Computer Algorithms

Segment 5

Graphs basic & Traversal Techniques

Terminology

- A ***graph*** is a collection of nodes, called ***vertices***, and a collection of line segments, called ***lines, edges,*** or ***arcs***, connecting pairs of vertices.
- In other words, a graph consists of 2 sets:
  - A set of lines
  - A set of vertices

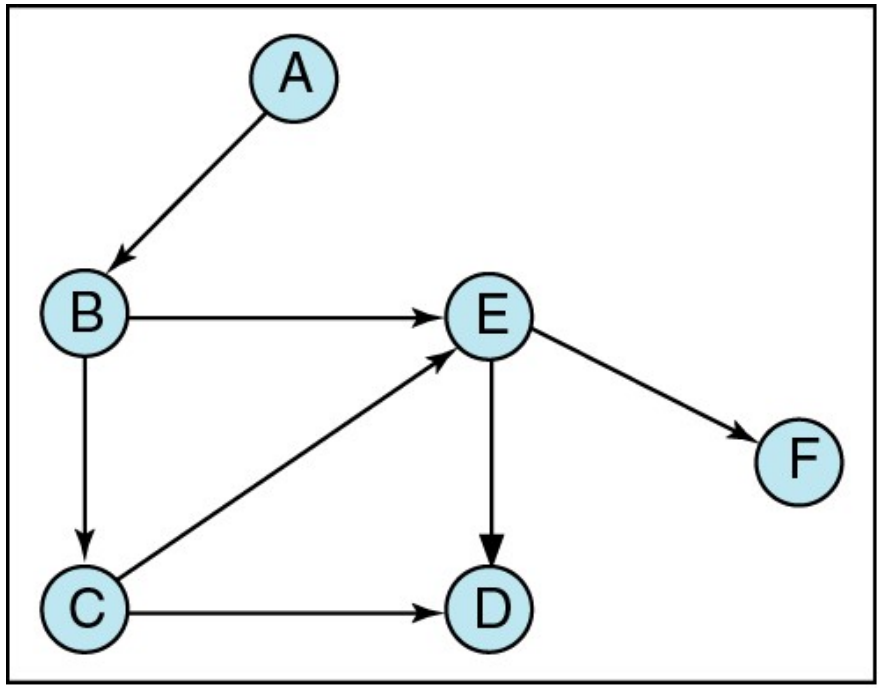
Terminology

- Graphs may be either directed or undirected.
- An ***undirected graph*** is a graph in which there is no direction associated with the lines.
- In an undirected graph, the flow between two vertices can go in either direction.

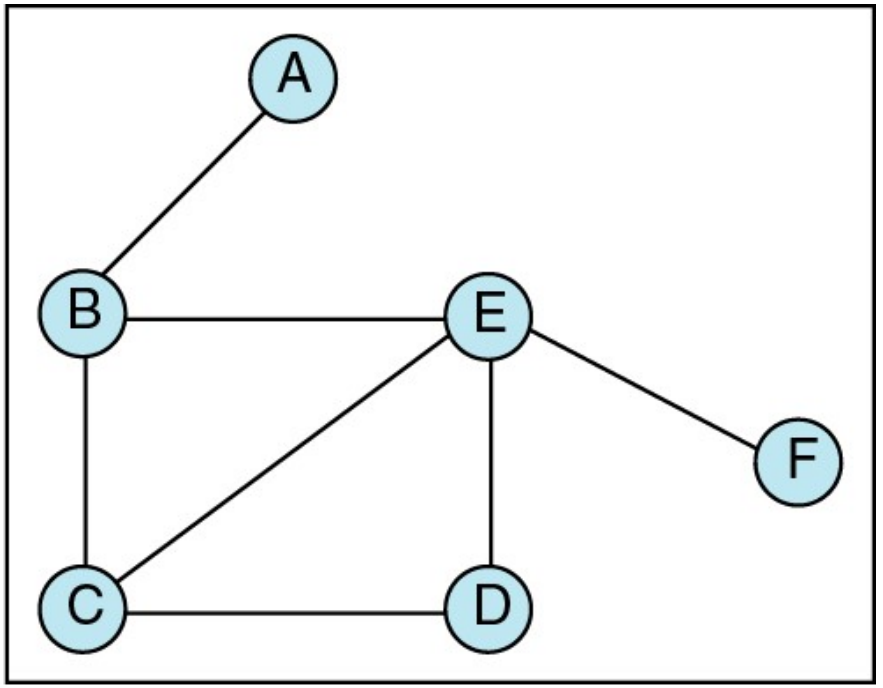
Terminology

- A ***directed graph*** or ***digraph*** is a graph in which each line has a direction associated with it.
- In a directed graph, the flow along an edge between 2 vertices can only follow the indicated direction.

Terminology



(a) Directed graph

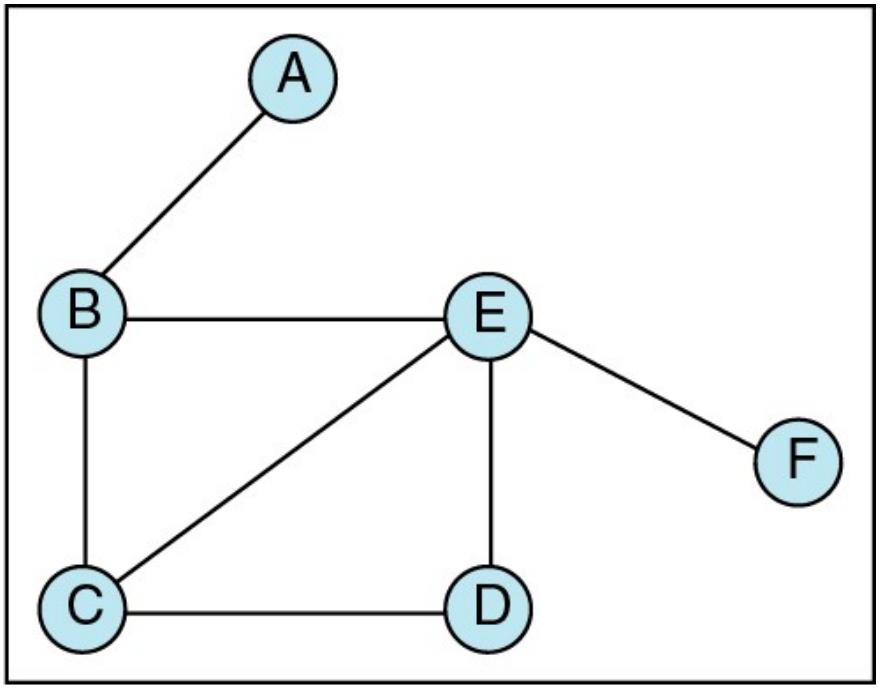


(b) Undirected graph

Terminology

- 2 vertices in a graph are said to be ***adjacent*** (or neighbors) if an edge directly connects them.

A and B are adjacent
D and F are not adjacent

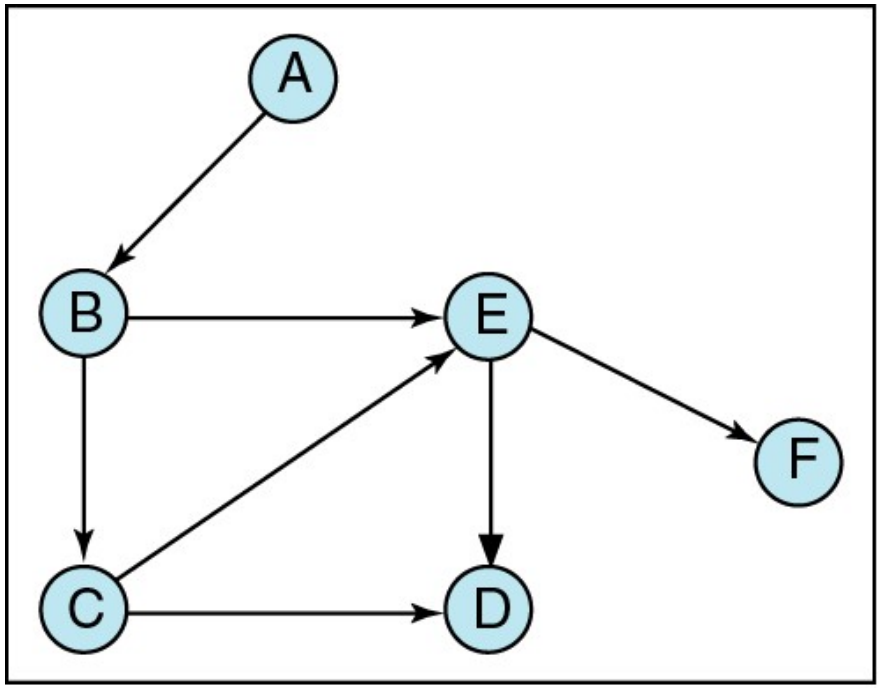


Terminology

- A ***path*** is a sequence of vertices in which each vertex is adjacent to the next one.

A,B,C,E is a path
A,B,E,F is a path

A,B,E,C is *NOT* a path because, although E is adjacent to C, C is not adjacent to E (note direction).



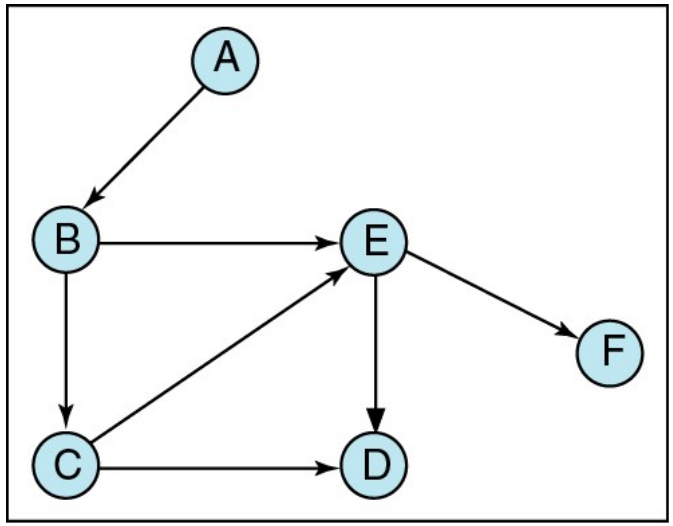
Terminology

- Note that both directed and undirected graphs have paths.
- In an undirected graph, you may travel in either direction.
- In a directed graph, you may only travel along the given direction of the arcs.

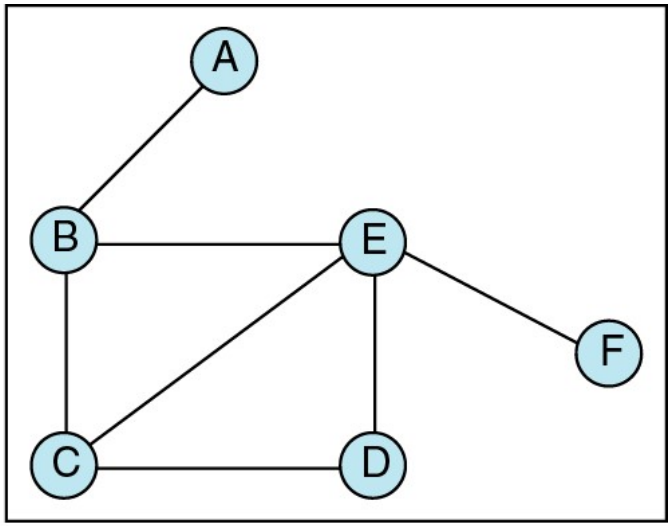
Terminology

- A ***cycle*** is a path that starts and ends with the same vertex.
- Note: a single vertex is not a cycle.

B, C, D, E, B is a cycle in the the undirected graph but not in the directed graph.



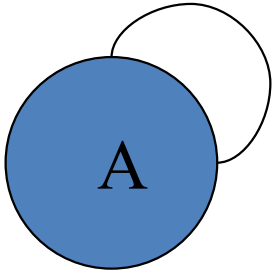
(a) Directed graph



(b) Undirected graph

Terminology

- A ***self edge*** or ***loop*** is an arc that begins and ends at the same vertex.



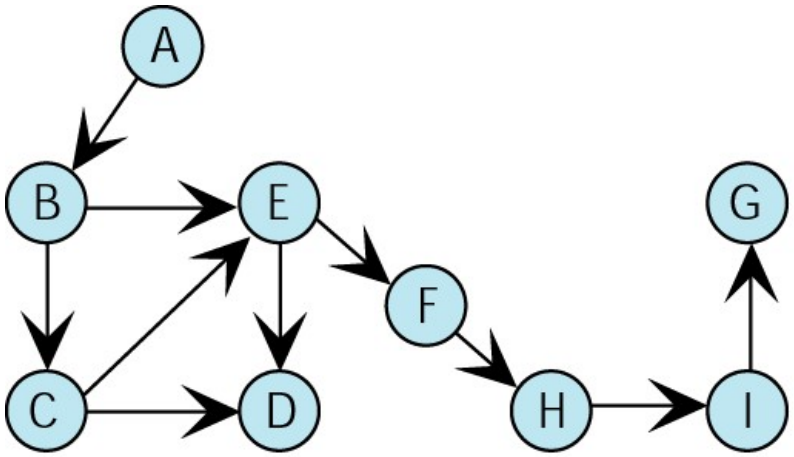
Terminology

- Two vertices are said to be ***connected*** if there is a path between them.
- A graph is said to be connected if, suppressing direction, there is a path from each vertex to every other vertex.
- Furthermore, a directed graph is ***strongly connected*** if there is a path from each vertex to every other vertex.

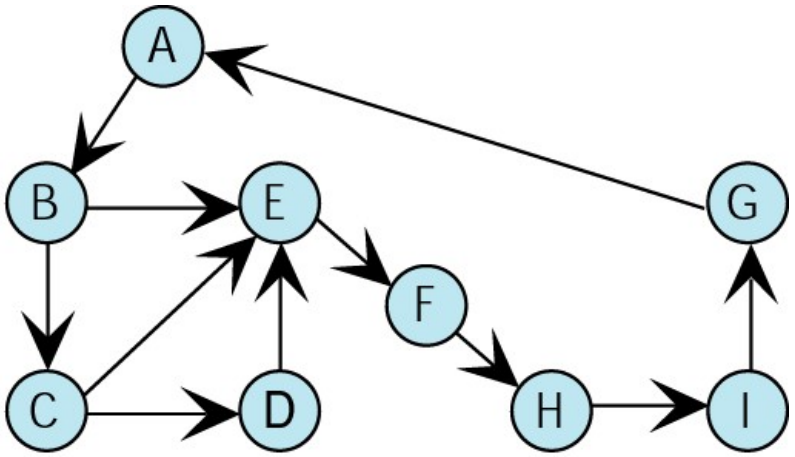
Terminology

- A directed graph is ***weakly connected*** if at least 2 vertices are not connected.
- A graph is ***disconnected*** or ***disjoint*** if it is not connected.

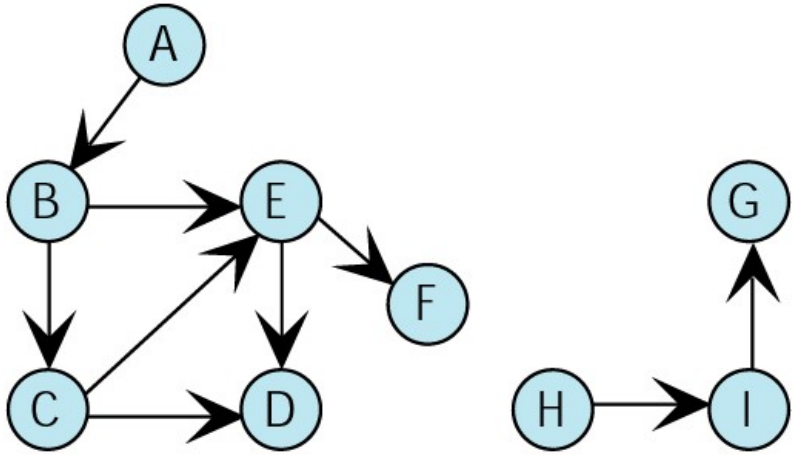
Terminology



(a) Weakly connected



(b) Strongly connected

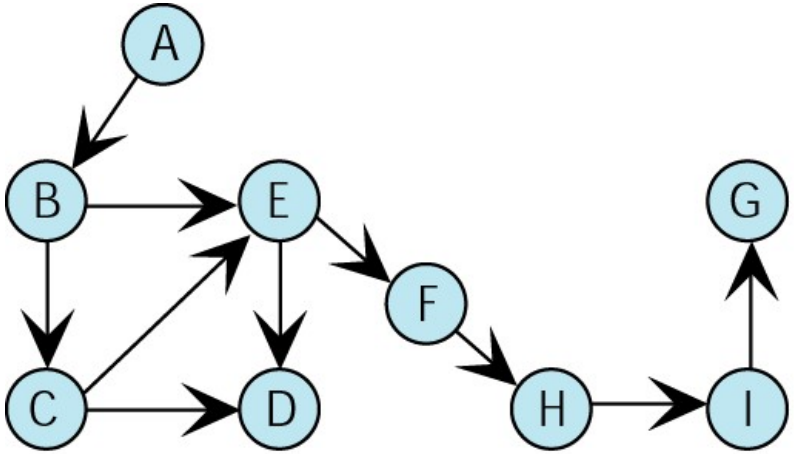


(c) Disjoint graph

Terminology

- The ***degree*** of a vertex is the number of edges incident to it.
- The ***outdegree*** of a vertex in a digraph is the number of arcs leaving the vertex.
- The ***indegree*** of a vertex in a digraph is the number of arcs entering the vertex.

Terminology



- The degree of vertex B is … 3
- The outdegree of vertex B is … 2
- The indegree of vertex B is … 1

Graphs

- If $(u, v) \in E$, then vertex v is adjacent to vertex u.
- Adjacency relationship is:
  - Symmetric if G is undirected.
  - Not necessarily so if G is directed.
- If G is connected:
  - There is a path between every pair of vertices.
  - $|E| \geq |V| - 1$.
  - Furthermore, if $|E| = |V| - 1$, then G is a tree.

Traverse Graph

- Because a vertex in a graph can have multiple parents, the traversal of a graph presents some problems not found in the traversal of linear lists and trees.
- Specifically, we must ensure that we process the data in each vertex once and only once.

Traverse Graph

- Because there are multiple paths to a vertex, we may arrive at it from more than one direction as we traverse the graph.
- The traditional solution to this problem is to include a ‘visited’ flag at each vertex.
- Before the traversal, we set the visited flag in each vertex to ‘off.’
- Then, as we traverse the graph, we set the visited flag to ‘on’ to indicate that the data in a given vertex has been processed.

Traverse Graph

- Note that a graph traversal algorithm is slightly different from a “print all vertices” algorithm.
- A graph-traversal algorithm will only visit all the nodes that it can reach.
- In other words, a graph traversal that starts at vertex V will visit all vertices W for which there exists a path between V and W.

Traverse Graph

- Hence, unlike a tree traversal which always visits all of the nodes in a tree, a graph traversal does not necessarily visit all of the vertices in the graph (unless the graph is connected).
- If the graph is not connected, a graph traversal that begins at vertex V will visit only a subset of the graph's vertices.
- This subset is called the ***connected component*** containing V.

Traverse Graph

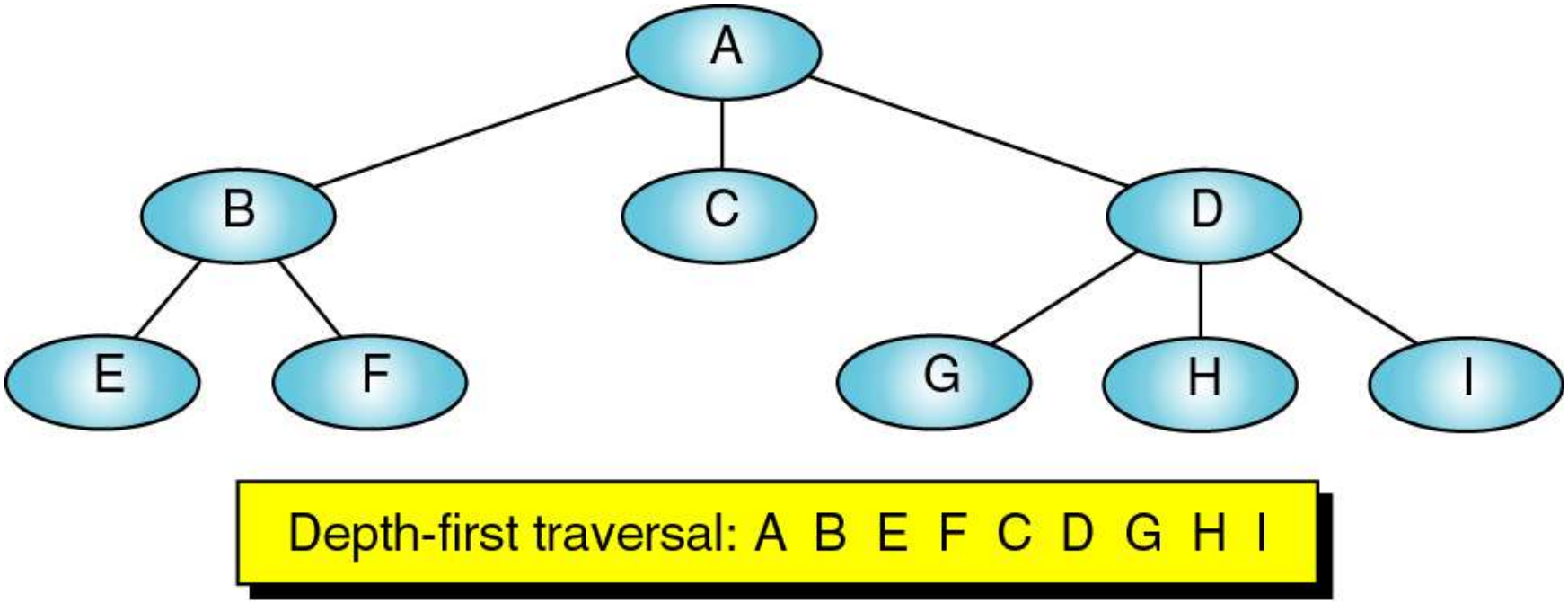
- There are 2 standard graph traversals:
  - Depth first
  - Breadth first
- Both of these methods use the ‘visited’ flag.

Depth-First Traversal

- In the depth-first traversal, we process all of a vertex's descendants before we move to an adjacent vertex.
- This concept is most easily seen when the graph is a tree.

Depth-First Traversal

- Here we show the preorder traversal, one of the standard depth-first traversals.



Depth-First Traversal

- In a similar manner, the depth-first traversal of a graph starts by processing the first vertex of the graph.
- After processing the first vertex, we select any vertex adjacent to the first vertex and process it.
- As we process each vertex, we select an adjacent vertex until we reach a vertex with no adjacent entries.
- This is similar to reaching a leaf in a tree.

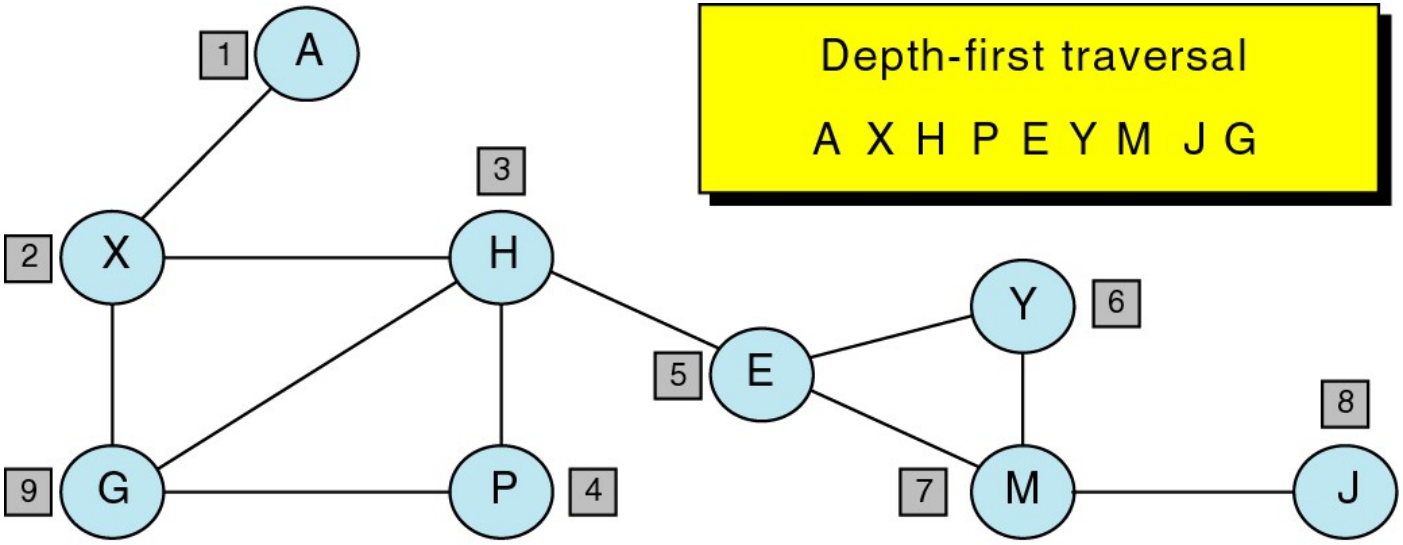
Depth-First Traversal

- We then back out of the structure, processing adjacent vertices as we go.
- It should be obvious that this logic requires a stack (or recursion) to complete the traversal.
- The order in which the adjacent vertices are processed depends on how the graph is physically stored.

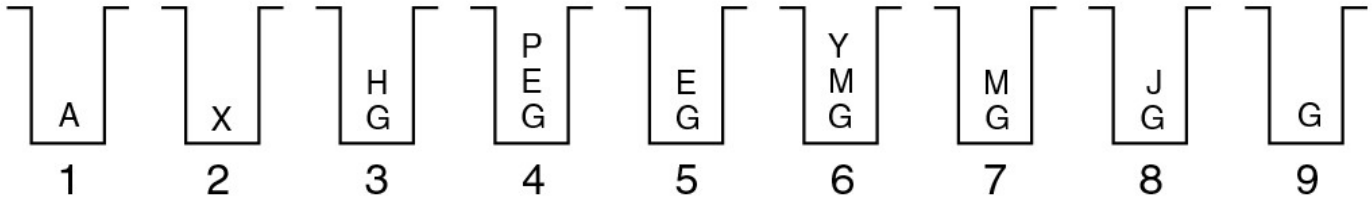
Depth-First Traversal

- On the next slide we'll walk through a depth-first traversal.
- The number in the box next to a vertex indicates the processing order.
- The stacks below the graph show the stack contents as we work our way down the graph and then as we back out.

Depth-First Traversal



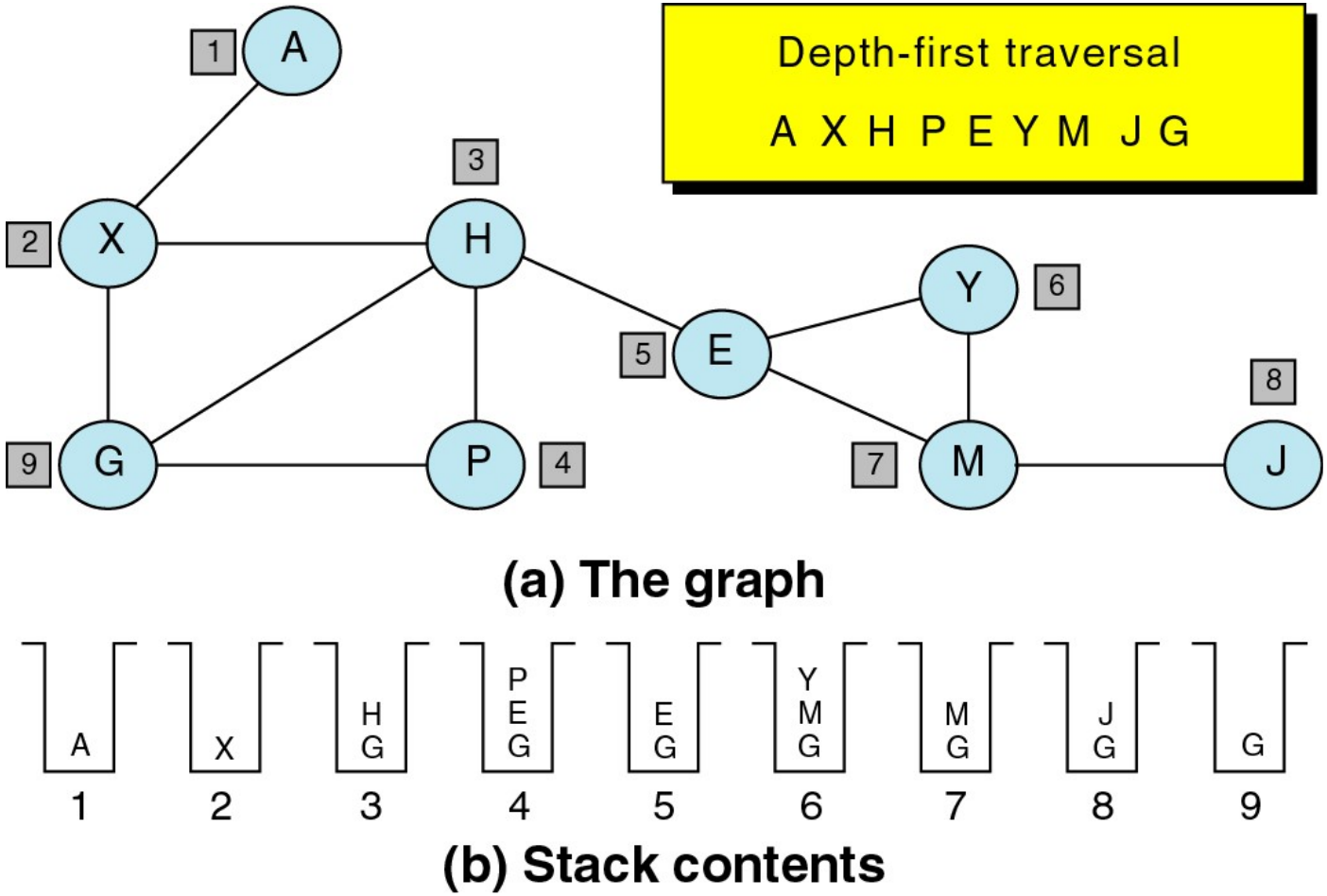
(a) The graph



(b) Stack contents

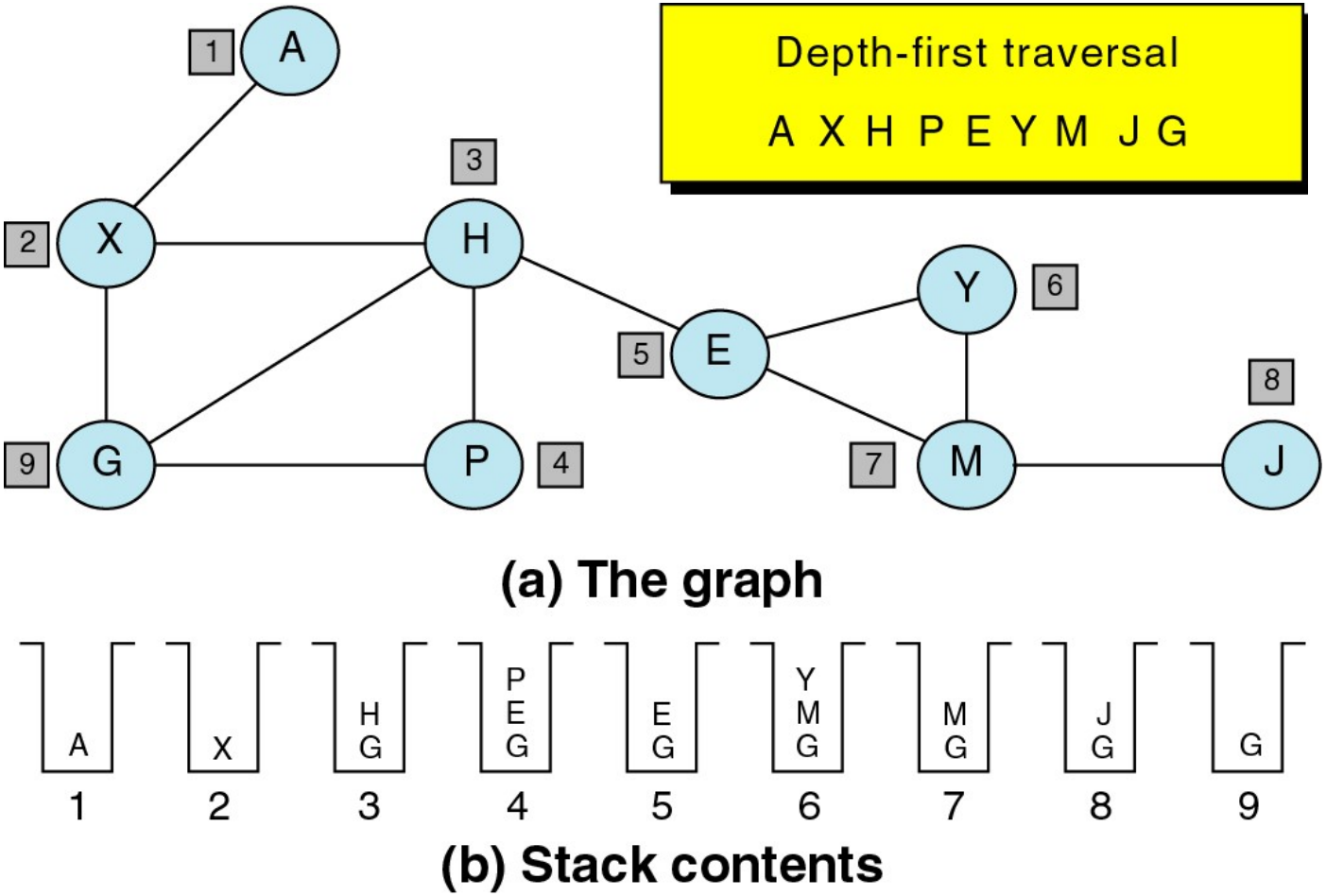
1. We begin by pushing the 1st vertex, A, onto the stack.

