

# Chapter 6

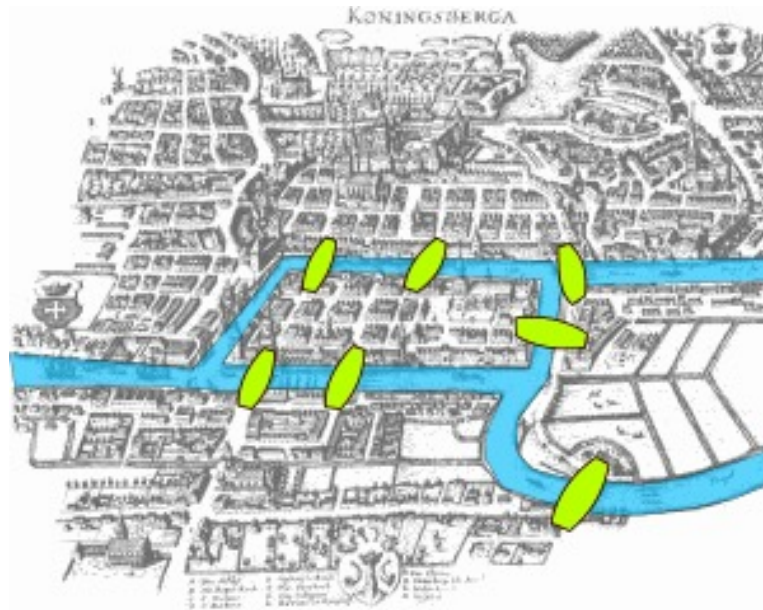
## Introduction to Graphs

# Topics to be Covered

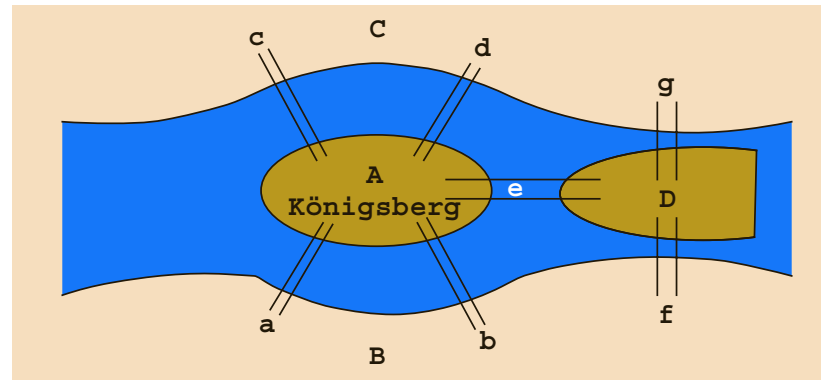
- Learn about graphs
- Become familiar with the basic terminology of graph theory
- Discover how to represent graphs in computer memory
- Examine and implement various graph traversal algorithms
- Examine and implement some applications of graph theory

# Introduction

- In 1736, in the town of Königsberg in Prussia, the river Pregel flows around the island Kneiphof and then divides into two branches



# Introduction

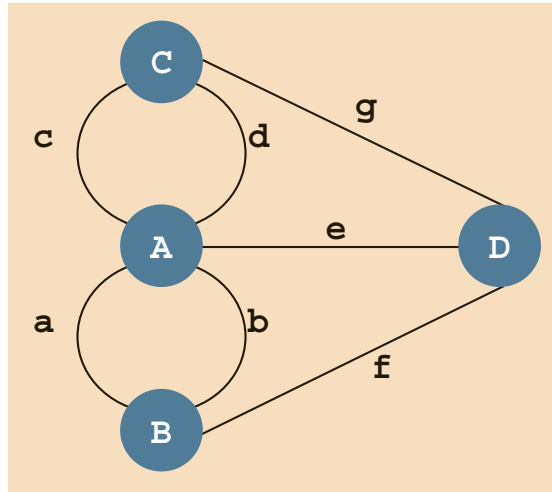


- The river has four land areas: A, B, C, and D
- These land areas are connected using seven bridges that are labeled  $a$ ,  $b$ ,  $c$ ,  $d$ ,  $e$ ,  $f$ , and  $g$
- *The Königsberg bridge problem*

Starting at one land area, is it possible to walk across all of the bridges exactly once and return to the starting land area?

# Introduction

- In 1736, Euler represented the *Königsberg bridge problem* as a *graph* ...



... and answered the question in the negative

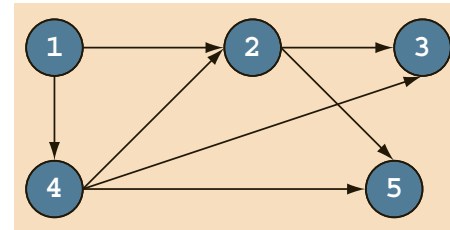
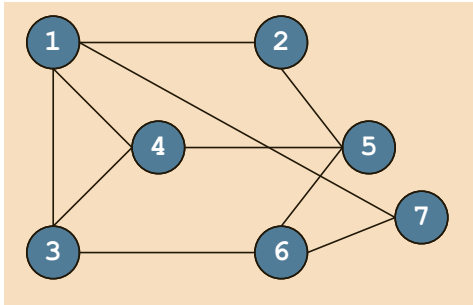
➡ This marked the birth of *graph theory*

# Graph Definitions and Notations

- A *graph*  $G$  is a pair,  $G = (V, E)$ , in which  $V$  is the set of vertices of  $G$ , and  $E$  is the set of edges in  $G$
- If the elements of  $E$  are ordered pairs,  $G$  is called a *directed graph* (or *digraph*); otherwise,  $G$  is called an *undirected graph*
  - In an undirected graph, the pairs  $(u, v)$  and  $(v, u)$  represent the same edge
  - If  $(u, v)$  is an edge in a digraph, the vertex  $u$  is called the *origin* of the edge, and the vertex  $v$  is called the *destination*

