Graphs

Contents

- Definition of Graphs and Related Concepts
- Terminology
- Representation of Graphs
- Graphs as ADTs
- Graph Traversal
- Applications of Graphs

Terminology

- Definition:
- A set of points that are joined by lines
- Graphs also represent the relationships among data items
- G = { V , E }; that is, a graph is a set of vertices and edges
- V(G): a finite, nonempty set of vertices
- E(G): a set of edges (pairs of vertices)
- A subgraph consists of a subset of a graph’s vertices and a subset of its edges

Terminology

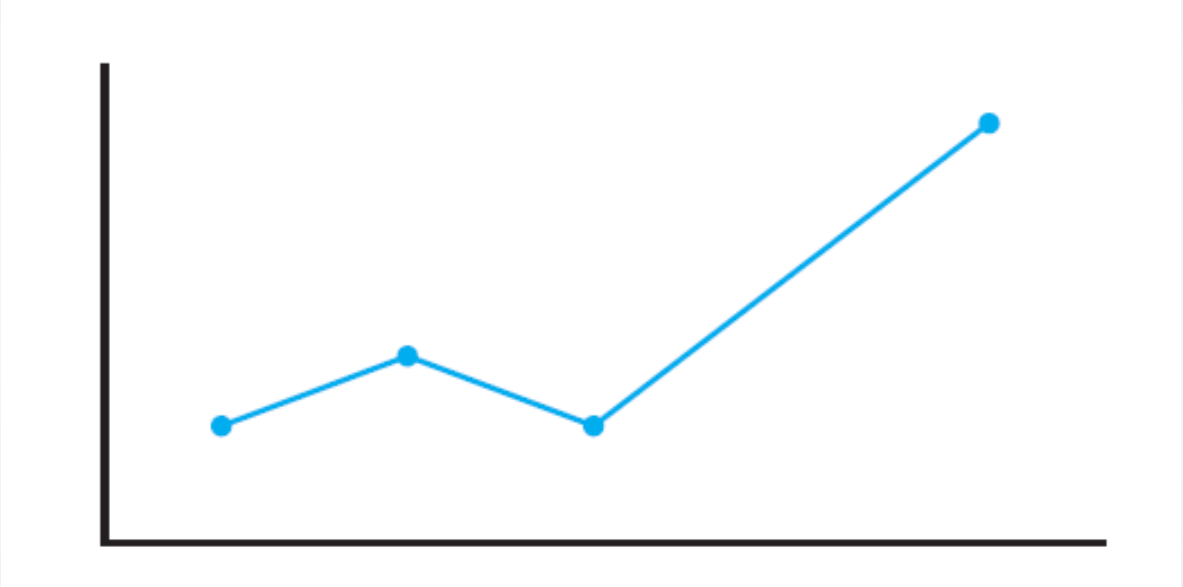


FIGURE 1 An ordinary line graph

Terminology

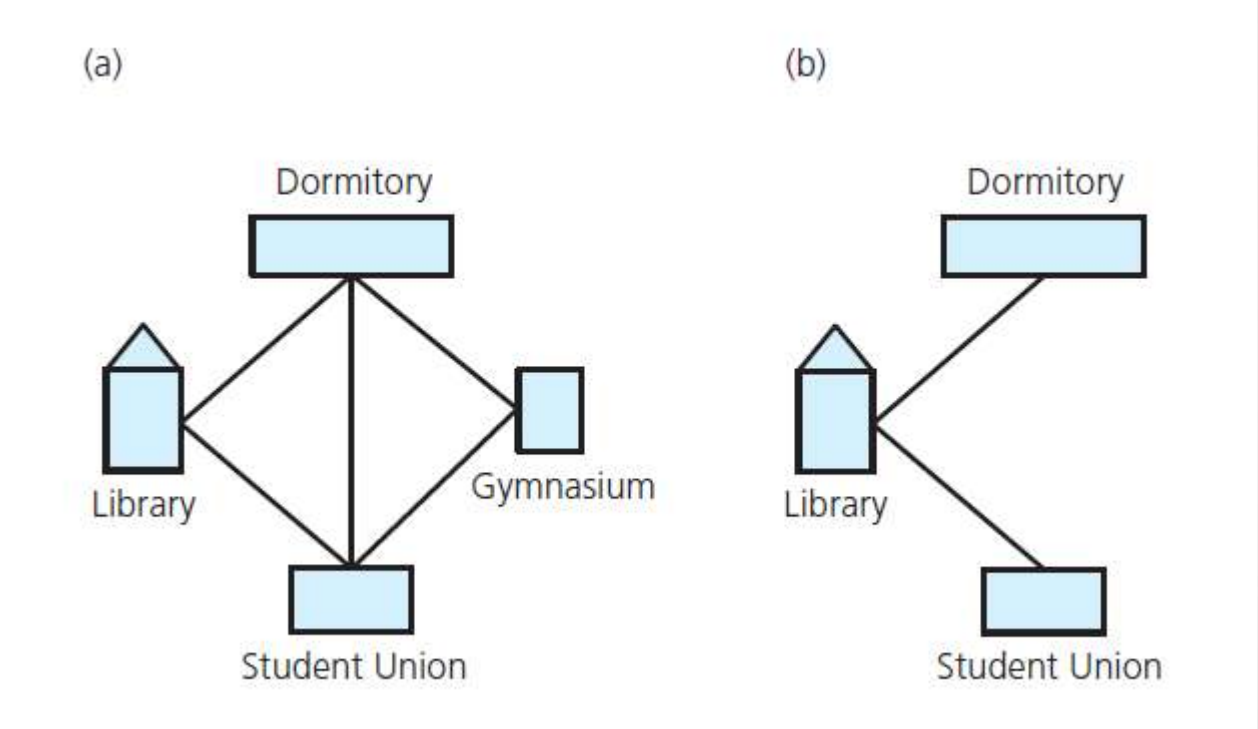


FIGURE 2 (a) A campus map as a graph;
(b) a subgraph

Terminology

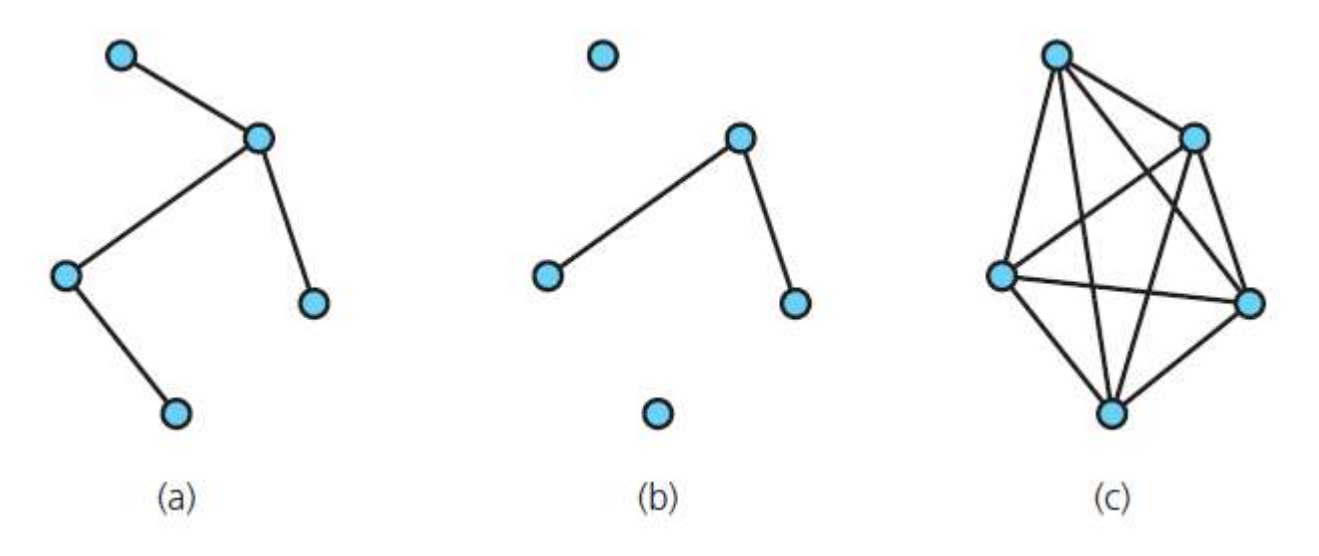


FIGURE 20-3 Graphs that are (a) connected; (b) disconnected; and (c) complete

Terminology

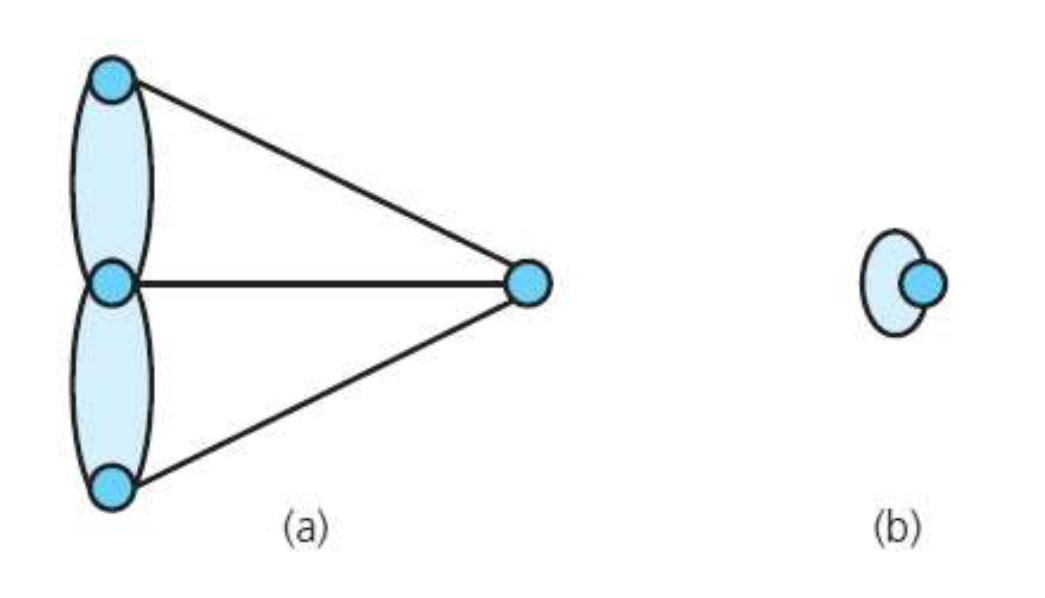


FIGURE 4 (a) A multigraph is not a graph;
(b) a self edge is not allowed in a graph