Depth-First Traversal



2. We then loop, pop the stack, and, after processing the vertex, push all of its adjacent vertices onto the stack. To process X at Step 2, therefore, we pop X from the stack, process it, and then push G and H onto the stack.

Depth-First Traversal

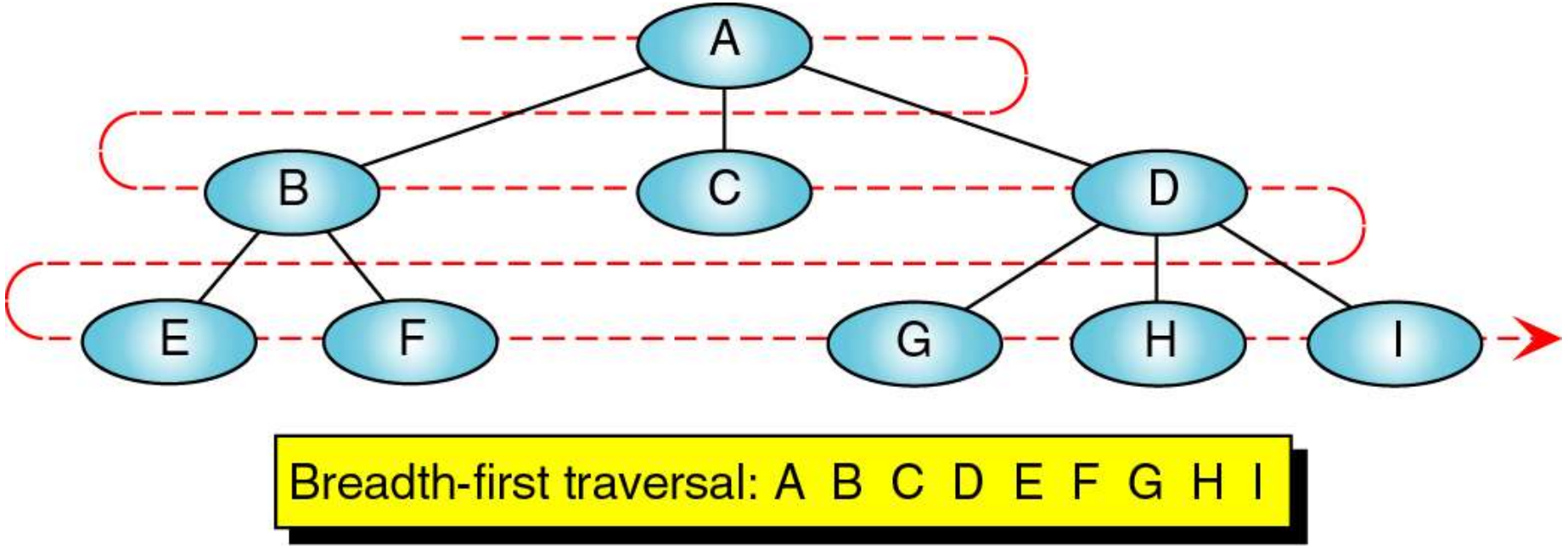


3. When the stack is empty, the traversal is complete.

Breadth-First Traversal

- In the breadth-first traversal of a graph, we process all adjacent vertices of a vertex before going to the next level.
- We also saw the breadth-first traversal of a tree.

Breadth-First Traversal



- Here we see that the breadth-first traversal starts at level 0 and then processes all the vertices in level 1 before going on to process the vertices in level 2.

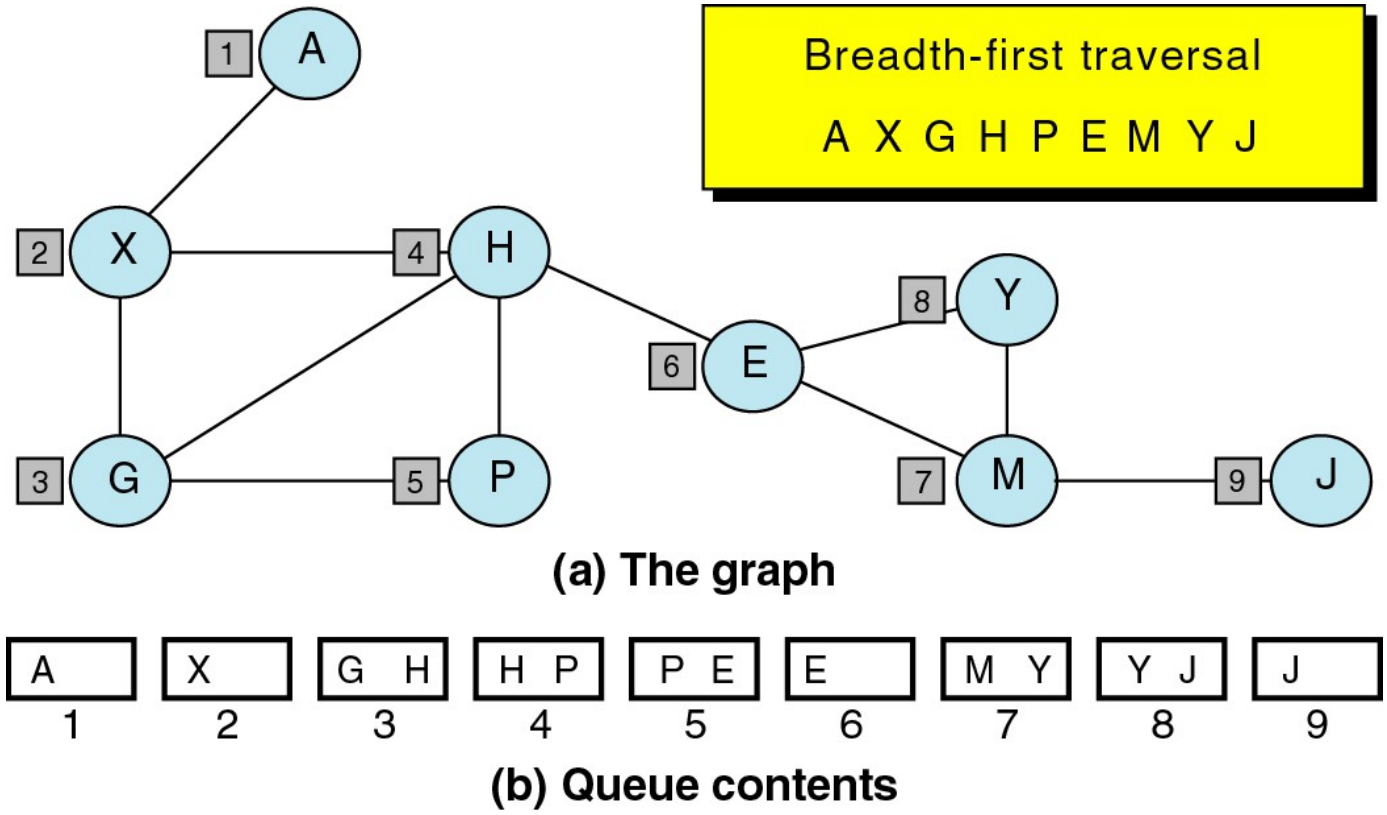
Breadth-First Traversal

- The breadth-first traversal of a graph follows the same approach as the BFT of a tree.
- We begin by picking a starting vertex.
- After processing it, we process all of its adjacent vertices.
- After we process all of the first vertex's adjacent vertices, we pick the first adjacent vertex and process all of its adjacent vertices, then the 2nd adjacent vertex and process all of its adjacent vertices, etc.

Breadth-First Traversal

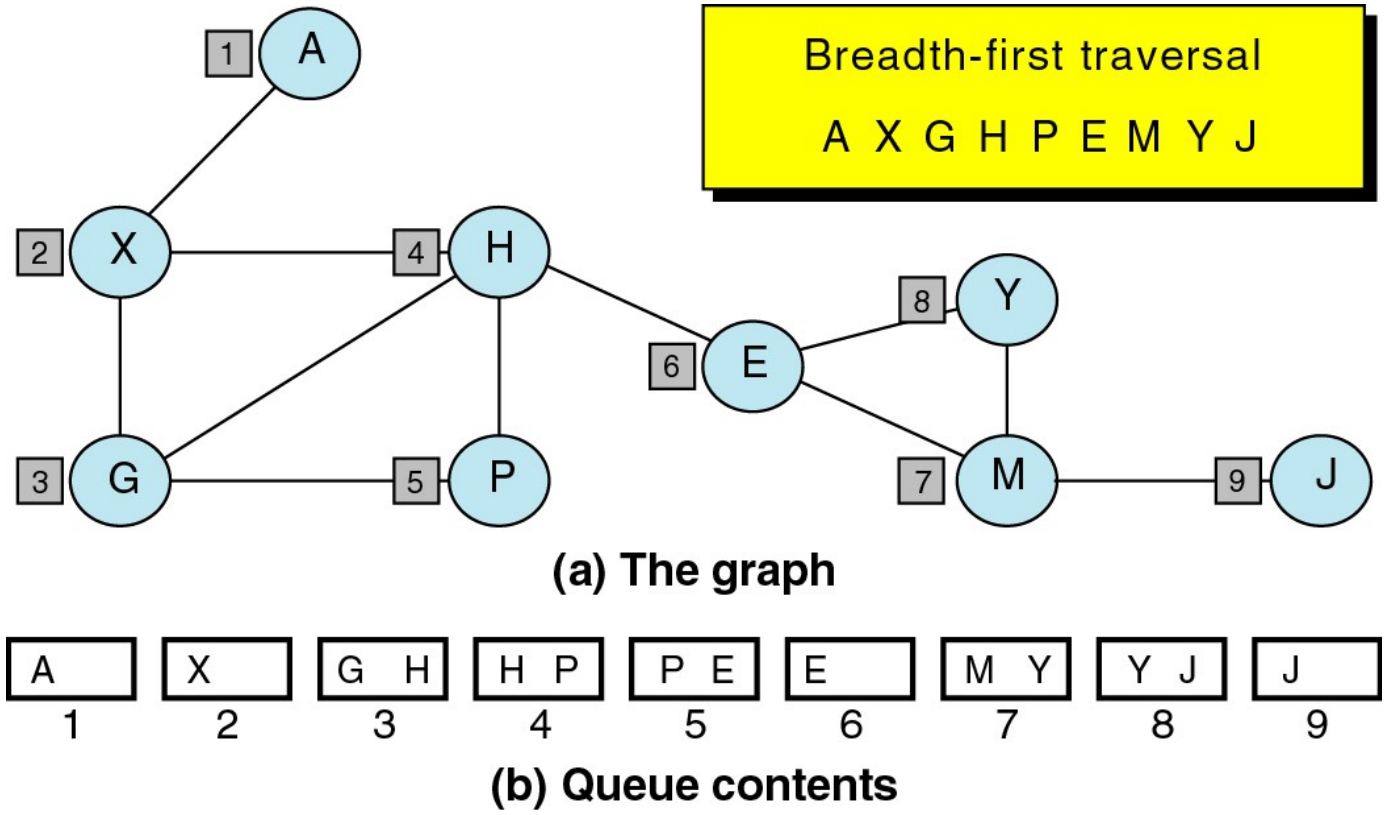
- The breadth-first traversal uses a queue rather than a stack.
- As we process each vertex, we place all of its adjacent vertices in a queue.
- Then, to select the next vertex to be processed, we remove a vertex from the queue and process it.

Breadth-First Traversal



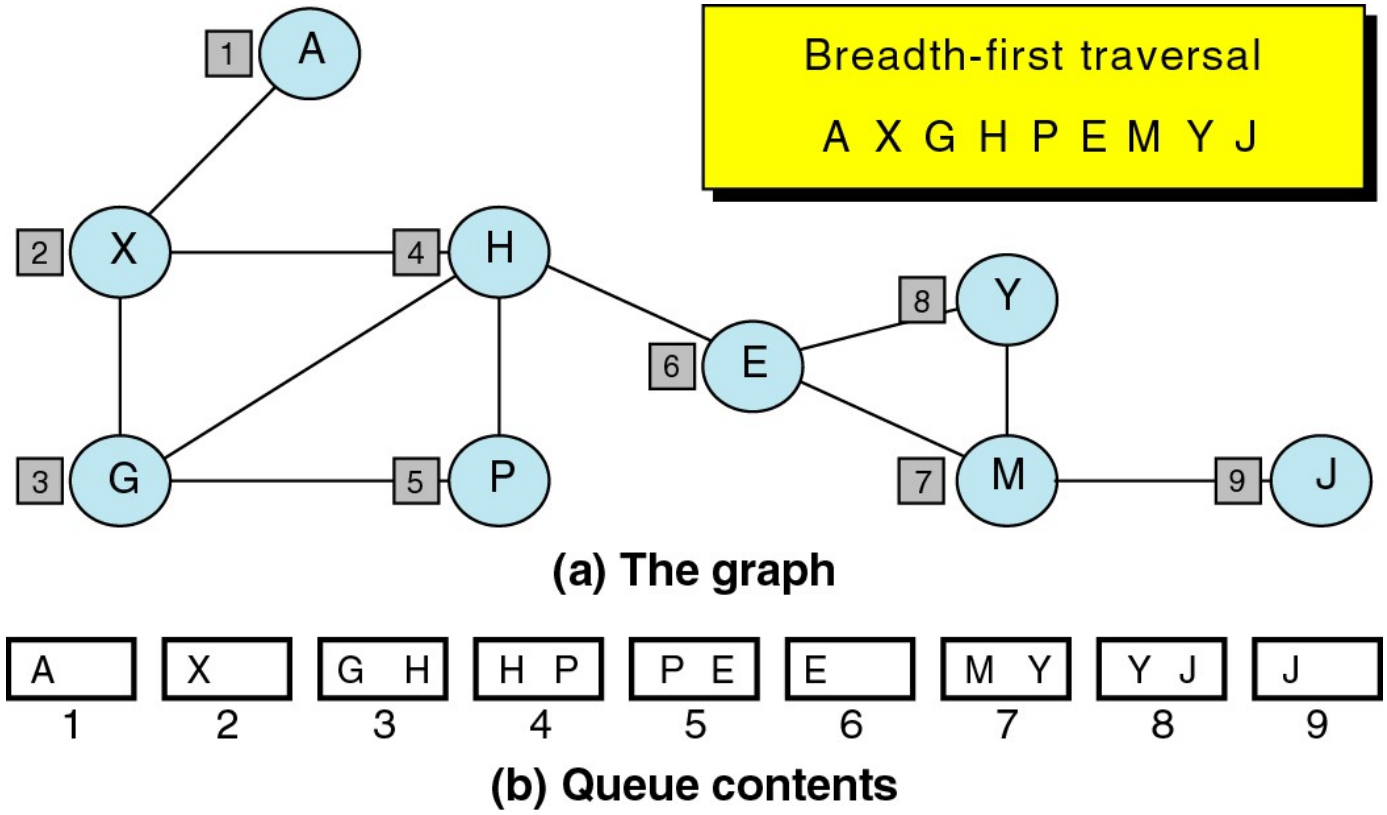
1. We begin by enqueuing the 1st vertex, A.

Breadth-First Traversal



2. We then loop, dequeuing the queue, and, after processing the vertex at the front of the queue, enqueue all of its adjacent vertices. To process X at Step 2, therefore, we dequeue X, process it, and then enqueue G and H.

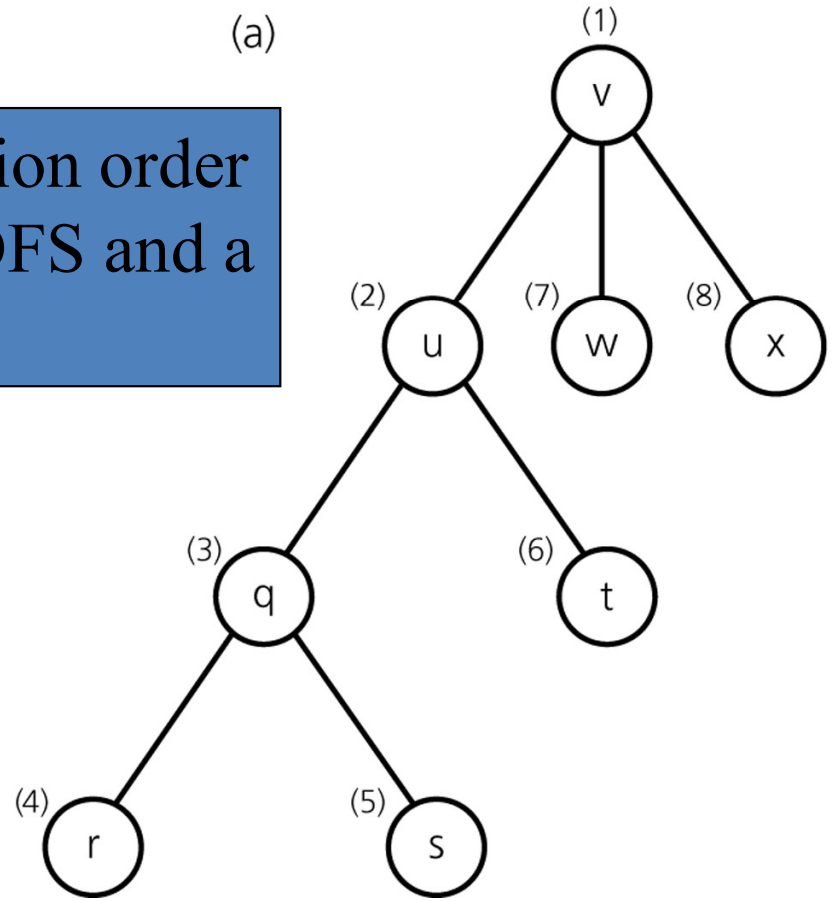
Breadth-First Traversal



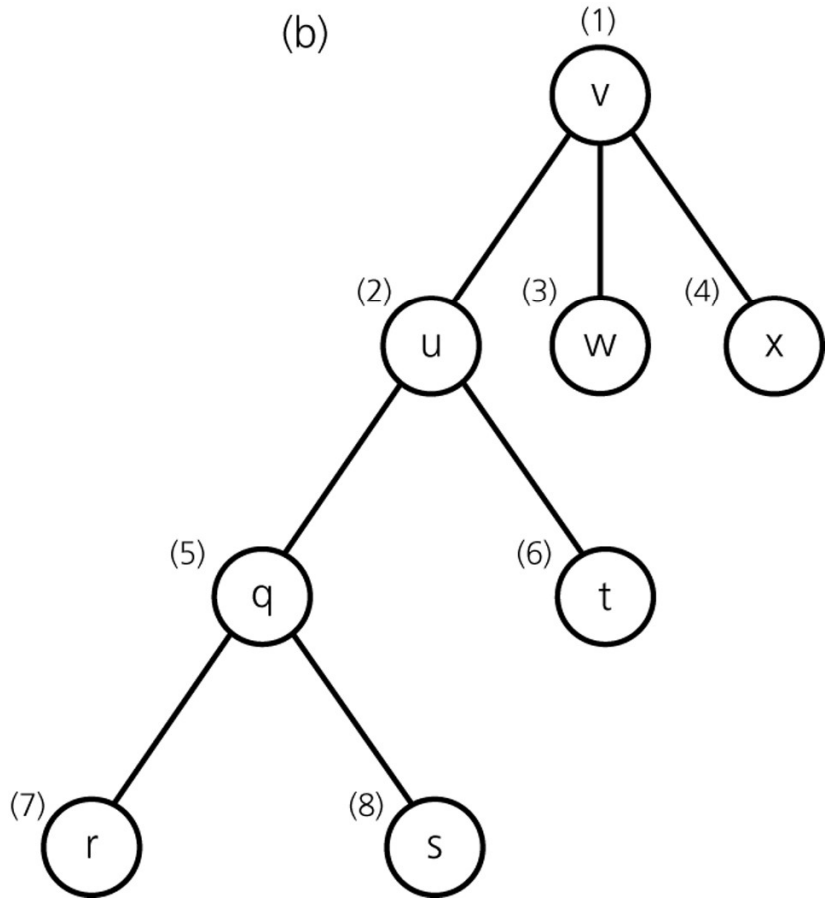
3. When the queue is empty, the traversal is complete.

Traversals Example

Visitation order for a DFS and a BFS.



(a) Depth-first search



(b) Breadth-first search

Graph Storage Structures

- To represent a graph, we need to store two sets:
  - The first set represents the vertices of the graph.
  - The second set represents the edges of the graph.
- The most common structures used to store these sets are arrays and linked lists.
- Although the arrays offer some programming and processing efficiencies, to use them, the number of vertices must be known in advance – a major limitation.

Adjacency Matrix

- The ***adjacency matrix*** uses a vector (1-dimensional array) to store the vertices and a matrix (2-dimensional array) to store the edges.
- If 2 vertices are adjacent – that is, if there is an edge between them, the matrix intersect has a value of 1.
- If there is no edge between them, the intersect is set to 0.

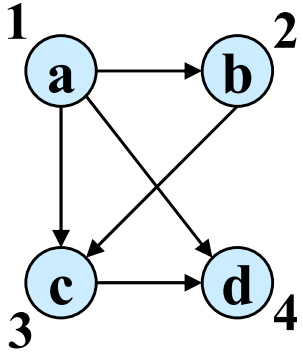
Adjacency Matrix

- If the graph is directed, then the intersection chosen in the adjacency matrix indicates the direction.

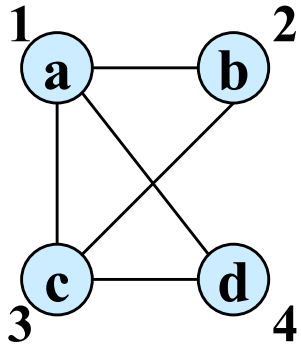
Adjacency Matrix

- $|V| \times |V|$ matrix A.
- Number vertices from 1 to $|V|$ in some arbitrary manner.
- A is then given by:

$$A[i,j] = a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

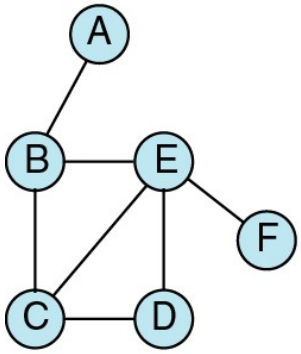


	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	1
4	0	0	0	0



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

Adjacency Matrix



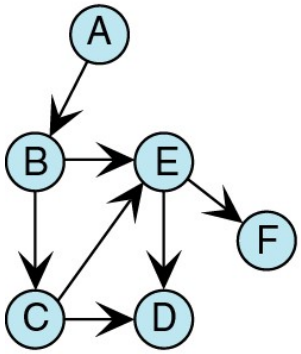
A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

Adjacency matrix

(a) Adjacency matrix for non-directed graph



A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

Adjacency matrix

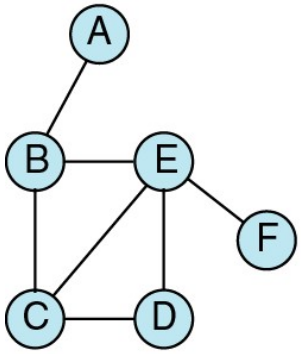
(a) Adjacency matrix for directed graph

Here we see the vertex vectors and the adjacency matrices for the graphs shown.

The vertex vector contains an entry for each node in the graph.

The adjacency matrix contains a ‘1’ at the appropriate intersection to represent an edge.

Adjacency Matrix

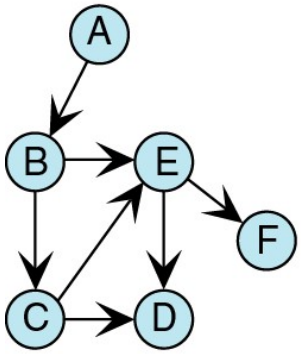


Vertex vector
A
B
C
D
E
F

Adjacency matrix

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

(a) Adjacency matrix for non-directed graph



Vertex vector
A
B
C
D
E
F

Adjacency matrix

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

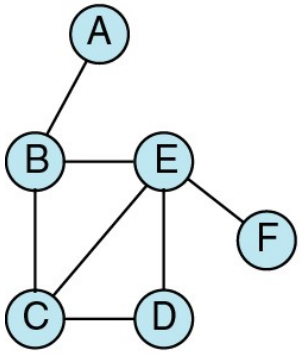
(a) Adjacency matrix for directed graph

For example, A and B are adjacent in the graph in (a).

Hence, at the intersection of A and B in the adjacency matrix, we place a '1' to denote an edge.

Similarly, for the digraph shown, we place a '1' at the intersection of A and B. However, because this is a digraph, there is only one '1'.

Adjacency Matrix



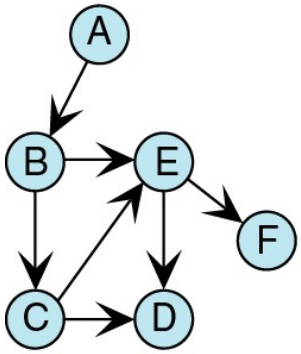
A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

Adjacency matrix

(a) Adjacency matrix for non-directed graph



A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

Adjacency matrix

(a) Adjacency matrix for directed graph

The intersection at which the '1' appears is determined by the direction of the arc between adjacent nodes of the digraph.

It doesn't matter which intersection gets the '1' – just remain consistent.

I like the notation:
matrix[i][j] = 1 if there is an edge from vertex i to vertex j

Space and Time

- **Space:** $\Theta(V^2)$.
  - Not memory efficient for large graphs.
- **Time:** to list all vertices adjacent to u: $\Theta(V)$.
- **Time:** to determine if $(u, v) \in E$: $\Theta(1)$.
- Can store weights instead of bits for weighted graph.

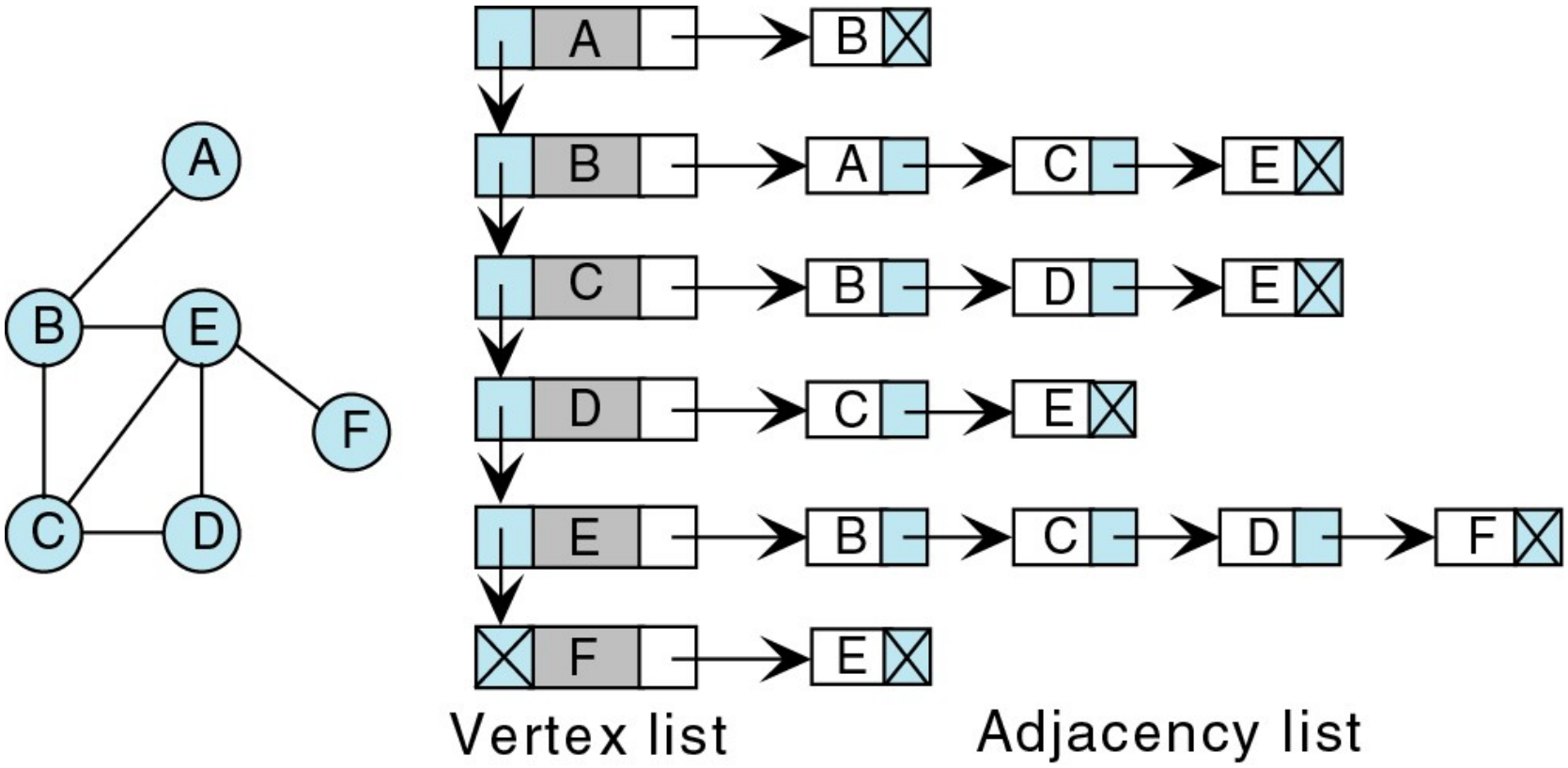
Adjacency Matrix

- In addition to the limitation that the size of the graph must be known before the program starts, there is another serious limitation in the adjacency matrix: only 1 edge can be stored between any 2 vertices.
- This can be an issue in modeling some structures (e.g., 2 different flights between the same cities).