# Graph Definitions and Notations

- A graph can be shown pictorially



- A graph is called a *weighted graph* if its edges are labeled with numeric values
- A graph  $H = (V_H, E_H)$  is called a *subgraph* of  $G$  if

$$V_H \subseteq V, E_H \subseteq E$$

# Graph Definitions and Notations

Let  $G$  be an undirected graph;  $u, v$  be two vertices of  $G$

- $u$  and  $v$  are called *adjacent* if there is an edge from one to the other; that is,  $(u, v) \in E$
- Let  $e = (u, v) \in E$ . Edge  $e$  is *incident on* the vertices  $u$  and  $v$
- An edge incident on a single vertex is called a *loop*
- If two edges,  $e_1$  and  $e_2$ , are associated with the same pair of vertices, then  $e_1$  and  $e_2$  are called *parallel edges*



# Graph Definitions and Notations

- A graph is called a *simple graph* if it has no loops and no parallel edges
- There is a *path* from  $u$  to  $v$  if there is a sequence of vertices  $u_1, u_2, \dots, u_n$  such that  $u = u_1, v = u_n$ , and  $(u_i, u_{i+1})$  is an edge for all  $i = 1, 2, \dots, n - 1$
- A *simple path* from  $u$  to  $v$  is a path from  $u$  to  $v$  with no repeated vertices
- A *cycle* is a simple path in which the first and last vertices are the same

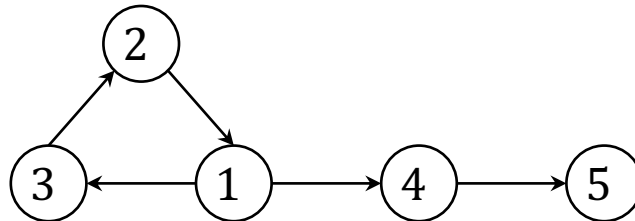
# Graph Definitions and Notations

- Vertices  $u$  and  $v$  are called *connected* if there is a path from  $u$  to  $v$
- $G$  is called *connected* if there is a path from any vertex to any other vertex
  - A subset of connected vertices is called a *connected component* of  $G$

# Graph Definitions and Notations

Let  $G$  be a directed graph;  $u, v$  be two vertices in  $G$

- If  $(u, v) \in E$  then we say that  $u$  is *adjacent to*  $v$  and  $v$  is *adjacent from*  $u$
- $G$  is called *strongly connected* if any two vertices in  $G$  are connected
  - A *strongly connected component* of  $G$  is a maximal strongly connected subgraph



# Graph Definitions and Notations

Let  $G$  be a directed graph;  $u, v$  be two vertices in  $G$

- If  $(u, v) \in E$  then we say that  $u$  is *adjacent to*  $v$  and  $v$  is *adjacent from*  $u$
- $G$  is called *strongly connected* if any two vertices in  $G$  are connected
  - A *strongly connected component* of  $G$  is a maximal strongly connected subgraph
- The *outdegree* of  $v$  is the number of directed edges leaving  $v$ ; the *indegree* of  $v$  is the number of directed edges entering  $v$

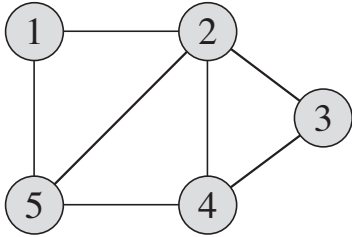
# Graph Representation

- A graph can be represented in several ways
  - How a graph is represented in memory depends on the specific application
- The most two common methods are *adjacency matrices* and *adjacency lists*
- Let  $G = (V, E)$  be a graph with  $n(= |V|)$  vertices
  - Let  $V = \{v_1, v_2, \dots, v_n\}$

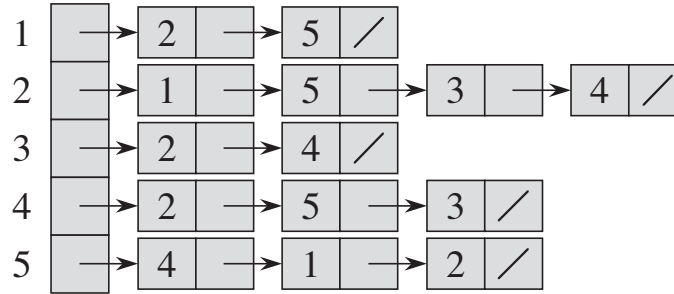
# Graph Representation: Adjacency Matrices

- The *adjacency matrix*  $A$  of  $G$  is a two-dimensional  $n \times n$  matrix such that
  - If there is an edge from  $v_i$  to  $v_j$ , the  $(i, j)^{th}$  entry of  $A$  is 1
  - Otherwise, the  $(i, j)^{th}$  entry is 0
- In an undirected graph, if  $(v_i, v_j) \in E$  then  $(v_j, v_i) \in E$ , so the  $(i, j)^{th}$  entry is as same as the  $(j, i)^{th}$  entry
  - ➡ The adjacency matrix of an undirected graph is symmetric

# Graph Representation: Examples



(a)

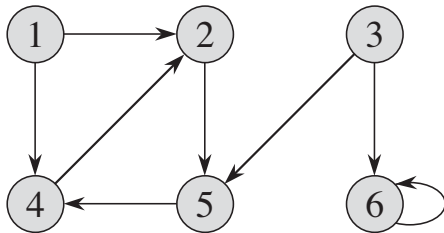


(b)

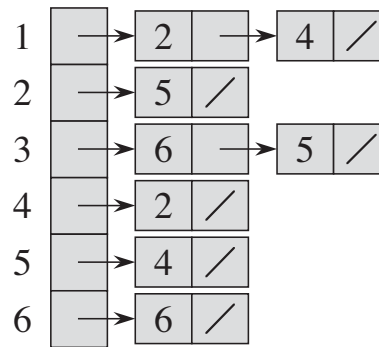
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Two representations of an undirected graph