Terminology

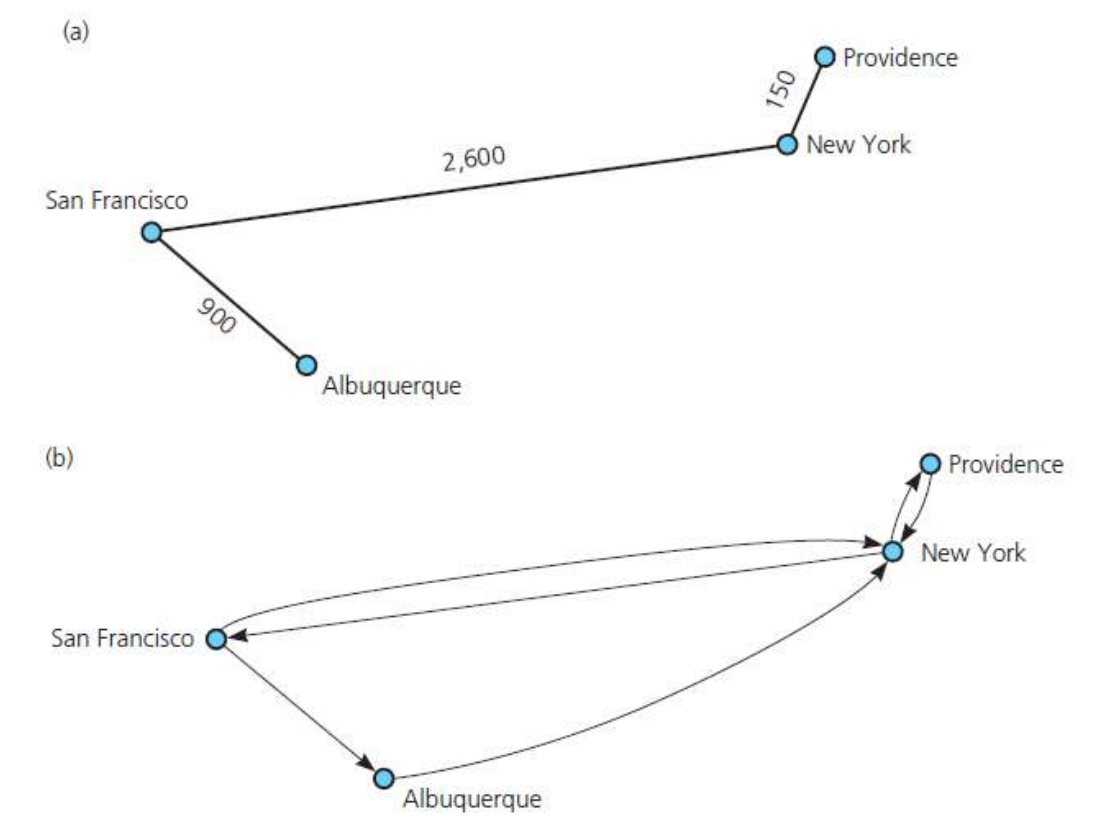


FIGURE 5 (a) A weighted graph;
(b) a directed graph

Terminology

- path: passes through vertex only once and there exist an edge from one vertex to the next vertex.
- Cycle: a path that begins and ends at same vertex
- Simple cycle: cycle that does not pass through other vertices more than once
- Connected graph: each pair of distinct vertices has a path between them

Terminology

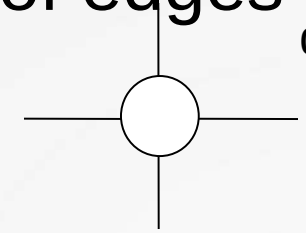
- Complete graph: each pair of distinct vertices has an edge between them
- Graph cannot have duplicate edges between vertices
- Multigraph: does allow multiple edges
- Weighted graph: When labels represent numeric values, graph is called a weighted graph

Terminology

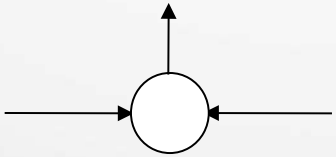
- Undirected graphs: edges do not indicate a direction
- Directed graph, or digraph: each edge has a direction

Concepts: Degree

- Undirected graph: The *degree* of a vertex is the number of edges touching it.



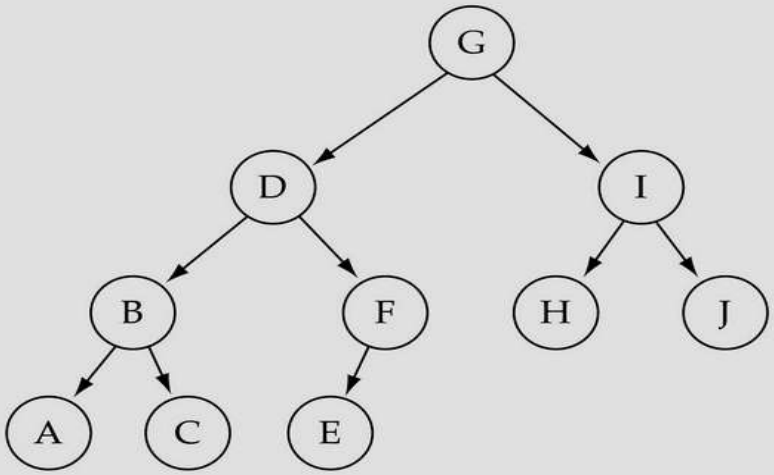
- For a directed graph, the *in-degree* is the number of edges entering the vertex, and the *out-degree* is the number leaving it. The *degree* is the *in-degree* + the *out-degree*.



Trees vs graphs

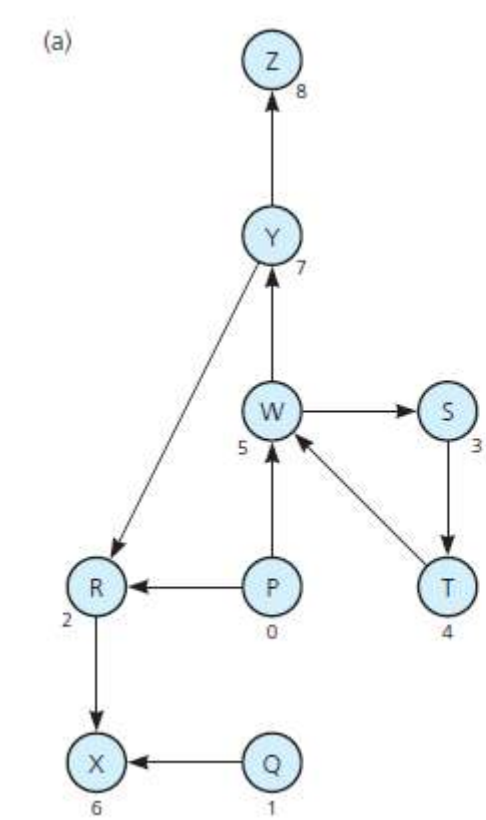
- Trees are special cases of graphs!!
- No unique node call root
- A cycle can be formed
- Applications

(c) Graph3 is a directed graph.



V(Graph3) = { A, B, C, D, E, F, G, H, I, J }
E(Graph3) = { (G, D), (G, J), (D, B), (D, F) (I, H), (I, J), (B, A), (B, C), (F, E) }

Representation of Graphs



(b)

		0 P	1 Q	2 R	3 S	4 T	5 W	6 X	7 Y	8 Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

FIGURE 6 (a) A directed graph and
(b) its adjacency matrix

Representation of Graphs

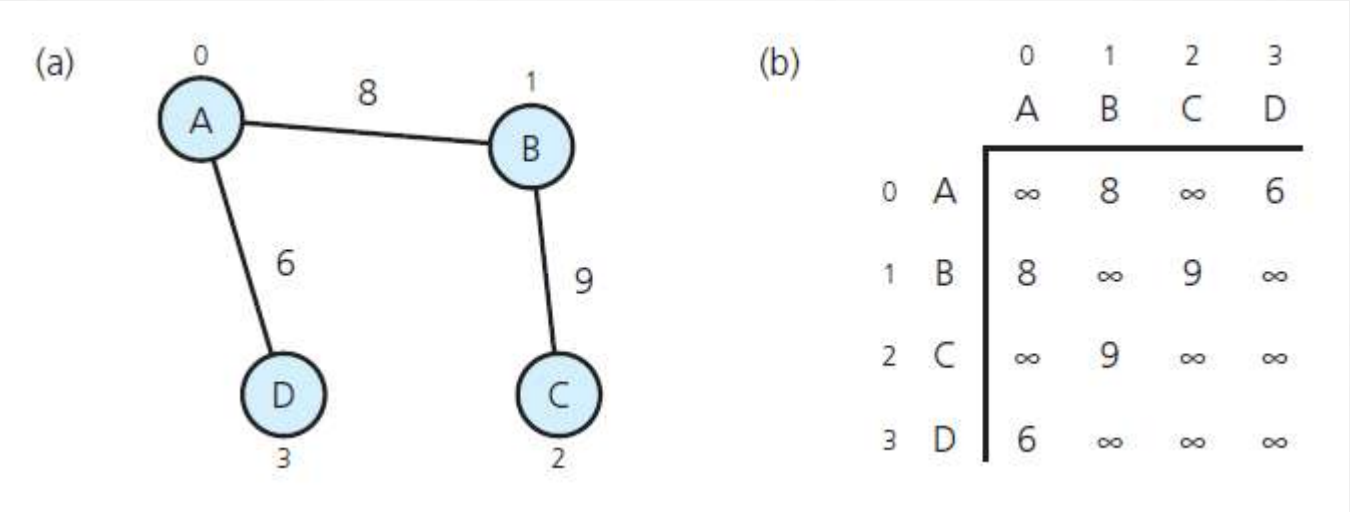


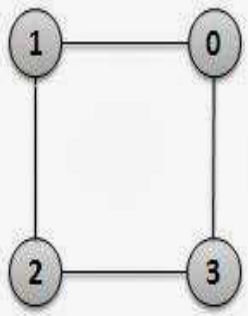
FIGURE 7 (a) A weighted undirected graph and (b) its adjacency matrix

Adjacency Matrix: Pros and Cons

- *advantages*
  - fast to tell whether edge exists between any two vertices *i* and *j* (and to get its weight)
- *disadvantages*
  - consumes a lot of memory on sparse graphs (ones with few edges)
  - redundant information for undirected graphs

Steps: Graph using adjacency matrix

- Create and display a graph using adjacency matrix



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

Adjacency Matrix Representation of Undirected Graph

1. Initialize g[i][j]=0;
2. Enter the no. of nodes required
3. for(i=0;i<n;i++)
{

for(j=0;j<n;j++)
{
cin>>graph[i][j];
}
}
4. for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
cout<<graph[i][j];
}
}