Adjacency List

- If the graph structure is unknown ahead of time, we can use linked lists to implement the vertex vector and adjacency matrix.
- In this case, they are referred to as a vertex list and an adjacency list.

Adjacency List



Here, we have a linked list of the nodes in the graph. Each node in the graph contains a pointer to the head of an adjacency list. The adjacency list is a linked list containing adjacent nodes.

Storage Requirement

- For directed graphs:
  - Sum of lengths of all adj. lists is

$$\sum_{v \in V} \text{out-degree}(v) = |E|$$

No. of edges leaving *v*

  - Total storage: $\Theta(V+E)$
- For undirected graphs:
  - Sum of lengths of all adj. lists is

$$\sum_{v \in V} \text{degree}(v) = 2|E|$$

No. of edges incident on *v*. Edge (*u*,*v*) is incident on vertices *u* and *v*.

  - Total storage: $\Theta(V+E)$

Pros and Cons: adj list

- Pros
  - Space-efficient, when a graph is sparse.
  - Can be modified to support many graph variants.
- Cons
  - Determining if an edge $(u,v) \in G$ is not efficient.
    - Have to search in u's adjacency list. $\Theta(\text{degree}(u))$ time.
    - $\Theta(V)$ in the worst case.

Graph Structure

- Now we will present a possible implementation of the graph data structure along with some of the basic graph operations.

Queues

- Like a stack, special kind of linear list
- One end is called *front*
- Other end is called *rear*
- Additions (insertions or enqueue) are done at the rear only
- Removals (deletions or dequeue) are made from the front only

Bus Stop Queue

- Remove a person from the queue

Bus Stop Queue

front

rear

Bus Stop Queue

front rear

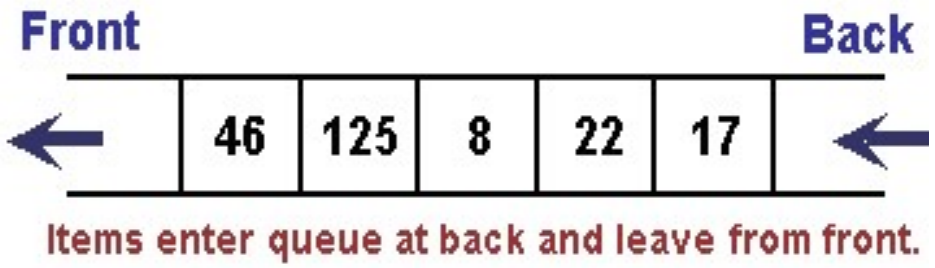
Bus Stop Queue

front rear

- Add a person to the queue
- A queue is a FIFO (First-In, First-Out) list.

What is a queue?

- A queue system is a linear list in which deletions can take place only at one end the “front” of the list, and the insertions can take place only at the other end of the list, the “back” . Is called a First-In-First-Out(FIFO)



Enqueue Pseudocode

This procedure insert an element Item into a queue

```
1.  Enqueue(Q, N, Front, Rear, Item)
2.  {
3.    if  Front=1 and Rear=N, or if  Front=Rear+1, then  // Q already filled
4.     overflow, and return
5.    if  Front = Null then     // Find the new value of Rear
6.          Front = 1 and Rear = 1     //Queue initially empty
7.    else if  Rear = N  then
8.          Rear = 1
9.    else
10.       Rear = Rear + 1
11. Q[Rear] = Item  // Insert new element
12.  Return
13. }
```

Reference: Data Structure- Schaum's, page 192

Dqueue Pseudocode

This procedure deletes an element from a queue and assigns it to the variable Item

```
1.  Dqueue(Q, N, Front, Rear, Item)
2.  {
3.    if  Front = Null then     // Q already empty
4.          Underflow, and Return
5.  Item = Q[Front]
6.  if  Front= Rear then      // Find the new value of Front
7.      Front = Null  and Rear = Null
8.    else if  Front = N then
9.           Front = 1
10.  else
11.        Front = Front + 1
12.  Return
13. }
```

Reference: Data Structure- Schaum's, page 192

Breadth-first Search

- **Input:** Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.
- **Output:**
  - $d[v]$ = distance (smallest # of edges, or shortest path) from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  - $\pi[v] = u$ such that (u, v) is last edge on shortest path $s \rightsquigarrow v$.
    - u is v's predecessor.
  - Builds breadth-first tree with root s that contains all reachable vertices.

Breadth-first Search

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - A vertex is “discovered” the first time it is encountered during the search.
  - A vertex is “finished” if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  - White – Undiscovered.
  - Gray – Discovered but not finished.
  - Black – Finished.
    - Colors are required only to reason about the algorithm. Can be implemented without colors.

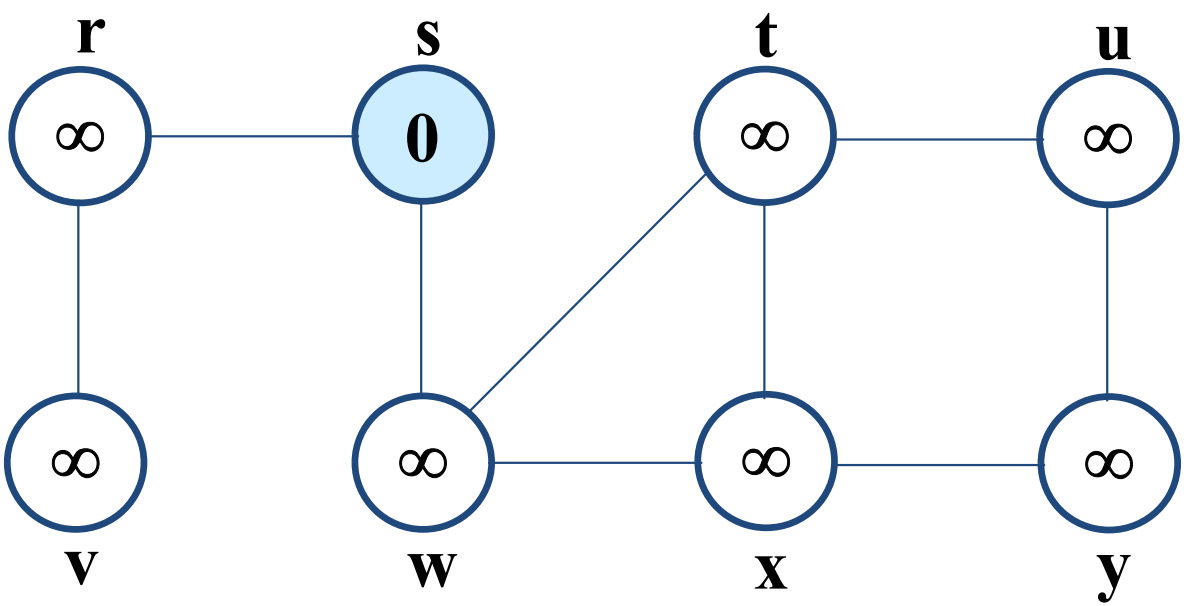
BFS(G,s)

```
1.  for each vertex u in V[G] – {s}
2           do color[u] ← white
3              d[u] ← ∝
4              π[u] ← nil
5   color[s] ← gray
6   d[s] ← 0
7    π[s] ← nil
8   Q ← Φ
9   enqueue(Q,s)
10  while Q ≠ Φ
11          do u ← dequeue(Q)
12                  for each v in Adj[u]
13                          do if color[v] = white
14                                  then color[v] ← gray
15                                       d[v] ← d[u] + 1
16                                       π[v] ← u
17                                       enqueue(Q,v)
18                  color[u] ← black
```

white: undiscovered
gray: discovered
black: finished

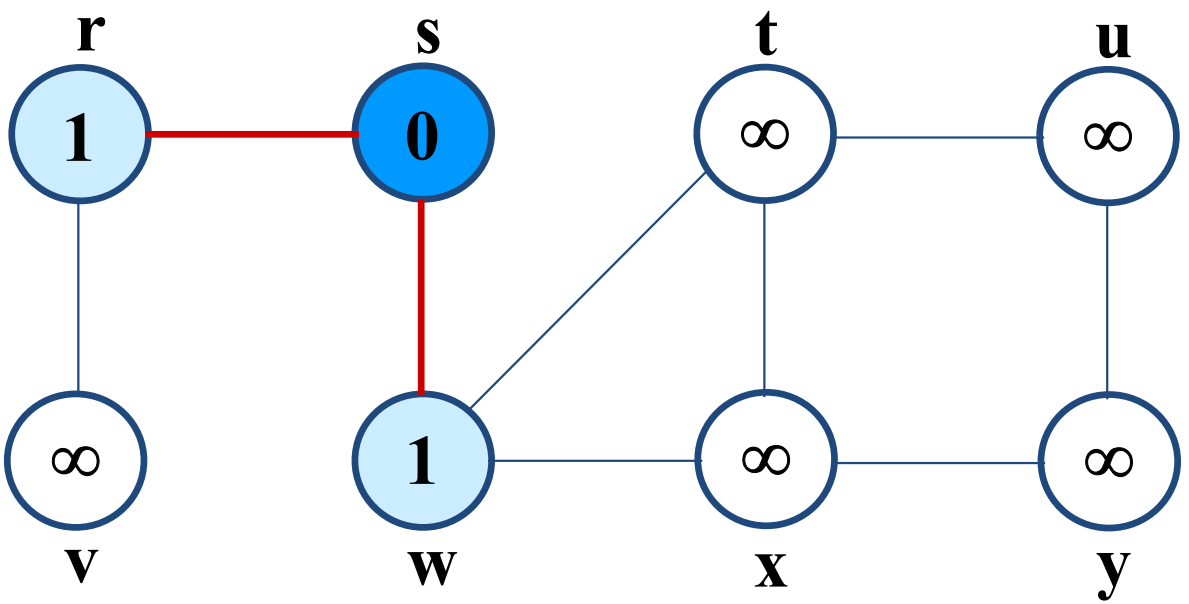
***Q*: a queue of discovered vertices**
color[*v*]: color of v
d[*v*]: distance from s to v
π[*u*]: predecessor of v

Example (BFS)



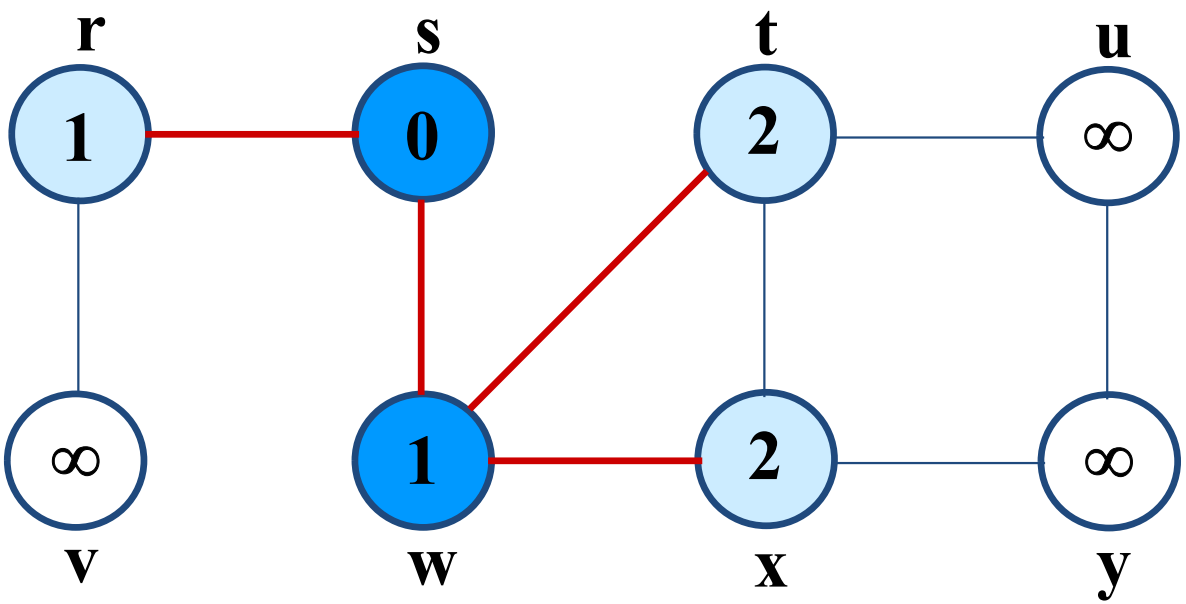
Q:	s
	0

Example (BFS)



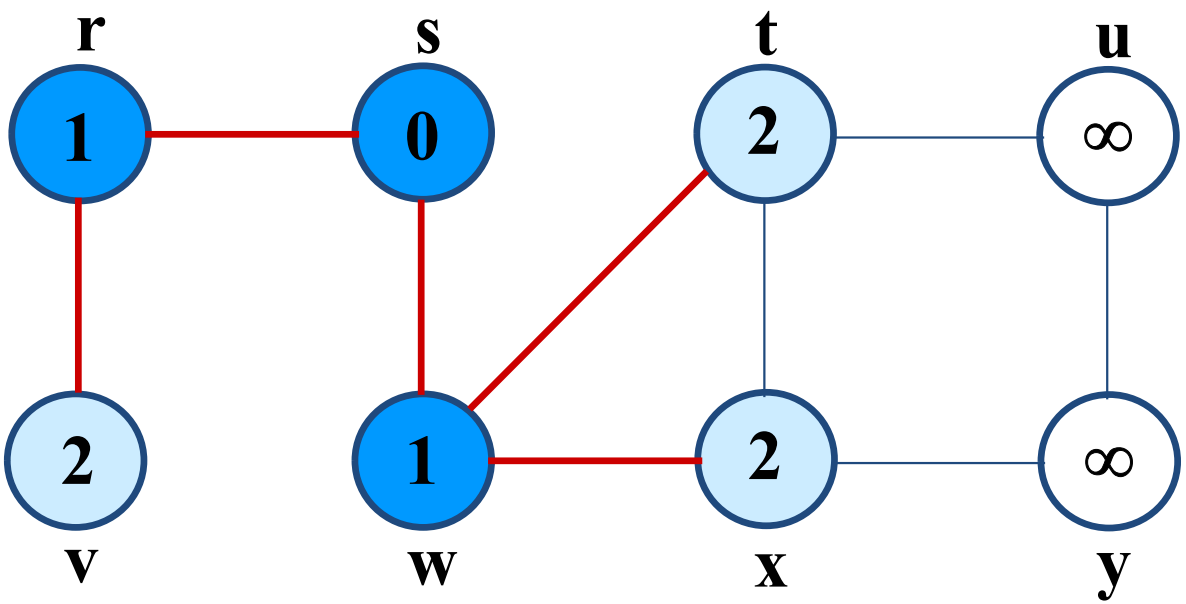
Q:	w	r
	1	1

Example (BFS)



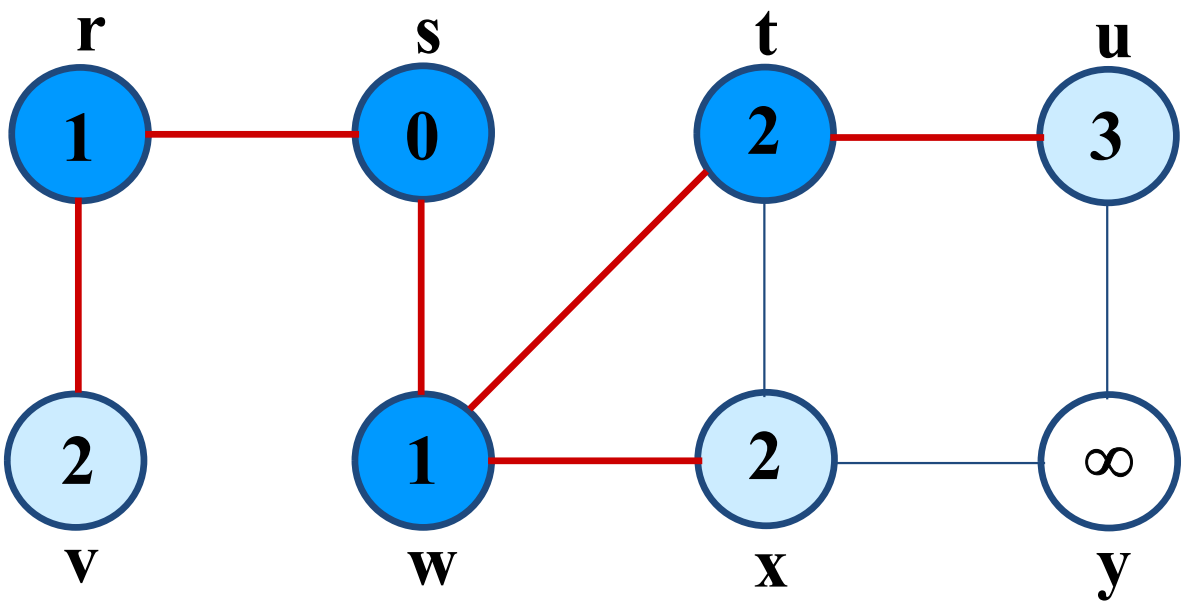
Q:	r	t	x
	1	2	2

Example (BFS)



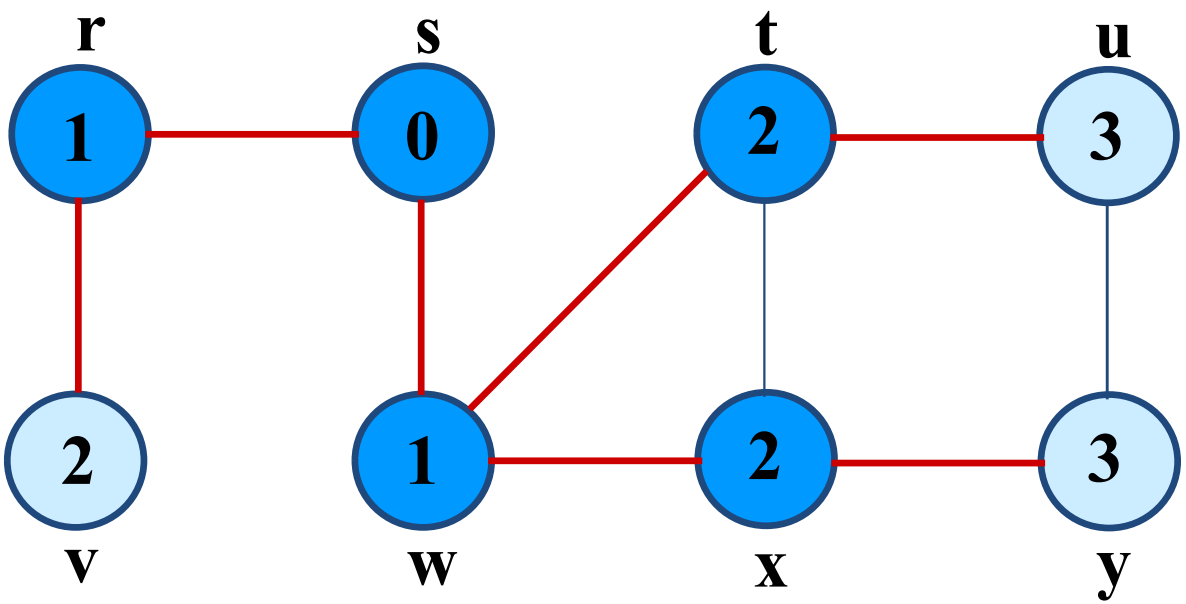
Q:	t	x	v
	2	2	2

Example (BFS)



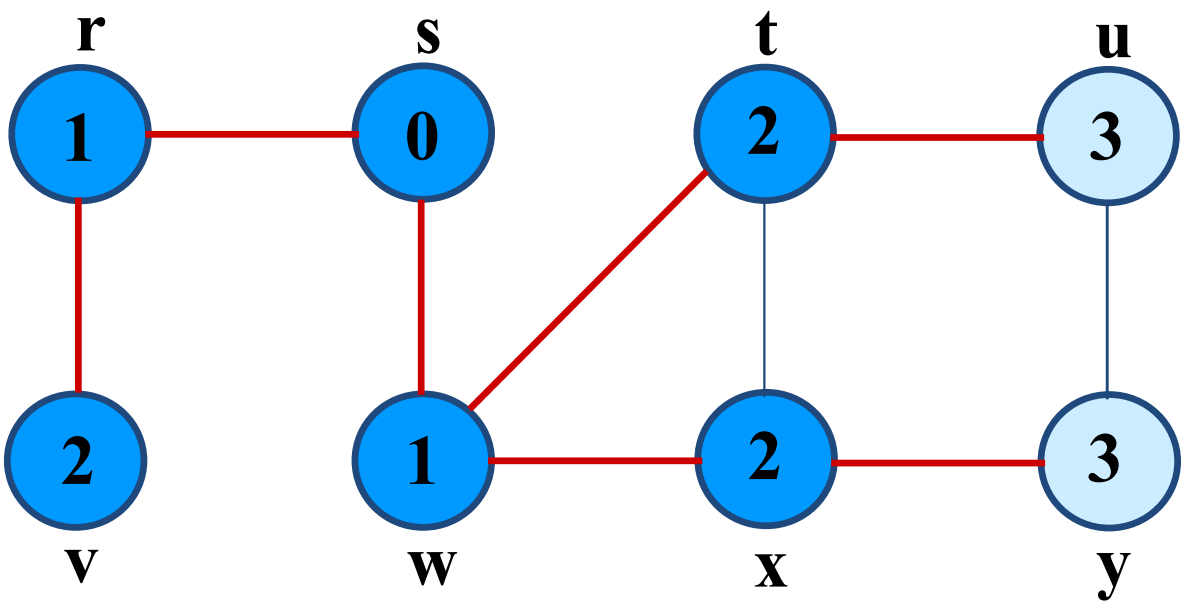
Q:	x	v	u
	2	2	3

Example (BFS)



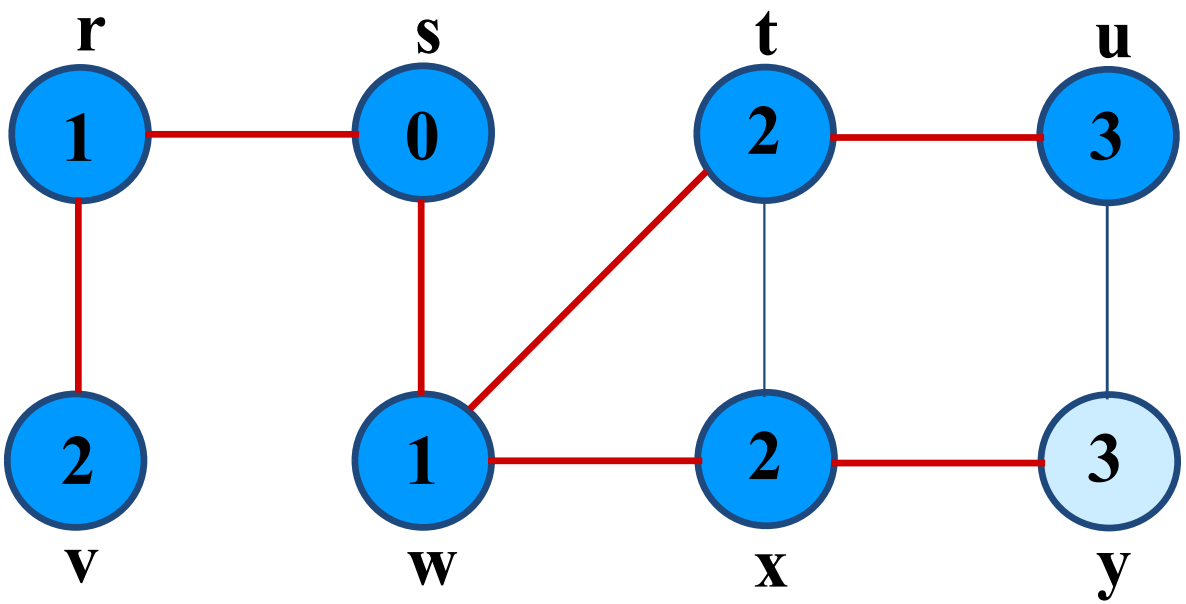
Q:	v	u	y
	2	3	3

Example (BFS)



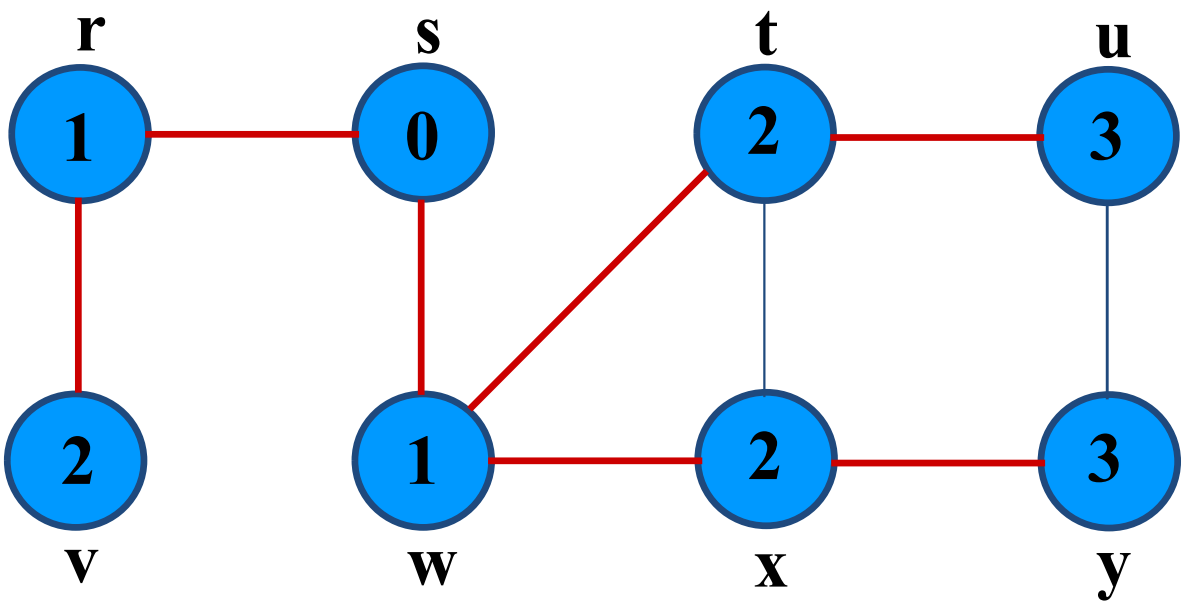
Q:	u	y
	3	3

Example (BFS)



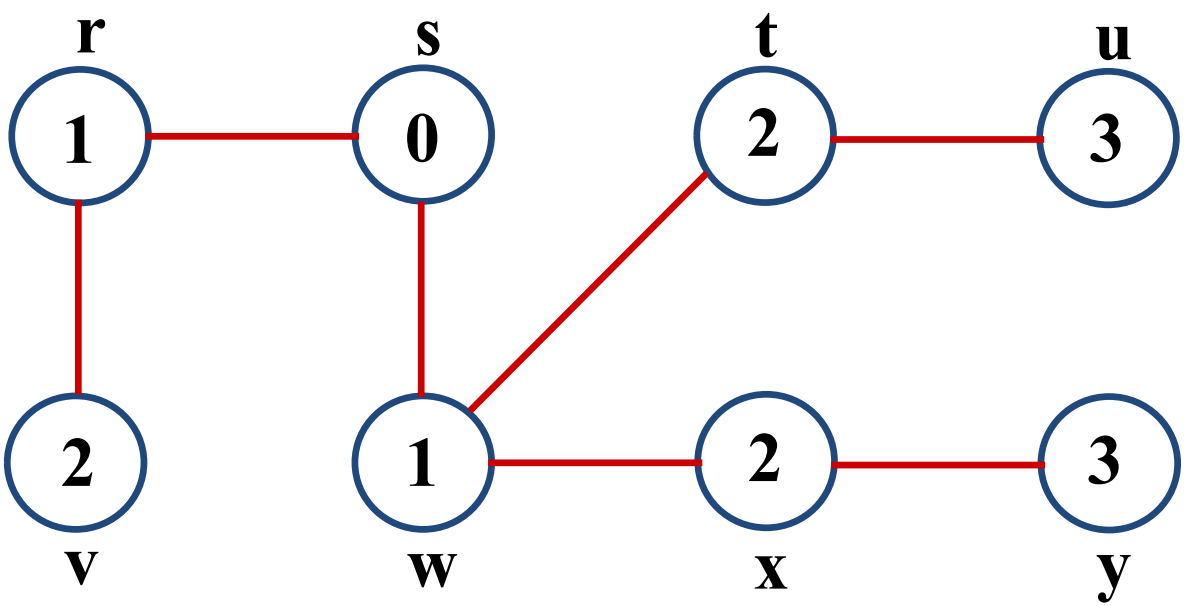
Q:	y
	3

Example (BFS)



Q: ∅

Example (BFS)



BF Tree

Analysis of BFS

- Initialization takes $O(V)$.
- Traversal Loop
  - After initialization, each vertex is enqueued and dequeued at most once, and each operation takes $O(1)$. So, total time for queuing is $O(V)$.
  - The adjacency list of each vertex is scanned at most once. The sum of lengths of all adjacency lists is $\Theta(E)$.
- Summing up over all vertices => total running time of BFS is $O(V+E)$, linear in the size of the adjacency list representation of graph.

Breadth-first Tree

- For a graph $G = (V, E)$ with source s, the **predecessor subgraph** of G is $G_\pi = (V_\pi, E_\pi)$ where
  - $V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$
  - $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$
- The predecessor subgraph G_π is a **breadth-first tree** if:
  - V_π consists of the vertices reachable from s and
  - for all $v \in V_\pi$, there is a unique simple path from s to v in G_π that is also a shortest path from s to v in G.
- The edges in E_π are called **tree edges**.
  $|E_\pi| = |V_\pi| - 1$.

Depth-first Search (DFS)

- Explore edges out of the most recently discovered vertex v.
- When all edges of v have been explored, backtrack to explore other edges leaving the vertex from which v was discovered (its *predecessor*).
- "Search as deep as possible first."
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

Depth-first Search

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
  - 2 **timestamps** on each vertex. Integers between 1 and 2|V|.
    - $d[v]$ = ***discovery time*** (v turns from white to gray)
    - $f[v]$ = ***finishing time*** (v turns from gray to black)
  - $\pi[v]$: predecessor of $v = u$, such that v was discovered during the scan of u's adjacency list.
- Uses the same coloring scheme for vertices as BFS.