(a)



(b)

	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)

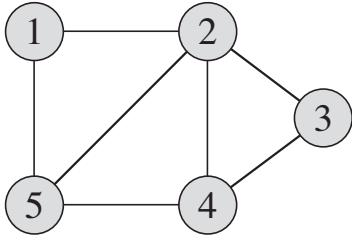
Two representations of a directed graph

# Graph Representation: Adjacency Lists

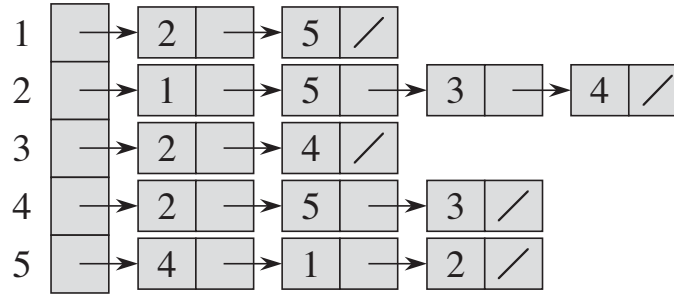
- With *adjacency lists*, corresponding to each vertex,  $u$ , there is a linked list such that each node of the linked list contains the vertex,  $v$ , such that  $(u, v) \in E$
- Technically, we use an array  $A$  of size  $n$ , such that  $A[i]$  is
  - a representation of the vertex  $v_i$
  - a *pointer* to the first node of the linked list containing the vertices *to which  $v_i$  is adjacent*



# Graph Representation: Examples



(a)

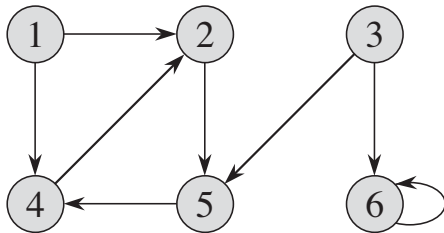


(b)

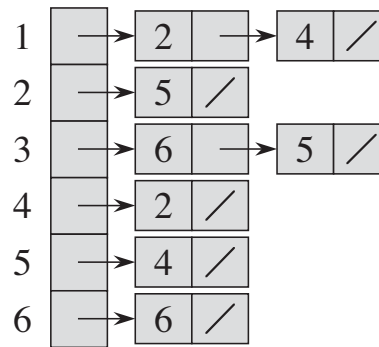
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Two representations of an undirected graph



(a)



(b)

	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

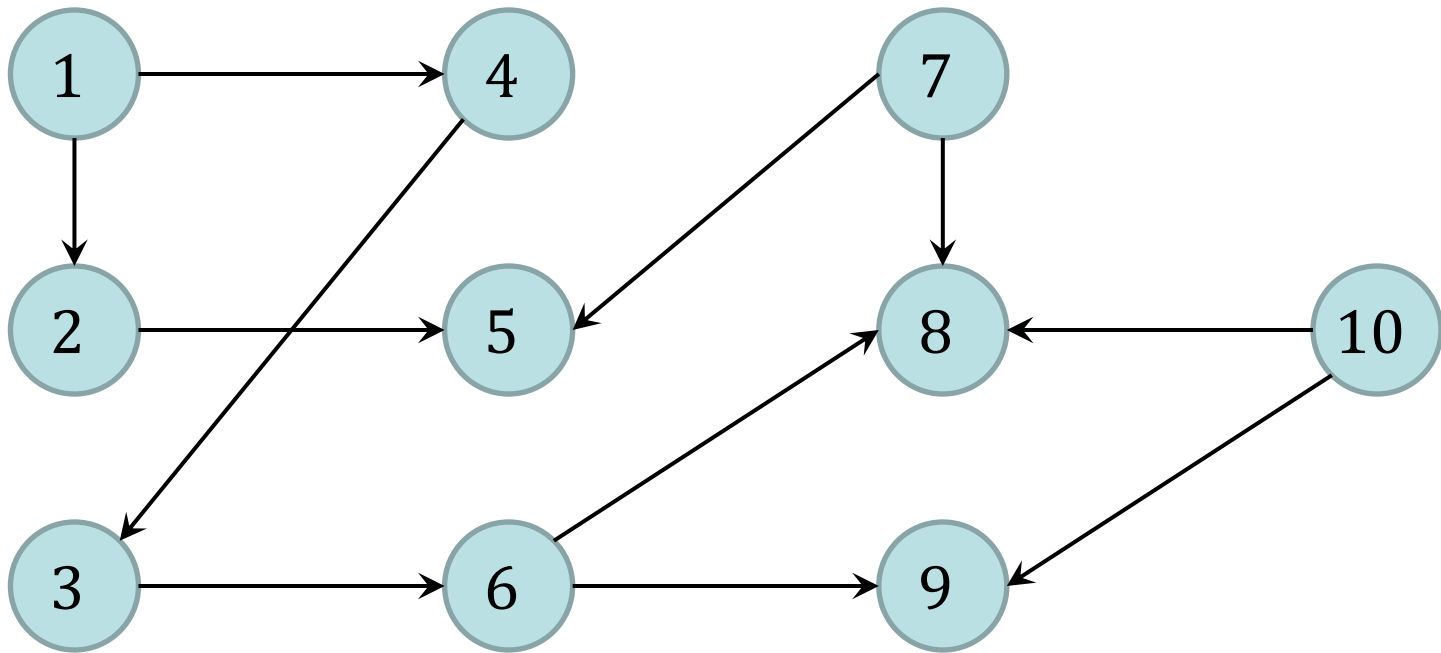
(c)

Two representations of a directed graph

# Graph Traversals

- Traversing a graph is a bit more complicated than traversing a binary tree
  - A binary tree has no cycles, but a graph might have cycles
  - We might not be able to traverse the entire graph from a single vertex
- In order to traverse the entire graph, we must ...
  - keep track of the vertices that have been visited
  - traverse the graph from each unvisited vertex

