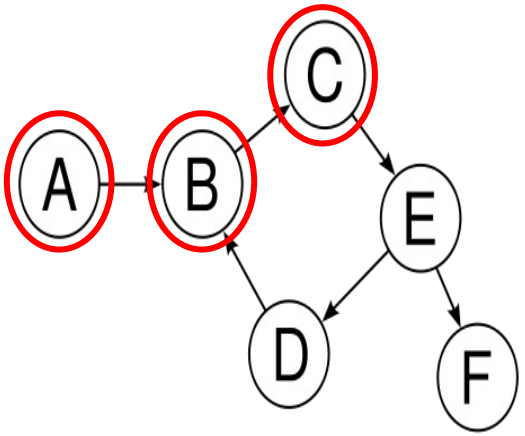


Graph Traversal - DFS

Stack

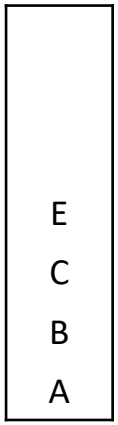


Output: A, B, C

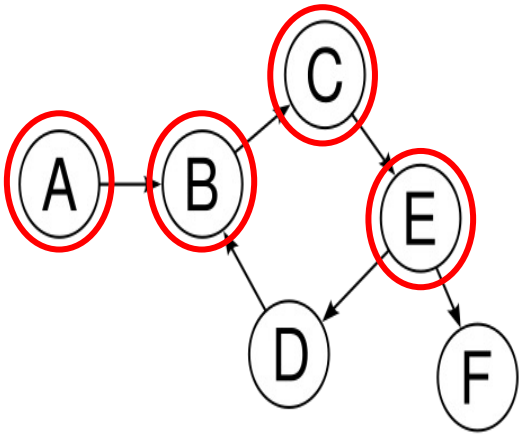


Graph Traversal - DFS

Stack

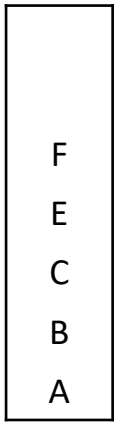


Output: A, B, C, E

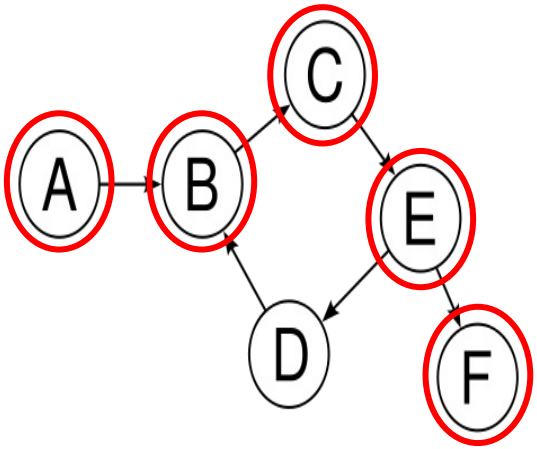


Graph Traversal - DFS

Stack

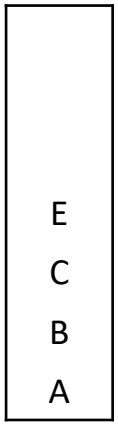


Output: A, B, C, E, F



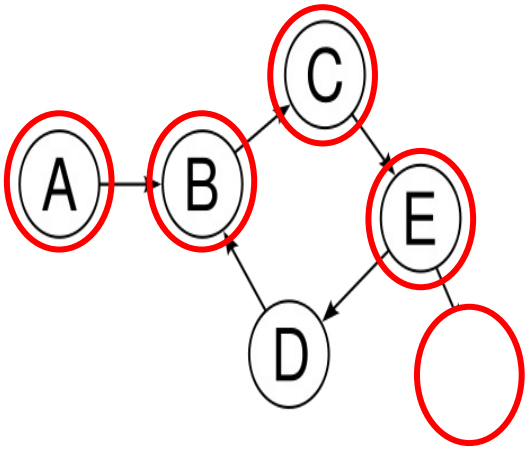
Graph Traversal - DFS

Stack



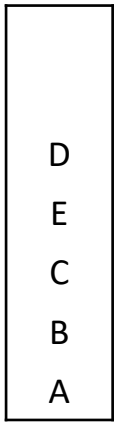
Pop

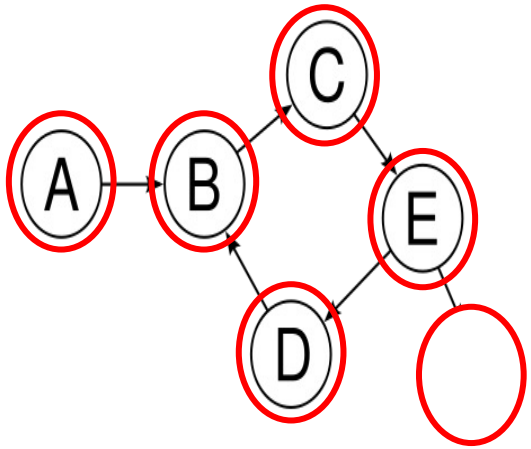
Output: A, B, C, E, F



Graph Traversal - DFS

Stack

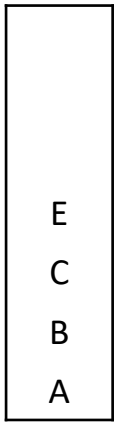




Output: A, B, C, E, F, D

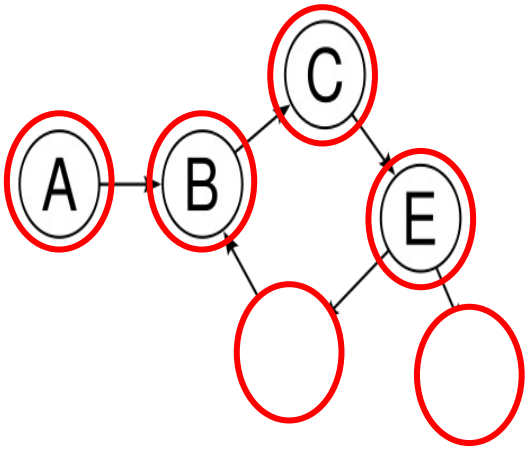
Graph Traversal - DFS

Stack



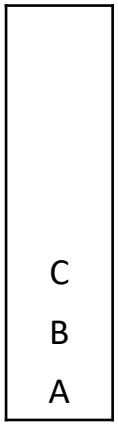
Pop

Output: A, B, C, E, F, D



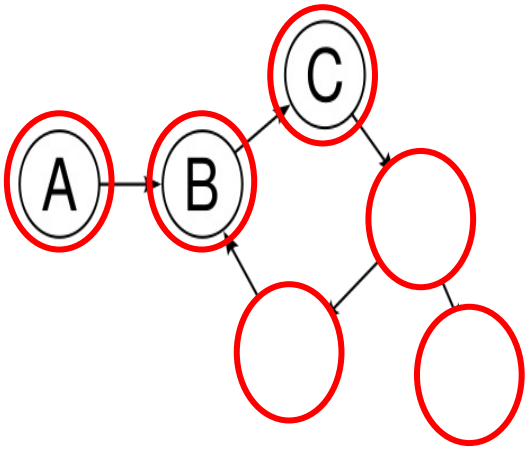
Graph Traversal - DFS

Stack



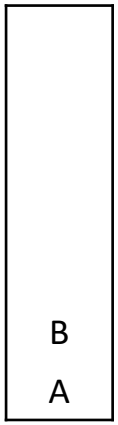
Pop

Output: A, B, C, E, F, D



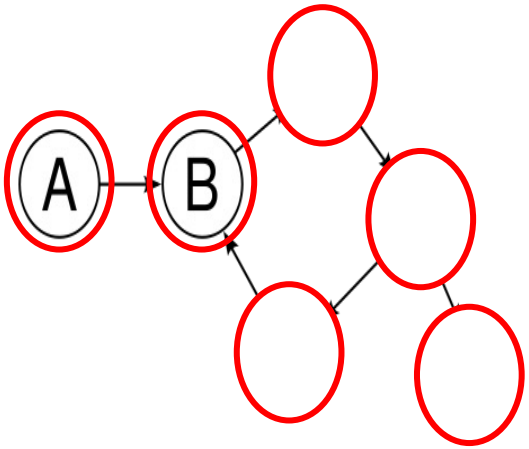
Graph Traversal - DFS

Stack



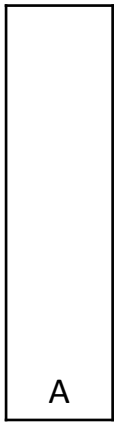
Pop

Output: A, B, C, E, F, D