Representation of Graphs

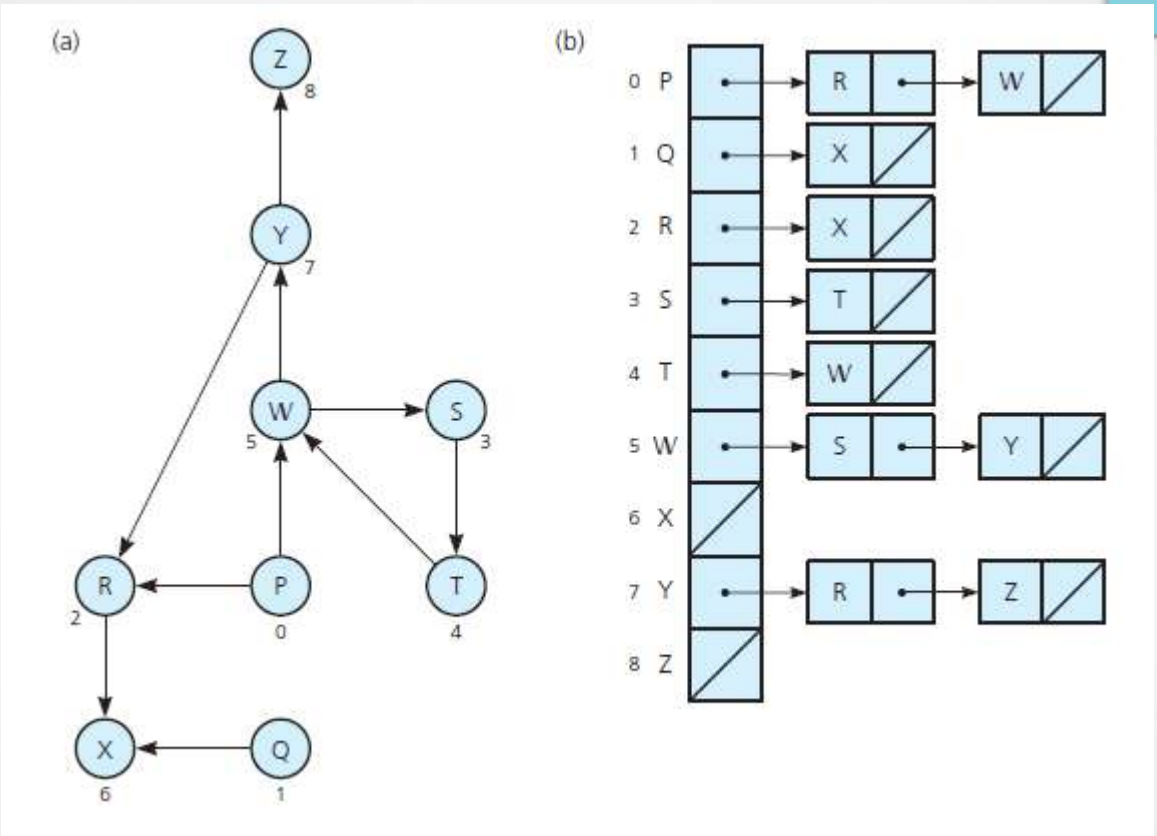


FIGURE 8 (a) A directed graph and
(b) its adjacency list

Representation of Graphs

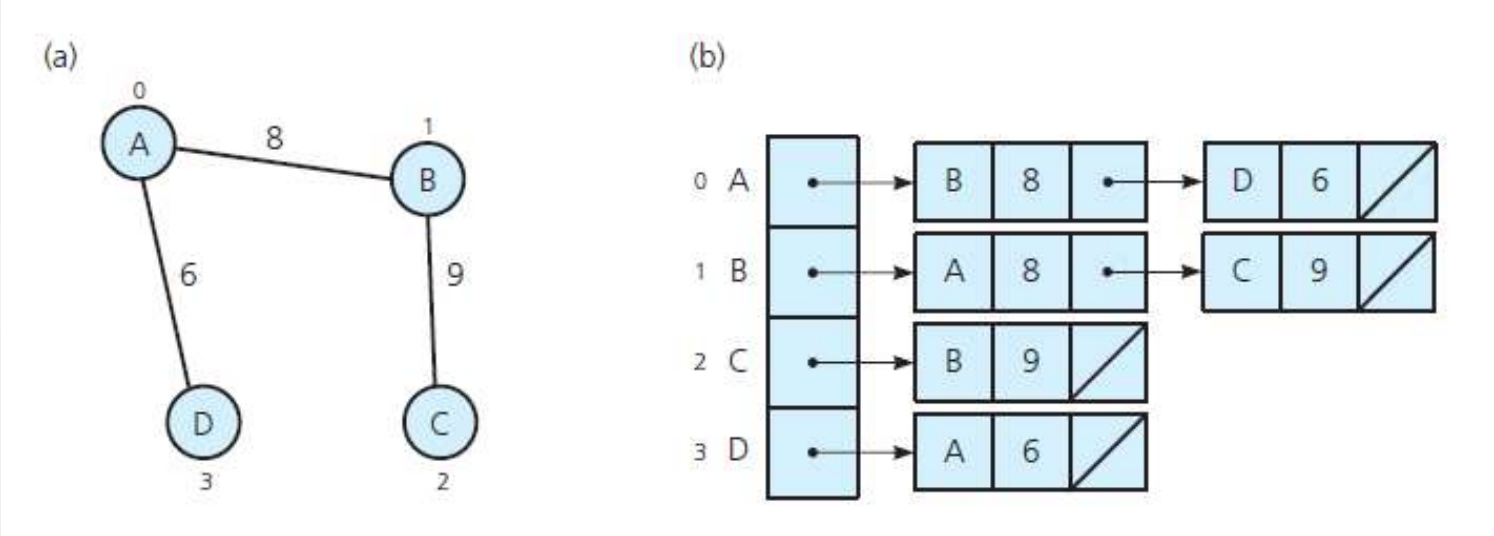


FIGURE 9 (a) A weighted undirected graph and (b) its adjacency list

Algorithm

```
class node
{
   int/char vertex;
   node *next;
} ;
node *G[10];
```

```
void read_graph()
{
   int i,vi,vj,no_of_edges;
   cout<<"\nEnter no of vertices : ";
   cin>>n;
   // initialise G[] with a null
   for(i=0;i<n;i++)
   {
         G[i]=NULL;
         // read edges and insert them in G[]
         cout<<"\nEnter no of edges : ";
         cin>>no_of_edges;
         for(i=0;i<no_of_edges;i++)
         {
              cout<<"\nEnter an edge (u,v) :";
              cin>>vi>>vj;
              insert(vi,vj);
         }
   }
}
void insert(int vi,int vj)
{
   node *p,*q;
   // acquire memory for the new node
   q=new node;
   q->vertex=vj;
   q->next=NULL;
   //insert the node in the linked list number vi
   if(G[vi]==NULL)
         G[vi]=q;
   else
   {
         // go to end of linked list
         p=G[vi];
         while(p->next!=NULL)
              p=p->next;
         p->next=q;
```

Adjacency List: Pros and Cons

- *advantages*:
  - new nodes can be added easily
  - new nodes can be connected with existing nodes easily
- *disadvantages*:
  - determining whether an edge exists between two nodes: O(average degree)

Adjacency matrix vs. adjacency list representation

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

Graph operations and Storage Structure

1.Create(): using adjacency matrix or adjacency list.

2.Display(): using BFS and DFS.

Storage Structure :

Graph creation using adjacency matix or adjacency list.

Graphs as ADTs

ADT graph operations

- Test whether graph is empty.
- Get number of vertices in a graph.
- Get number of edges in a graph.
- See whether edge exists between two given vertices.
- Indegree()
- Outdegree()
- Display()

Graph Traversals

- Visits all of the vertices that it can reach
  - Happens if and only if graph is connected
- Connected component is subset of vertices visited during traversal that begins at given vertex

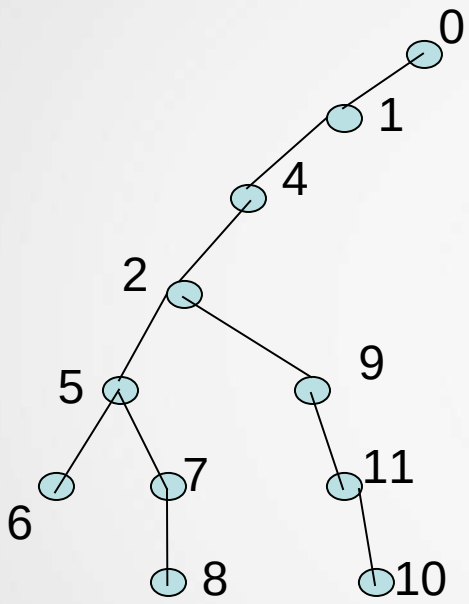
Graph Traversal (Contd.)

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

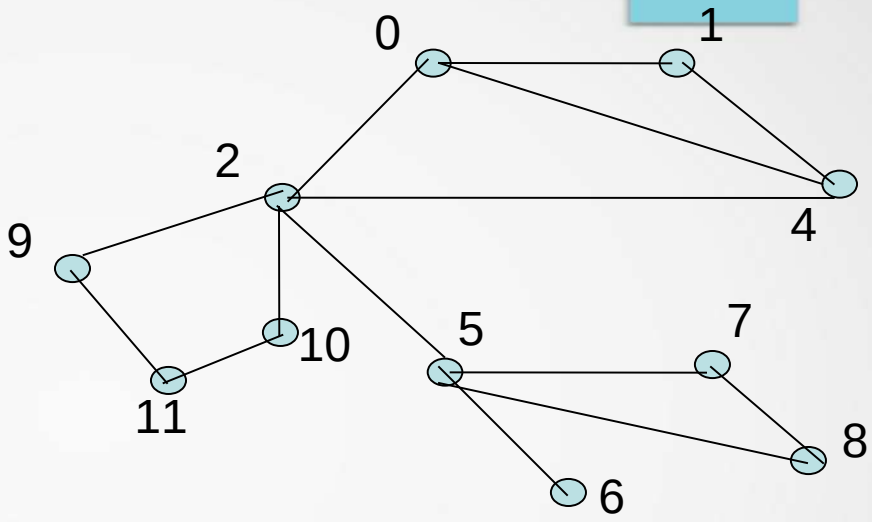
Depth-First Search

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

Illustration of DFS



DFS Tree



Graph G

Implementation of DFS

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

DFS (Pseudo Code)

```
 procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5            v = S.pop()
6            if v is not labeled as discovered:
7                label v as discovered
8                for all edges from v to w in
   G.adjacentEdges(v) do
9                    S.push(w)
```

Depth-First Search

- Goes as far as possible from a vertex before backing up
- Recursive algorithm

```
// Traverses a graph beginning at vertex v by using a
// depth-first search: Recursive version.
dfs(v: Vertex)

   Mark v as visited
   for (each unvisited vertex u adjacent to v)
      dfs(u)
```

Depth-First Search

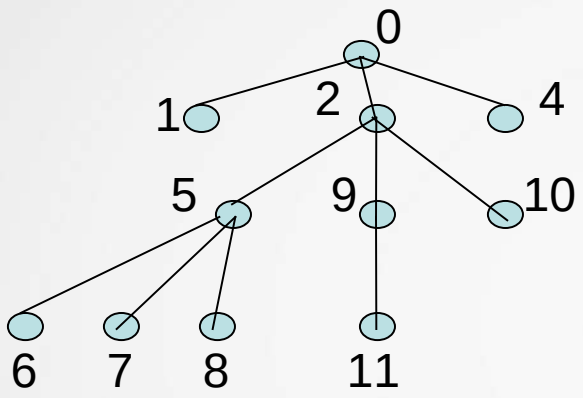
- Iterative algorithm, using a stack

```
// Traverses a graph beginning at vertex v by using a
// depth-first search: Iterative version.
dfs(v: Vertex)

   s= a new empty stack

   // Push v onto the stack and mark it
   s.push(v)
   Mark v as visited

   // Loop invariant: there is a path from vertex v at the
   // bottom of the stack s to the vertex at the top of s
   while (!s.isEmpty())
   {
      if (no unvisited vertices are adjacent to the vertex on the top of the stack)
         s.pop()  // Backtrack

      else
      {
         Select an unvisited vertex u adjacent to the vertex on the top of the stack
         s.push(u)
         Mark u as visited
      }
   }
```

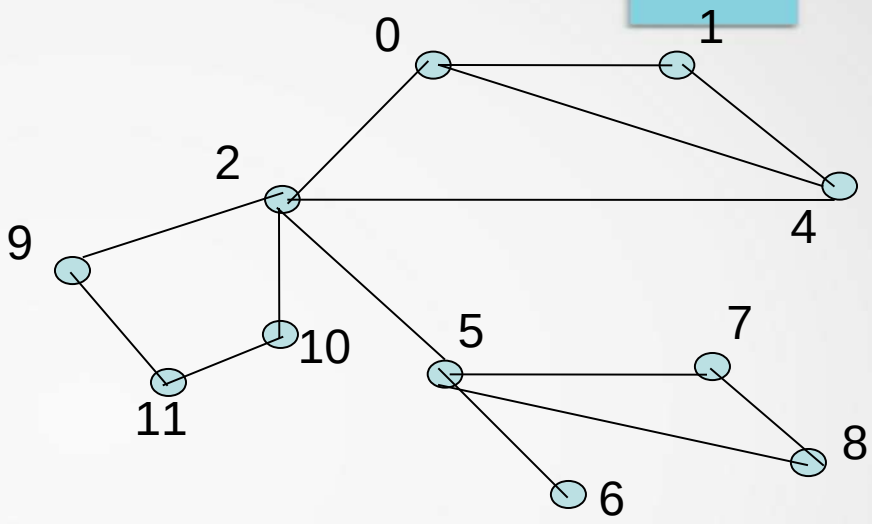
Breadth-First Search

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

Illustration of BFS



BFS Tree



Graph G

Implementation of BFS

- Observations:
  - the first node visited in each level is the first node from which to proceed to visit new nodes.
- This suggests that a queue is the proper data structure to remember the order of the steps.