# Depth First Traversal



*A depth first ordering of the vertices of the graph is*

1, 2, 5, 4, 3, 6, 8, 9, 7, 10

# Depth First Traversal: A Recursive Algorithm

$\text{dft}(v)$  // *DFT starts at the vertex  $v$*

Visit  $v$  and mark it as visited

for each vertex  $u$  to which  $v$  is adjacent

if  $u$  is an unvisited vertex

$\text{dft}(u)$

Mark each vertex in  $V$  as unvisited

for each vertex  $v$  in  $V$

if  $v$  is an unvisited vertex

$\text{dft}(v)$

# Depth First Traversal: Pseudo-code

```
dft(graph, v, visited) {  
    visited[v] = true;  
    cout << v;  
    for (p = graph[v]; p; p = p->next;)  
        if (!visited[p->vertex])  
            dft(graph, p->vertex, visited);  
}
```

```
visited[1 .. n] = false;  
for (v = 1; v ≤ n; v++)  
    if (!visited[v])  
        dft(graph, v, visited);
```

# Depth First Traversal: A Non-recursive Algorithm

Mark each vertex in  $V$  as unvisited  
for each vertex  $v$  in  $V$

    if  $v$  is an unvisited vertex

        Push  $v$  onto the empty stack

    while the stack is not empty

        Pop  $u$  off the stack

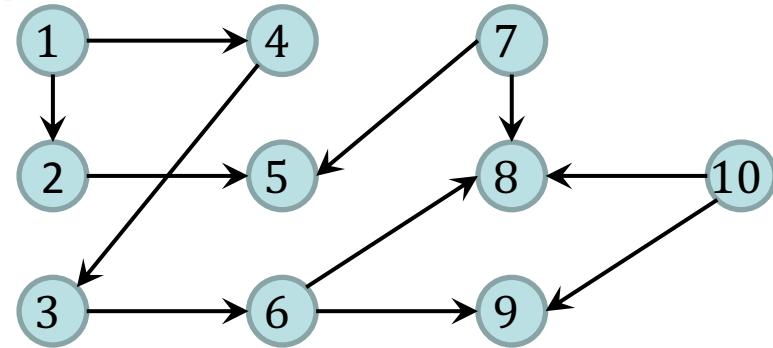
        if  $u$  is an unvisited vertex

            Visit  $u$  and mark it as visited

        for each vertex  $w$  to which  $u$  is adjacent

            if  $w$  is an unvisited vertex

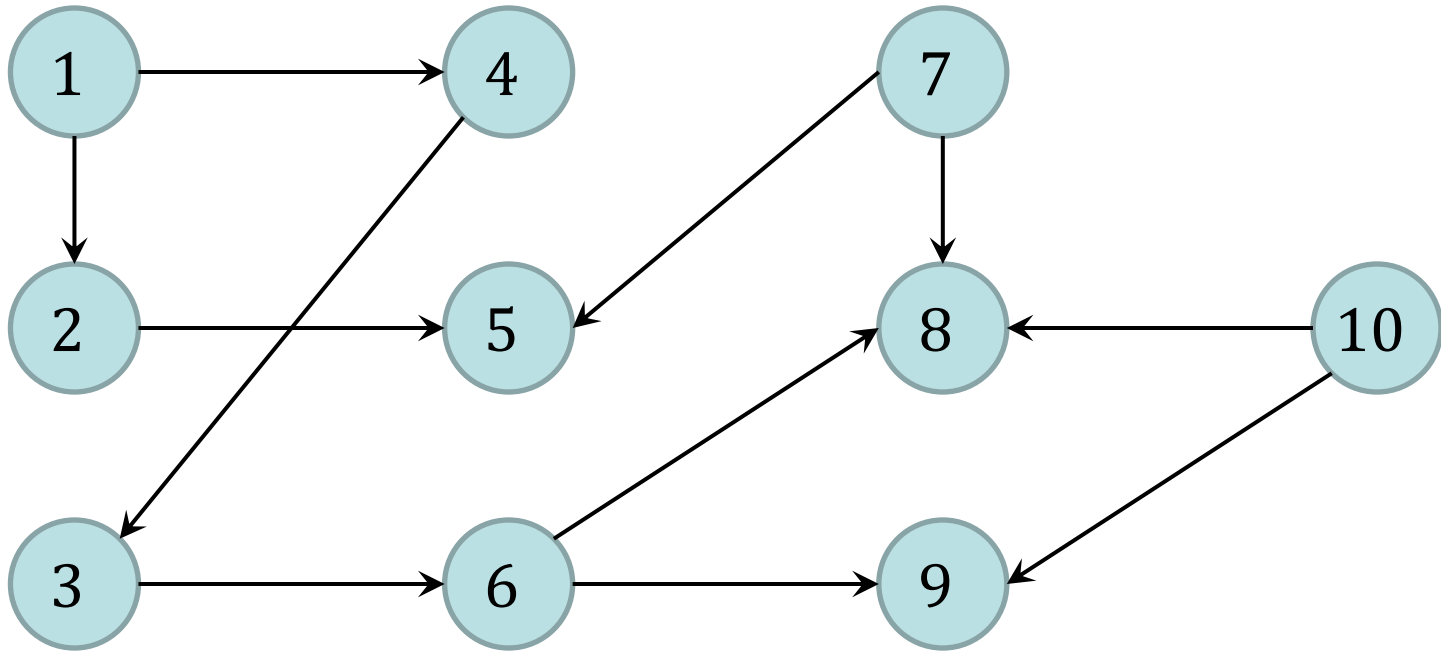
                Push  $w$  onto the stack