Pseudo-code

DFS(*G*)

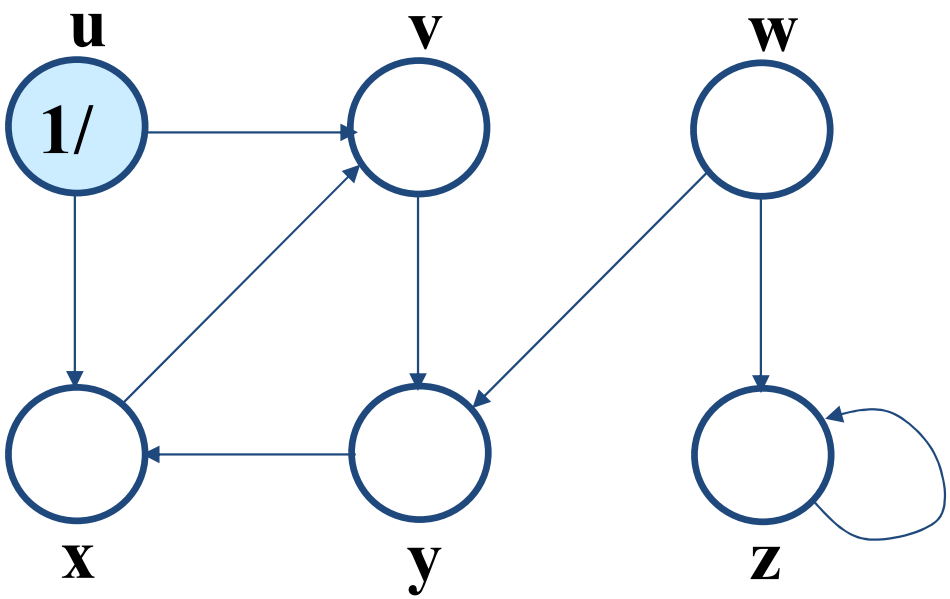
1. **for** each vertex $u \in V[G]$
2. **do** $color[u] \leftarrow$ white
3. $\pi[u] \leftarrow$ NIL
4. $time \leftarrow 0$
5. **for** each vertex $u \in V[G]$
6. **do if** $color[u]$ = white
7. **then** DFS-Visit(u)

DFS-Visit(*u*)

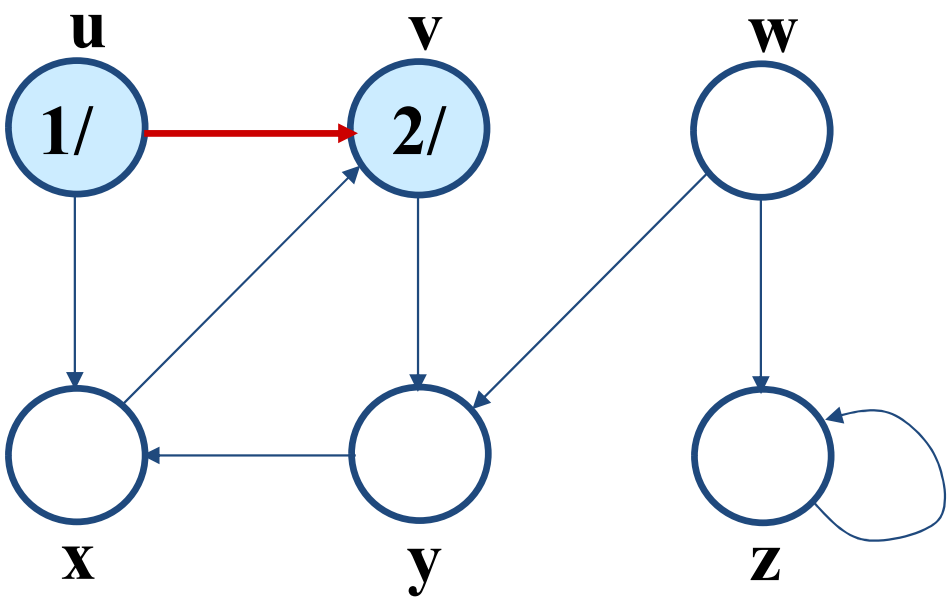
1. ***color*[*u*] ← GRAY ∇ White vertex *u* has been discovered**
2. ***time* ← *time* + 1**
3. ***d*[*u*] ← *time***
4. **for each *v* ∈ *Adj*[*u*]**
5. **do if *color*[*v*] = WHITE**
6. **then π[*v*] ← *u***
7. **DFS-Visit(*v*)**
8. ***color*[*u*] ← BLACK ∇ Blacken *u*; it is finished.**
9. ***f*[*u*] ← *time* ← *time* + 1**

Uses a global timestamp *time*.

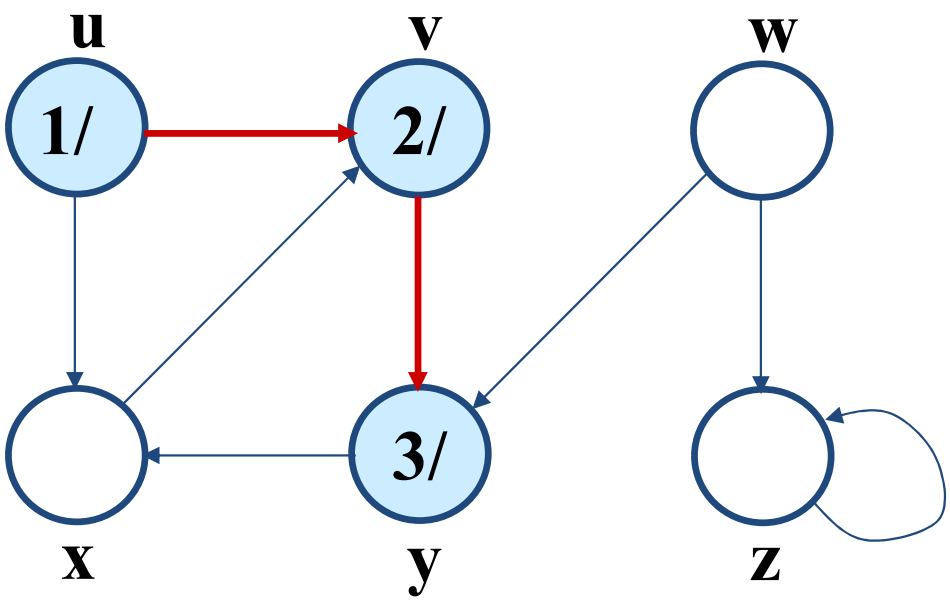
Example (DFS)



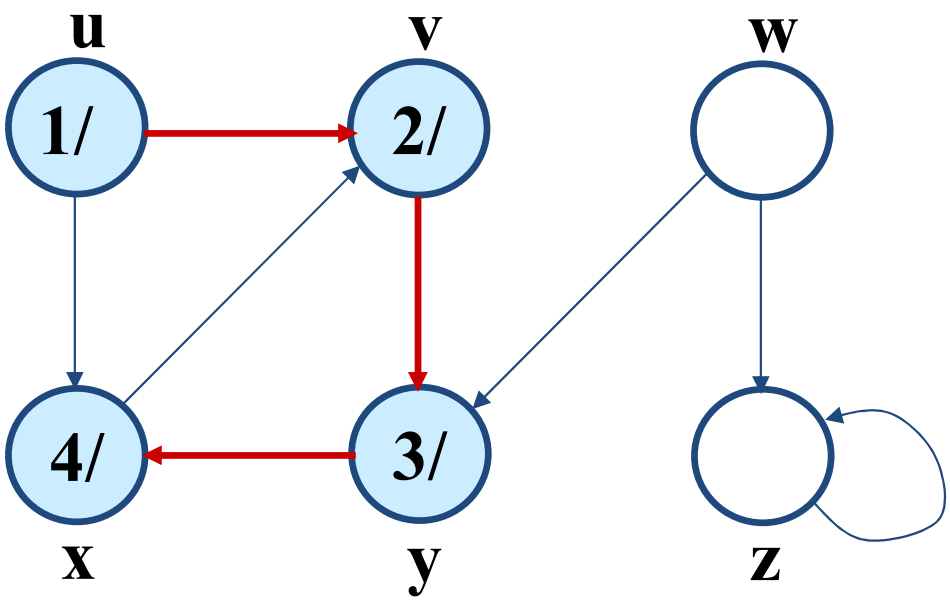
Example (DFS)



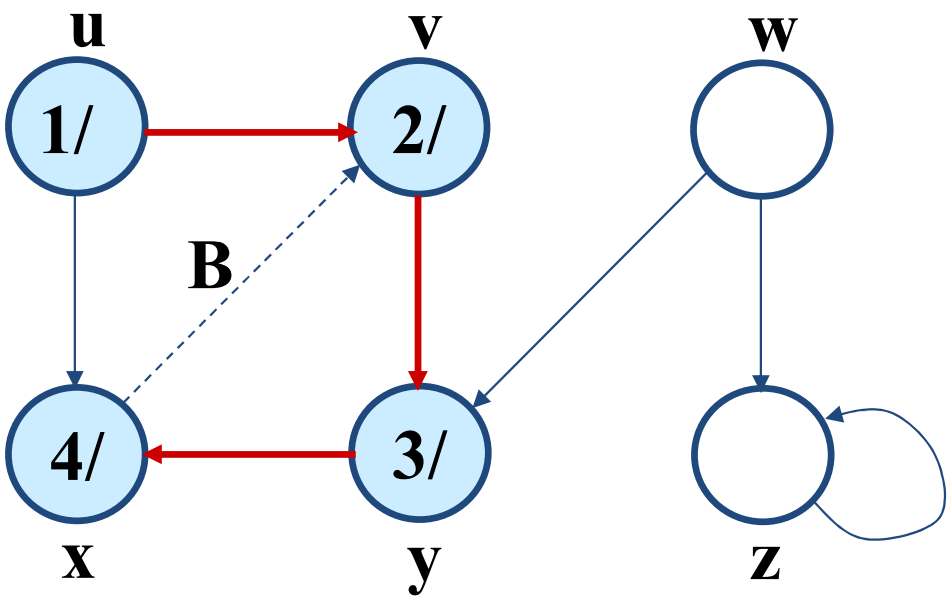
Example (DFS)



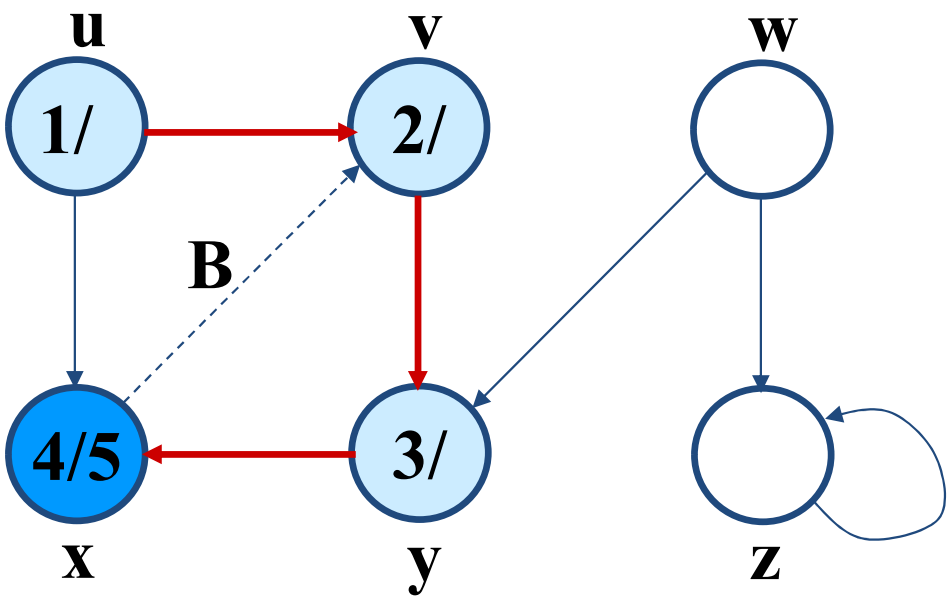
Example (DFS)



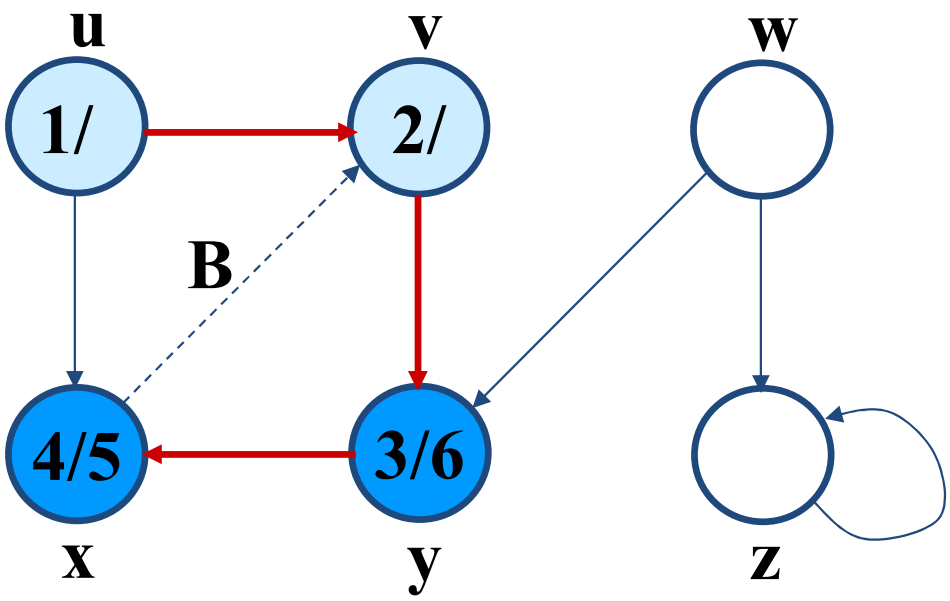
Example (DFS)



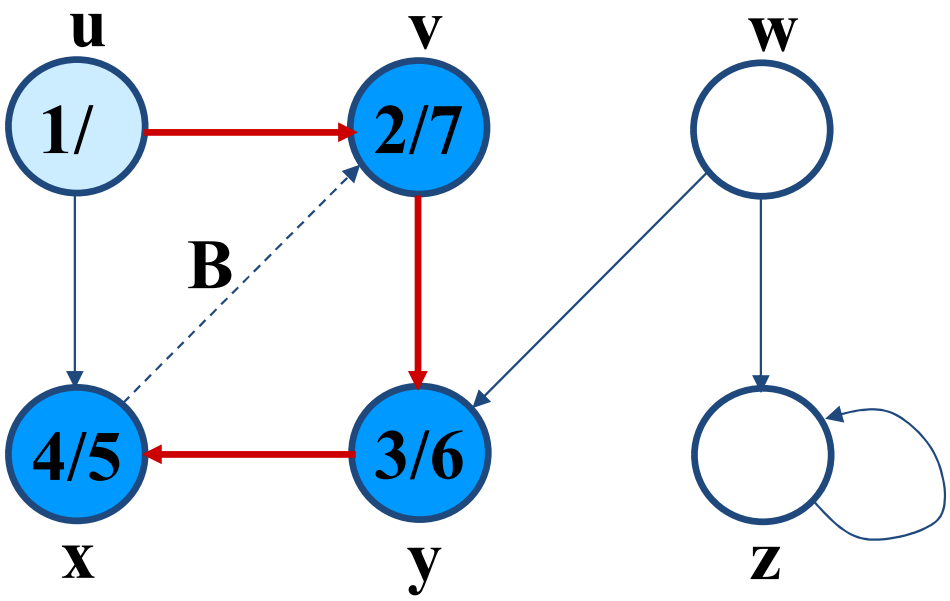
Example (DFS)



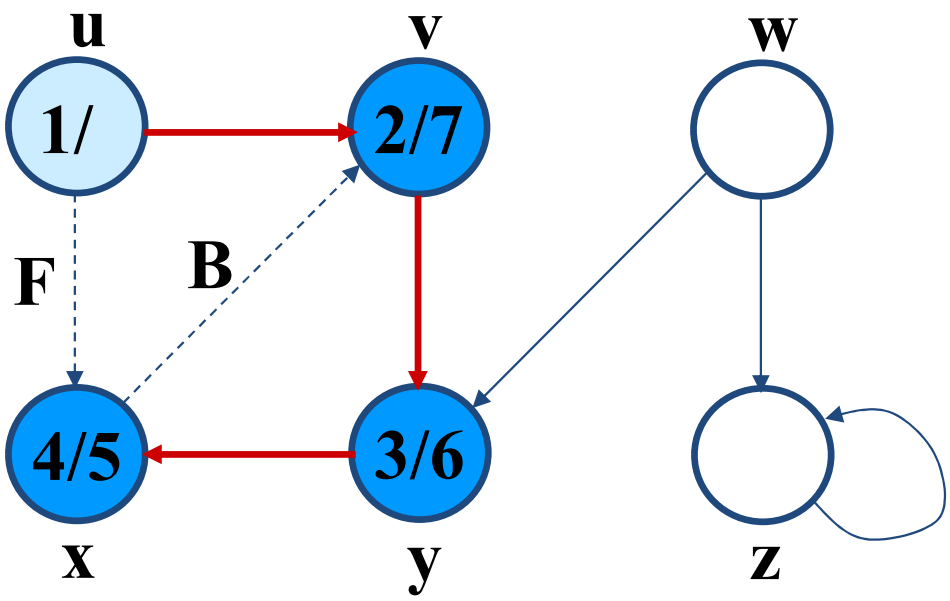
Example (DFS)



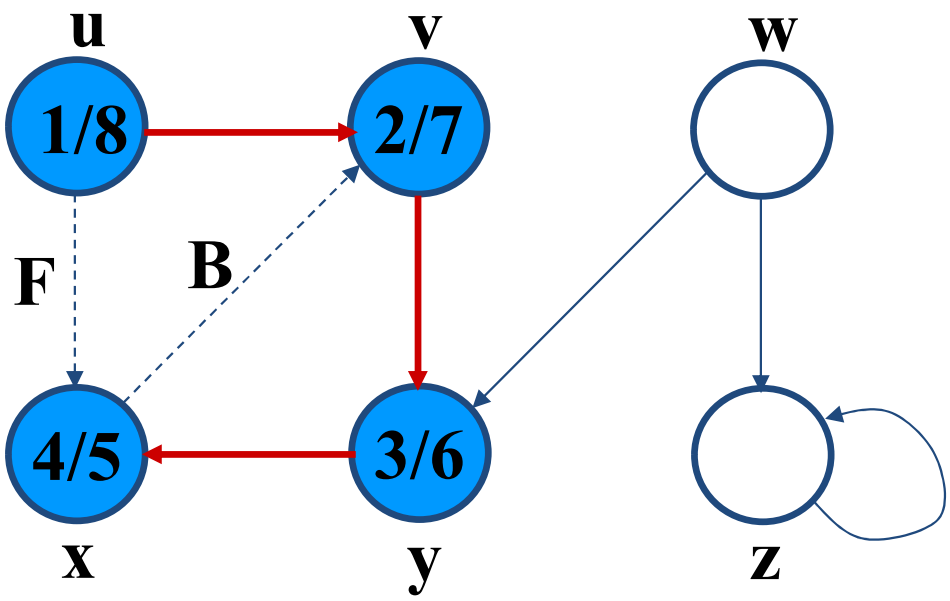
Example (DFS)



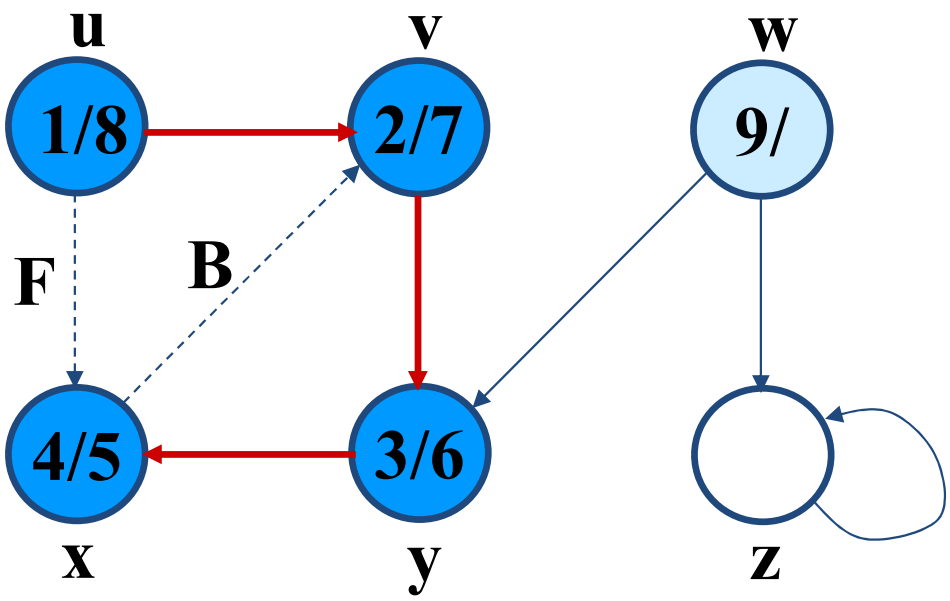
Example (DFS)



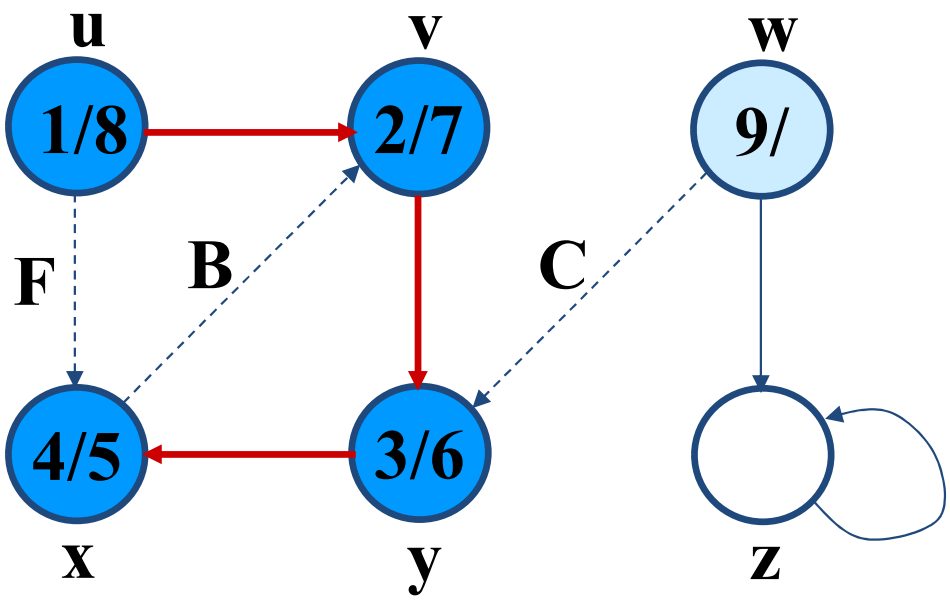
Example (DFS)



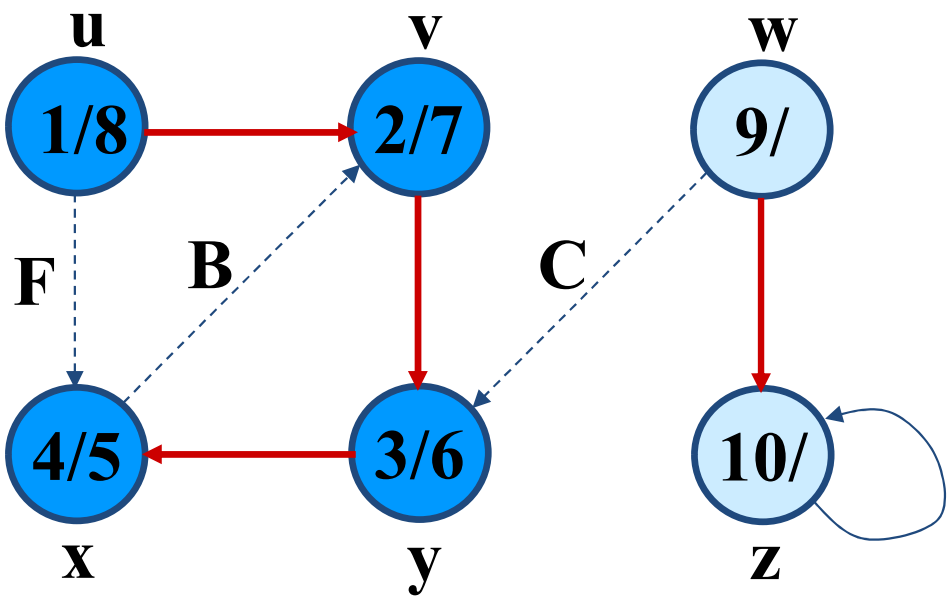
Example (DFS)



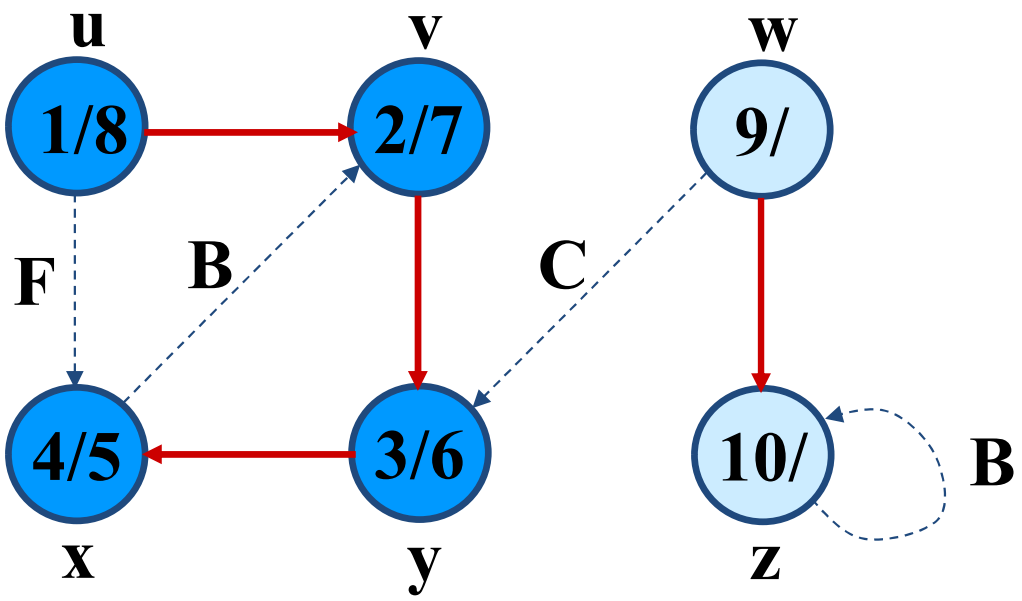
Example (DFS)



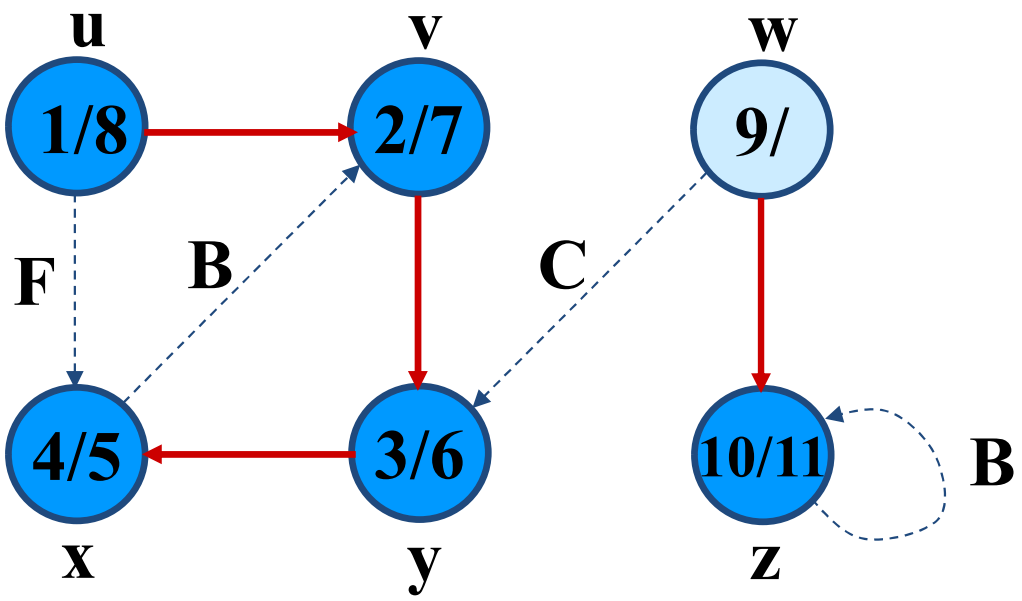
Example (DFS)



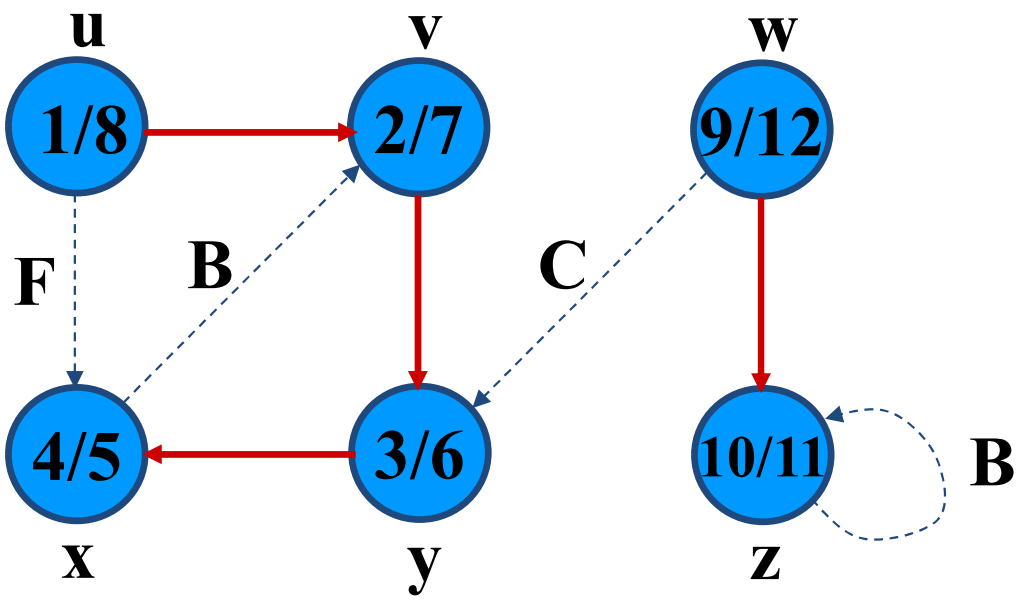
Example (DFS)



Example (DFS)



Example (DFS)



Analysis of DFS

- Loops on lines 1-2 & 5-7 take $\Theta(V)$ time, excluding time to execute DFS-Visit.
- DFS-Visit is called once for each white vertex $v \in V$ when it's painted gray the first time. Lines 3-6 of DFS-Visit is executed $|\text{Adj}[v]|$ times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$
- Total running time of DFS is $\Theta(V+E)$.

Depth-First Trees

- Predecessor subgraph defined slightly different from that of BFS.
- The predecessor subgraph of DFS is $G_\pi = (V, E_\pi)$ where $E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\}$.
  - How does it differ from that of BFS?
  - The predecessor subgraph G_π forms a *depth-first forest* composed of several *depth-first trees*. The edges in E_π are called *tree edges*.

Definition:
Forest: An acyclic graph G that may be disconnected.