We will redo the BFS on the previous graph, but this time with queues

BFS (Pseudo Code)

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

Breadth-First Search

- Visits all vertices adjacent to vertex before going forward
- Breadth-first search uses a queue

```
// Traverses a graph beginning at vertex v by using a
// breadth-first search: Iterative version.
bfs(v: Vertex)

   q = a new empty queue

   // Add v to queue and mark it
   q.enqueue(v)
   Mark v as visited

   while (!q.isEmpty())
   {
```

```
   while (!q.isEmpty())
   {
      q.dequeue(w)

      // Loop invariant: there is a path from vertex w to every vertex in the
      for (each unvisited vertex u adjacent to w)
      {
         Mark u as visited
         q.enqueue(u)
      }
   }
```

Depth-First Search

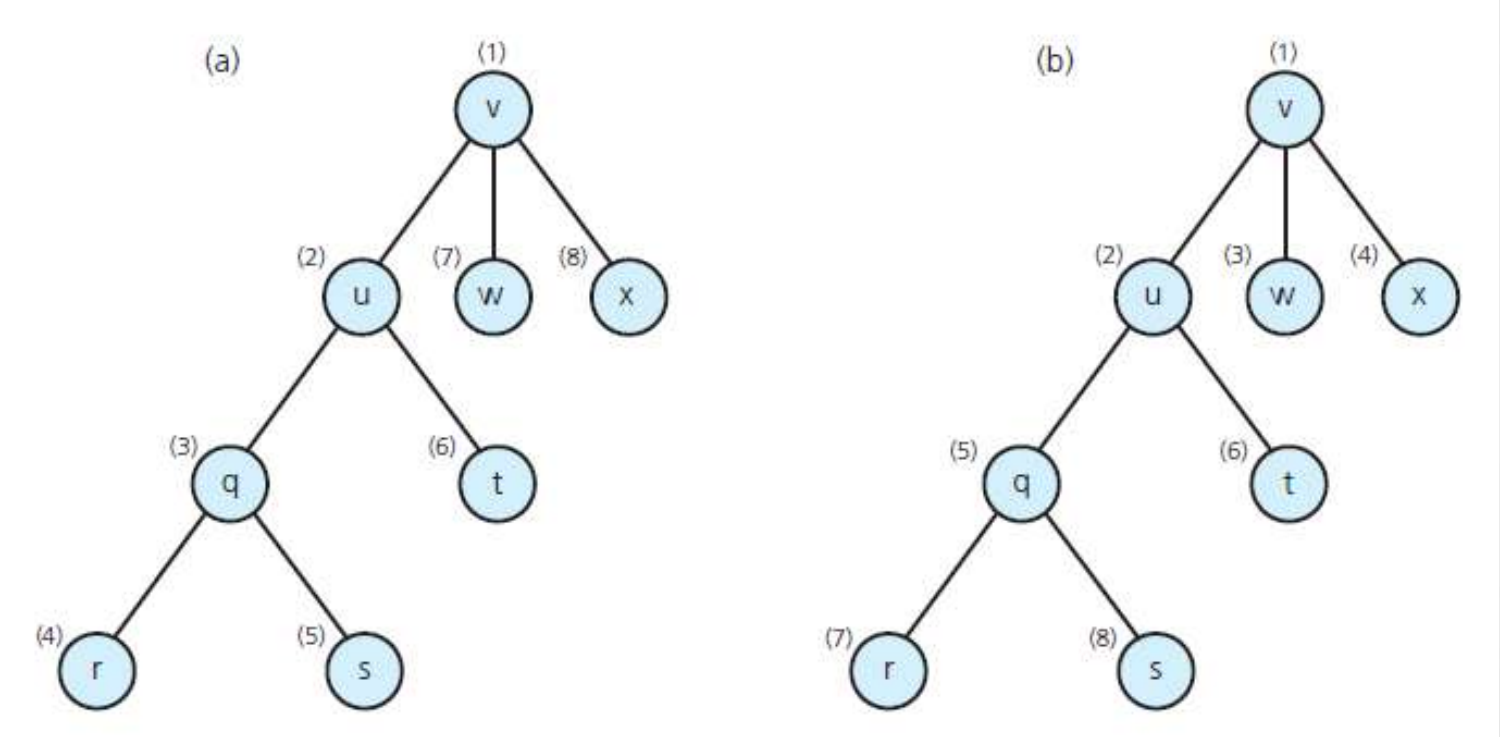


FIGURE 10 Visitation order for (a) a depth-first search; (b) a breadth-first search

Optimization problems

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A “greedy algorithm” sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Example: Counting money

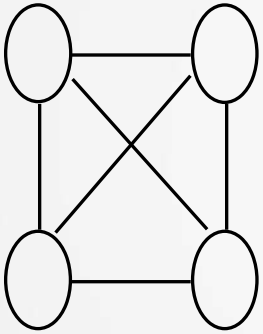
- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm would do this would be:
  At each step, take the largest possible bill or coin that does not overshoot
  - Example: To make $6.39, you can choose:
    - a $5 bill
    - a $1 bill, to make $6
    - a 25¢ coin, to make $6.25
    - A 10¢ coin, to make $6.35
    - four 1¢ coins, to make $6.39
- For US money, the greedy algorithm always gives the optimum solution

A failure of the greedy algorithm

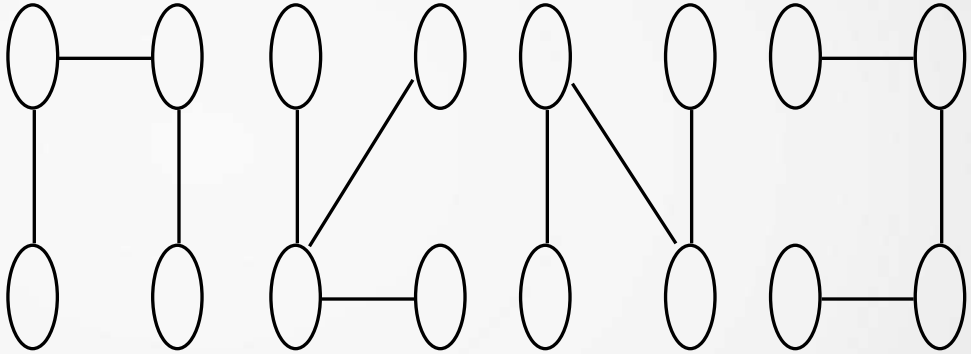
- In some (fictional) monetary system, “krons” come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
  - This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
  - This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution

spanning trees

A tree is a connected undirected graph with no cycles. It is a spanning tree of a graph G if it spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G).

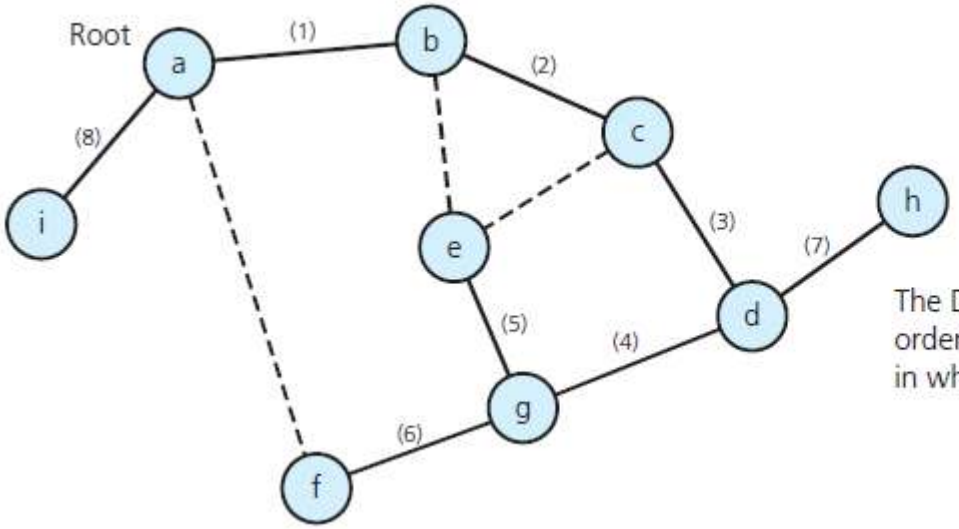


A connected, undirected graph



Four of the spanning trees of the graph

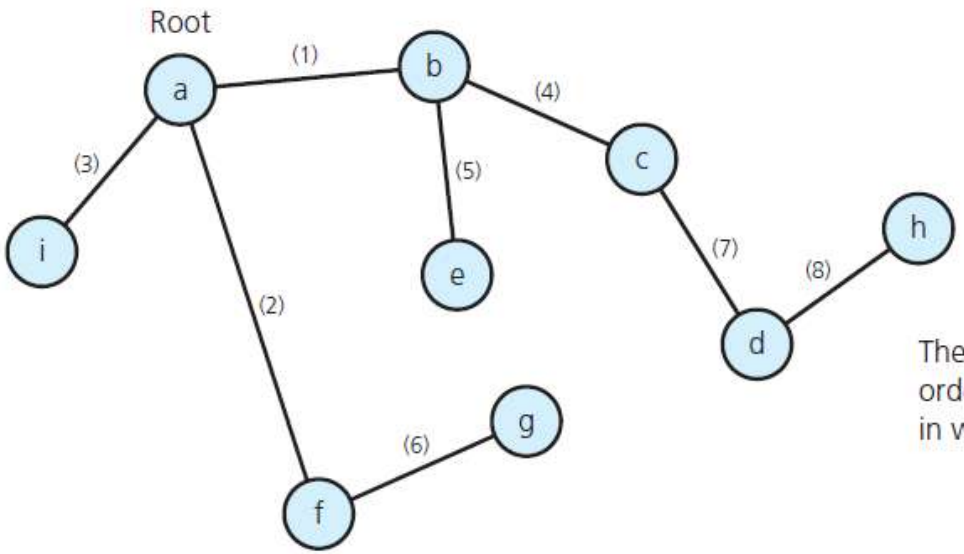
Spanning Trees



The DFS spanning tree algorithm visits vertices in this order: a, b, c, d, g, e, f, h, i. Numbers indicate the order in which the algorithm marks edges.

FIGURE 20 The DFS spanning tree rooted at vertex a for the graph in Figure 20-11

Spanning Trees



The BFS spanning tree algorithm visits vertices in this order: a, b, f, i, c, e, g, d, h. Numbers indicate the order in which the algorithm marks edges.