# Depth First Traversal: Pseudo-code

```
dft(graph) {  
    visited[1 .. n] = false;  
    for (v = 1; v ≤ n; v++)  
        if (!visited[v]) {  
            push(S, v);  
            while (!isEmpty(S)) {  
                u = pop(S);  
                if (!visited[u]) {  
                    cout << u;  
                    visited[u] = true;  
                    for (p = graph[u]; p; p = p->next)  
                        if (!visited[p->vertex])  
                            push(S, p->vertex);  
                }  
            }  
        }  
}
```

# Breadth First Traversal



*A breadth first ordering of the vertices of the graph is*

1, 2, 4, 5, 3, 6, 8, 9, 7, 10



# Breadth First Traversal: Algorithm

Mark each vertex in  $V$  as unvisited

**for** each vertex  $v$  in the graph

**if**  $v$  is an unvisited vertex

        Add  $v$  to the queue

        Mark  $v$  as visited

**while** the queue is not empty

        Extract  $u$  from the queue

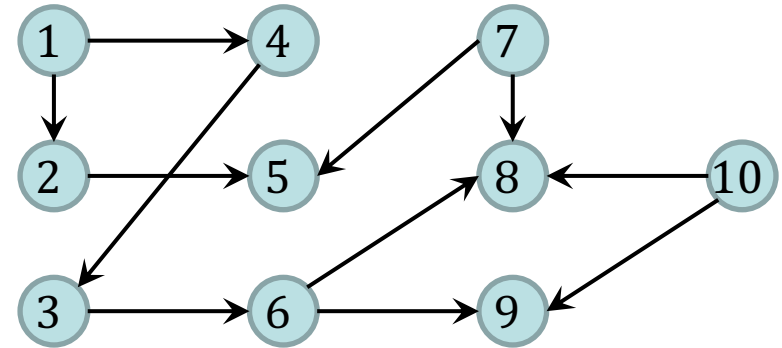
        Visit  $u$

**for** each vertex  $w$  to which  $u$  is adjacent

**if**  $w$  is an unvisited vertex

                Add  $w$  to the queue

                Mark  $w$  as visited



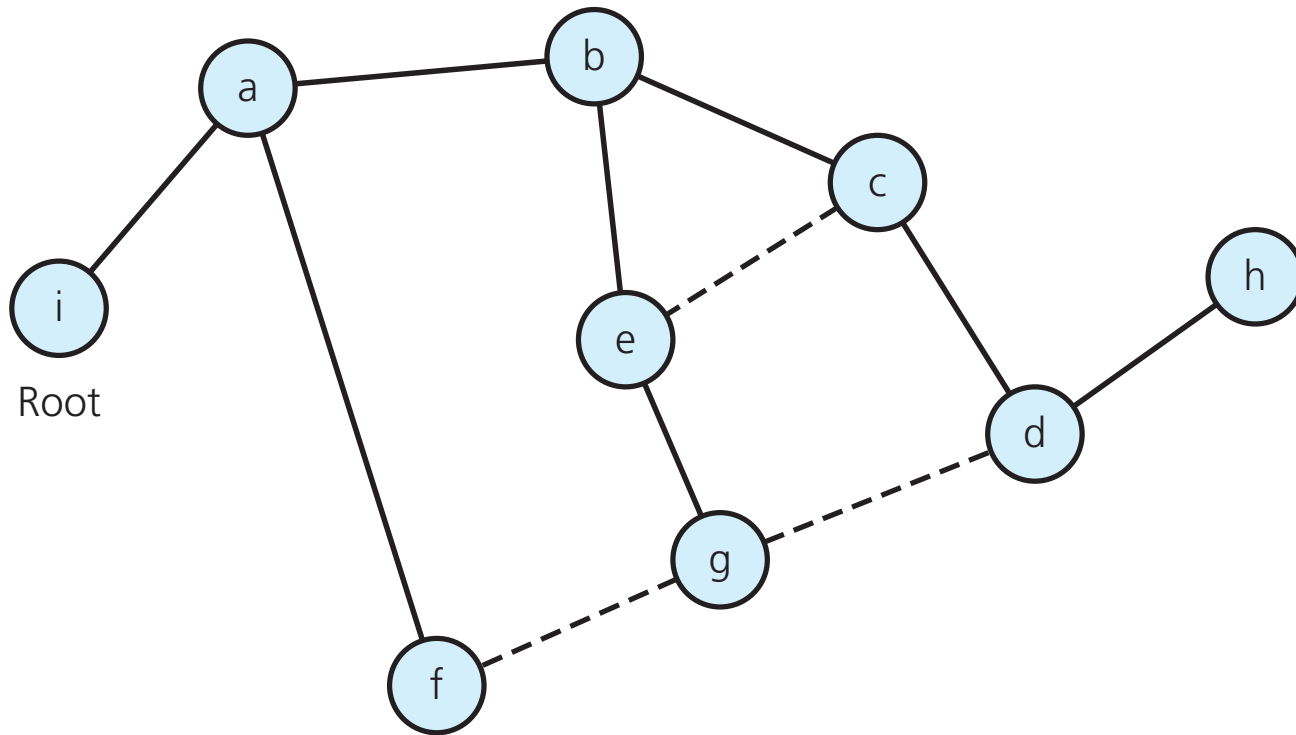
# Breadth First Traversal: Pseudo-code

```
bft(graph) {  
    visited[1 .. n] = false;  
    for (v = 1; v ≤ n; v++)  
        if (!visited[v]) {  
            enqueue(Q, v);  
            visited[v] = true;  
            while (!isEmpty(Q)) {  
                u = dequeue(Q);    cout << u;  
                for (p = graph[u]; p; p = p->next)  
                    if (!visited[p->vertex]) {  
                        enqueue(Q, p->vertex);  
                        visited[p->vertex] = true;  
                    }  
            }  
        }  
}
```