TREE

- A tree is an undirected graph which is connected and acyclic. It is easy to show that if graph *G*
  - G (V,E) is connected.
  - G (V,E) is acyclic.
  - |E| =|V| -1

Minimum Spanning Tree(MST)

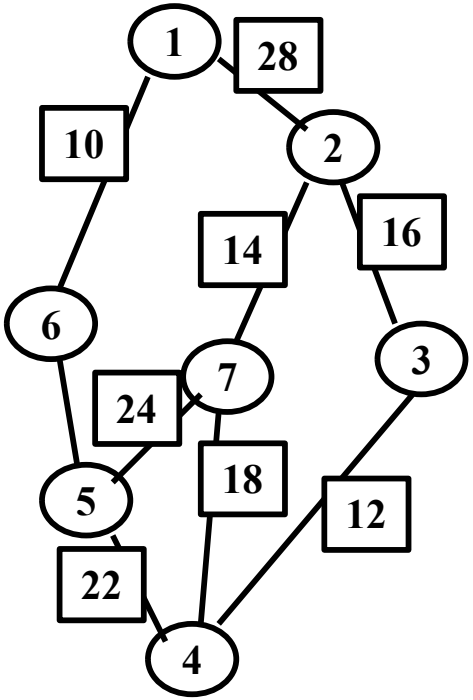
- In circuit design, sometimes we need to make short some of the pins of the circuit by connecting them with wire.
- To interconnect a set of n pins, we can use an arrangement of n-1 wires, each connecting 2 pins.
- Of all arrangements, the one that uses the least amount of wire is usually the most desirable.
- We can model this wiring problem with a connected, undirected weighted graph G(V,E).
- We will find an acyclic subset T from E that connects all the vertices and whose total weight is minimized.
- Since T is acyclic and connects all vertices, it is called a spanning tree.
- The spanning tree which has minimum weight is called minimum spanning tree.

Spanning Trees

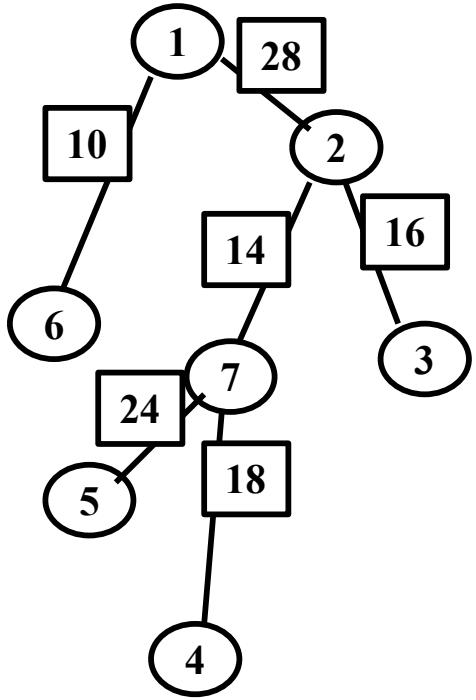
- A *spanning tree* in an undirected graph *G(V,E)* is a subset of edges $T \subseteq E$ that are acyclic and connect all the vertices in V.
- It follows from the above conditions that a spanning tree must consist of exactly n-1 edges.
- Now suppose that each edge has a weight associated with it: $w : E \to Z$ Say that the weight of a tree T is the sum of the weights of its edges;

$$w(T) = \sum_{e \in T} w(e)$$

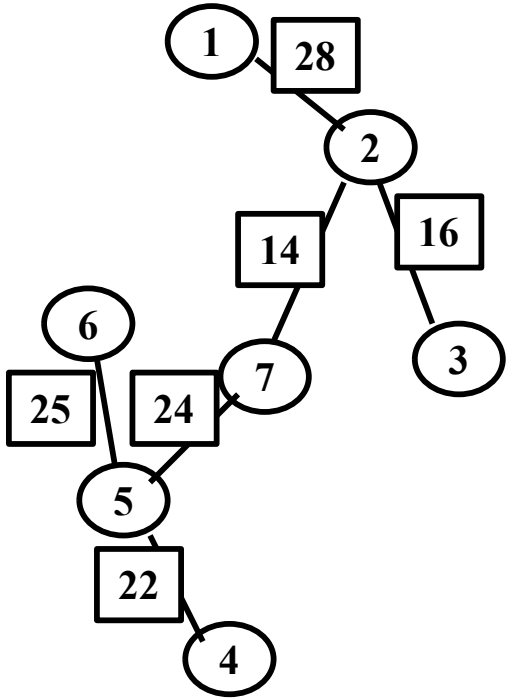
Example: Spanning Tree



connected graph



spanning tree
cost = 110

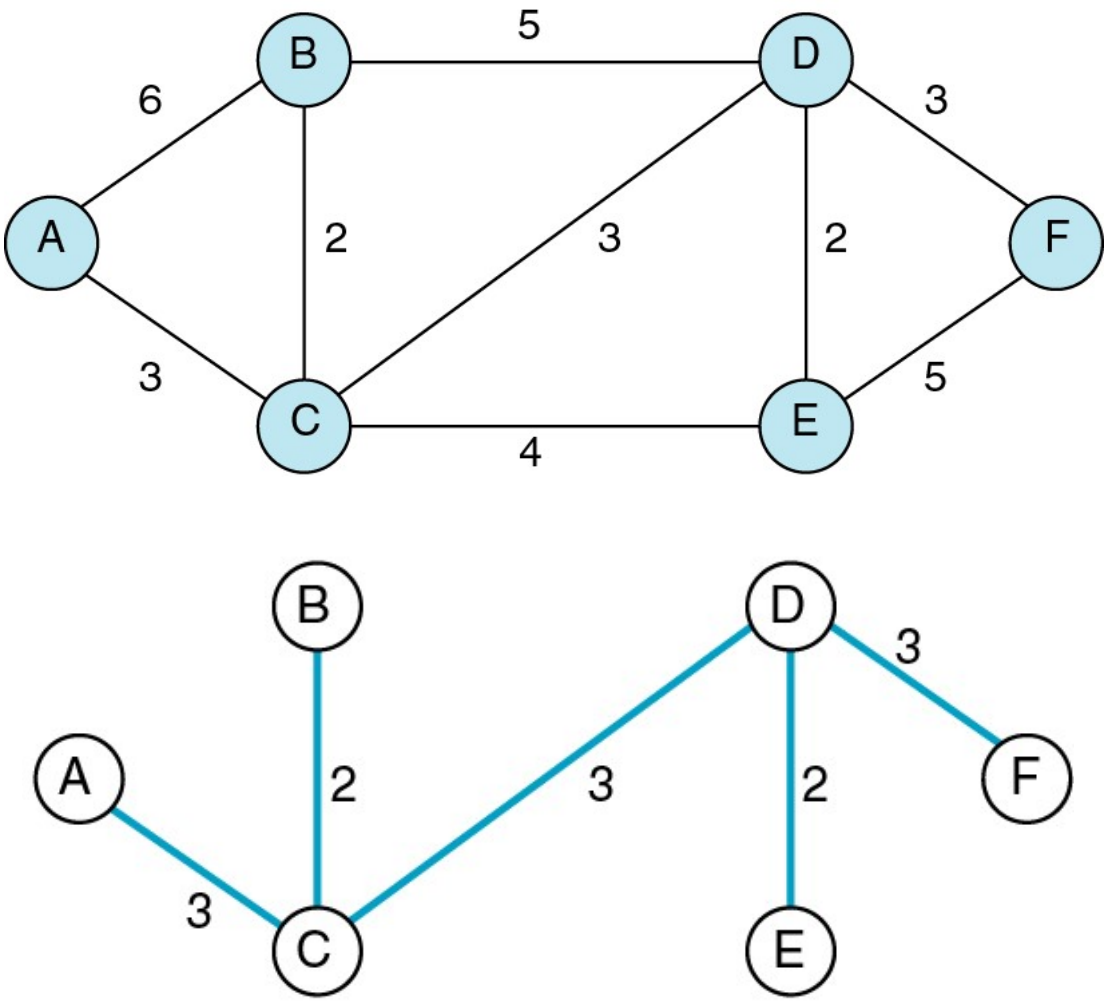


spanning tree
cost = 129

Minimum Spanning Tree

- A ***spanning tree*** is a tree that contains all of the vertices in the graph and enough of its edges to form a tree.
- The ***minimum spanning tree*** of a network is a spanning tree in which the sum of its edge weights is guaranteed to be minimal.
- It is possible to have more than one minimum spanning tree – however the weights of the different MSTs will be the same.
- The *minimum spanning tree* in a weighted graph *G(V,E)* is one which has the smallest weight among all spanning trees in *G(V,E).*

Minimum Spanning Tree



The bottom graph is a spanning tree of the top graph.

Note that every node from the top graph is represented in the spanning tree.

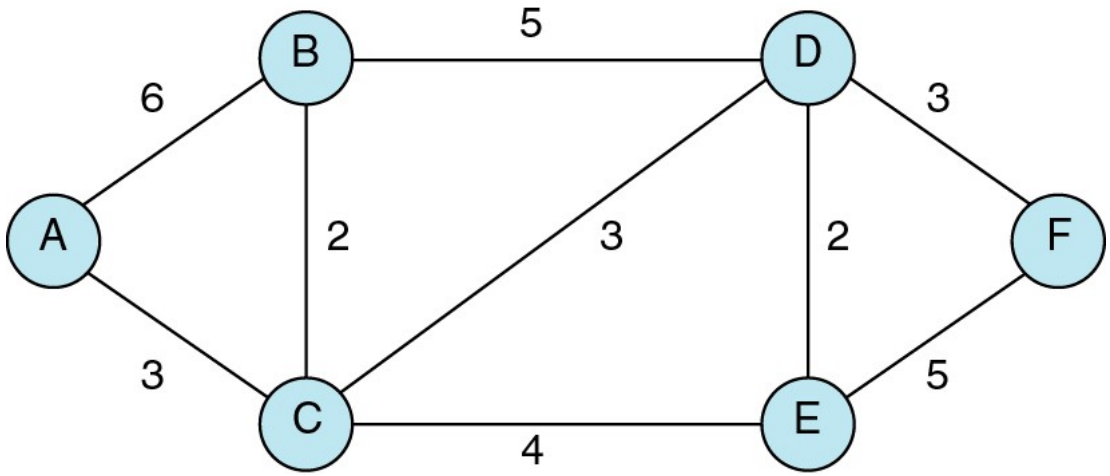
The bottom graph also happens to be the minimum spanning tree.

Minimum Spanning Tree

- There are many applications for minimum spanning trees, all with the requirement to minimize some aspect of the graph, such as the distance among all the vertices in the graph.
- For example, given a network of computers, we could create a graph that connects all of the computers.
- The MST gives us the shortest length of cable that can be used to connect all of the computers while ensuring that there is a path between any two computers.

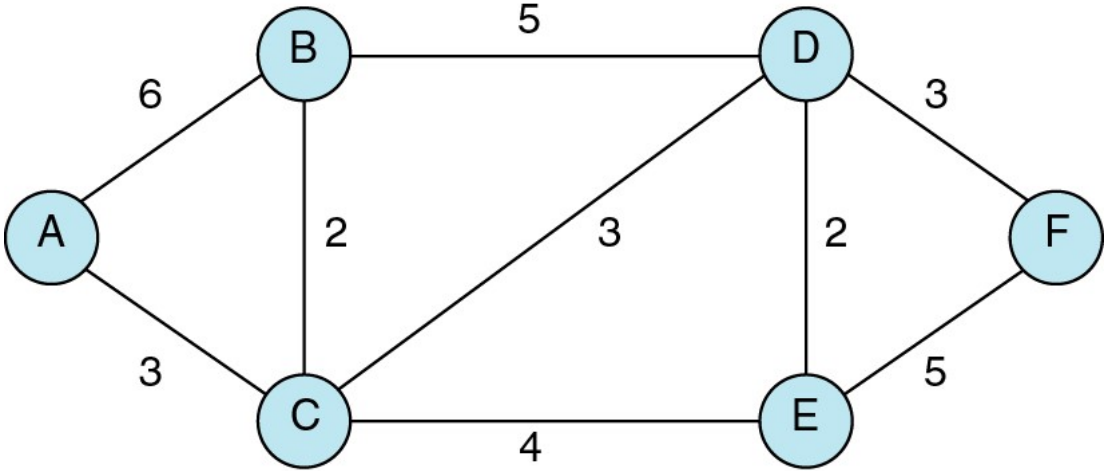
Minimum Spanning Tree

- Before we develop a formal algorithm to determine the MST of a graph, we will manually determine the MST of an example graph.



Minimum Spanning Tree

- We can start with any vertex, so we'll just start with A.
- Then, we add the vertex that gives the minimum-weighted edge with A.
- Our options are AB or AC – we choose AC because 3 is less than 6

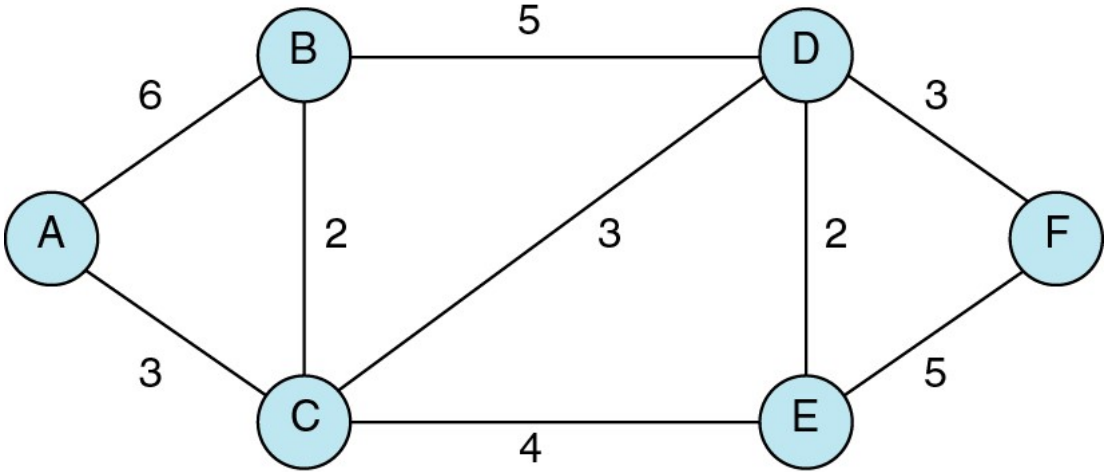


MST

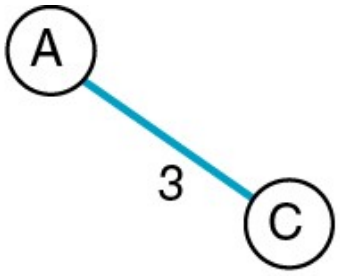
(a) Insert first vertex

Minimum Spanning Tree

- Now, we have vertices A and C in our MST.
- From these 2, we choose the edge with the minimum weight that connects a vertex in our MST to a vertex not already included in our MST.



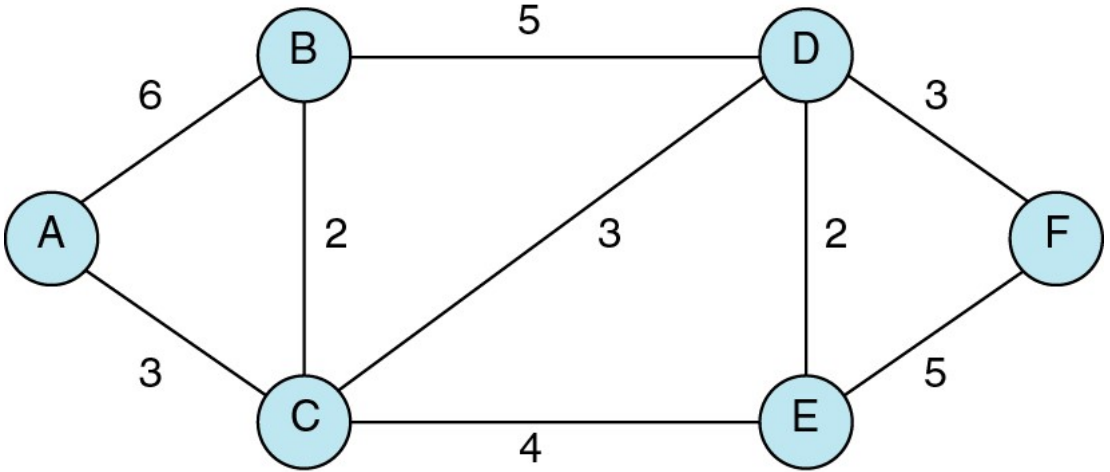
MST



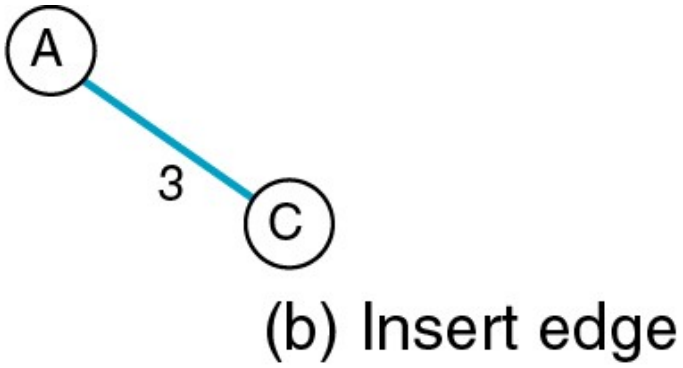
(b) Insert edge

Minimum Spanning Tree

- Our options are
  - CB, weight = 2
  - CD, weight = 3
  - CE, weight = 4
  - AB, weight = 6
- We choose CB because it has the minimum weight.



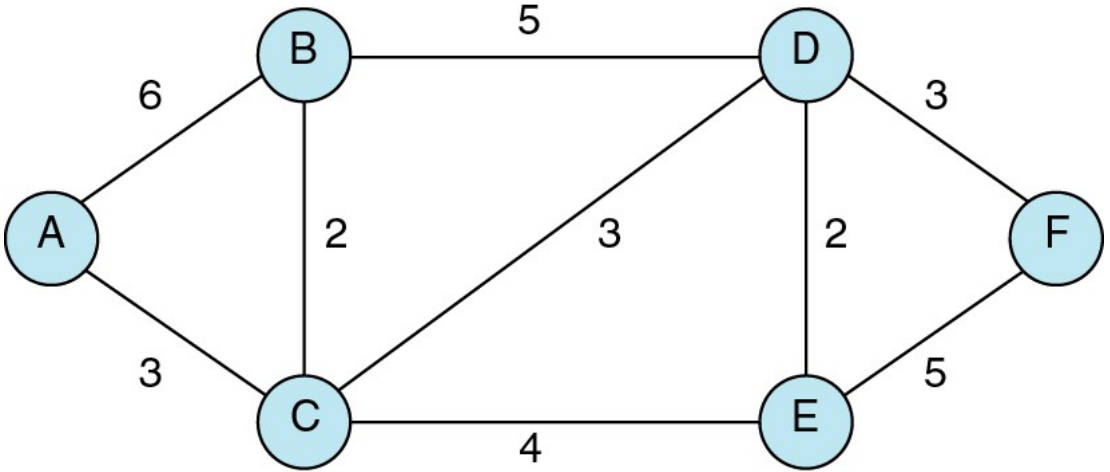
MST

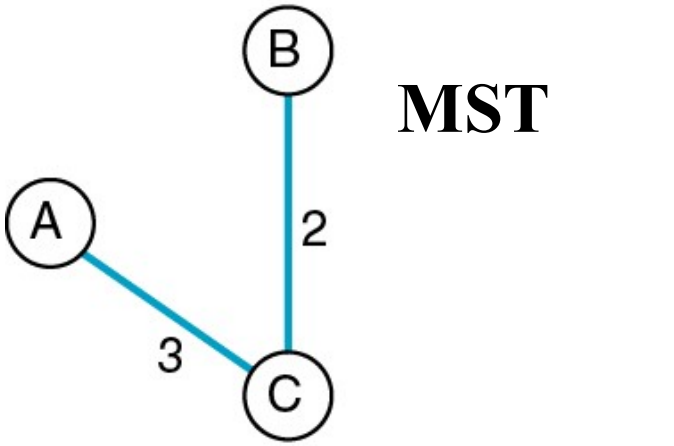


(b) Insert edge

Minimum Spanning Tree

- Now we continue with the same process – here are our choices:
  - BD, weight = 5
  - CD, weight = 3
  - CE, weight = 4
- We choose CD because it has the minimum weight.



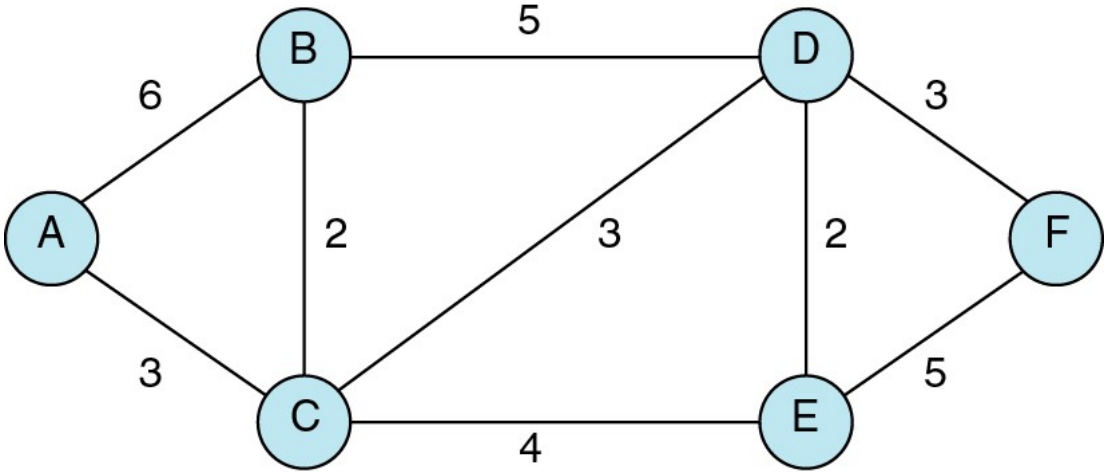


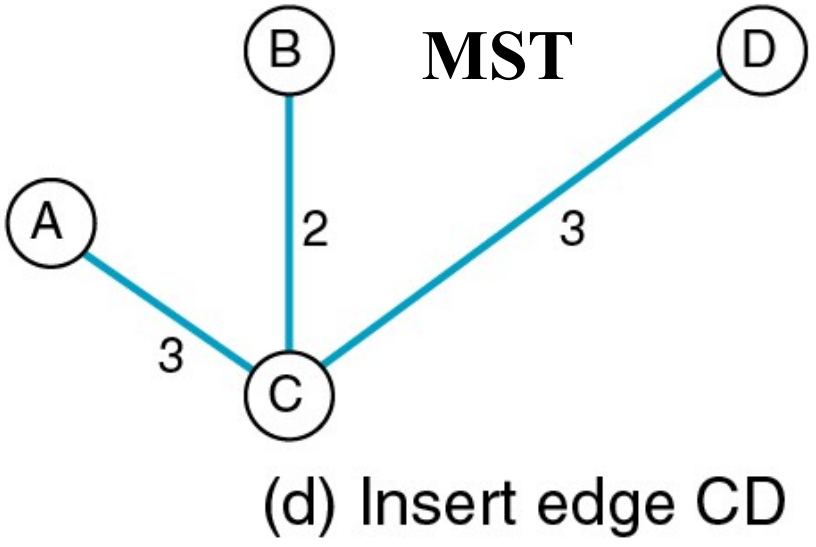
MST

(c) Insert edge BC

Minimum Spanning Tree

- Now we continue with the same process – here are our choices:
  - CE, weight = 4
  - DE, weight = 2
  - DF, weight = 3
- We choose DE because it has the minimum weight.

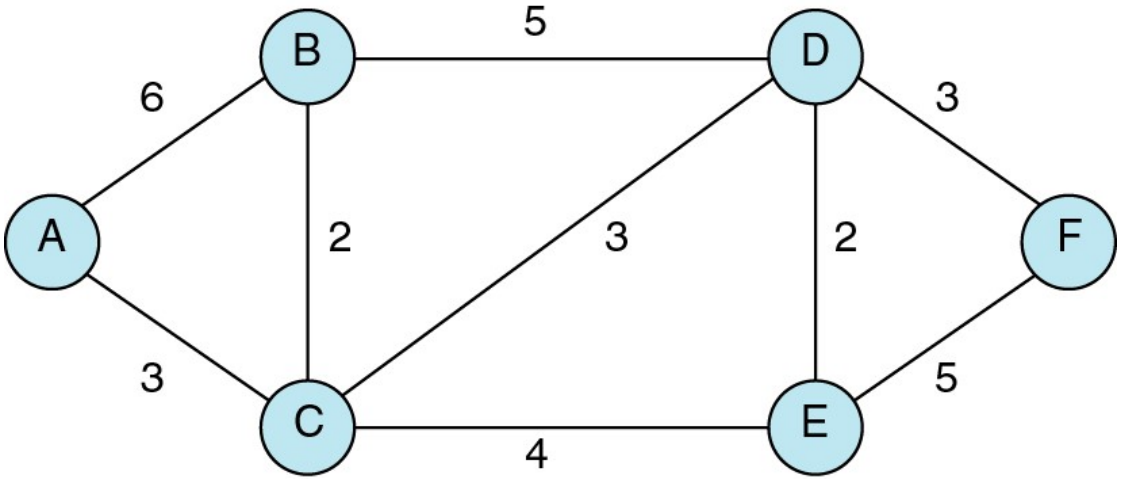


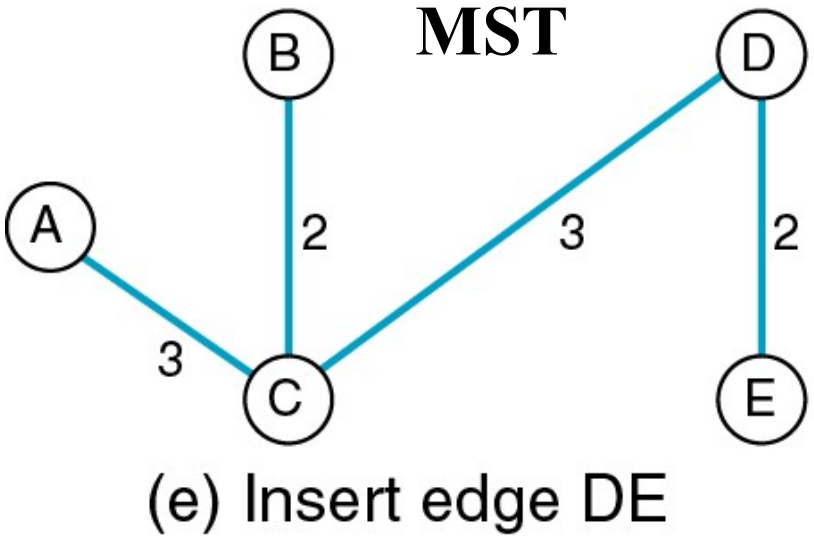


(d) Insert edge CD

Minimum Spanning Tree

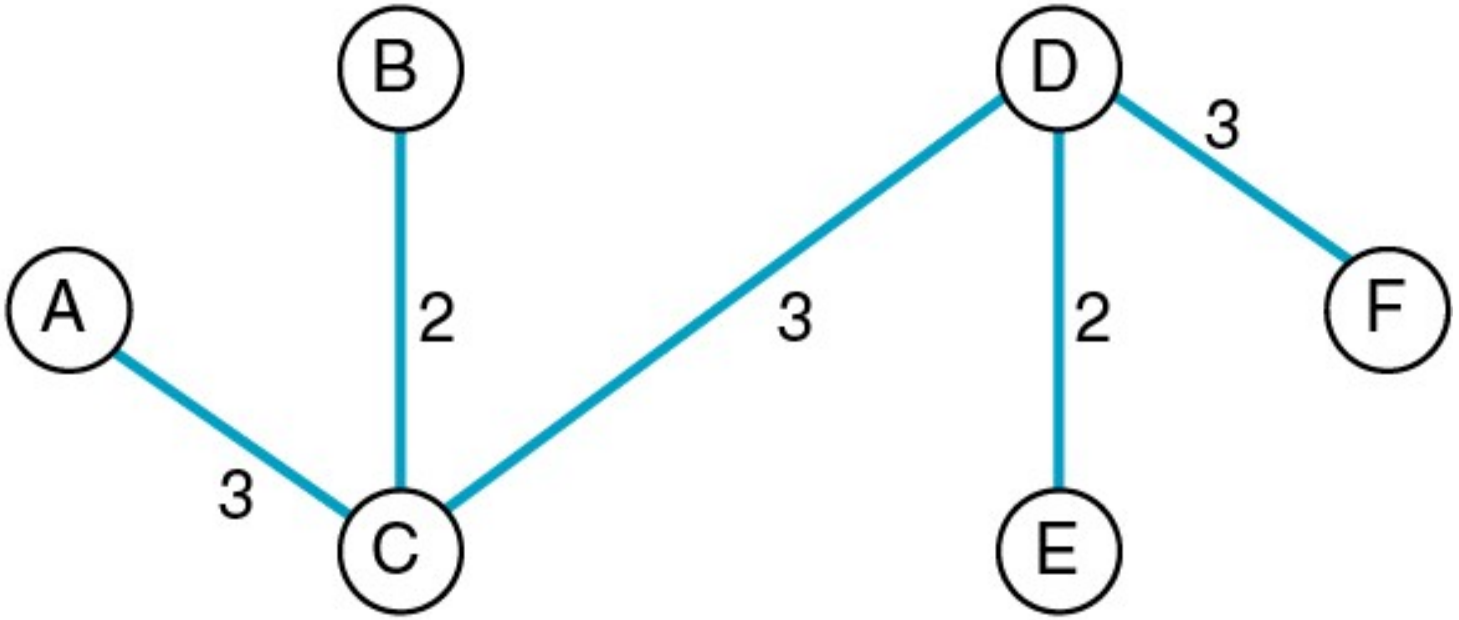
- Now we are down to our last node. Our choices are:
  - EF, weight = 5
  - DF, weight = 3
- We choose DF because it has the minimum weight.





Minimum Spanning Tree

- Here is our final MST.



Minimum Spanning Tree

- The algorithm just described is Prim's algorithm.
- Prim's algorithm finds a minimum spanning tree that begins at any vertex.
- Hence, the MST produced by Prim's algorithm may vary depending on which vertex you start at.

Prim's Algorithm

- A greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge.
- The algorithm will start with a tree that includes only a minimum-cost edge of graph G.
- Then, edges are added to this tree one by one.
- The next edge *(i,j)* to be added is such that *i* is a vertex already added in the tree, *j* is a vertex not yet included, and the cost of *(i,j)*, *cost*[*i,j*], is minimum among all edges (*k,l*) such that vertex *k* is in the tree and vertex *l* is not in the tree.
- To determine this edge (*i,j*) efficiently, we associate with each vertex *j* not yet included in the tree a value *near*[*j*]
- The value near[*j*] is a vertex in the tree such that *cost*[*j, near*[*j*]] is minimum among all choices for *near*[*j*].

Prim's Algorithm (cont.)

- We define *near*[*j*]=0 for all vertices j that are already in the tree.
- The next edge to include is defined by the vertex *j such* that *near*[*j*] ≠ 0 (*j* not already in the tree) and *cost*[*j*, *near*[*j*]] is minimum.