FIGURE 21 The BFS spanning tree rooted at vertex a for the graph in Figure 20-11

Spanning Trees

- Tree: an undirected connected graph without cycles
- Observations about undirected graphs
  1. Connected undirected graph with n vertices must have at least n – 1 edges.
  2. Connected undirected graph with n vertices, *exactly* n – 1 edges cannot contain a cycle
  3. A connected undirected graph with n vertices, *more than* n – 1 edges must contain at least one cycle

Minimum spanning tree Algorithms

Kruskal's algorithm

-Select the shortest edge in a network

-Select the next shortest edge which does not create a cycle

-Repeat step 2 until all vertices have been connected

Prim's algorithm

-Select any vertex

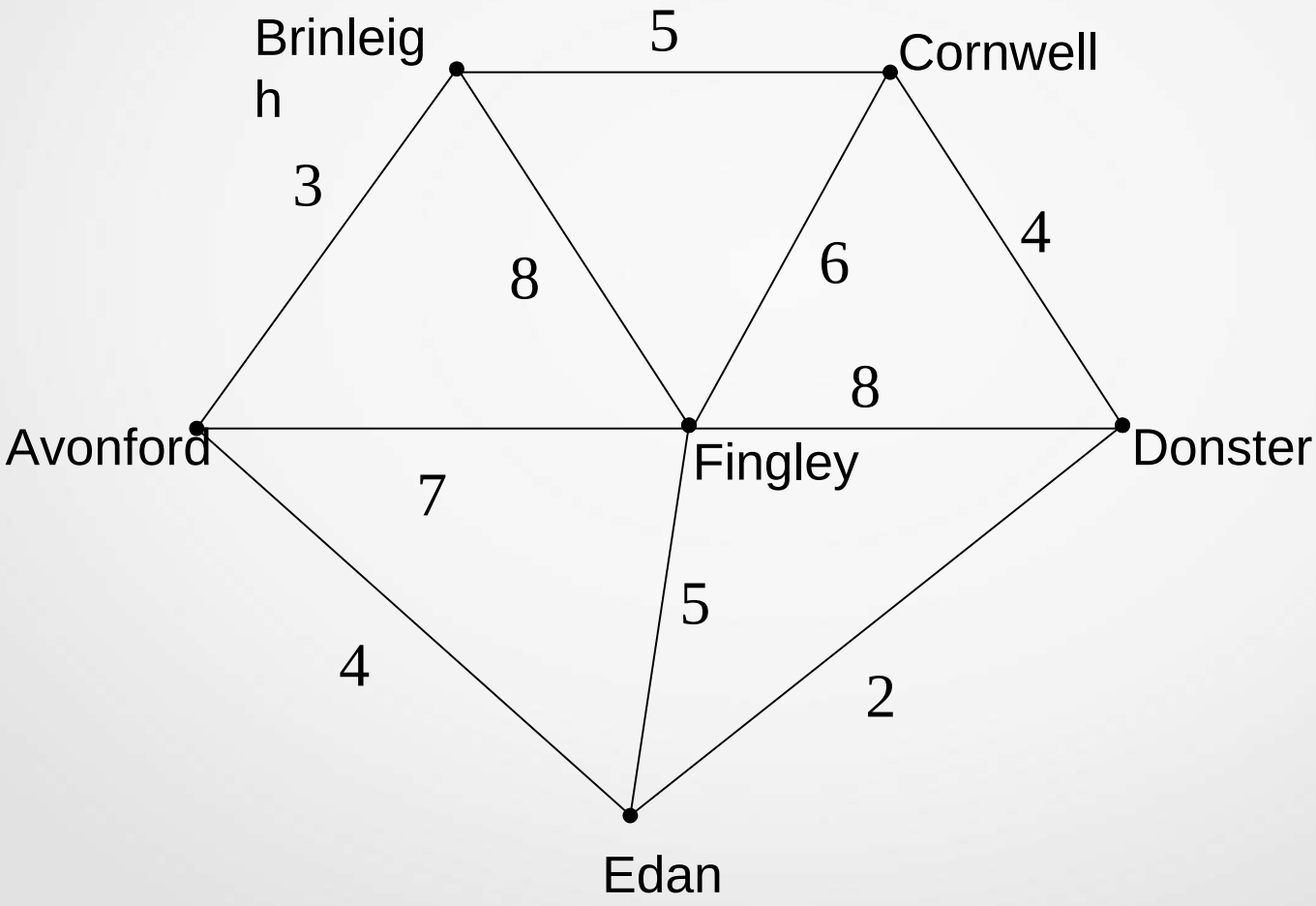
-Select the shortest edge connected to that vertex

-Select the shortest edge connected to any vertex already connected

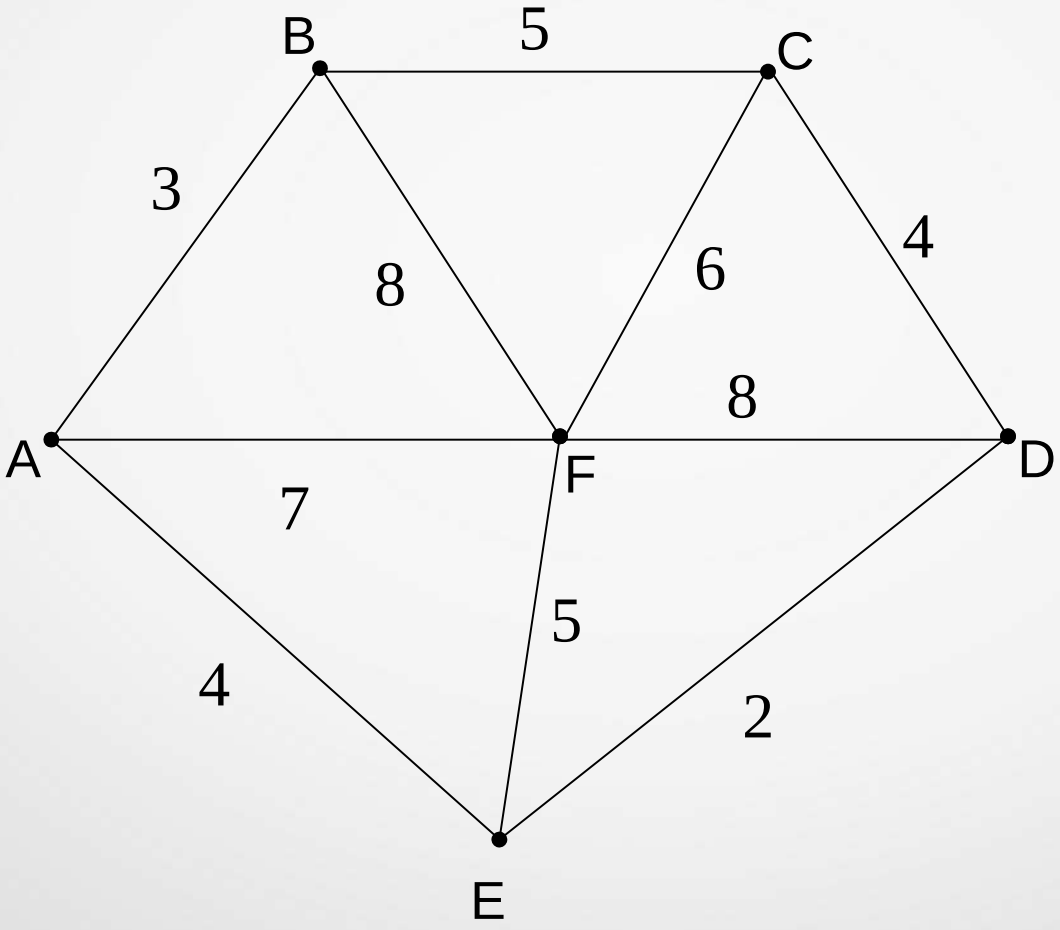
-Repeat step 3 until all vertices have been connected

Example

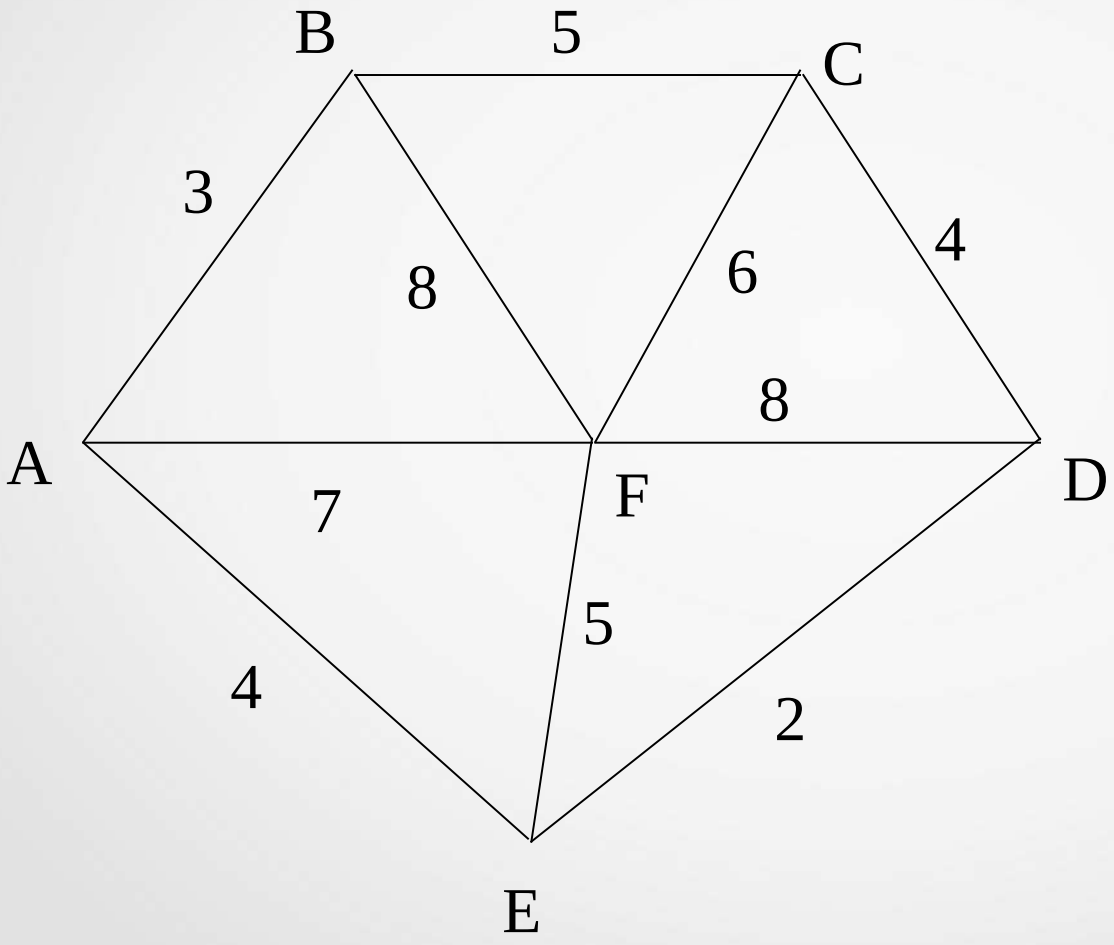
A cable company want to connect five villages to their network which currently extends to the market town of Avonford. What is the minimum length of cable needed?