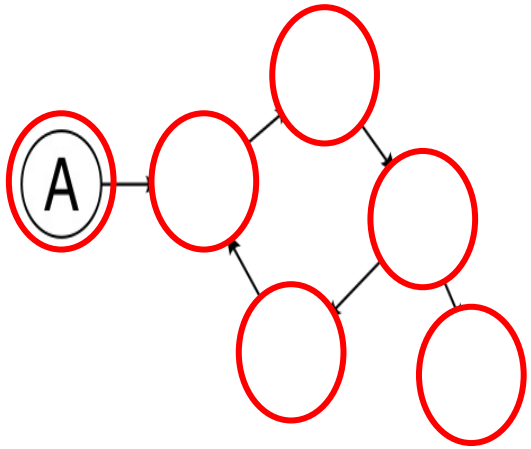
Graph Traversal - DFS

Stack



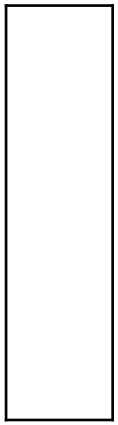
Pop

Output: A, B, C, E, F, D



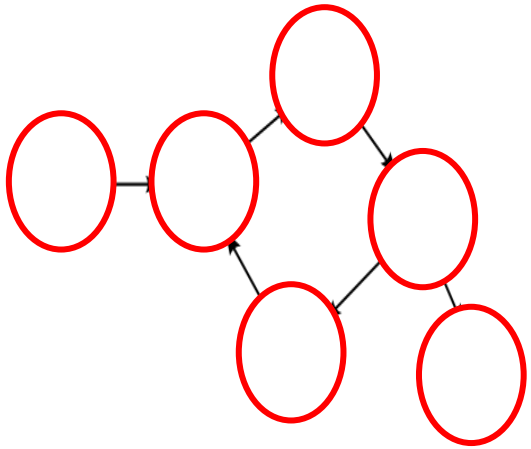
Graph Traversal - DFS

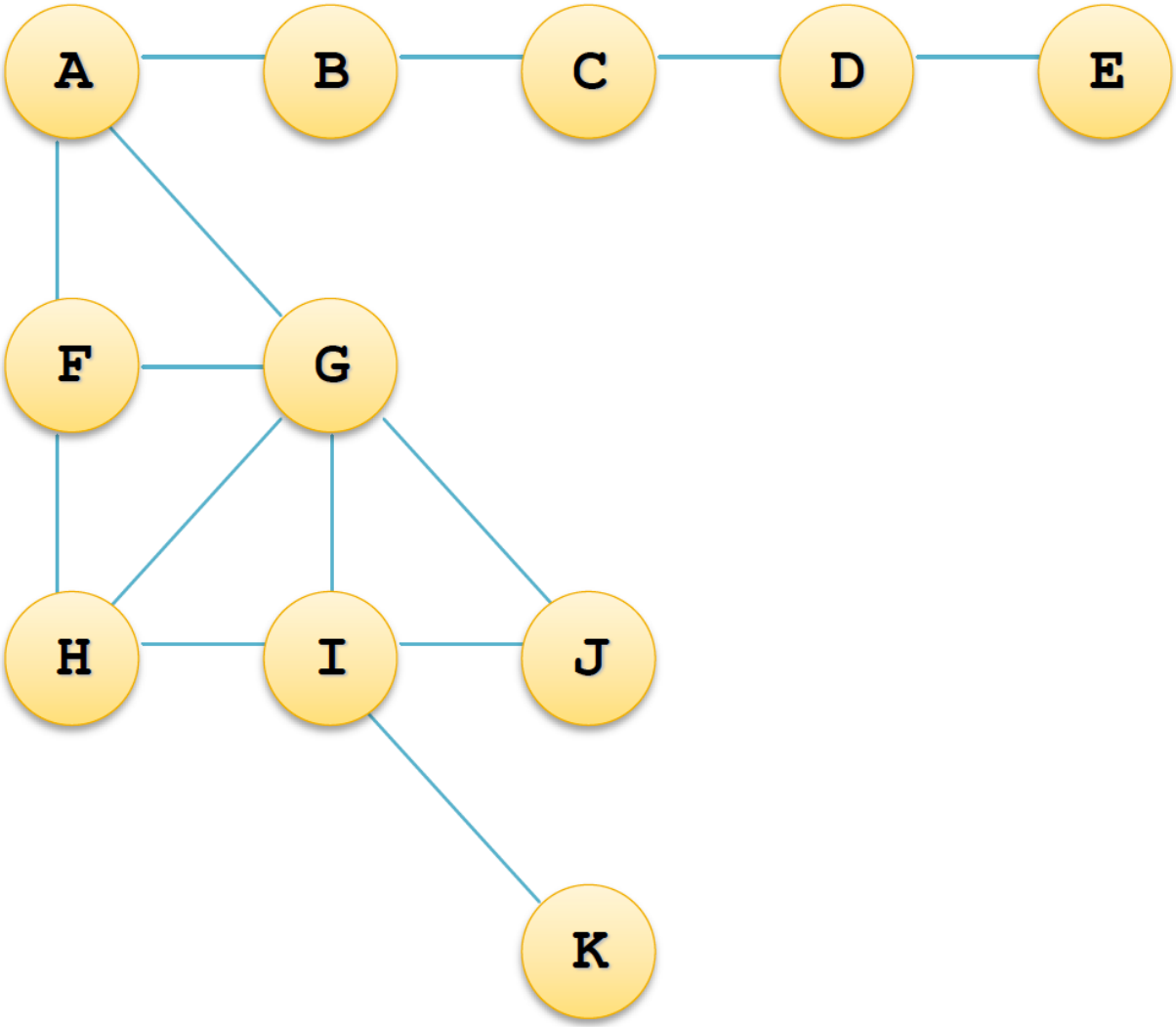
Stack



Done!

Output: A, B, C, E, F, D

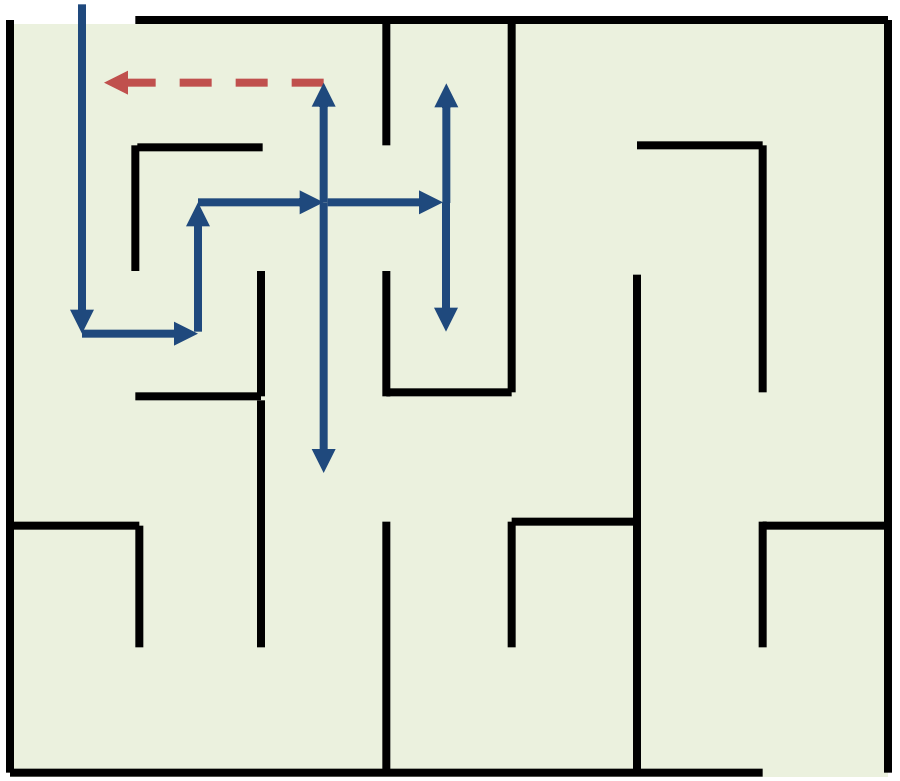




stack	visited
J K	A
E I	B
D H	C
C G	D
B F	E
A	F
	G
	H
	I
	J
	K

DFS and Maze Traversal

- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



Analysis of DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$

Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices ***u*** and ***z*** using the template method pattern
- We call ***DFS*(*G, u*)** with ***u*** as the start vertex
- We use a stack ***S*** to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex ***z*** is encountered, we return the path as the contents of the stack

```
Algorithm pathDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  if v = z
    return S.elements()
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        S.push(e)
        pathDFS(G, w, z)
        S.pop(e)
      else
        setLabel(e, BACK)
  S.pop(v)
```