# Spanning Trees


- A tree is a special kind of undirected graph, one that is connected but that has no cycles
  - Although all trees are graphs, not all graphs are trees
- A *spanning tree* of a connected undirected graph  $G$  is a subgraph of  $G$  that contains all of  $G$ 's vertices and enough of its edges to form a tree
- There may be several spanning trees for a given graph

# Spanning Trees: Example



# Spanning Trees: Some Observations

- A connected undirected graph that has  $n$  vertices ...
  - ... must have at least  $n - 1$  edges
  - ... and exactly  $n - 1$  edges cannot contain a cycle
  - ... and more than  $n - 1$  edges must contain at least one cycle

 To obtain the spanning tree of a connected graph of  $n$  vertices, we must connect its  $n$  vertices with  $n - 1$  edges

# Spanning Trees: Algorithm

- Beginning at a vertex, our computer visits all other vertices in the graph
  - Each vertex will only be visited once
- As our computer traverses the graph, it also marks the edge that it follows
- After the traversal is complete, the graph's vertices and marked edges form a spanning tree
  - The unmarked edges can be removed from the graph

# The DFS Spanning Tree

dfsTree( $v$ )

Mark  $v$  as visited

for each vertex  $u$  to which  $v$  is adjacent

if  $u$  is an unvisited vertex

Mark the edge between  $v$  and  $u$

dfsTree( $u$ )

# The BFS Spanning Tree

bfsTree( $v$ )

Add  $v$  to an empty queue

Mark  $v$  as visited

**while** the queue is not empty

    Extract  $u$  from the queue

**for** each vertex  $w$  to which  $u$  is adjacent

**if**  $w$  is an unvisited vertex

            Mark edge between  $u$  and  $w$

            Add  $w$  to the queue

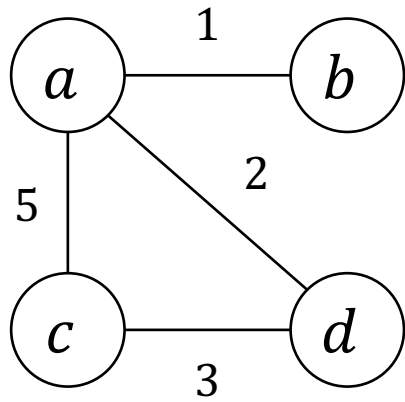
            Mark  $w$  as visited



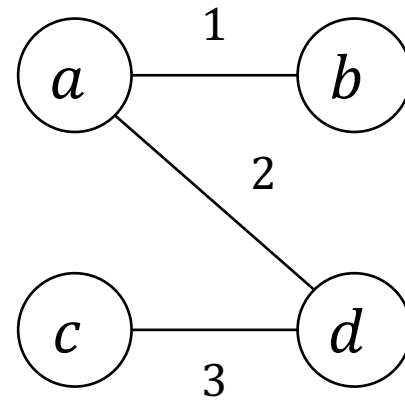
# Minimum Spanning Trees

- A *minimum spanning tree* (or *mst*) of an undirected weighted connected graph is its spanning tree of the smallest weight
  - The *weight of a tree* is defined as the sum of the weights on all its edges
- It has direct applications to the design of all kinds of networks by providing the cheapest way to achieve connectivity

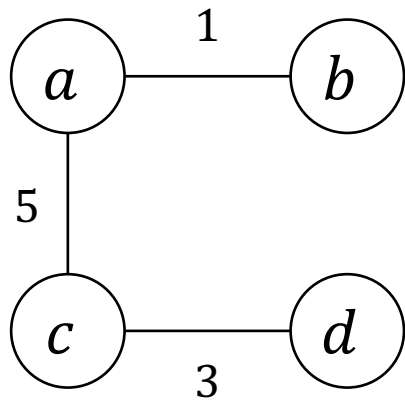
# Example



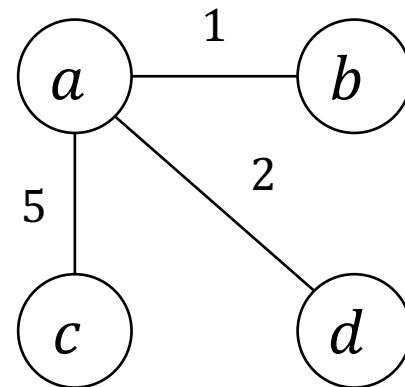
*Graph*



$$w(T_1) = 6$$



$$w(T_2) = 9$$



$$w(T_3) = 8$$

# Minimum Spanning Trees: Prim's Algorithm

- The algorithm constructs a *mst* through a sequence of expanding subtrees:
  - The initial subtree consists of a single vertex selected arbitrarily from the set  $V$
  - On each iteration, the algorithm expands the current tree by attaching to it the *nearest vertex* not in that tree
  - The algorithm stops after all the graph's vertices have been included in the tree being constructed
- ➡ The total number of such iterations is  $|V| - 1$

# An Outline of Prim's Algorithm

Prim( $G = (V, T)$ )

$$V_T = \{v_1\}$$

$$E_T = \emptyset$$

**for**  $i = 1$  to  $|V| - 1$

Find a minimum-weight edge  $e^* = (u^*, v^*)$  among all edges  $(u, v)$  such that  $u \in V_T, v \in V \setminus V_T$