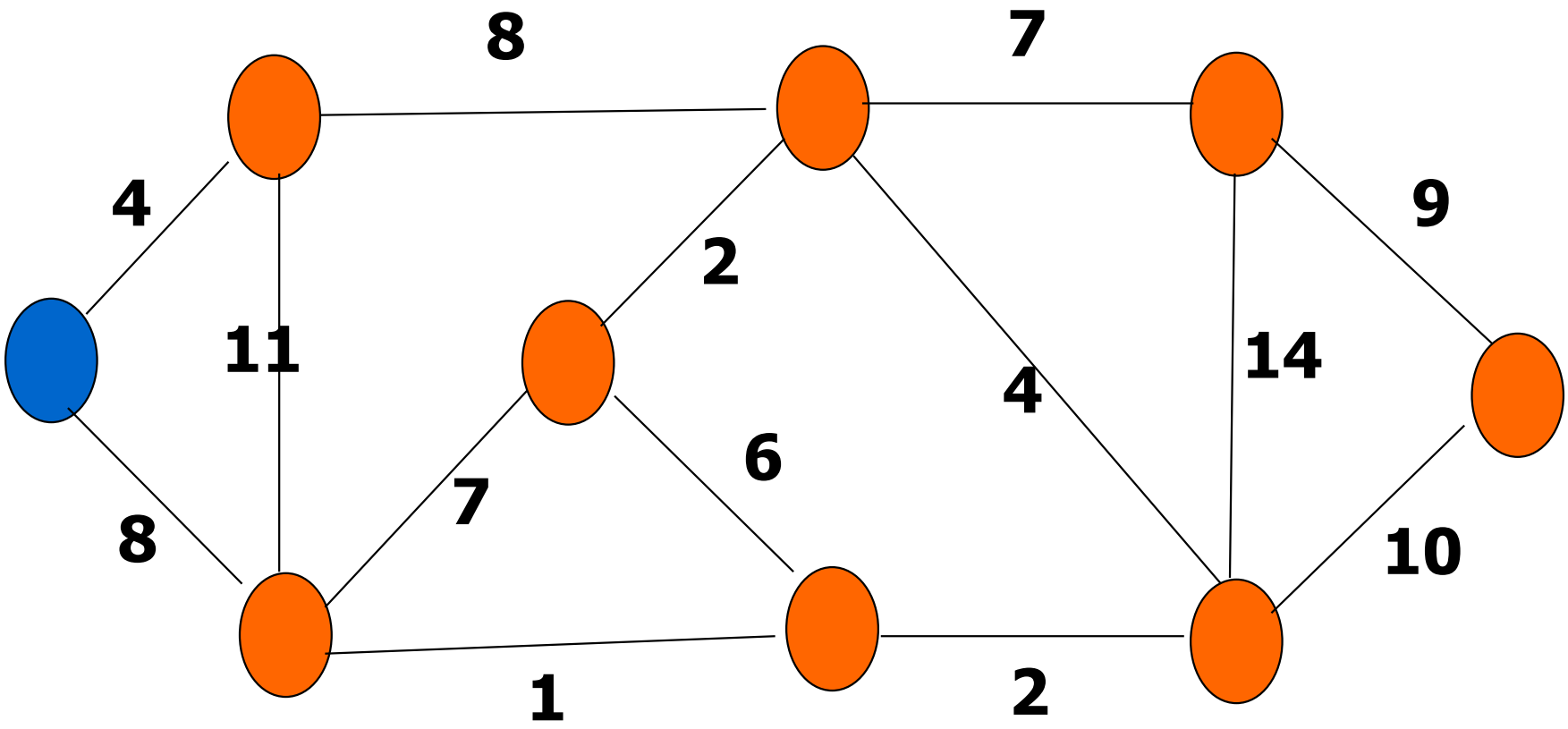
Prim's Algorithm Pseudocode

```
1.  Algorithm Prim(E, cost, n, t)
2.  // E is the set of edges in G. cost[1:n, 1:n] is the cost adjacency matrix of
3.  // an n vertex graph such that cost[i,j] is either a positive real number or
4.  //  ∞ if no edge (i,j) exists. A minimum spanning tree is computed and
5.  // stored as a set of edges in the array t[1:n-1,1:2]. (t[i,1],t[i,2]) is an
6.  // edge in the minimum-cost spanning tree.
7.  //The final cost is returned.
8.   {
9.     Let (k,l) be an edge of minimum cost in E;
10.    mincost := cost[k,l];
11.    t[1,1] :=k; t[1,2] :=l;
12.    for i:=1 to n do                    // Initialized near
13.       if (cost[i,l]  <  cost[i,k]) then near[i] :=l;
14.       else near[i] :=k;
15.  near[k] :=near[l] :=0;
```

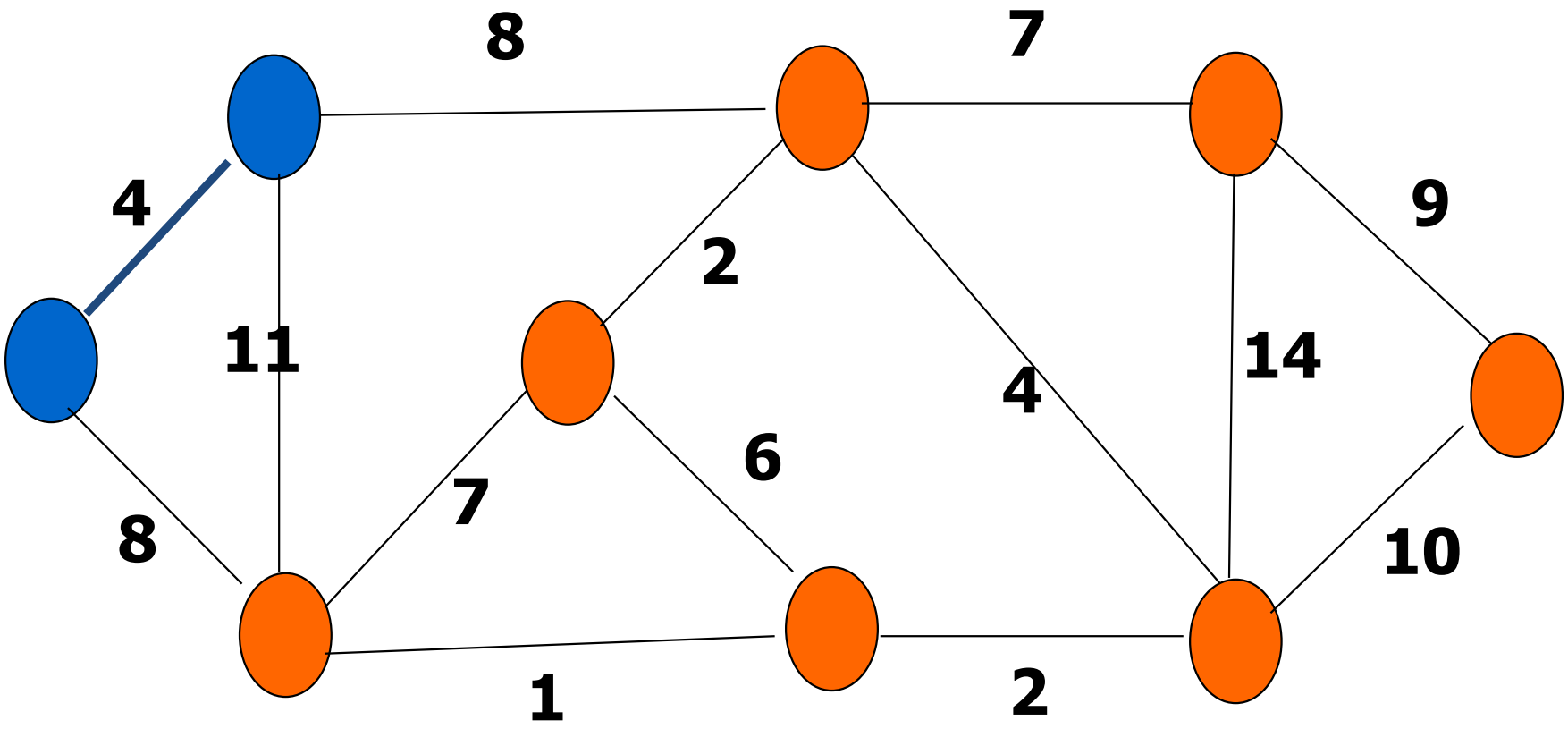
Prim's Algorithm Pseudocode (cont.)

```
16.   for i:=2 to n-1 do
17.    {// Find n-2 additional edges for t
18.      Let j be an index such that near[j] ≠0 and
19.      cost[ j, near[j]] is minimum;
20.      t[i,1] :=j; t[i,2] := near[j];
21.      mincost := mincost+cost[j, near[j]];
22.      near[j] :=0;
23.      for k:=1 to n do // Update near[ ].
24.           if ((near[k] ≠0) and (cost[k, near[k]]  > cost[k,j]) ) then
25.                near[k] :=j;
26.    }
27.    return mincost;
28.  }   //Reference: Computer algorithm-SAHNI, page 221
```

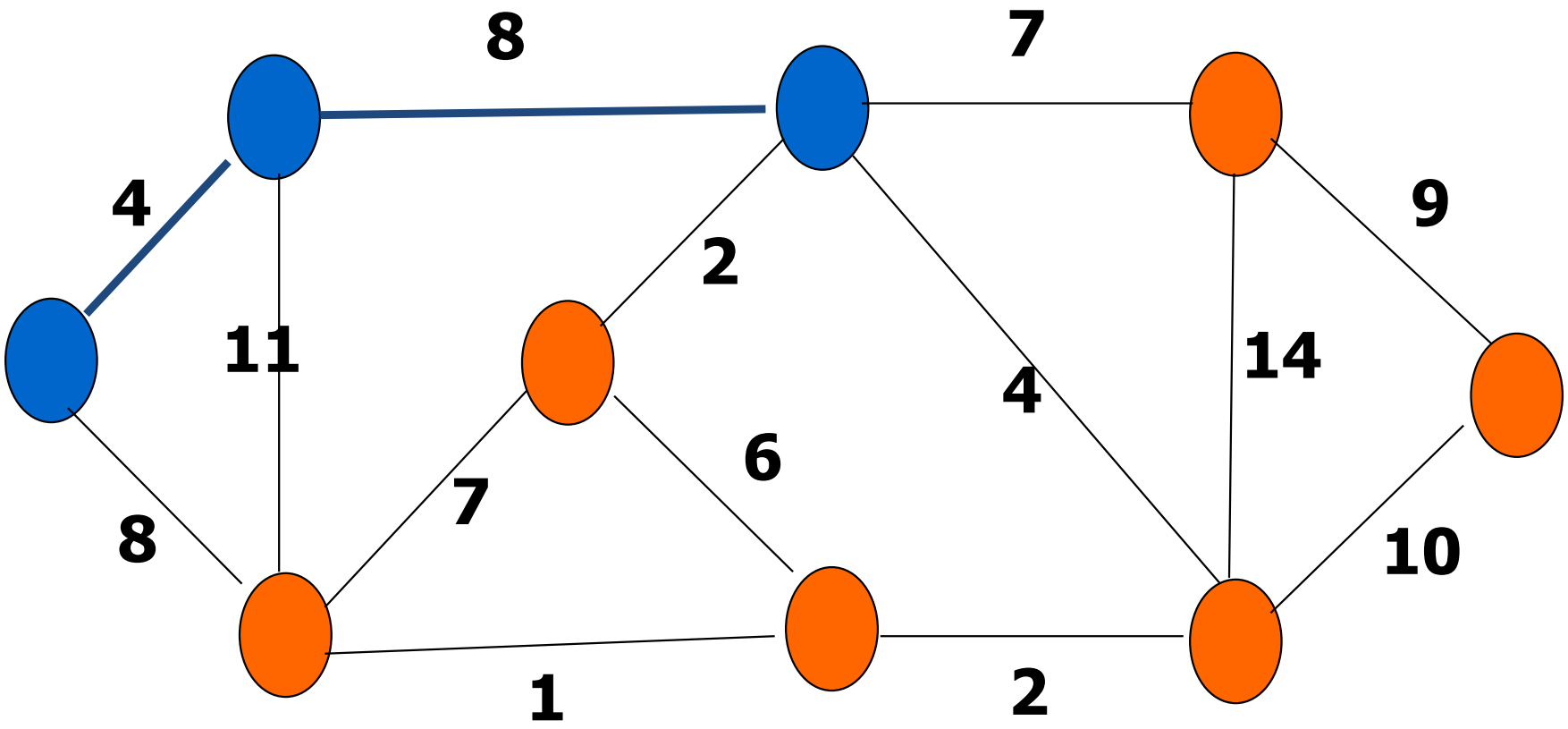
Prim's Algorithm Example



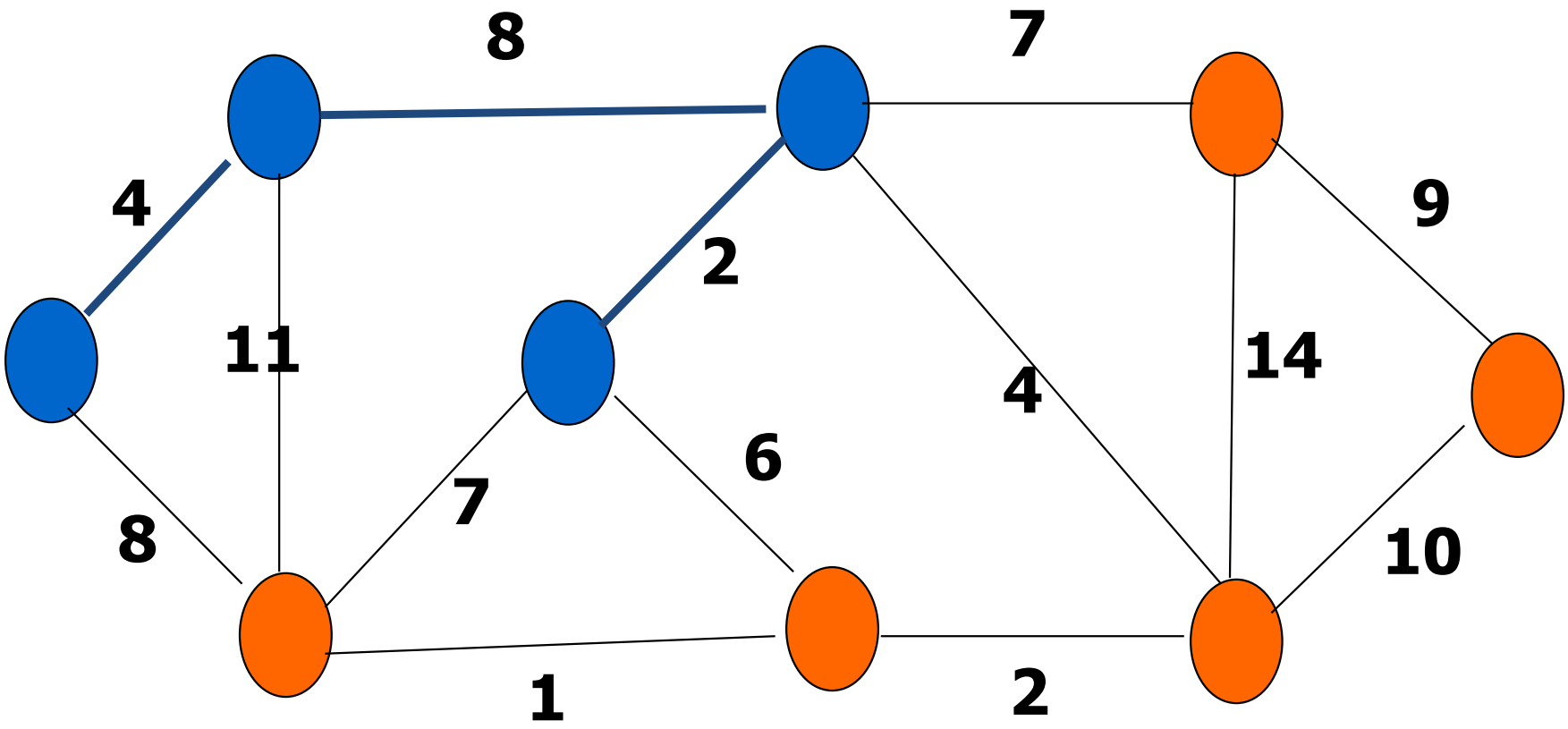
Prim’s Algorithm Example(2)



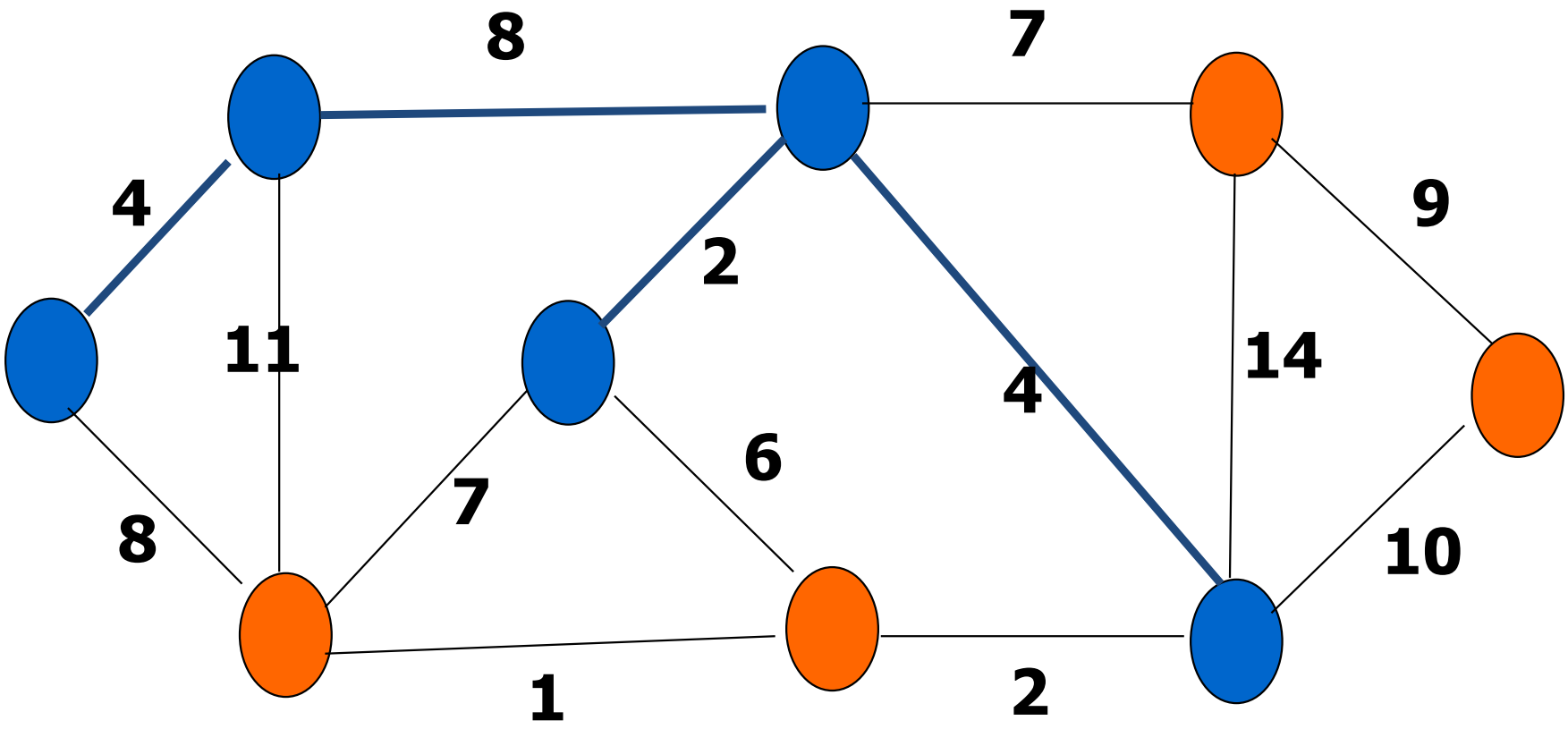
Prim’s Algorithm Example(3)



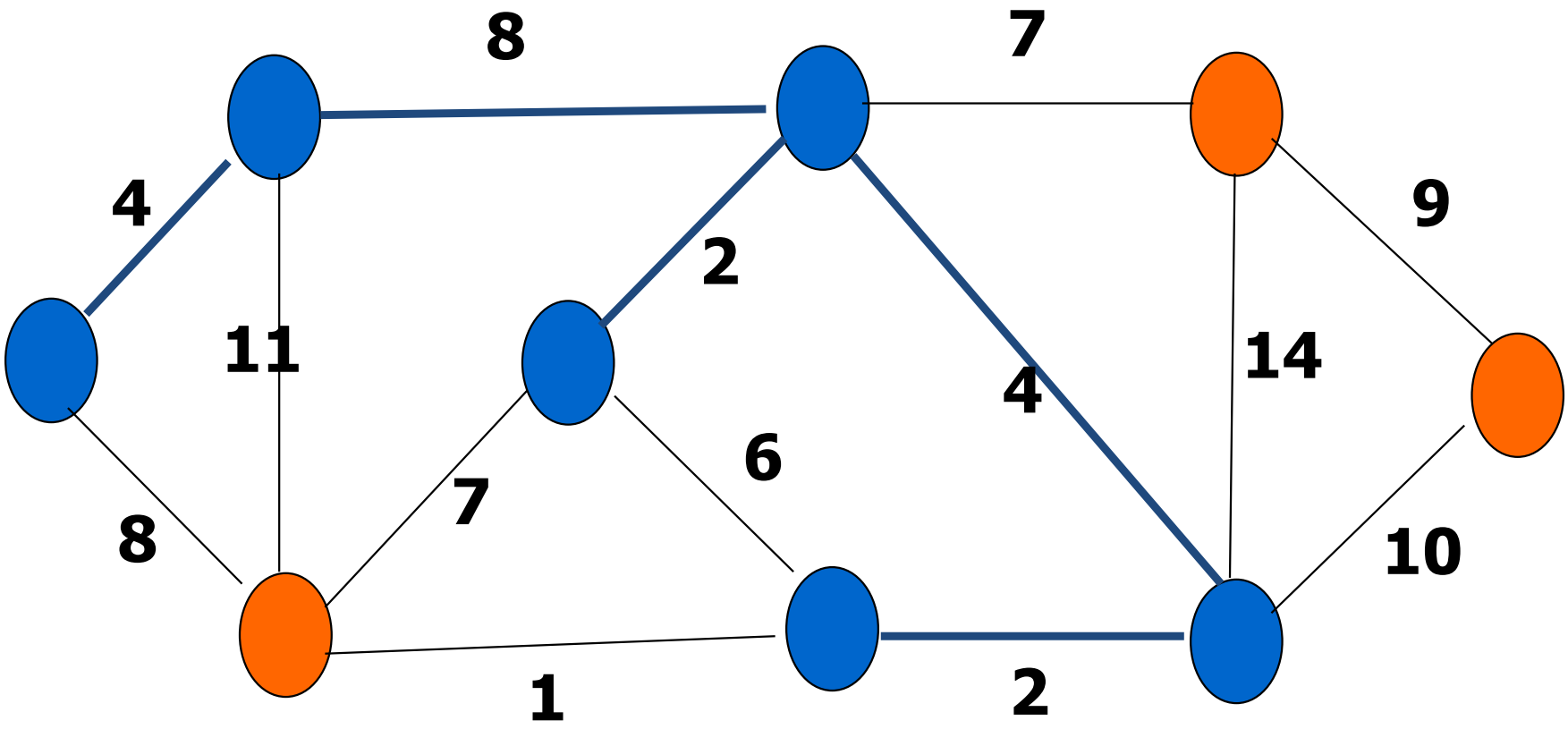
Prim’s Algorithm Example(4)



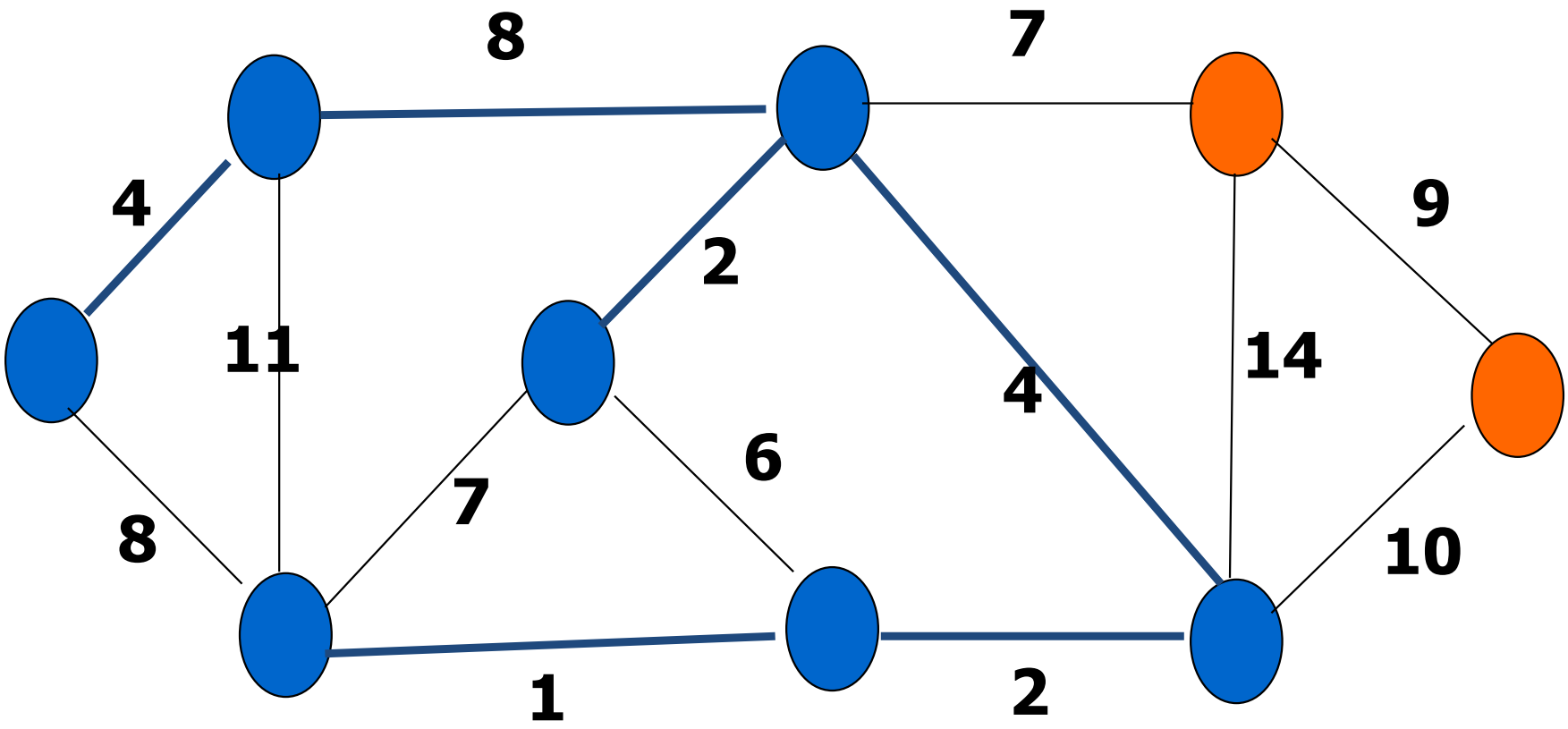
Prim’s Algorithm Example(5)



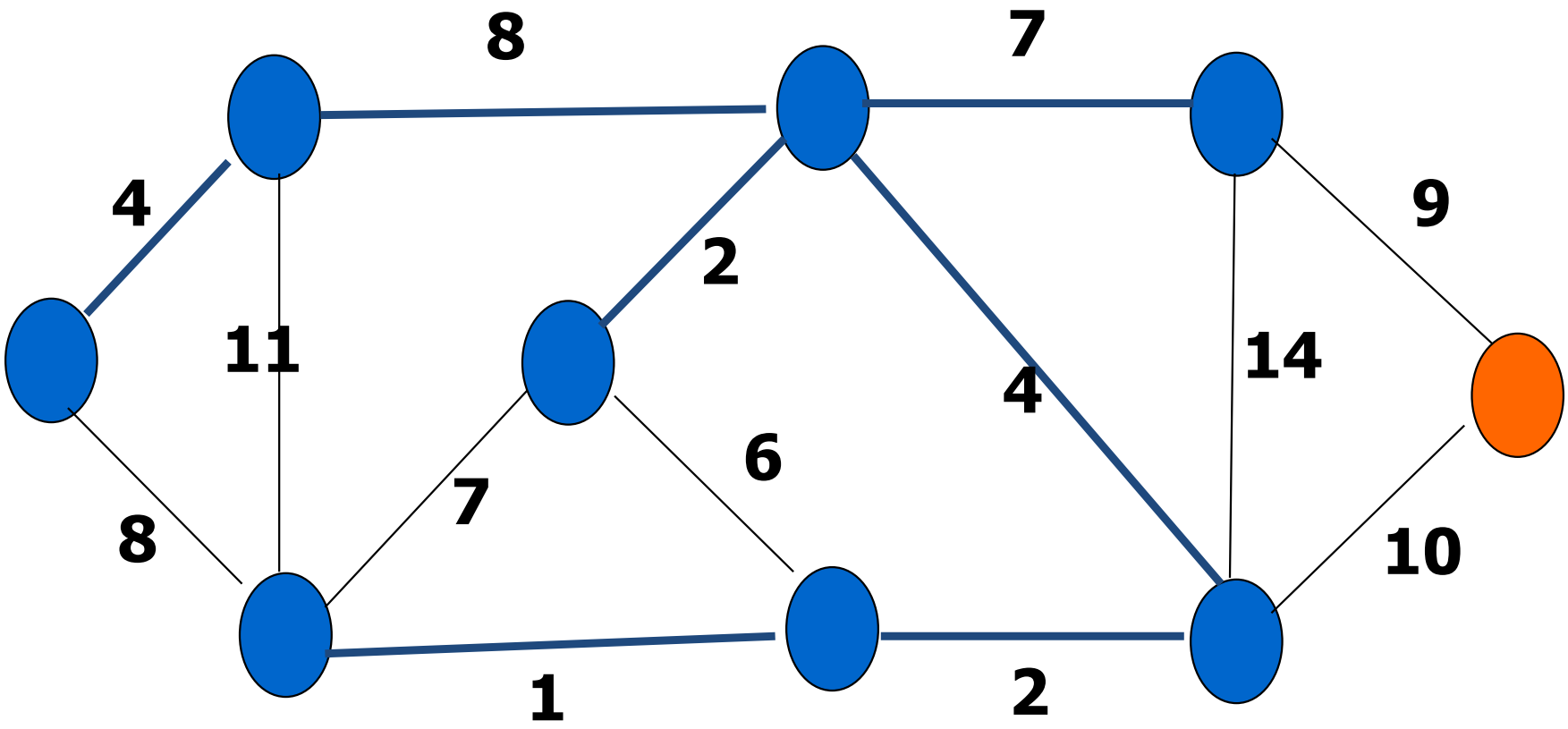
Prim’s Algorithm Example(6)