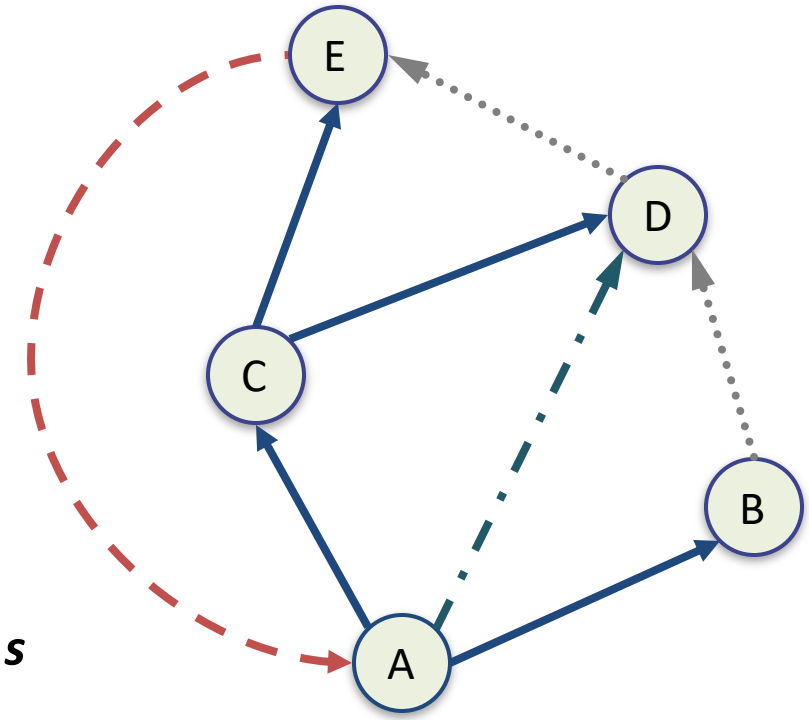
Cycle Finding

- We can use DFS to find a simple cycle
- We use a stack ***S*** to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (***v, w***) is encountered, we return the cycle as the portion of the stack from the top to vertex ***w***

```
Algorithm cycleDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            S.push(e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                pathDFS(G, w, z)
                S.pop(e)
            else
                T ← new empty stack
                repeat
                    o ← S.pop()
                    T.push(o)
                until o = w
                return T.elements()
    S.pop(v)
```

Directed DFS

- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- In the directed DFS algorithm, we have four types of edges
  - discovery edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex ***s*** determines the vertices reachable from ***s***



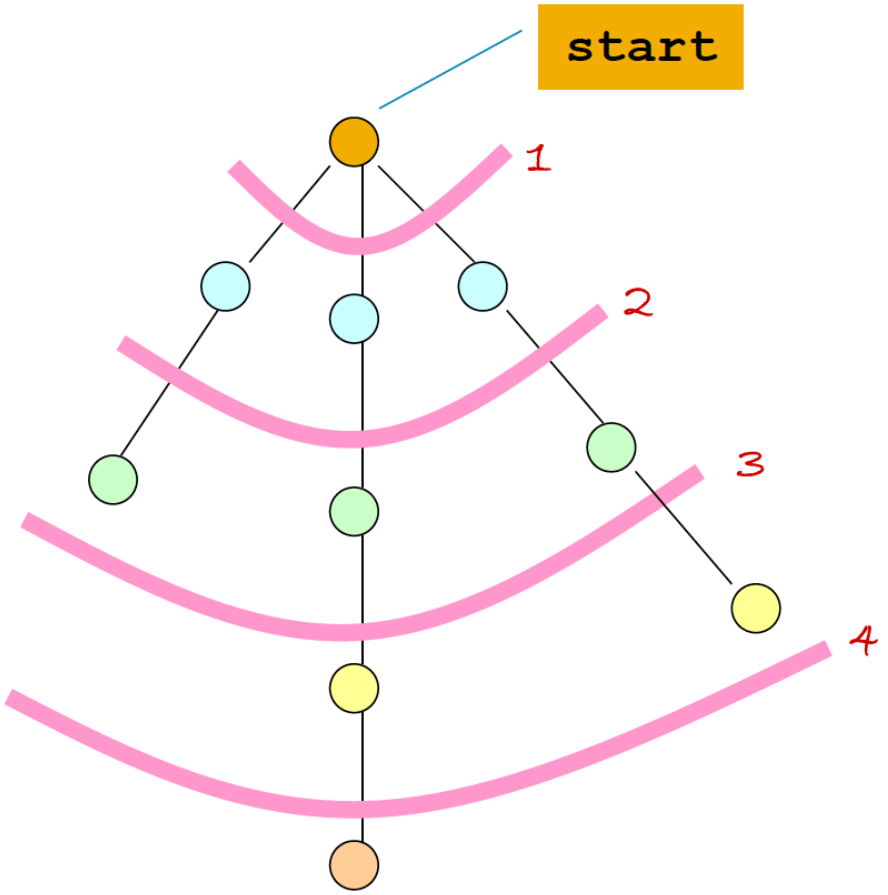
Breadth-First Search

- A general technique for traversing a graph G that
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G

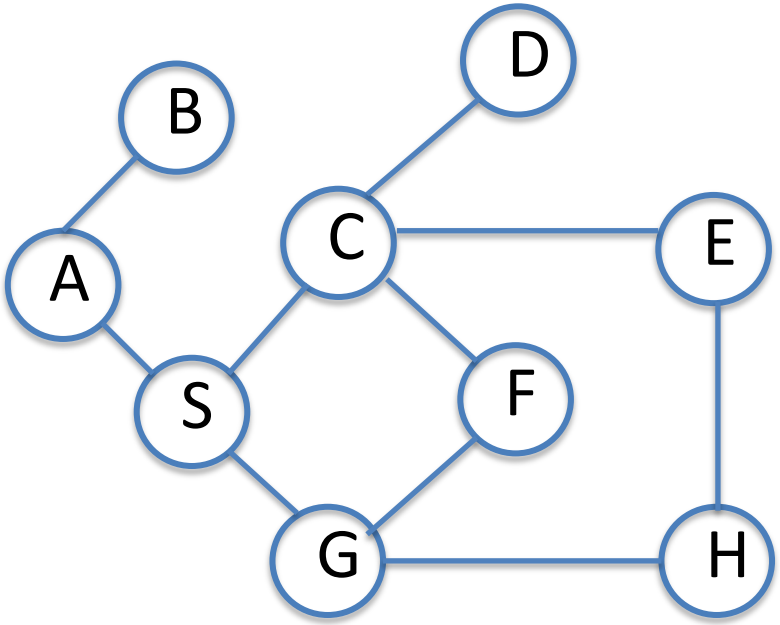
- For a graph with n nodes and m edges it takes $O(n + m)$

- BFS can be extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

- After visiting a vertex, *v*, visit every vertex adjacent to *v* before moving on
- Use a queue to store nodes
  - Queues are FIFO
- BFS:
  - visit and insert start
  - while (*q* not empty)
  - remove node from *q* and make it *current*
  - visit and insert the unvisited nodes adjacent to *current*



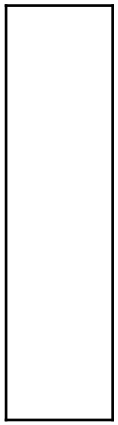
Graph Traversal - BFS

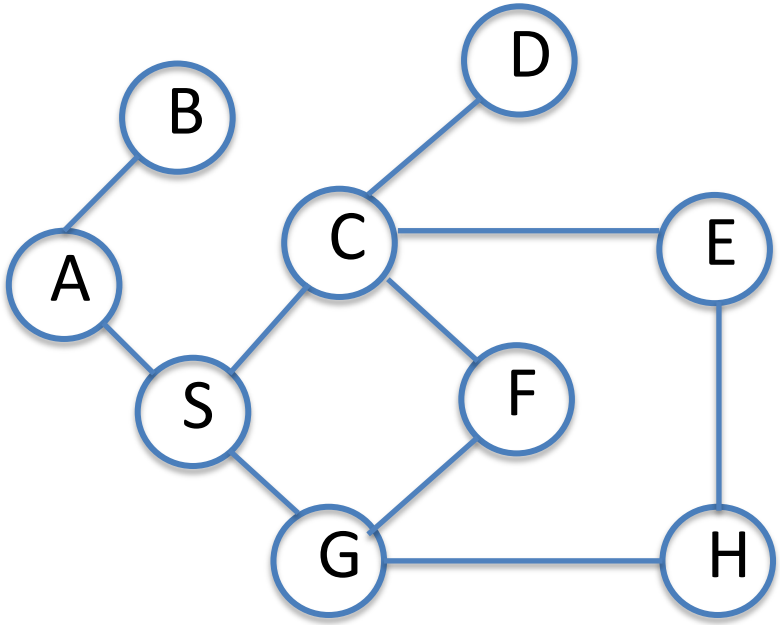


A node is fully explored before any other can begin.