$$V_T = V_T \cup \{v^*\}$$

$$E_T = E_T \cup \{e^*\}$$

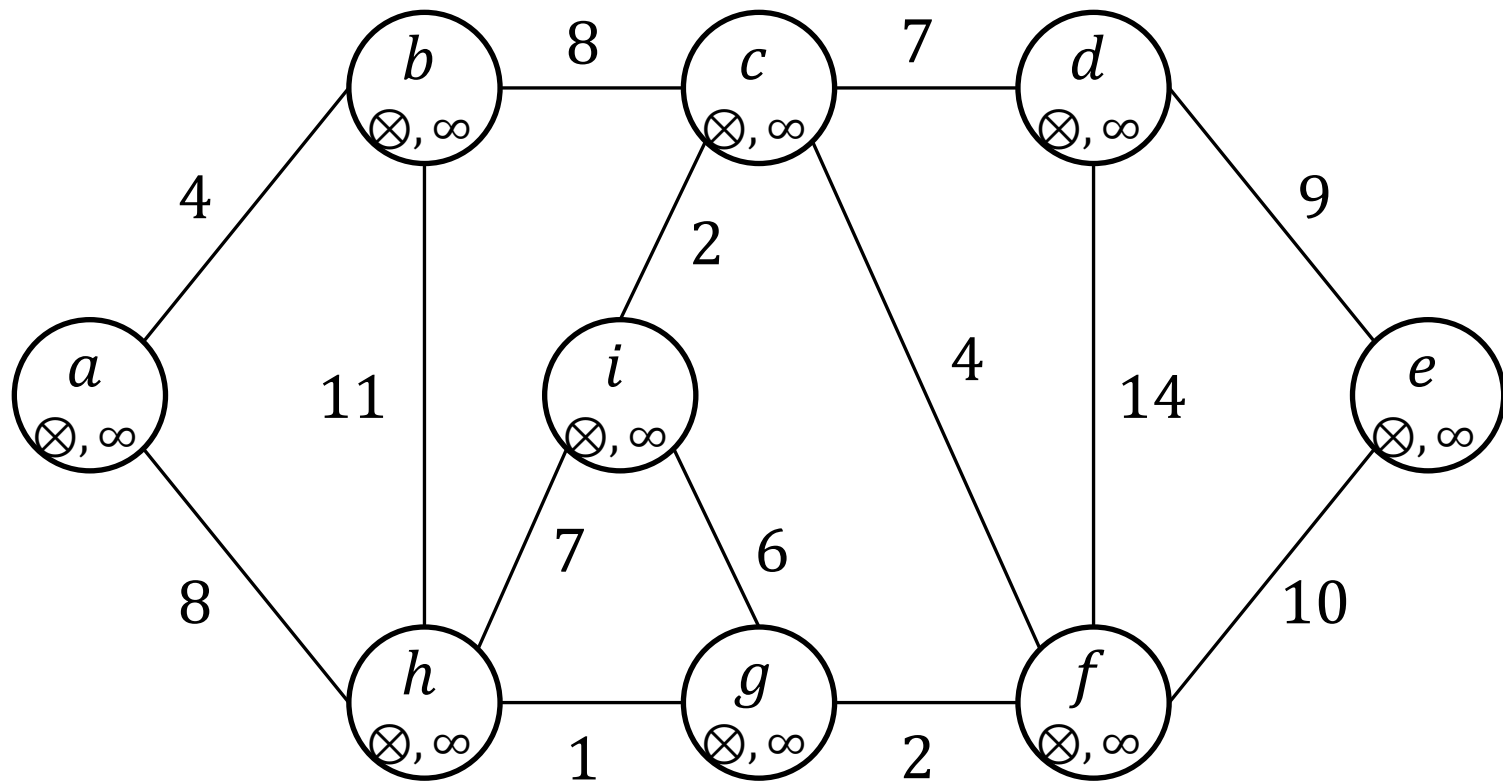
**return**  $G_T = (V_T, E_T)$

# An Implementation of Prim's Algorithm

- Each vertex in  $V \setminus V_T$  needs to remember the shortest edge connecting the vertex to a vertex in  $V_T$
- Each vertex has two labels:
  - “parent” label: The name of the nearest tree vertex ( $\in V_T$ )
  - “distance” label: The weight of the corresponding edge
- A vertex that is not adjacent to any of the tree vertices:
  - “parent” label is NIL
  - “distance” label is  $\infty$

# An Implementation of Prim's Algorithm

- A vertex ( $\in V \setminus V_T$ ) with the smallest distance label is the next one to be added to the tree being constructed  $G_T$ 
  - Ties can be broken arbitrarily
- Let  $v^*$  be the next vertex to be added to  $G_T$ 
  - Move  $v^*$  from the set  $V \setminus V_T$  to the set  $V_T$
  - For each remaining vertex  $v$  in  $V \setminus V_T$  that is connected to  $v^*$  by a shorter edge than the  $v$ 's current distance label, update its labels by  $v^*$  and  $G[v^*][v]$ , respectively



```

Prim(G, root) {
    for (each vertex  $v \in V$ ) {
         $v.dist = \infty$ ;
         $v.parent = NIL$ ;
    }
     $root.dist = 0$ ;       $V_T = \emptyset$ ;
    createQueue(pQ, V);
    while (!isEmpty(pQ)) {
         $v^* = extractQueue(pQ)$ ;
        add  $v^*$  to  $V_T$ ;
        for (each  $v \in pQ$  that is adjacent to  $v^*$ )
            if ( $G[v^*][v] < v.dist$ ) {
                 $v.dist = G[v^*][v]$ ;
                 $v.parent = v^*$ ;
                updateQueue(pQ, v);
            }
    }
}

```



# Single-Source Shortest-Paths Problem

- For a given vertex called the *source* in a weighted connected graph, find shortest paths to all its other vertices
- The most widely applications of the problem are ...
  - transportation planning
  - packet routing in communication networks, including the Internet

# Dijkstra's Algorithm: The General Idea

- The algorithm finds the shortest path from the *source* to a vertex nearest to it, then to a second nearest, ...
- ➡ Before the  $i^{th}$  iteration starts, the algorithm has already identified the shortest paths to  $i - 1$  other vertices nearest to the *source*
  - These vertices, the *source*, and the edges of the shortest paths leading to them from the *source* form a subtree  $T_i$
- The next vertex nearest to the *source* can be found among the vertices adjacent to the vertices of  $T_i$

# An Outline of Dijkstra's Algorithm

//  $d_u$  the length of the shortest path from the *source* to  $u$

Dijkstra( $G = (V, T)$ )