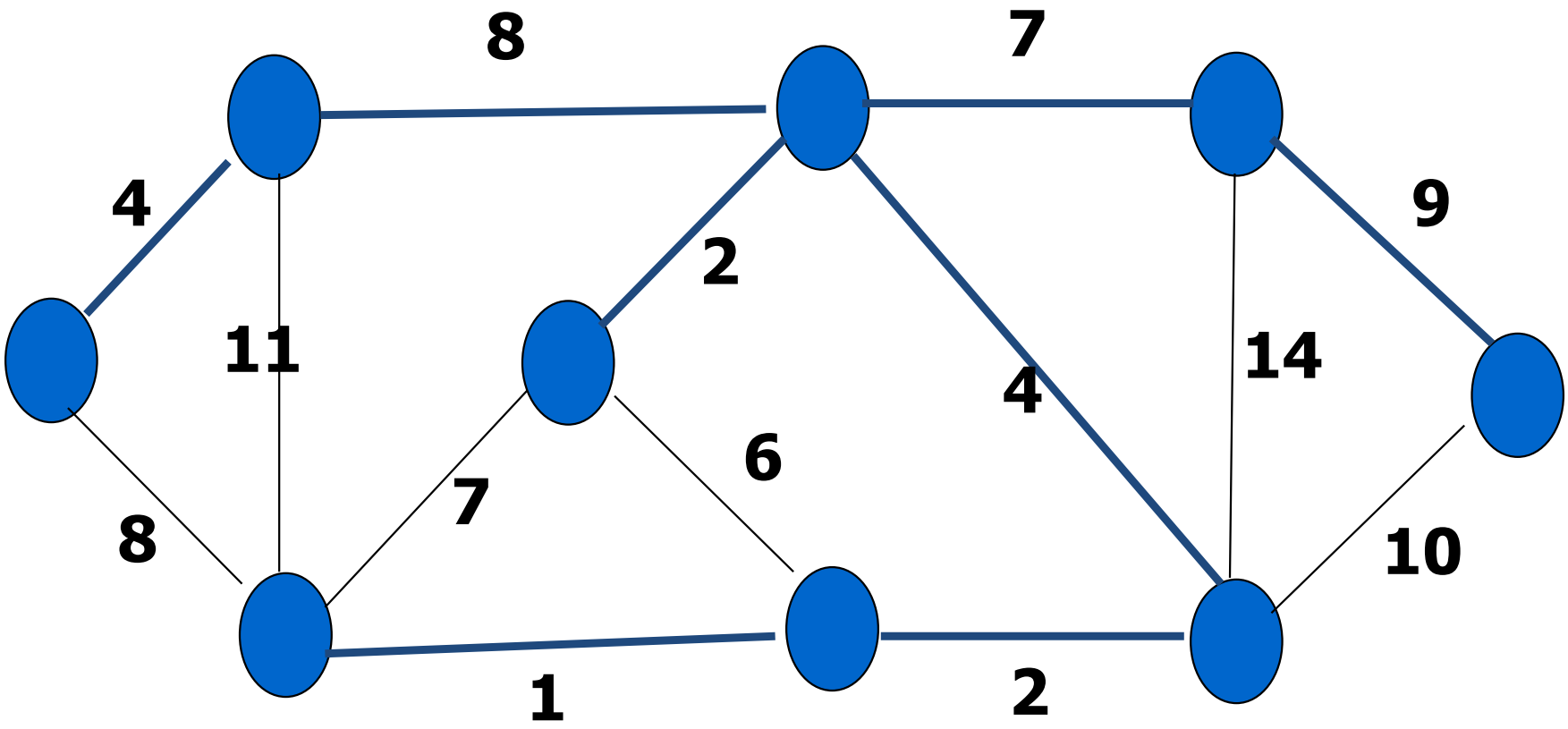
Prim’s Algorithm Example(7)



Prim’s Algorithm Example(8)



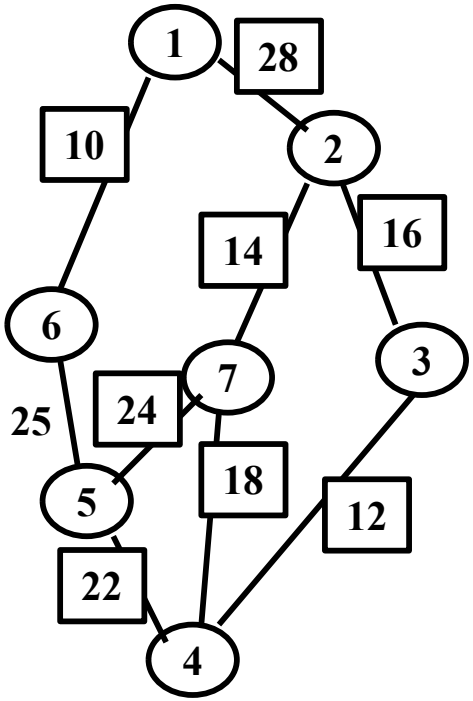
Prim’s Algorithm Example(9)



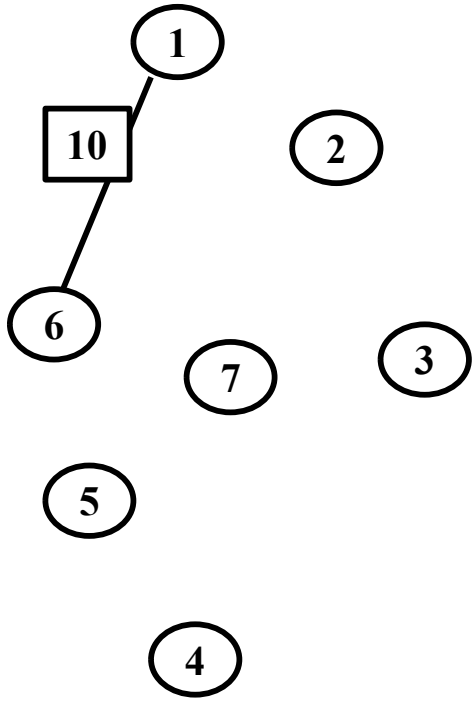
Prim's Algorithm (complexity)

- The time required by algorithm Prim is $O(n^2)$, where n is the number of vertex in the graph G. To see this, note that line 9 takes $O(|E|)$ time and line 10 takes $\Theta(1)$ time. The **for** loop of line 12 takes $\Theta(n)$ time. Lines 18 and 19 and **for** loop of line 23 require $O(n)$ time. So, each iteration of the for loop of line 16 takes $O(n)$ time. The total time for the **for** loop of line 16 is therefore $O(n^2)$. Hence, Prim runs in $O(n^2)$ time.

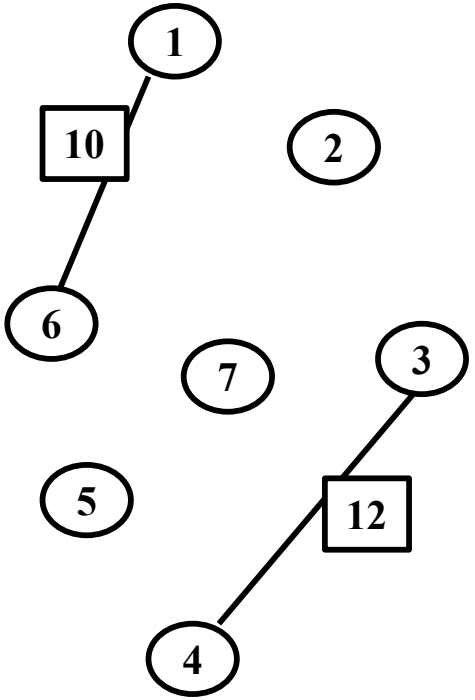
Kruskal's Algorithm



a

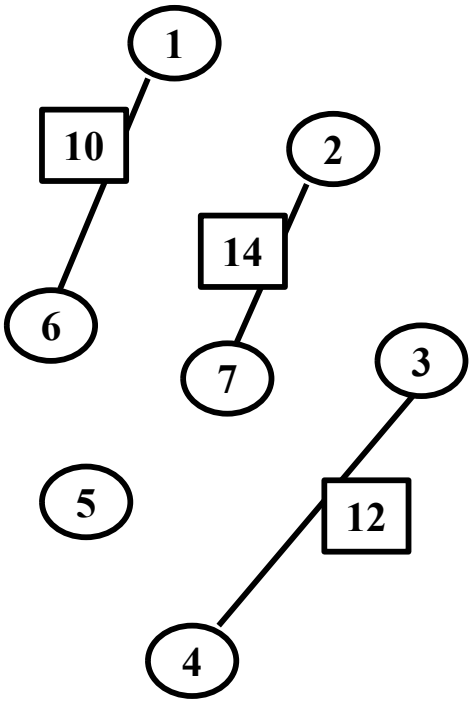


b

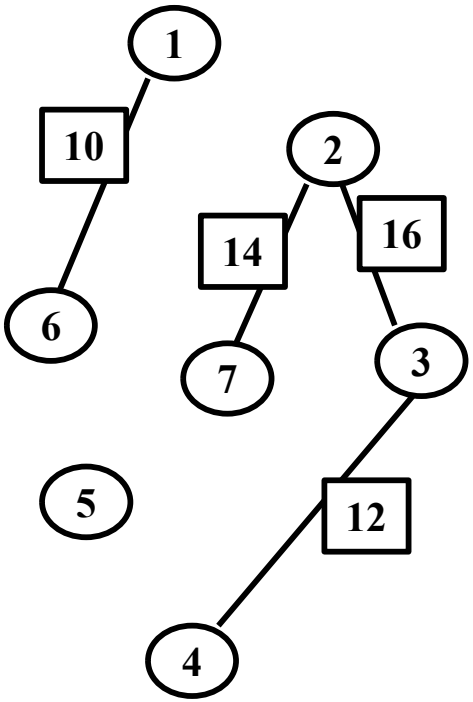


c

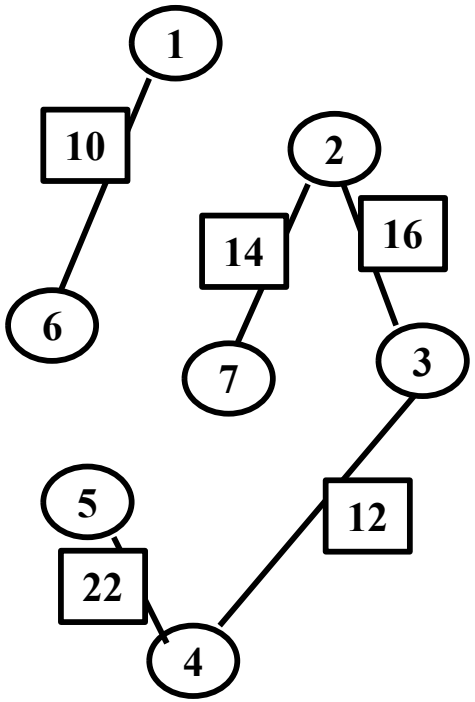
Kruskal's Algorithm (cont.)



d



e

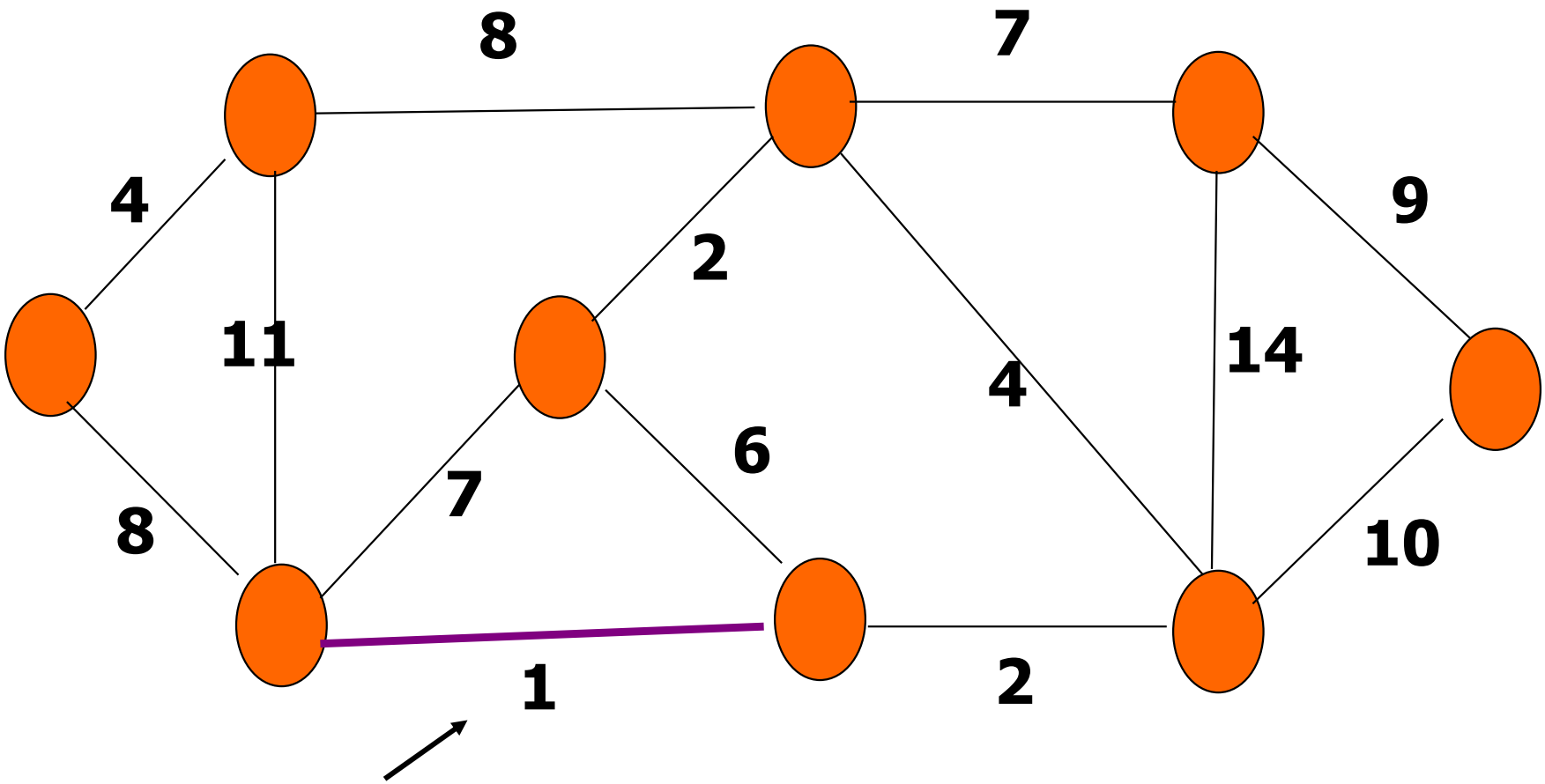


f

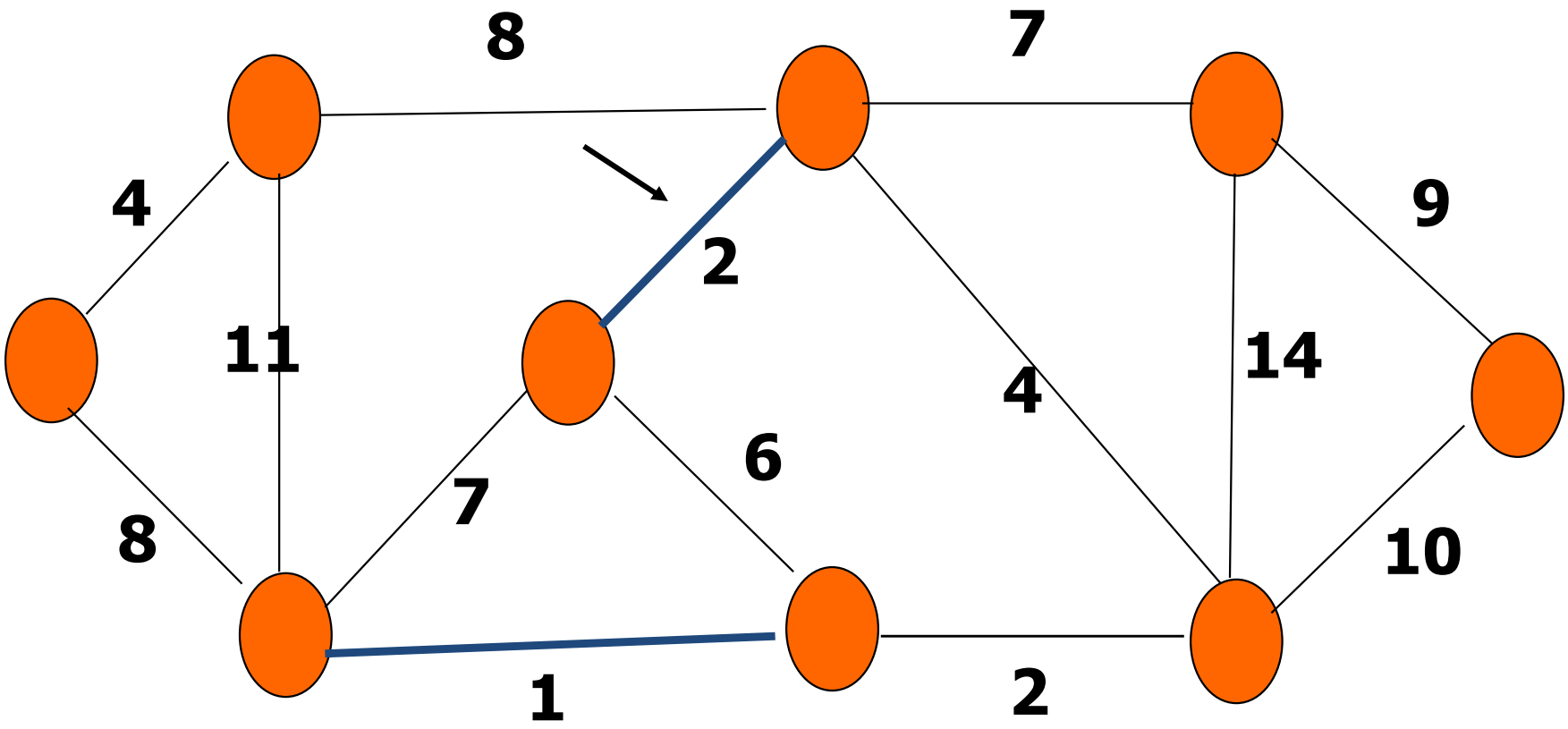
Kruskal’s Algorithm

- This is a greedy algorithm. A greedy algorithm chooses some local optimum (ie. picking an edge with the least weight in a MST).
- Kruskal's algorithm works as follows:
  - Take a graph with 'n' vertices, keep adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added.
  - NOTE: Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost

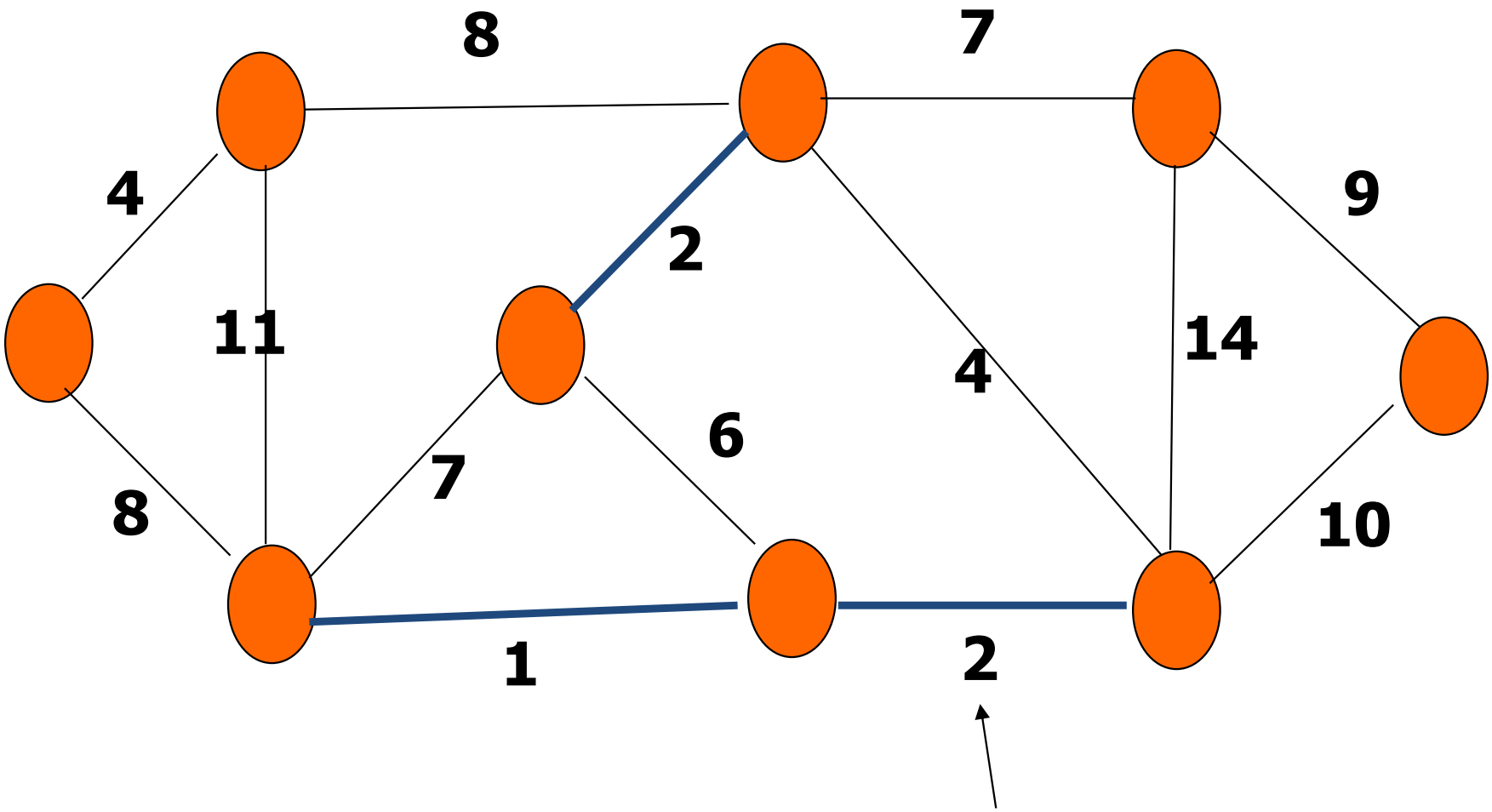
Kruskal's Algorithm Example(1)



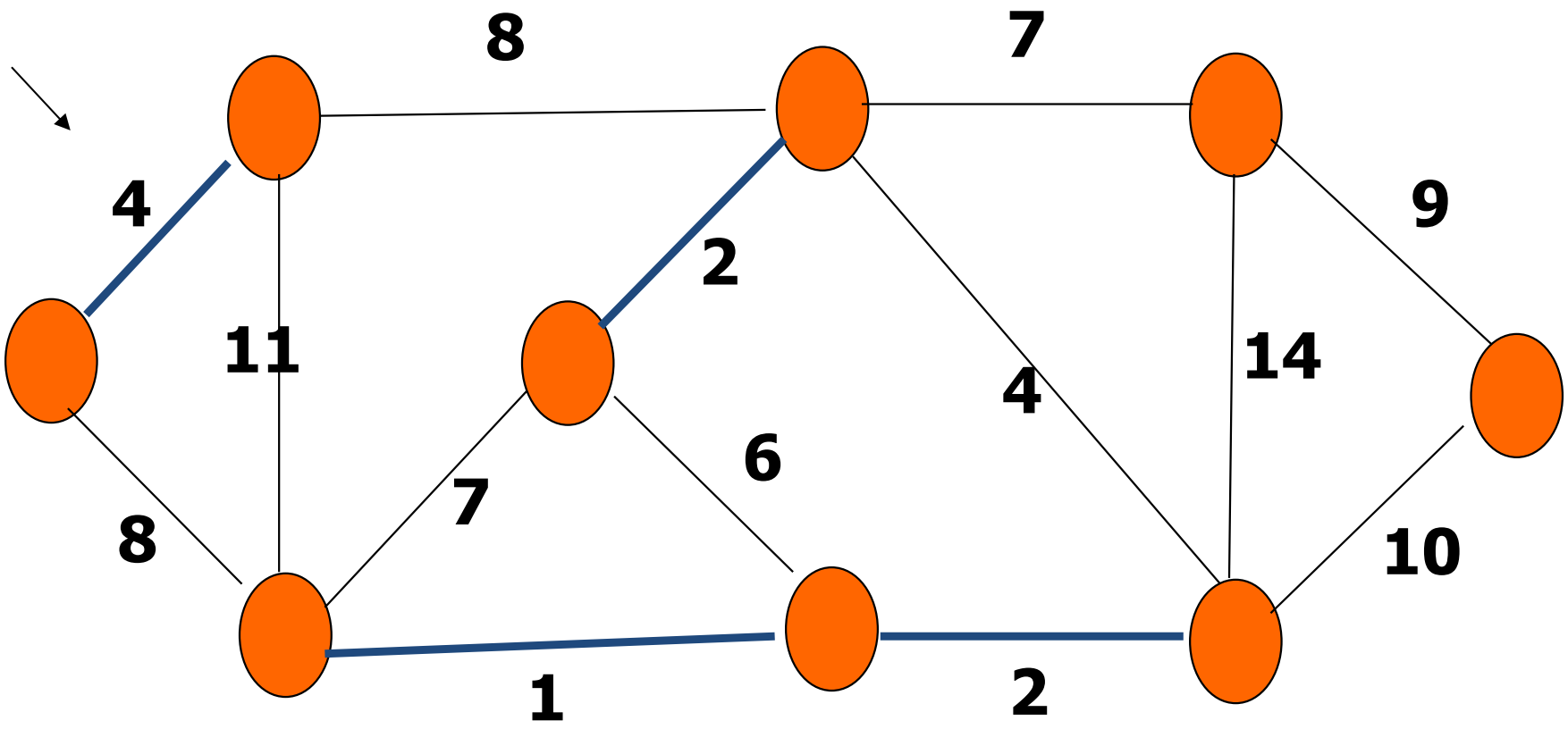
Kruskal's Algorithm Example(2)



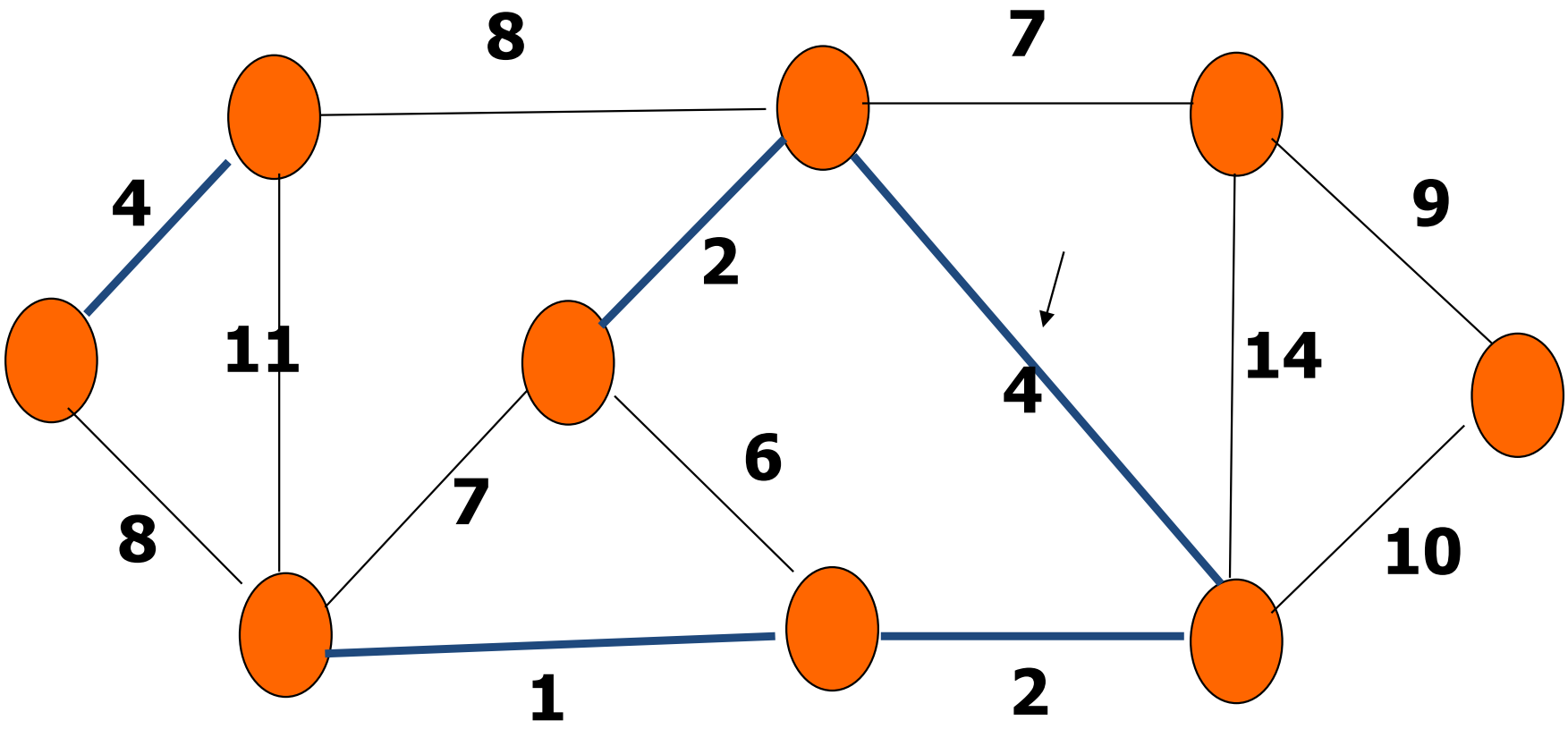
Kruskal's Algorithm Example(3)



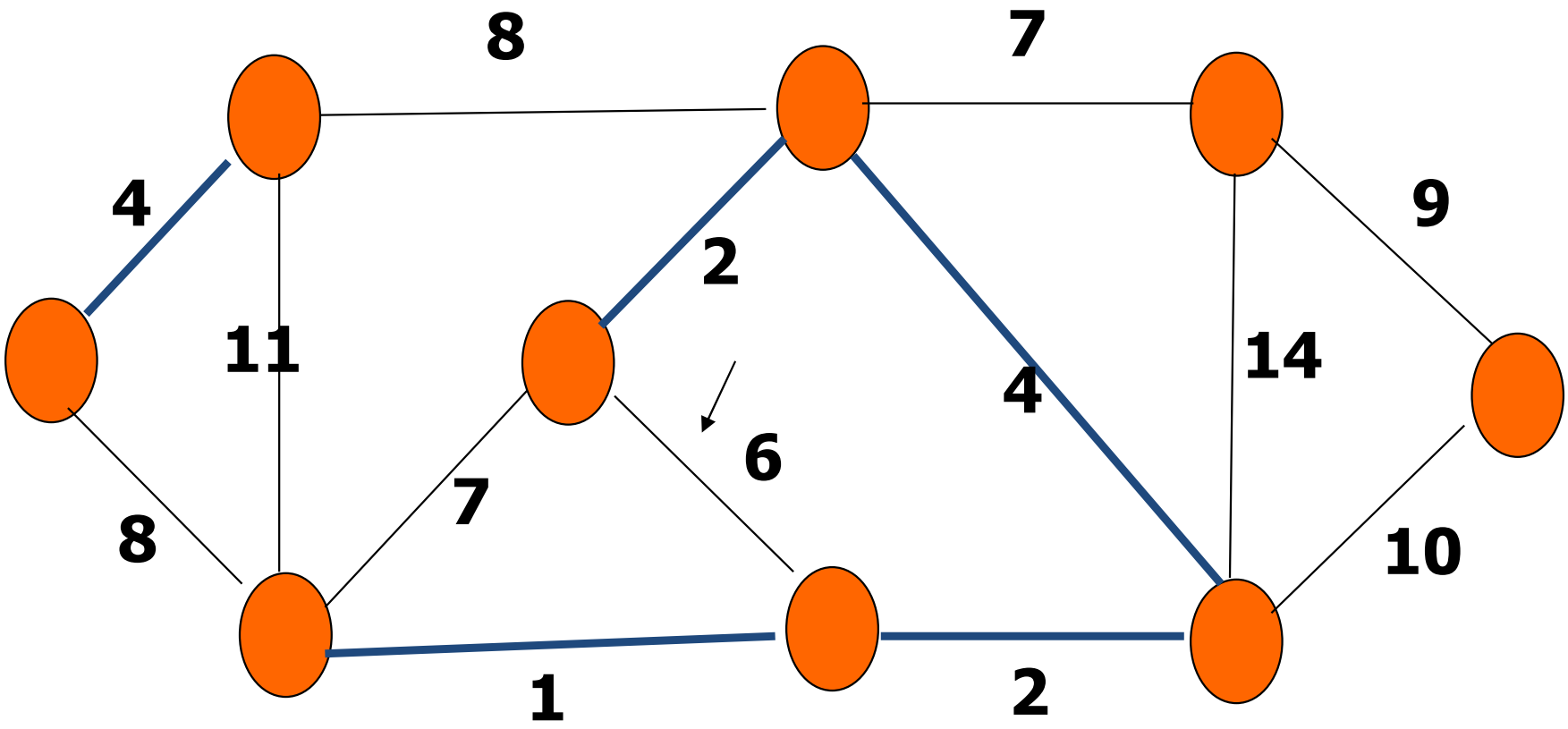
Kruskal's Algorithm Example(4)