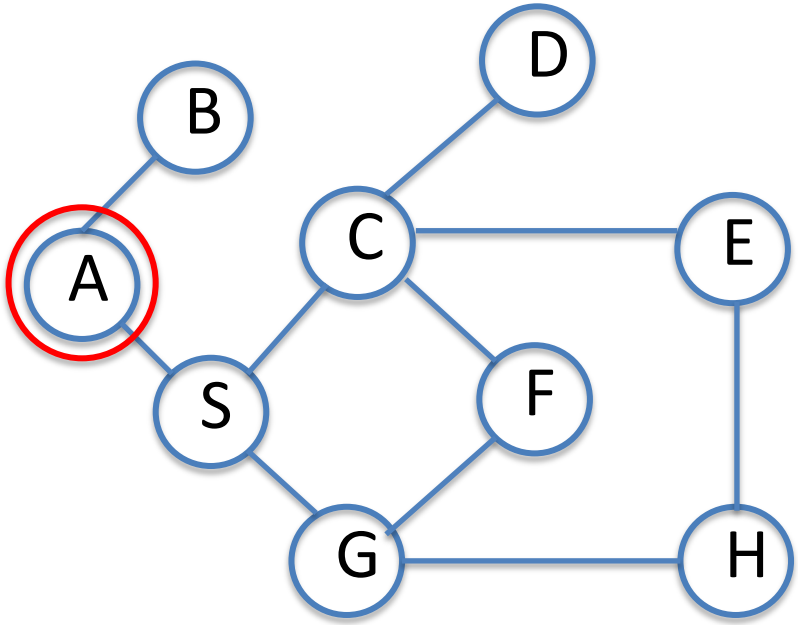
Graph Traversal - BFS

Queue

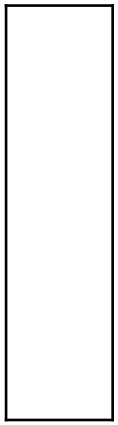




Graph Traversal - BFS

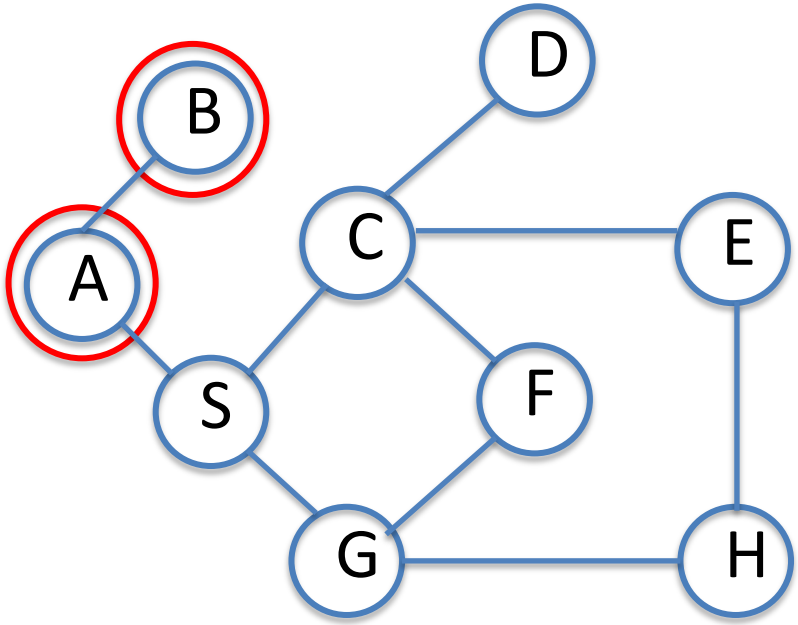


Queue

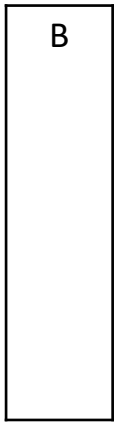


Output: A

Graph Traversal - BFS

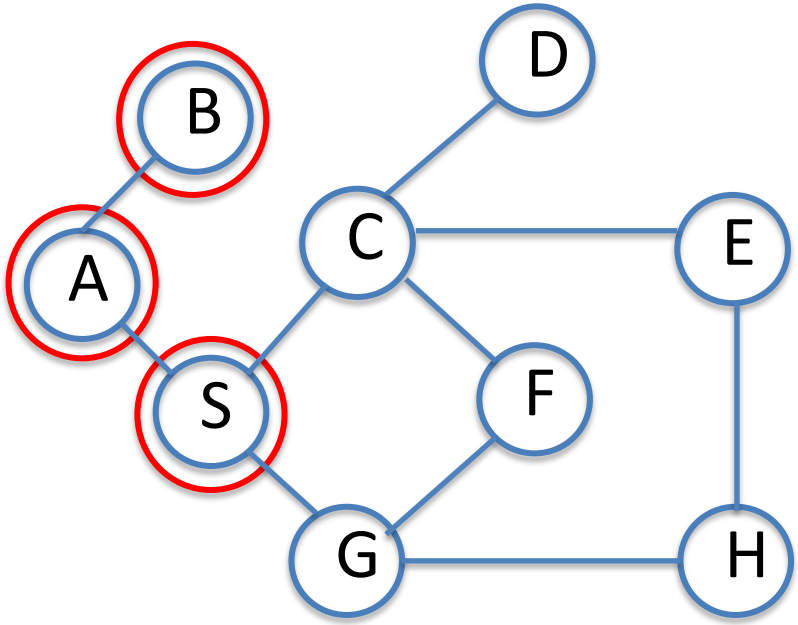


Queue



Output: A, B,

Graph Traversal - BFS



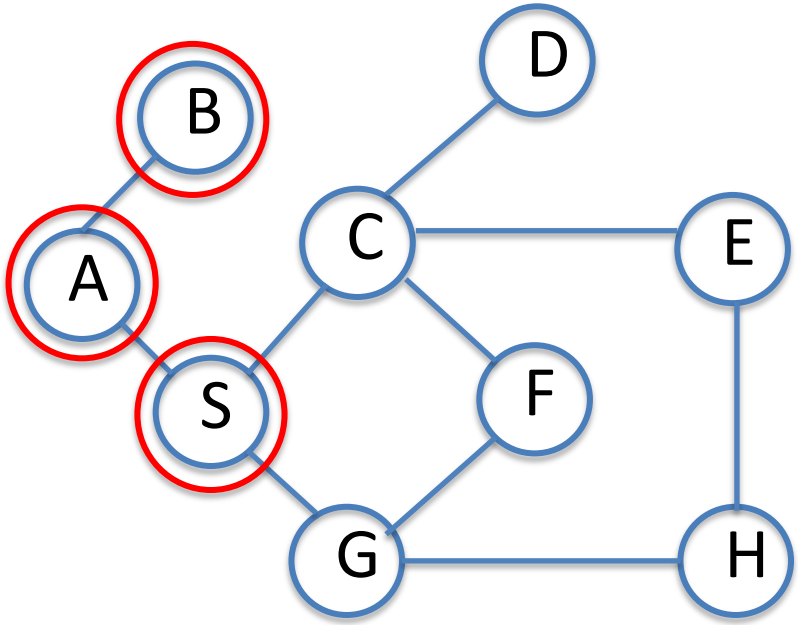
Queue

B

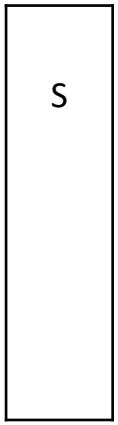
S

Output: A, B, S

Graph Traversal - BFS

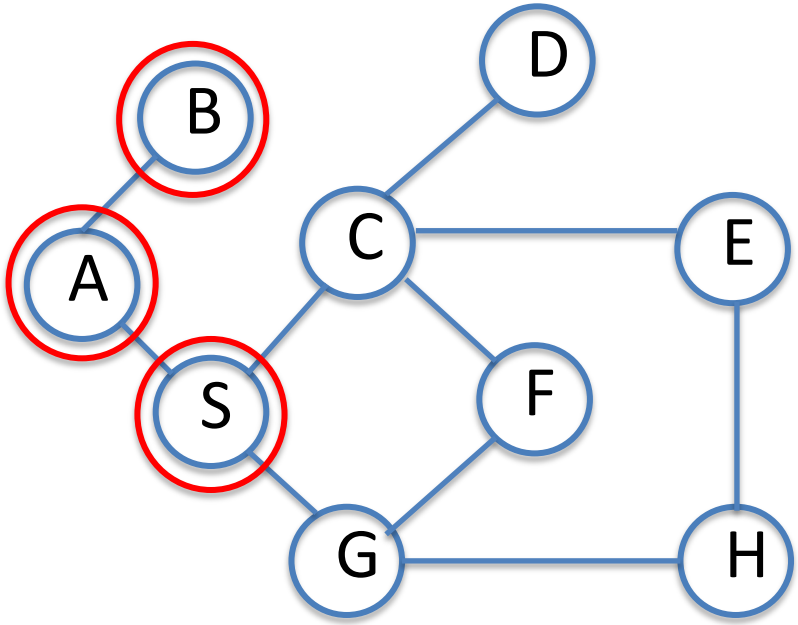


Queue

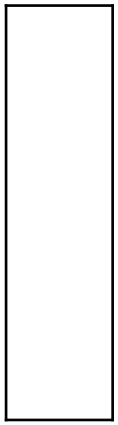


Output: A, B, S

Graph Traversal - BFS

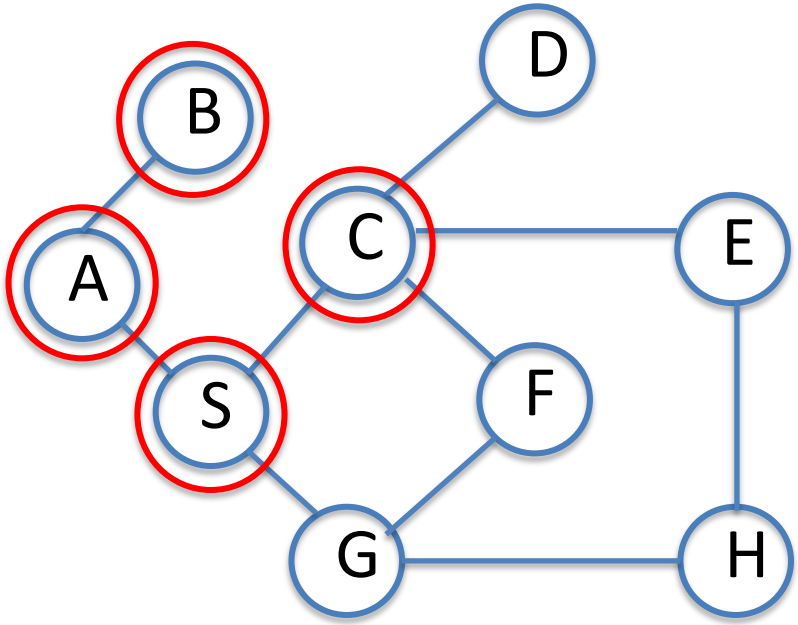


Queue

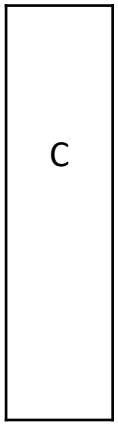


Output: A, B, S

Graph Traversal - BFS