We model the situation as a network, then the problem is to find the minimum connector for the network



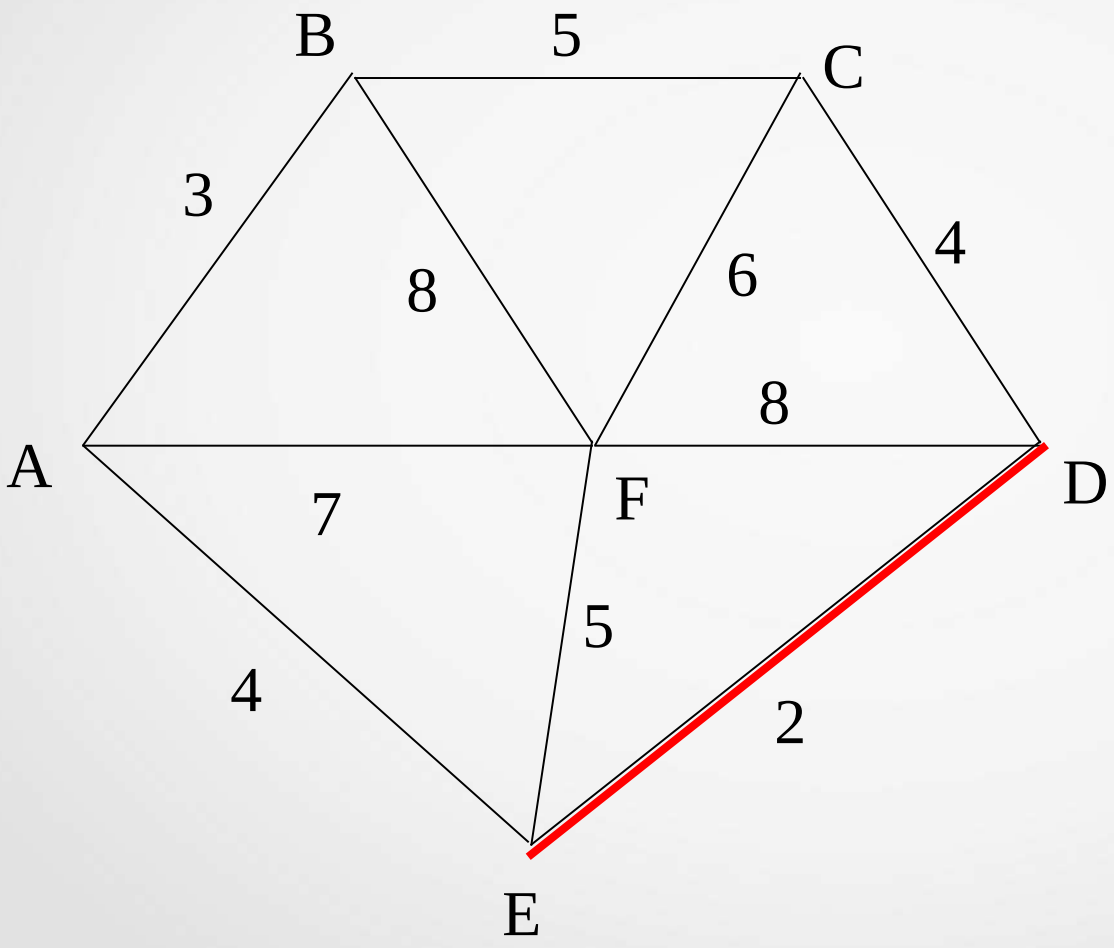
Kruskal's Algorithm



List the edges in order of size:

ED 2
AB 3
AE 4
CD 4
BC 5
EF 5
CF 6
AF 7
BF 8
CF 8

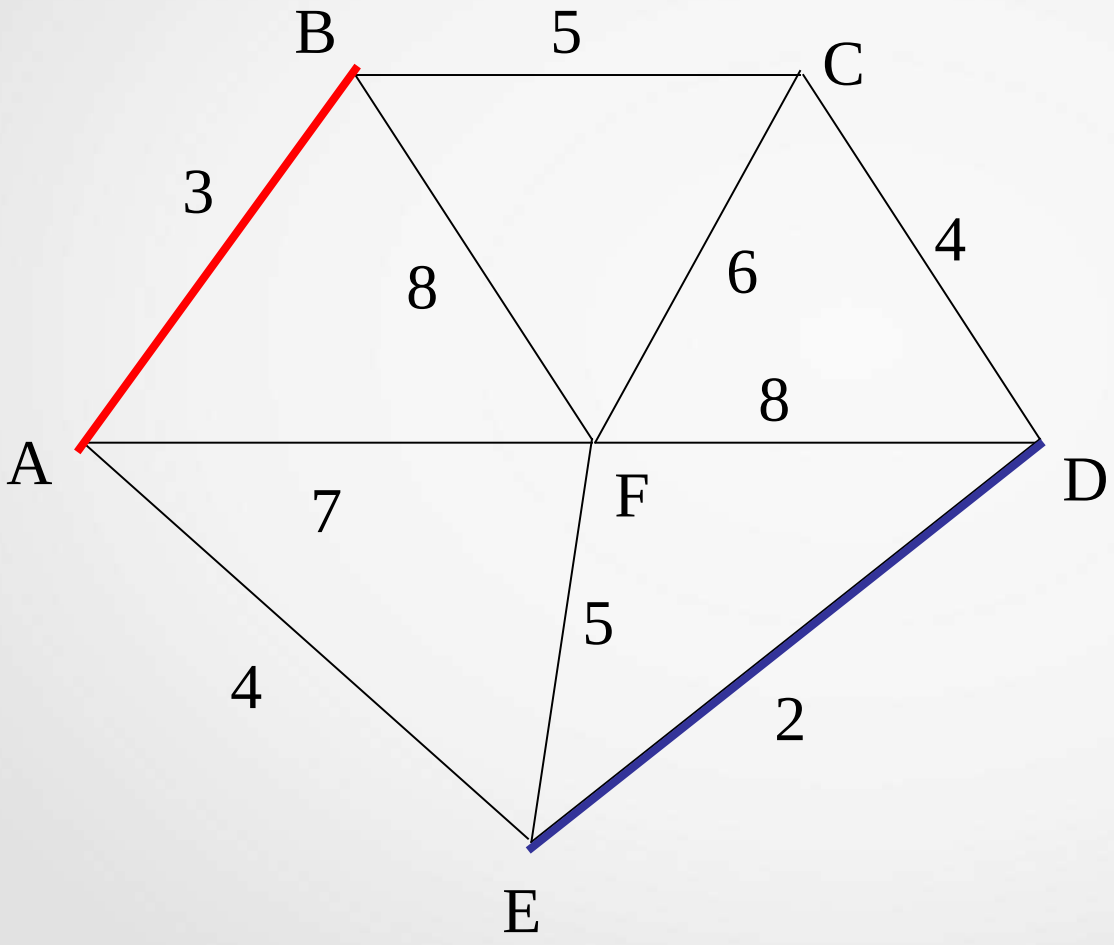
Kruskal's Algorithm



Select the shortest edge in the network

ED 2

Kruskal's Algorithm

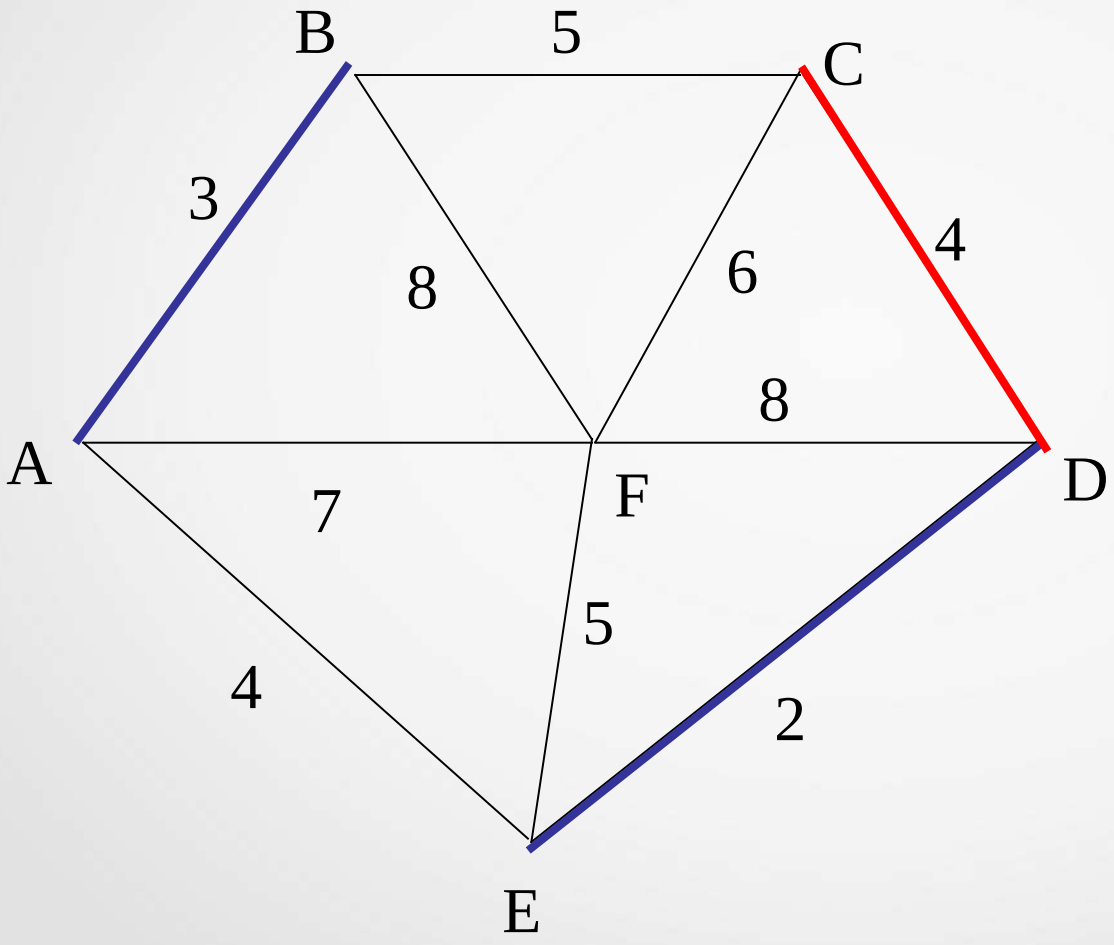


Select the next shortest edge which does not create a cycle

ED 2

AB 3

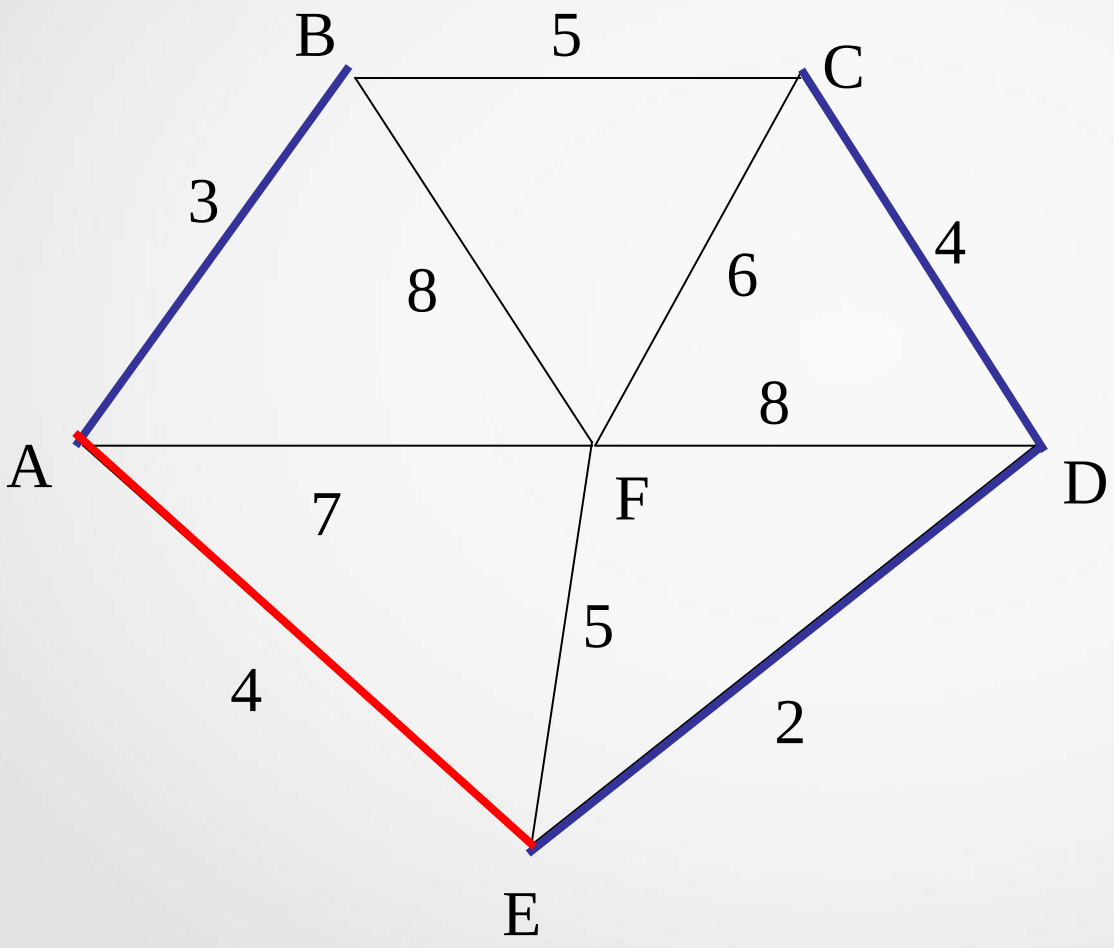
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3
CD 4 (or AE 4)

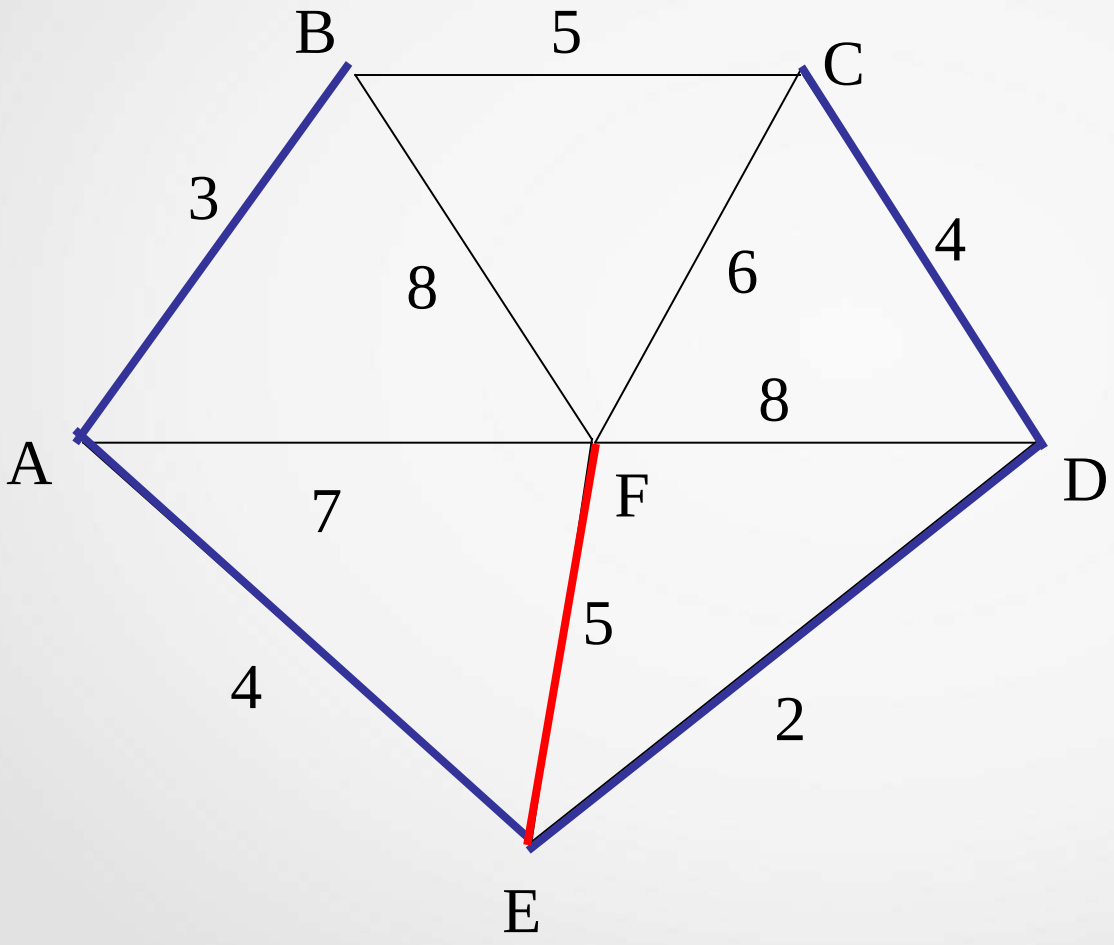
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3
CD 4
AE 4

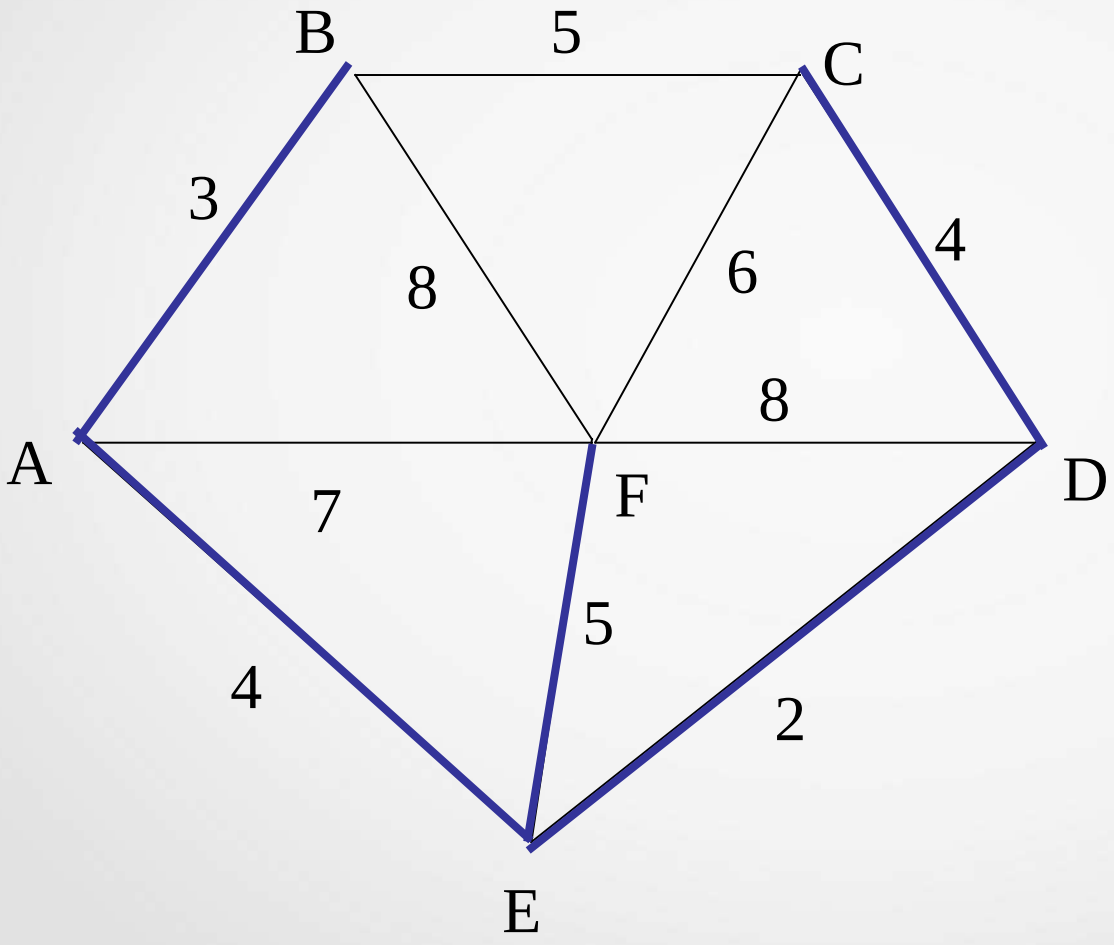
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3
CD 4
AE 4
BC 5 – forms a cycle
EF 5

Kruskal's Algorithm



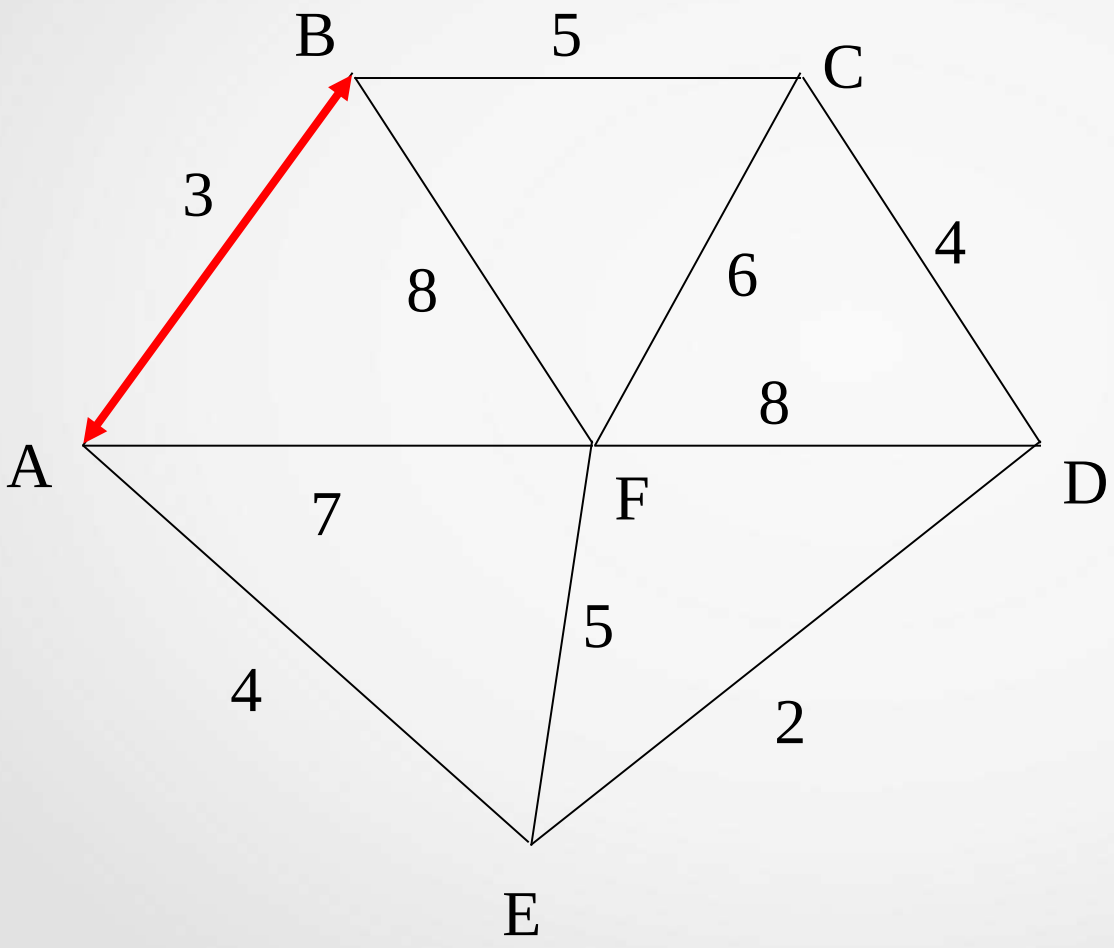
All vertices have been connected.