$V_T = \{v_1\}$  //  $v_1$  is the *source*

$E_T = \emptyset$

**for**  $i = 1$  to  $|V| - 1$

Find an edge  $e^* = (u^*, v^*)$  among all edges  $(u, v)$

such that  $u \in V_T, v \in V \setminus V_T$  and  $d_{u^*} + G[u^*][v^*] \leq d_u + G[u][v]$

$V_T = V_T \cup \{v^*\}$

$E_T = E_T \cup \{e^*\}$

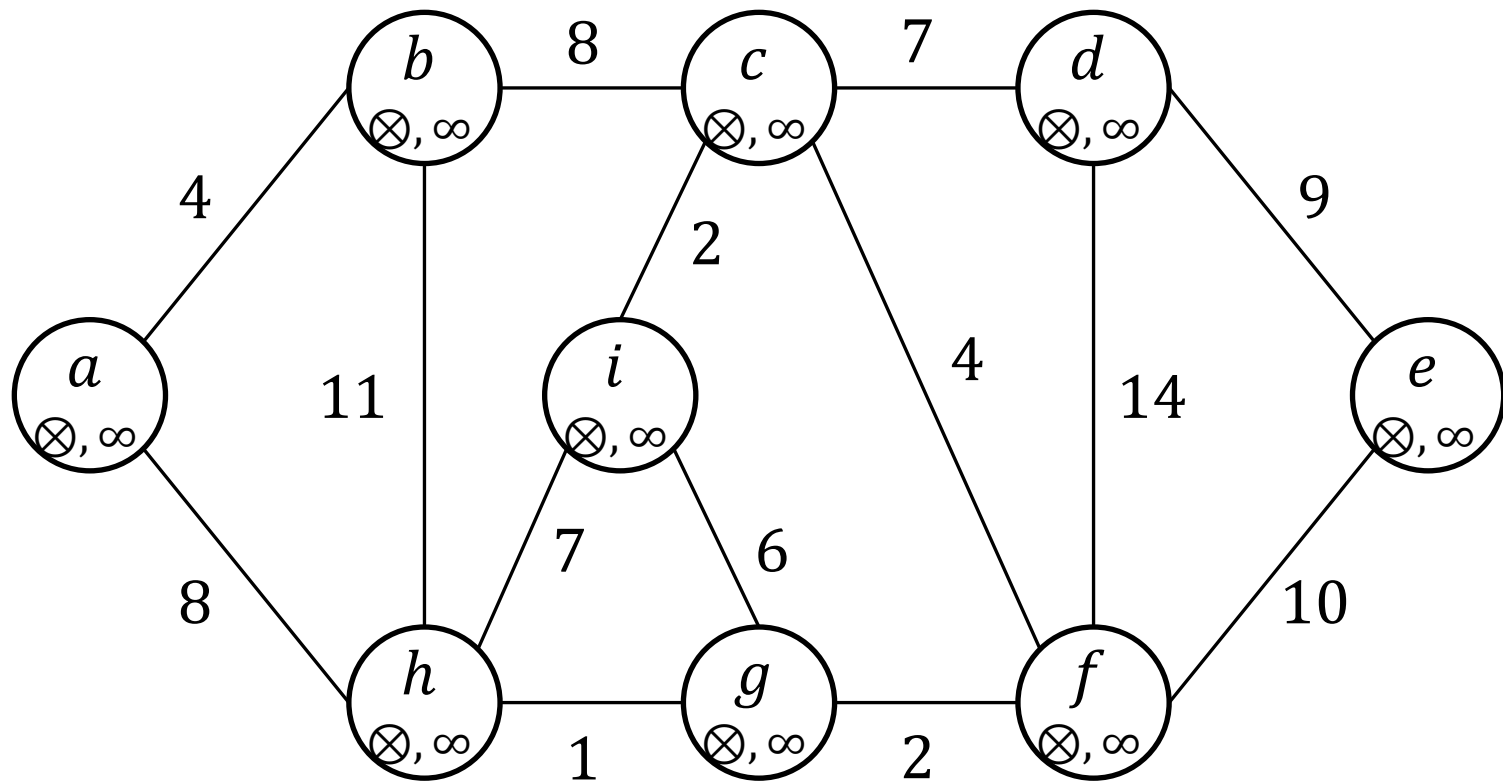
**return**  $G_T = (V_T, E_T)$

# An Implementation of Dijkstra's Algorithm

- Each vertex has two labels:
  - “parent” label: The name of the nearest tree vertex ( $\in V_T$ )
  - “distance” label: The length of the shortest path from the source to this vertex found by the algorithm so far
- ➡ When a vertex is added to  $V_T$ , this label indicates the length of the shortest path from the *source* to that vertex
- A vertex that is not adjacent to any of the tree vertices:
  - “parent” label is NIL
  - “distance” label is  $\infty$

# An Implementation of Dijkstra's Algorithm

- Let  $v^*$  be the next vertex to be added to  $G_T$ 
  - Move  $v^*$  from the set  $V \setminus V_T$  to the set  $V_T$
  - For each remaining vertex  $v$  in  $V \setminus V_T$  that is connected to  $v^*$  such that  $d_{v^*} + G[v^*][v] < d_v$ , update the labels of  $v$  by  $v^*$  and  $d_{v^*} + G[v^*][v]$ , respectively



```

Dijkstra(G(V, E), source) {
  for (each vertex v ∈ V) {
    v.dist = ∞;
    v.parent = NIL;
  }
  source.dist = 0;  VT = ∅;
  createQueue(pQ, V);
  while (!isEmpty(pQ)) {
    v* = extractQueue(pQ);
    add v* to VT;
    for (each v ∈ pQ that is adjacent to v*)
      if (v*.dist + G[v*][v] < v.dist) {
        v.parent = v*;
        v.dist = v*.dist + G[v*][v];
        updateQueue(pQ, v);
      }
  }
}

```