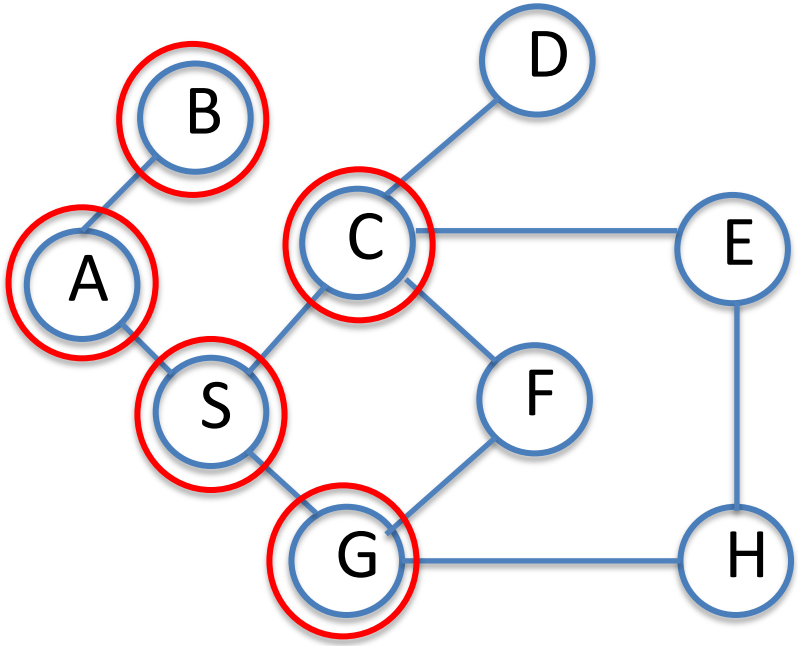


Queue



Output: A, B, S, C

Graph Traversal - BFS



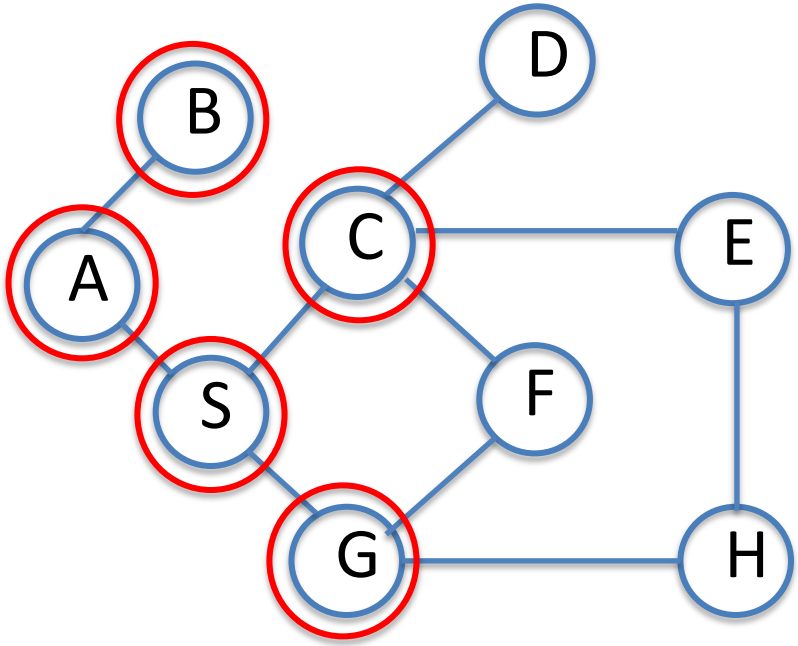
Queue

C

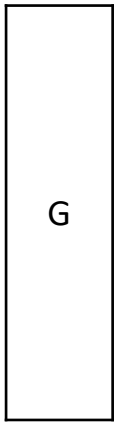
G

Output: A, B, S, C, G

Graph Traversal - BFS

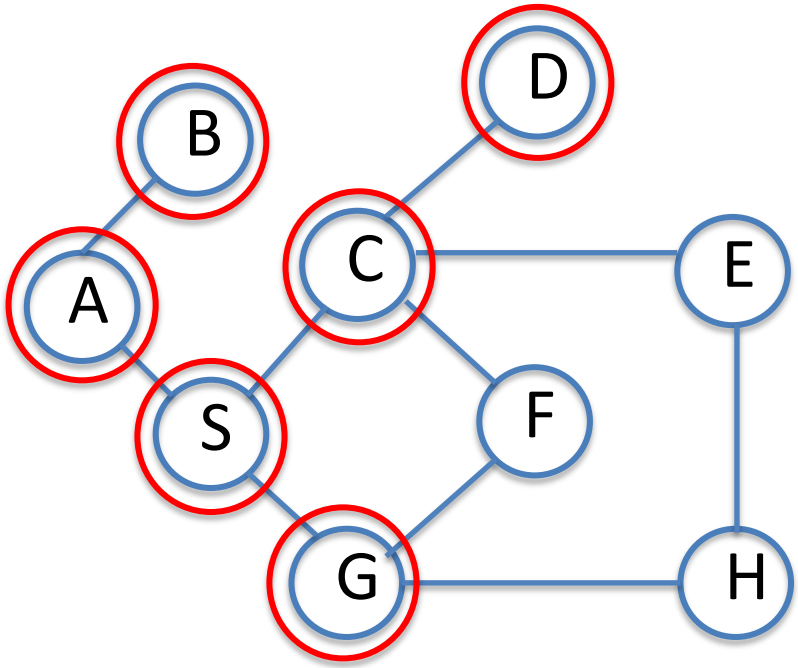


Queue



Output: A, B, S, C, G

Graph Traversal - BFS



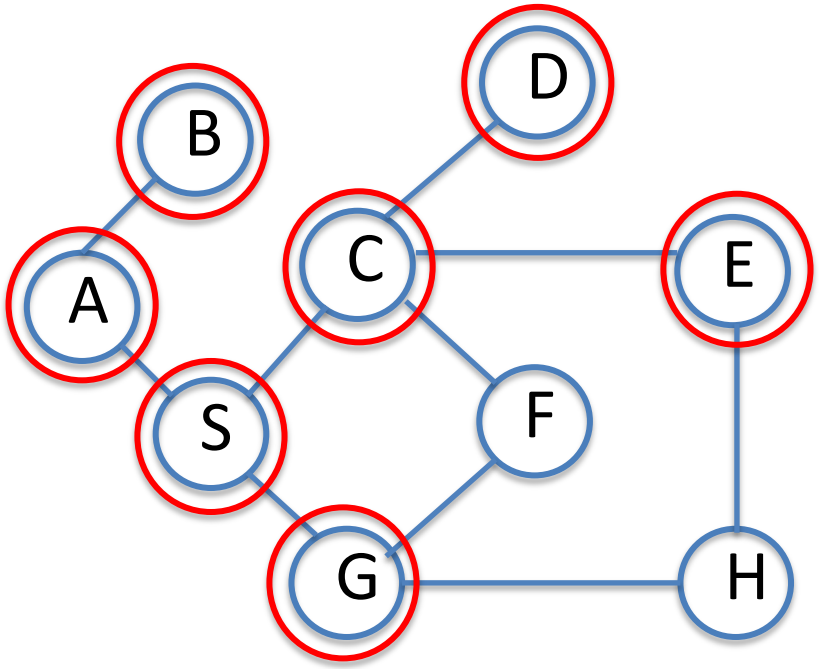
Queue

G

D

Output: A, B, S, C, G, D

Graph Traversal - BFS

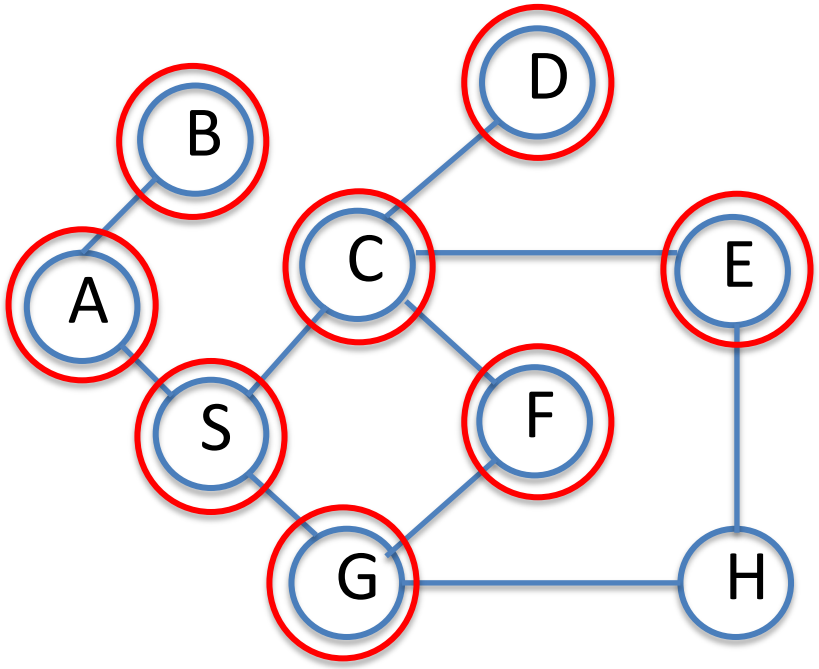


Queue

G
D
E

Output: A, B, S, C, G, D, E

Graph Traversal - BFS

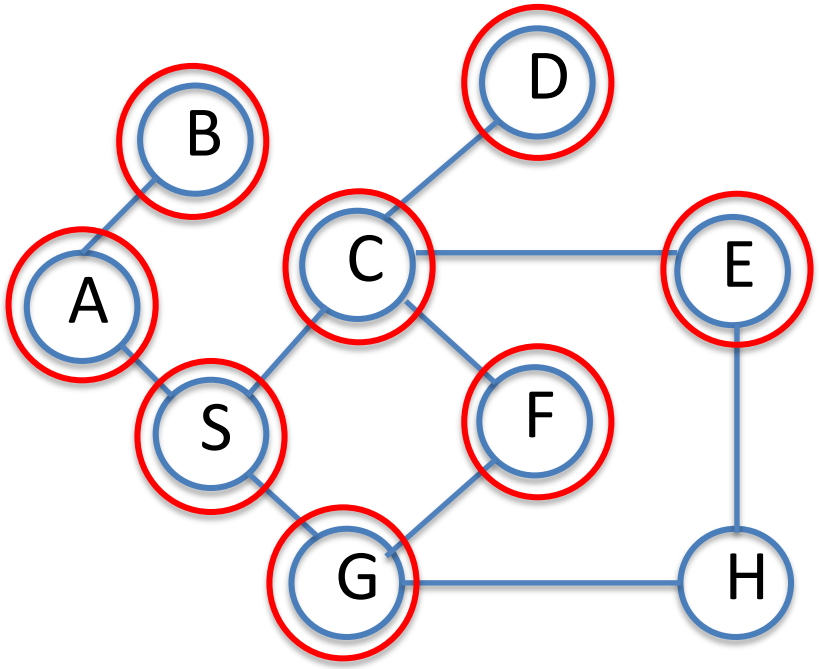


Queue

G
D
E
F

Output: A, B, S, C, G, D, E, F

Graph Traversal - BFS



Queue

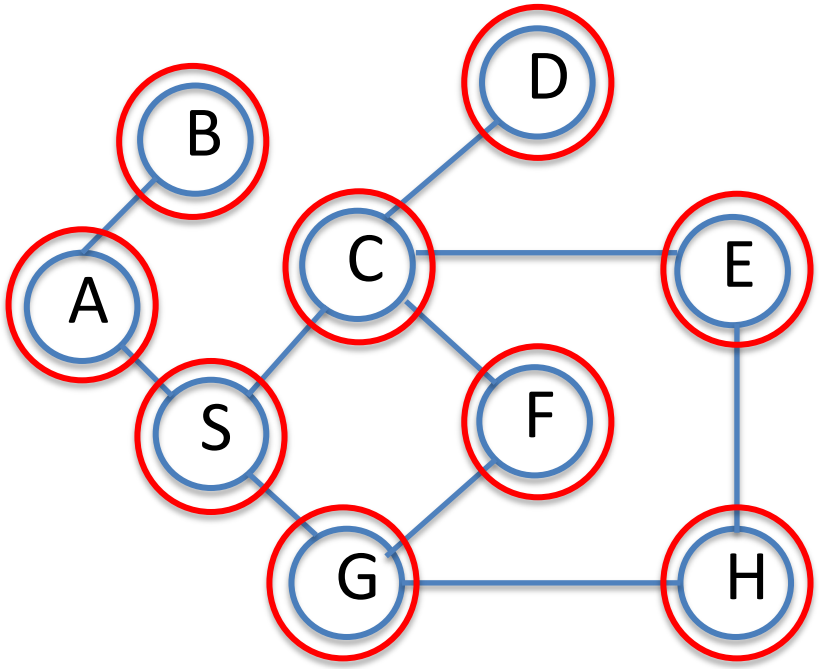
D

E

F

Output: A, B, S, C, G, D, E, F