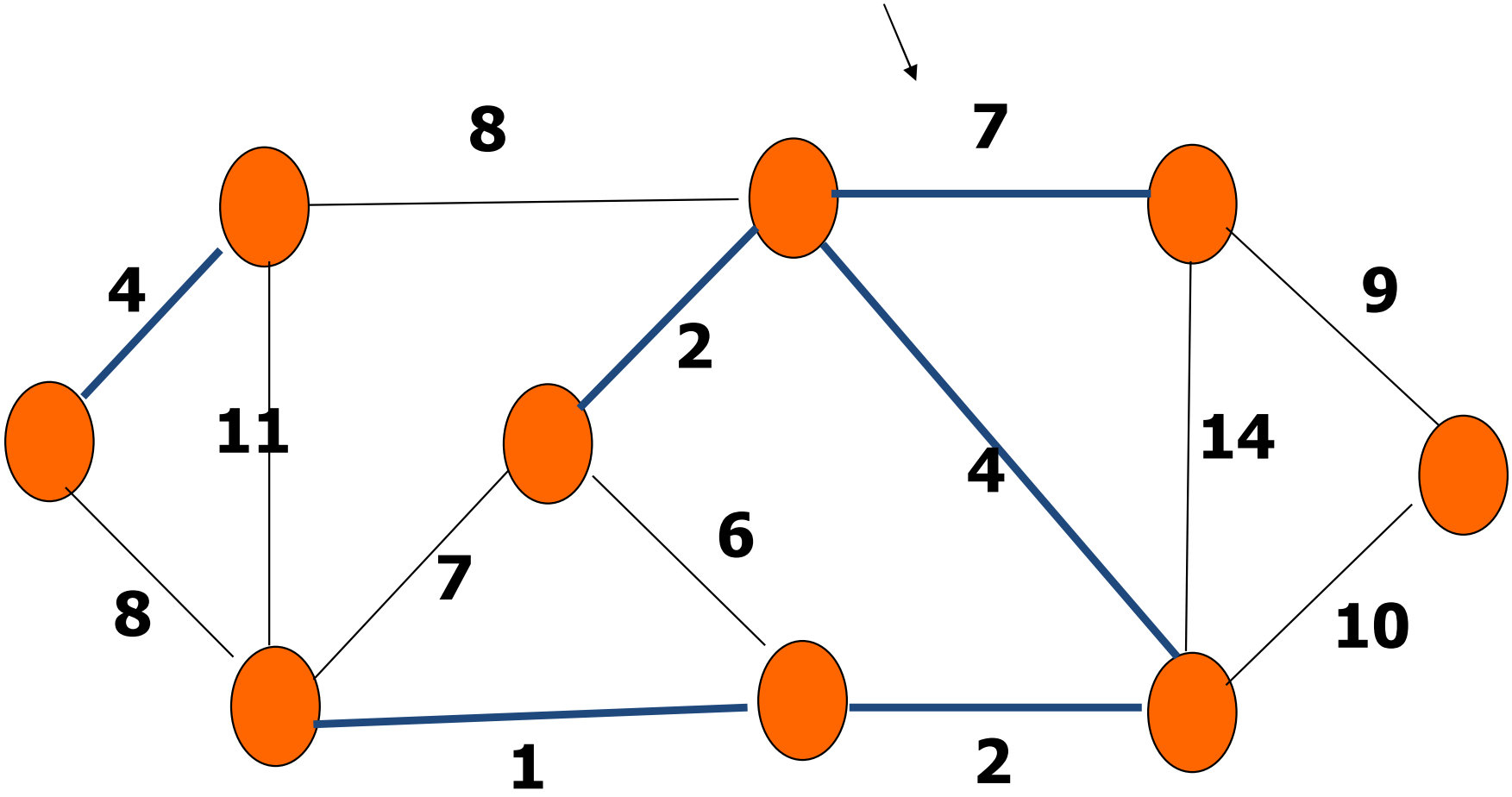
Kruskal's Algorithm Example(5)



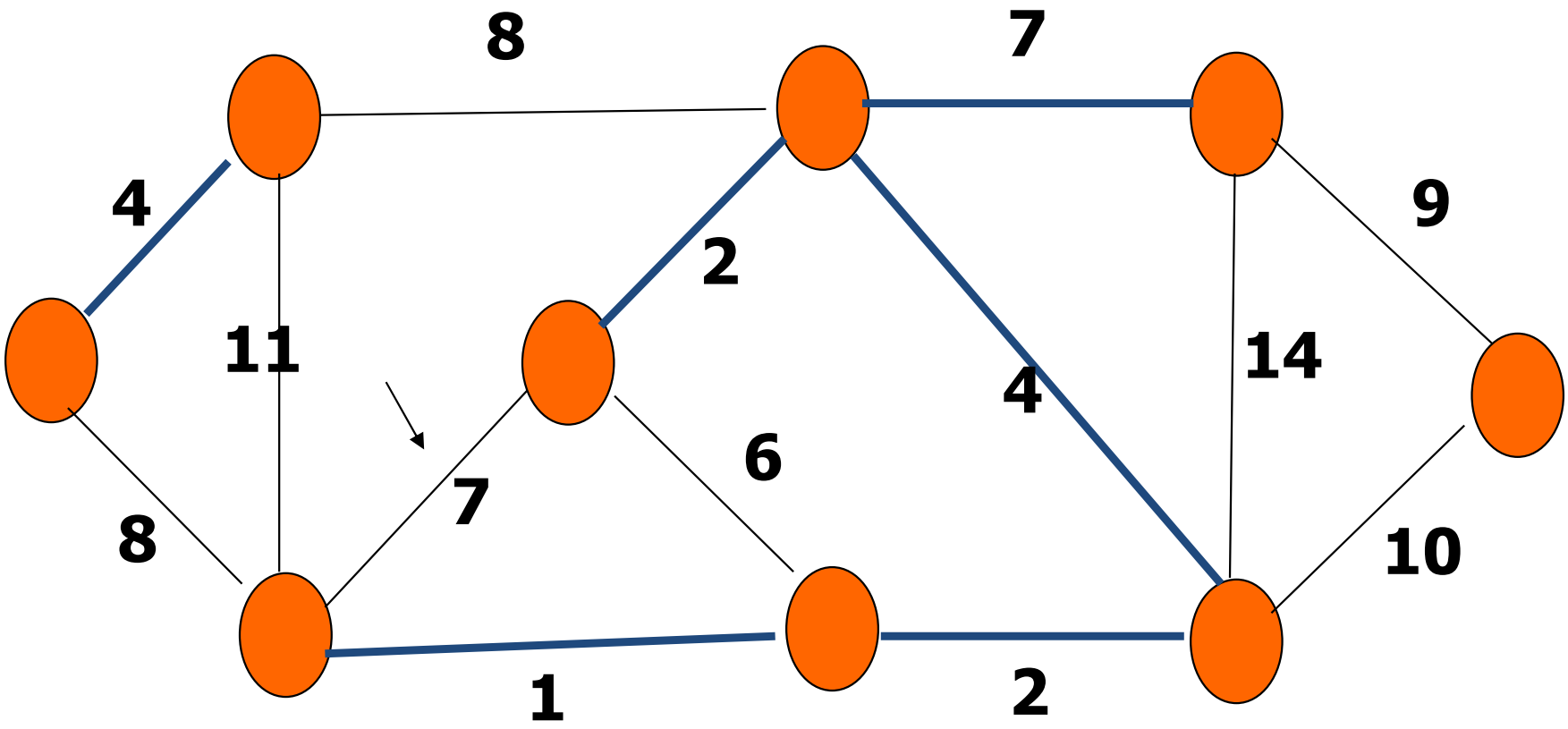
Kruskal's Algorithm Example(6)



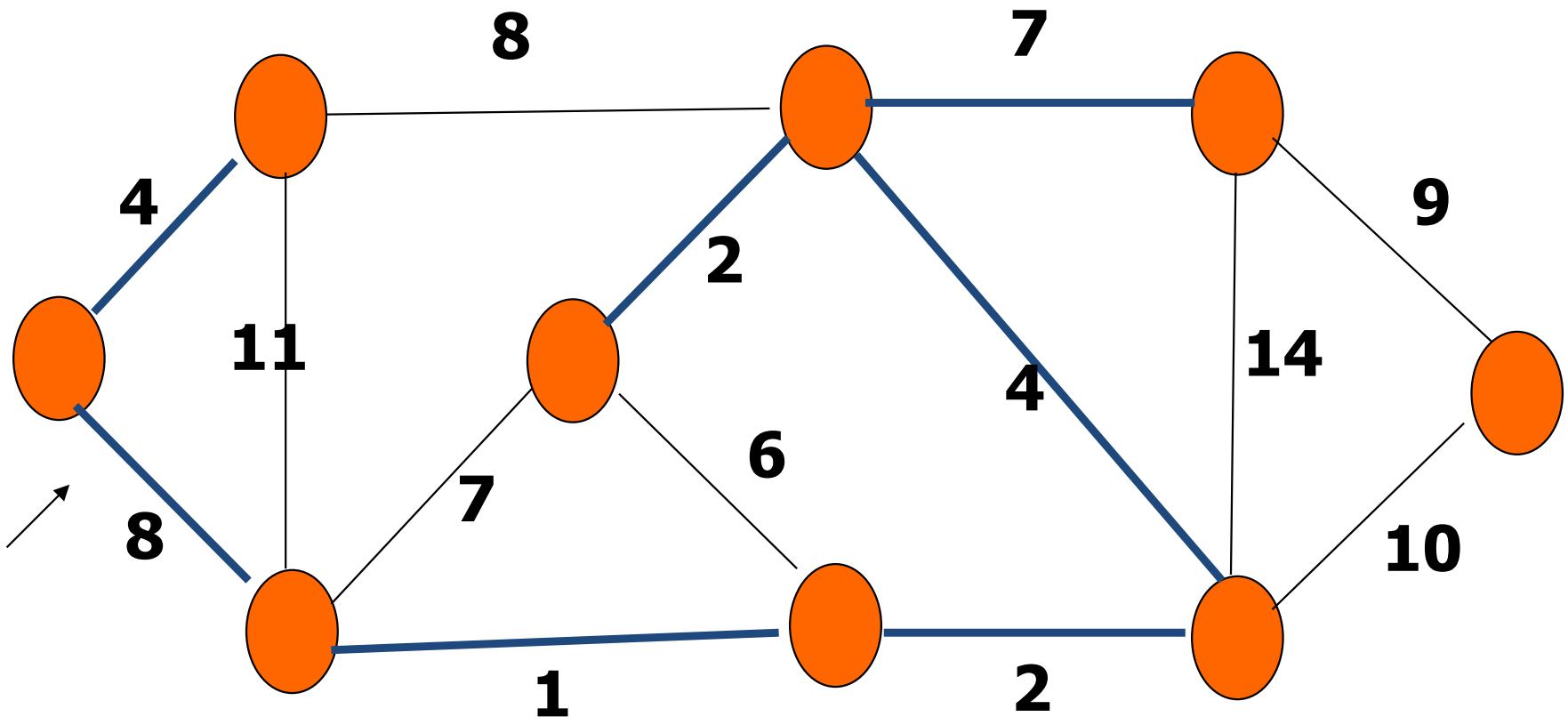
Kruskal's Algorithm Example(7)



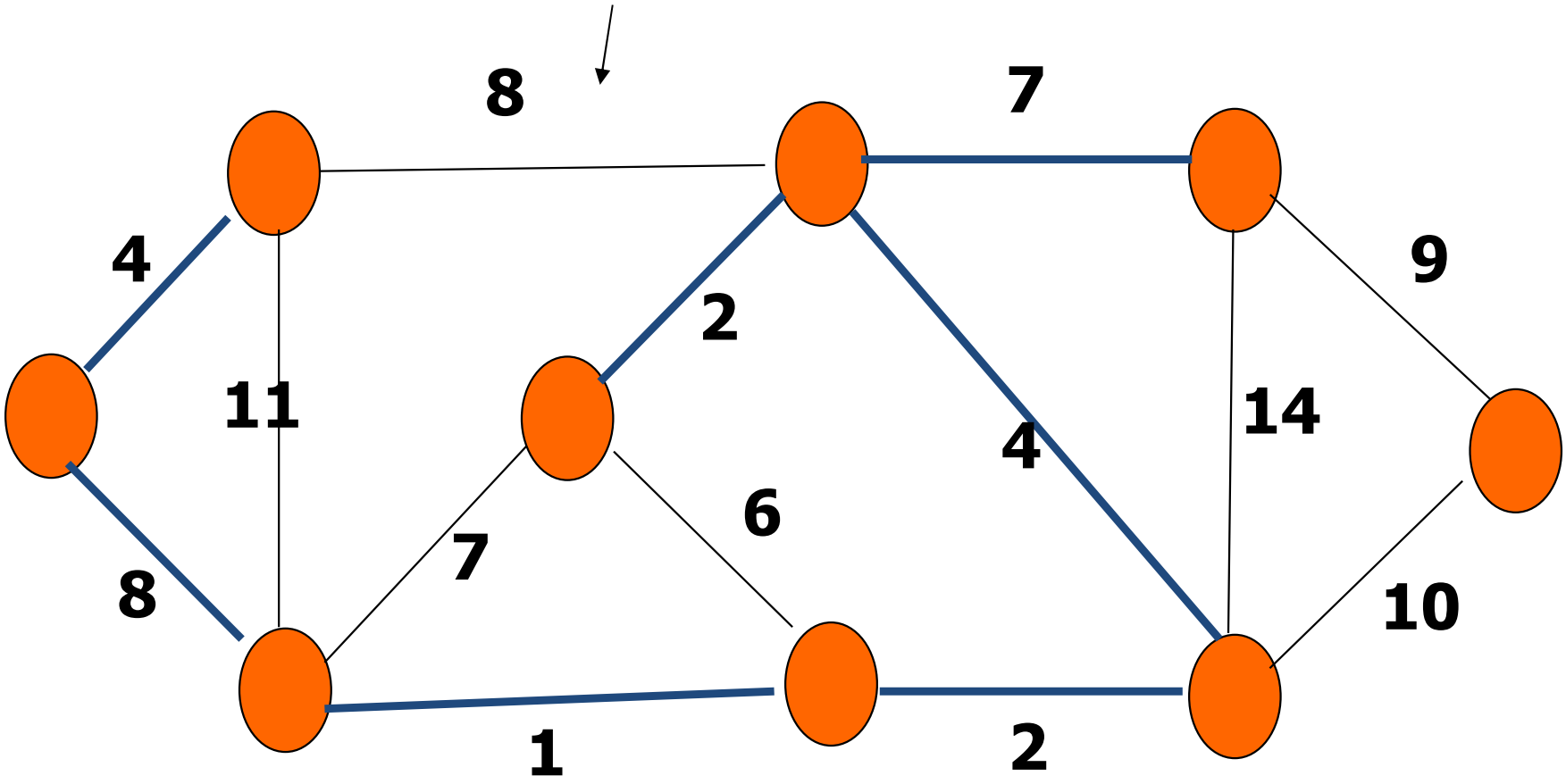
Kruskal's Algorithm Example(8)



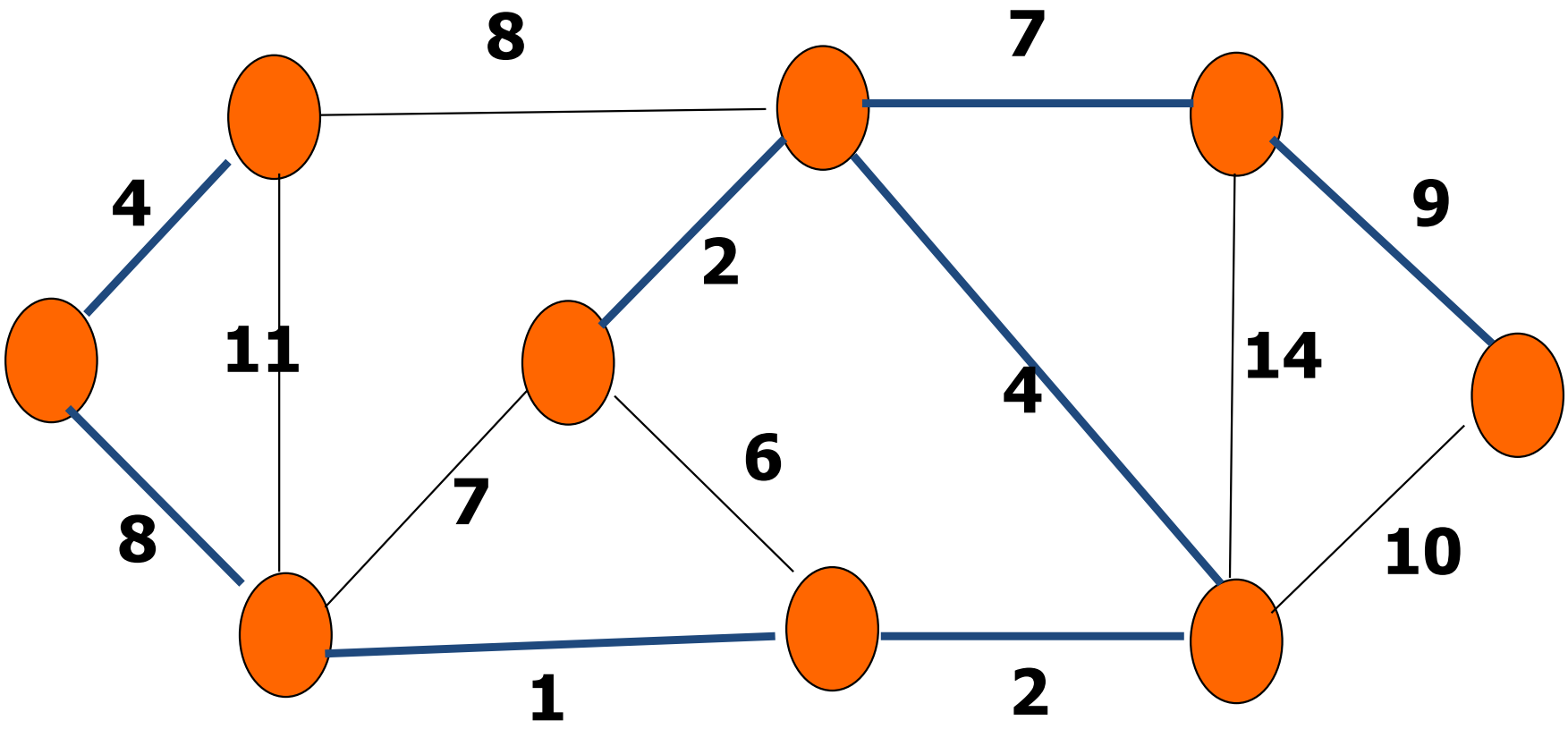
Kruskal's Algorithm Example(9)



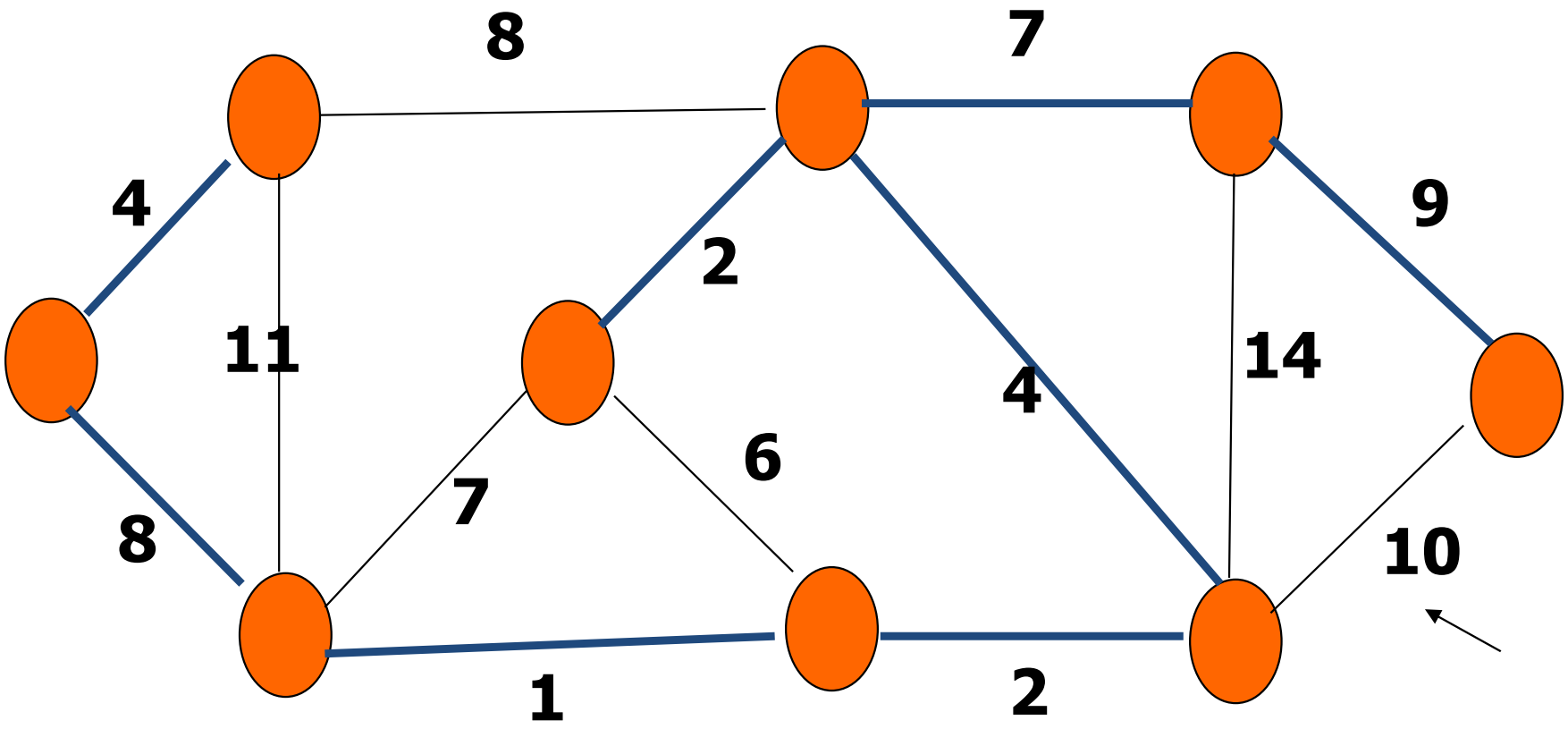
Kruskal's Algorithm Example(10)



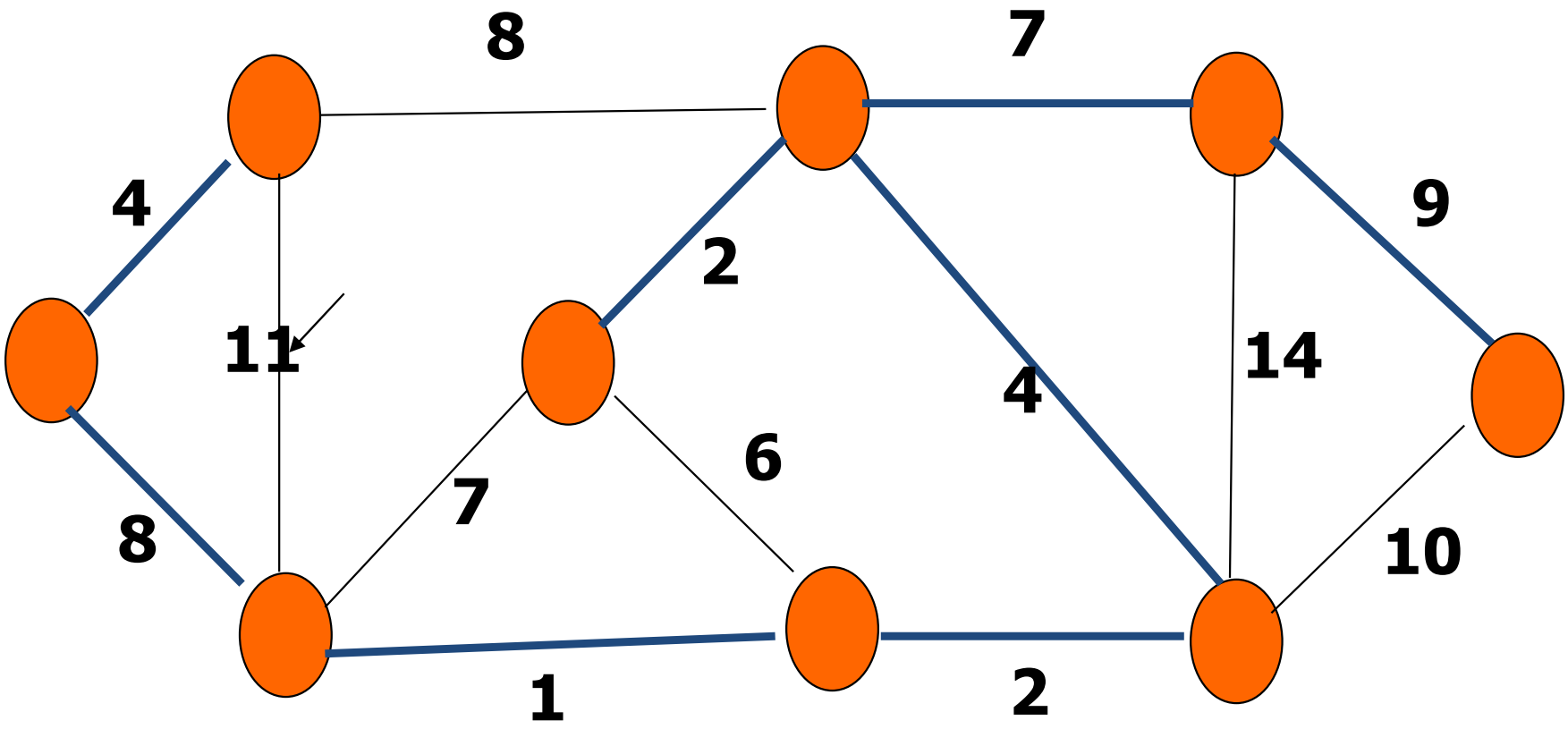
Kruskal's Algorithm Example(11)



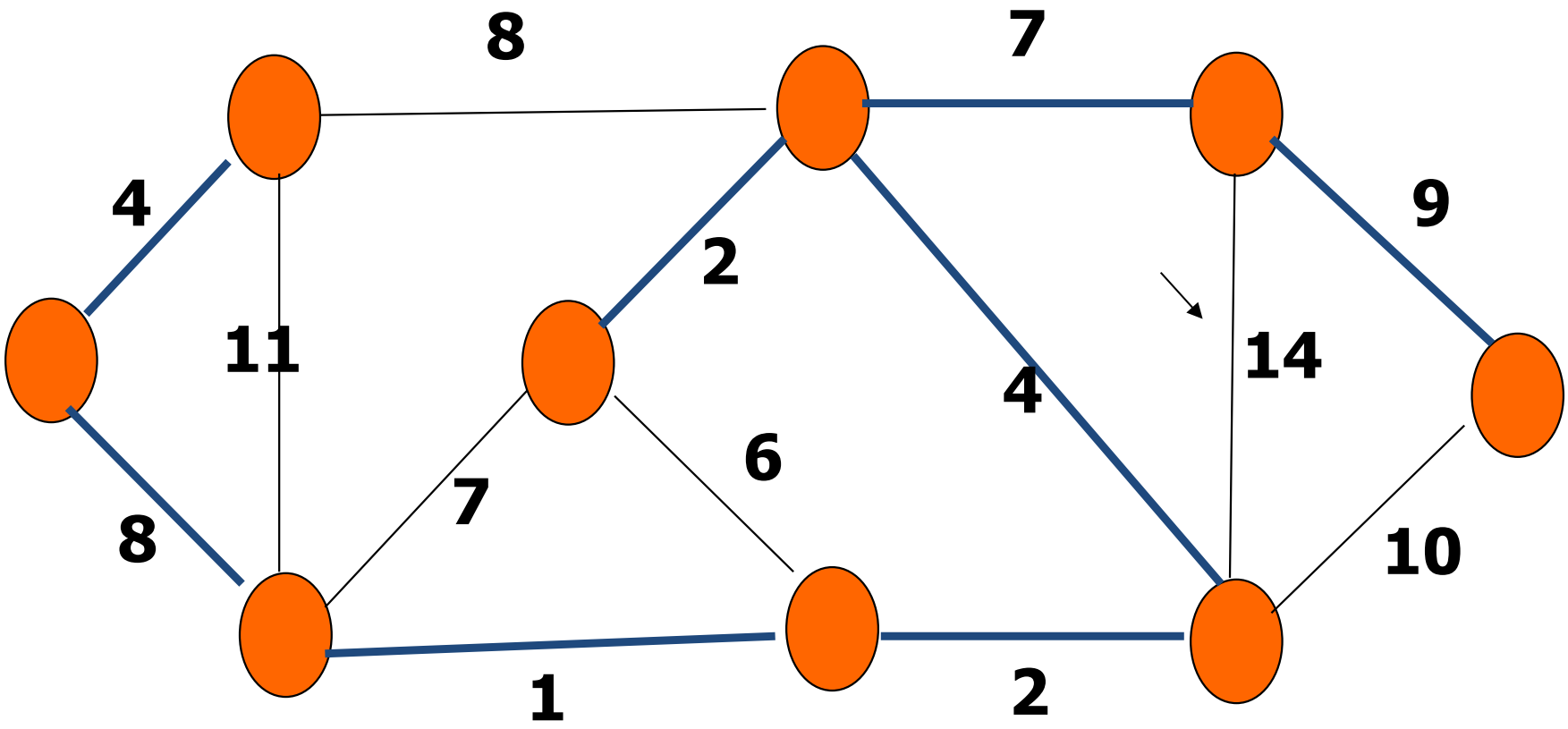
Kruskal's Algorithm Example(12)



Kruskal's Algorithm Example(13)



Kruskal's Algorithm Example(14)



Kruskal's Algorithm Pseudocode

```
1.  Algorithm Kruskal(E, cost, n, t)
2.  // E is the set of edges in G. G has n vertices. cost[u,v] is the cost of
3.  // edge (u,v). t is the set of edges in the minimum-cost spanning tree.
4.  // The final cost is returned.
5.  {
6.      Construct a heap out of the edges using Heapify;
7.     for i:=1 to n do  parent [i] := -1;
8.       // Each vertex is in a different set.
9.     i:= 0; mincost := 0.0;
10.     while((i < n-1) and (heap not empty)) do
11.     {
12.       Delete a minimum cost edge (u,v) from the heap
13.       and reheapify.
14.       j := Find(u); k:=Find(v);
```

Kruskal's Algorithm Pseudocode (cont.)

```
15.          if ( j ≠ k) then
16.          {
17.             i:=i+1;
18.             t[i,1] :=u; t[i,2] := v;
19.             mincost := mincost+cost[u, v];
20.             Union (j,k);
21.          }
22.       }
23.    if (i  ≠ n-1) then  write ("No spanning tree");
24.    else return mincost;
25.  }
```

Complexity is $O(|E|\log|E|)$

Reference: Computer algorithm-SAHNI, page 224