Graph Traversal - BFS

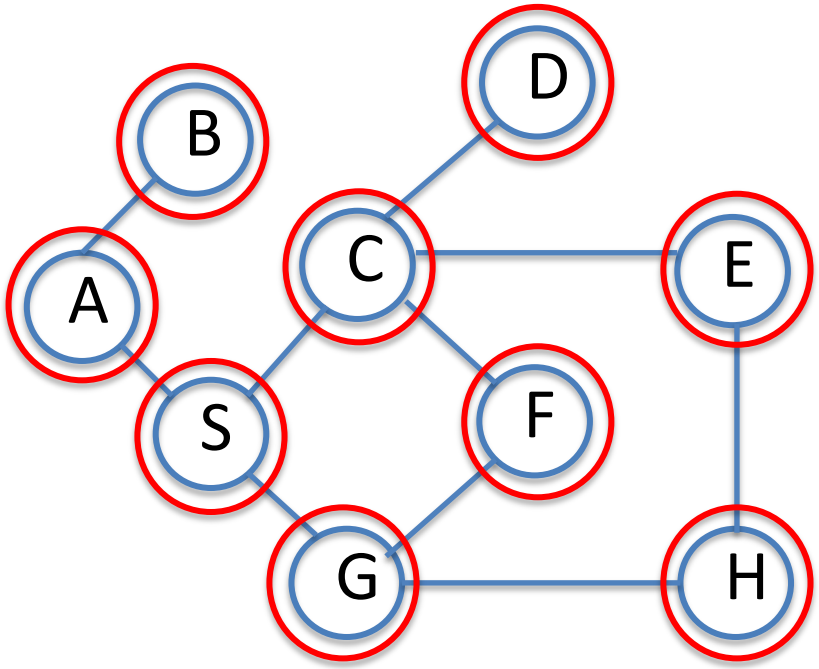


Queue

D
E
F
H

Output: A, B, S, C, G, D, E, F, H

Graph Traversal - BFS



Queue

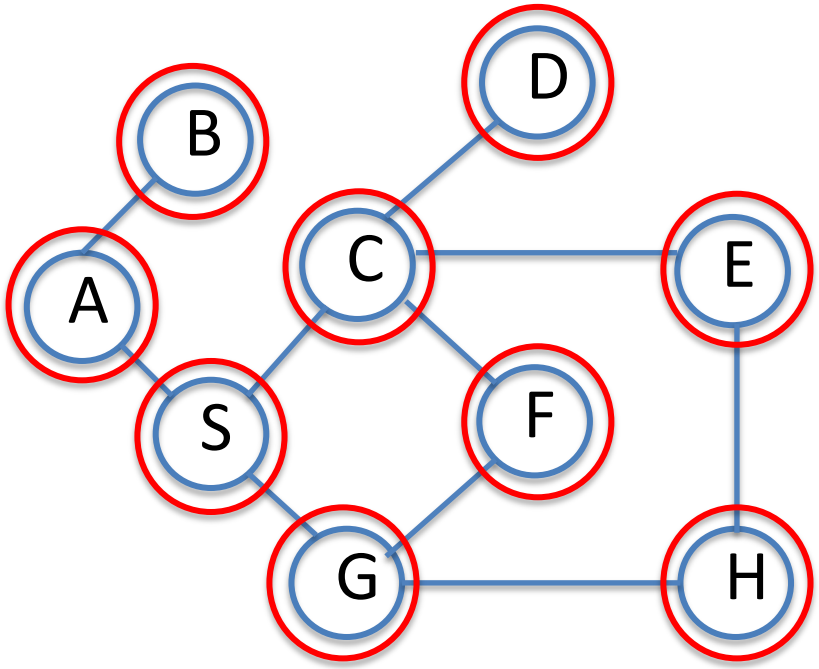
E

F

H

Output: A, B, S, C, G, D, E, F, H

Graph Traversal - BFS

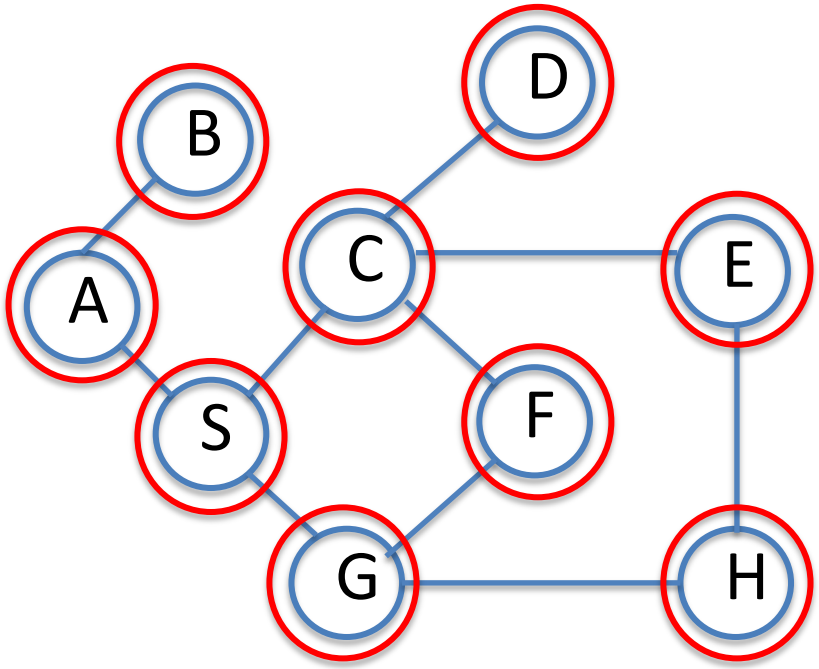


Queue

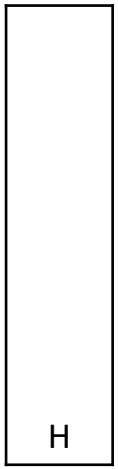
F
H

Output: A, B, S, C, G, D, E, F, H

Graph Traversal - BFS

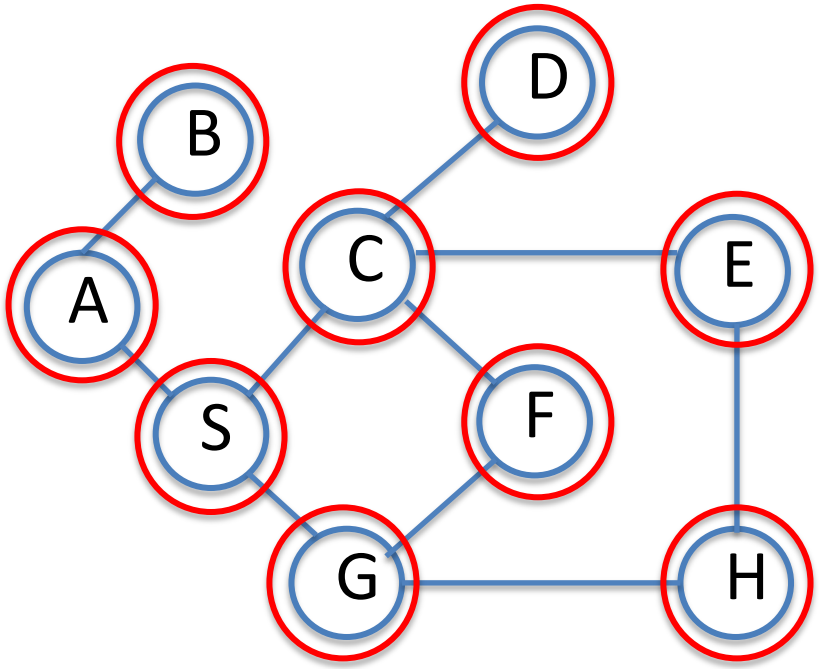


Queue

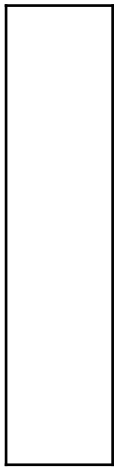


Output: A, B, S, C, G, D, E, F, H

Graph Traversal - BFS

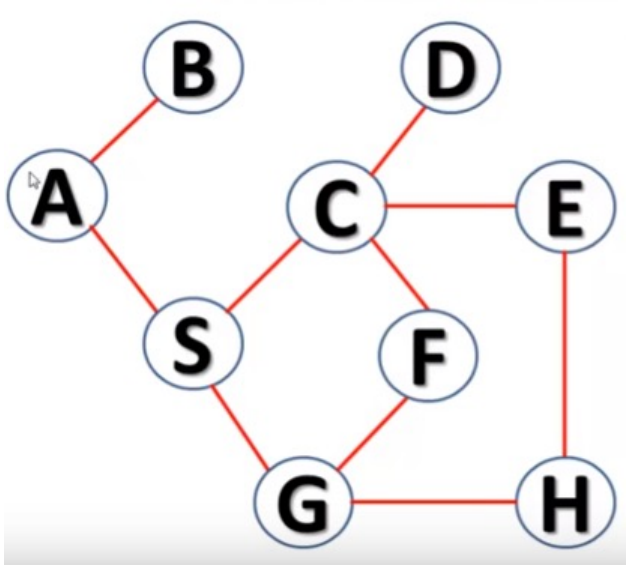


Queue



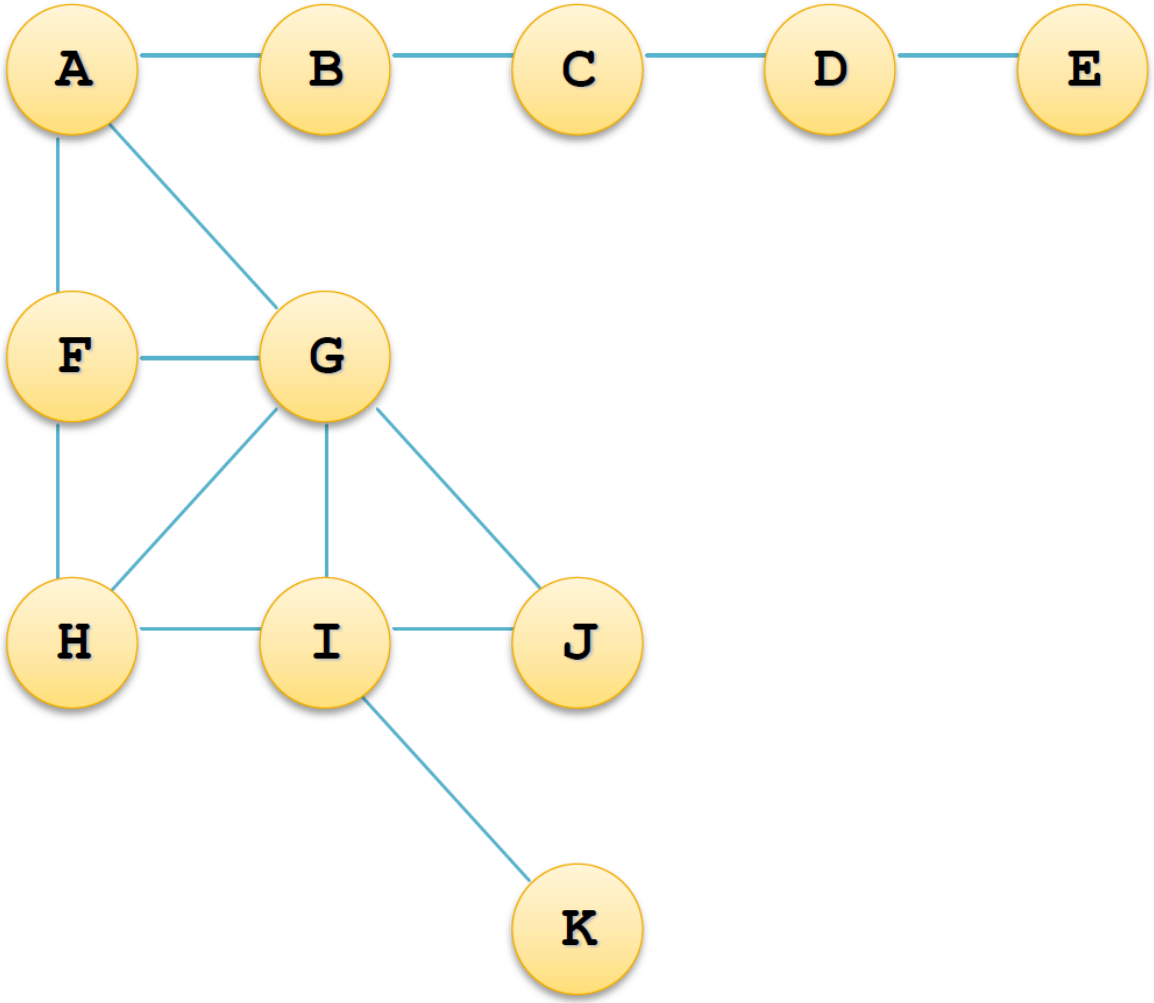
Output: A, B, S, C, G, D, E, F, H

DFS and BFS



DFS: A B S C D E H G F

BFS: A B S C G D E F H



queue	visited
A	A
B	B
F	F
G	G
C	C
H	H
I	I
J	J
D	D
K	K
E	E

BFS Algorithm

```
Algorithm BFS(G, s)
    L0 ← new empty sequence
    L0.addLast(s)
    setLabel(s, VISITED)
    i ← 0
    while Li.isNotEmpty()
        Li+1 ← new empty sequence
        for all v ∈ Li.elements()
            for all e ∈ G.incidentEdges(v)
                if getLabel(e) = UNEXPLORED
                    w ← opposite(v,e)
                    if getLabel(w) = UNEXPLORED
                        setLabel(e, DISCOVERY)
                        setLabel(w, VISITED)
                        Li+1.addLast(w)
                    else
                        setLabel(e, CROSS)
        i ← i +1
```

Analysis

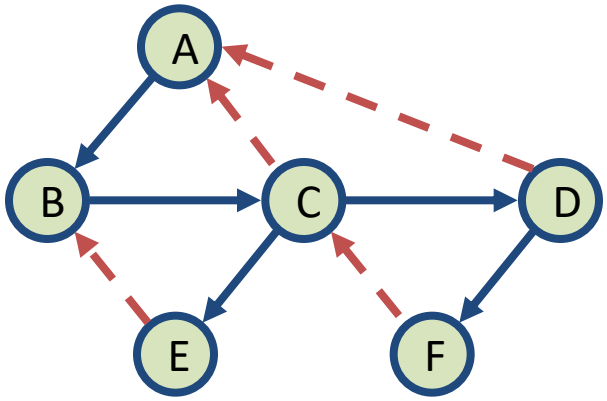
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$

Applications

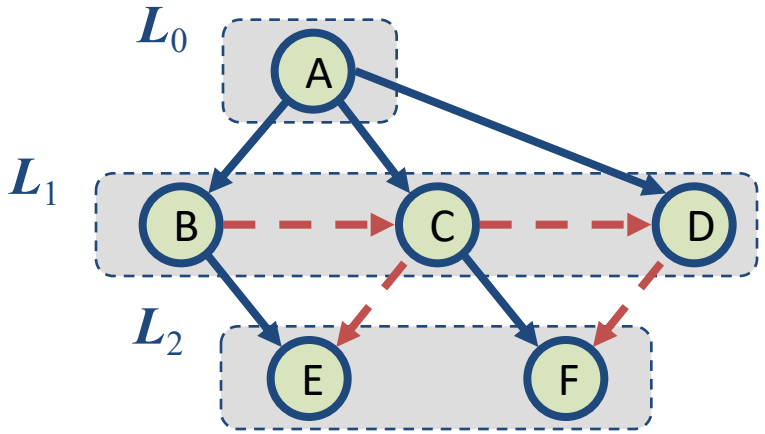
- Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	



DFS



BFS

BFS – Algorithm

1. Store source vertex S in a queue and mark as processed
2. while queue is not empty
   Read vertex v from the queue
   for all neighbors w:
   If w is not processed
      Mark as processed
      Append in the queue
      Record the parent of w to be v (necessary only if we need the shortest path tree)

Depth-First Search with Stack

Initialization:
mark all vertices as unvisited,
visit(s)

while the stack is not empty:
pop **(v,w)**
if w is not visited
add **(v,w)** to tree T
visit(w)

Procedure visit(v)
mark **v** as visited
for each edge **(v,w)**
push **(v,w)** in the stack