The solution is

ED 2
AB 3
CD 4
AE 4
EF 5

Total weight of tree: 18

Prim's Algorithm



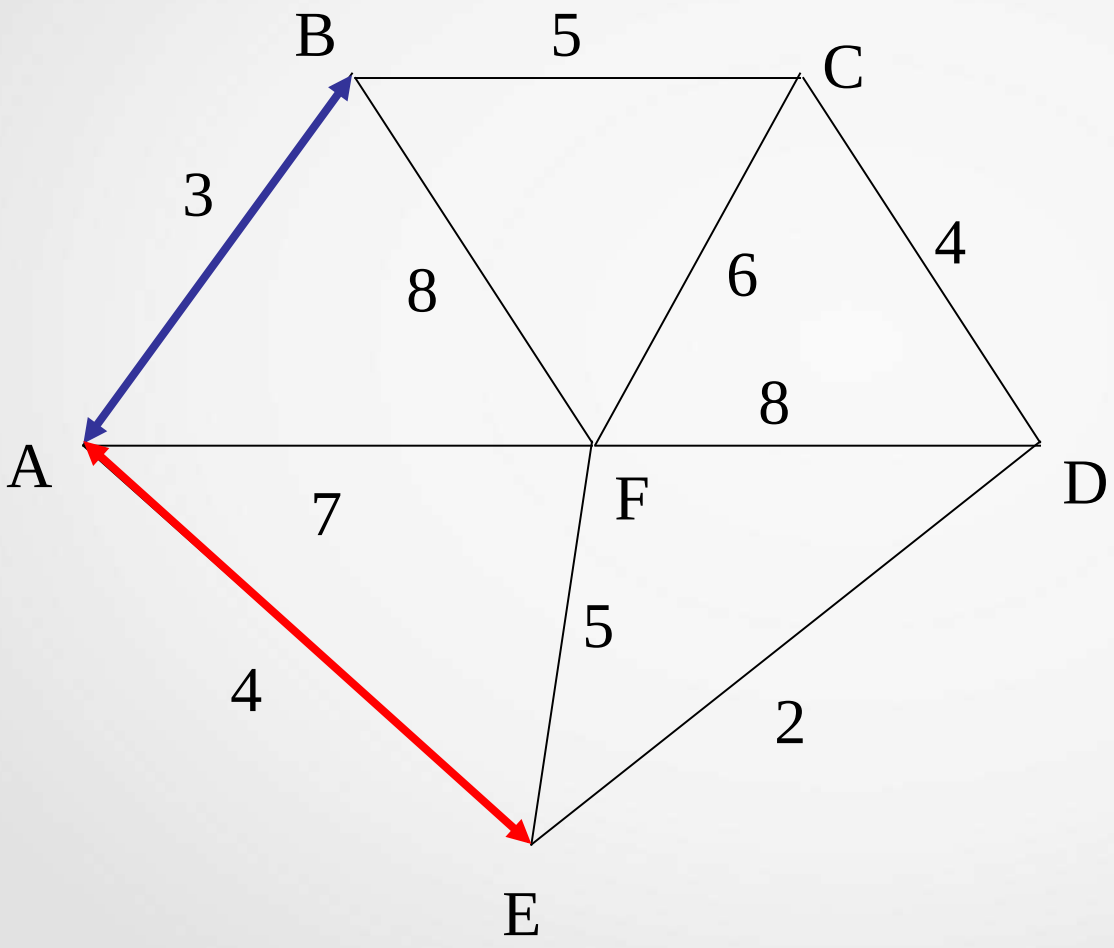
Select any vertex

A

Select the shortest edge connected to that vertex

AB 3

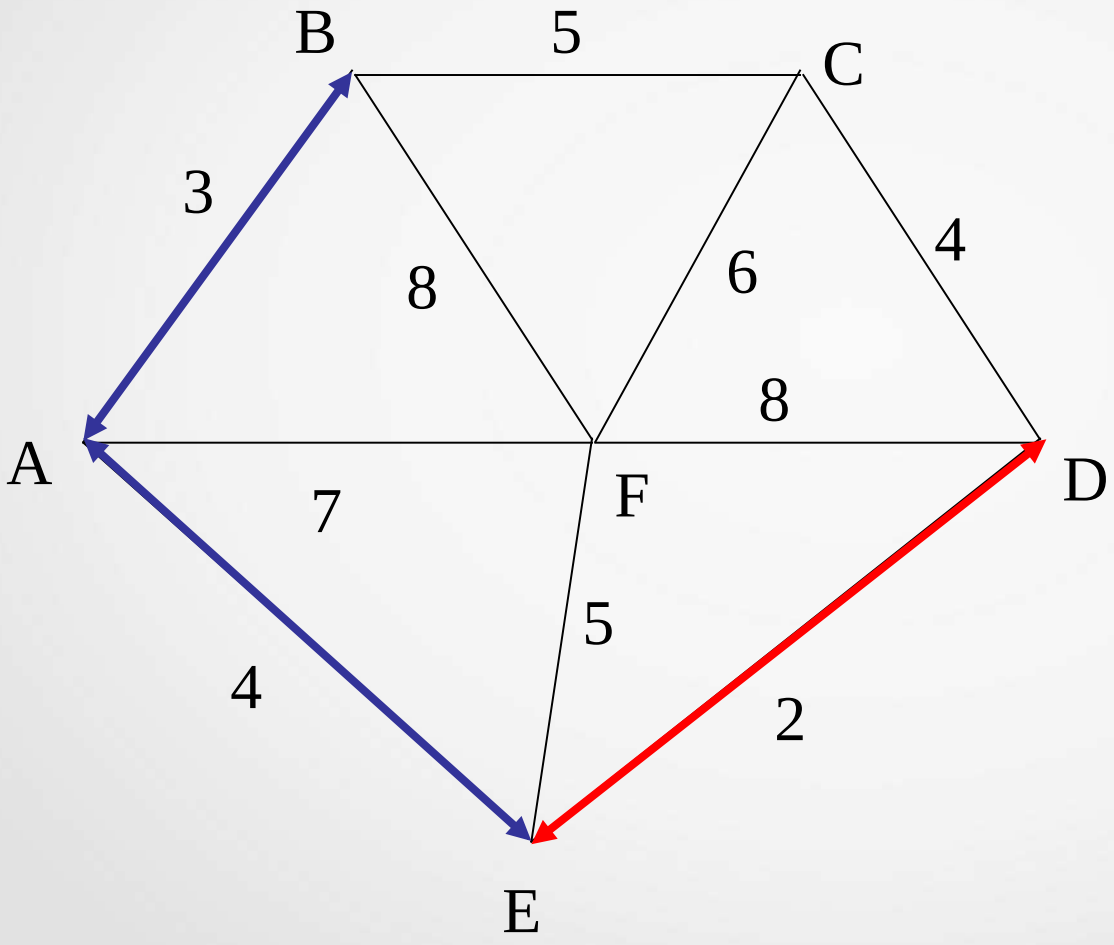
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

AE 4

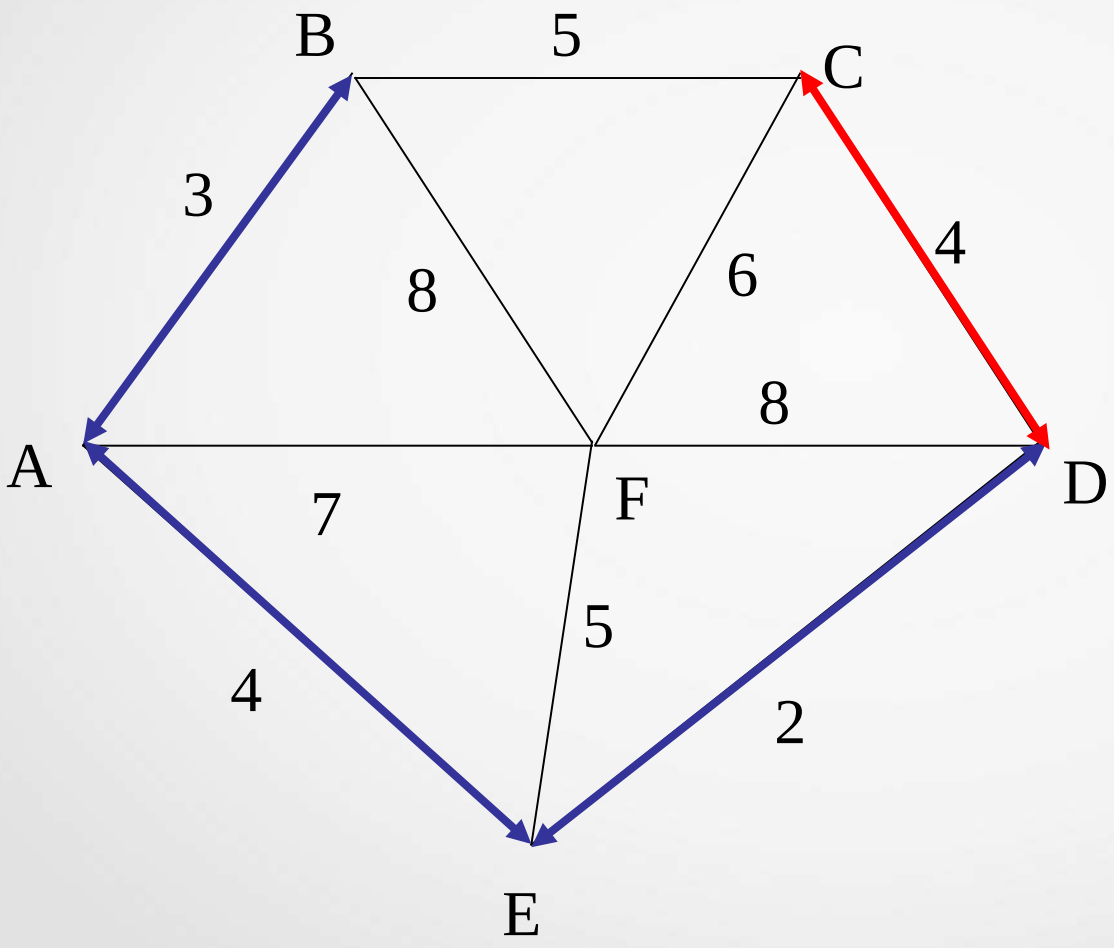
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

ED 2

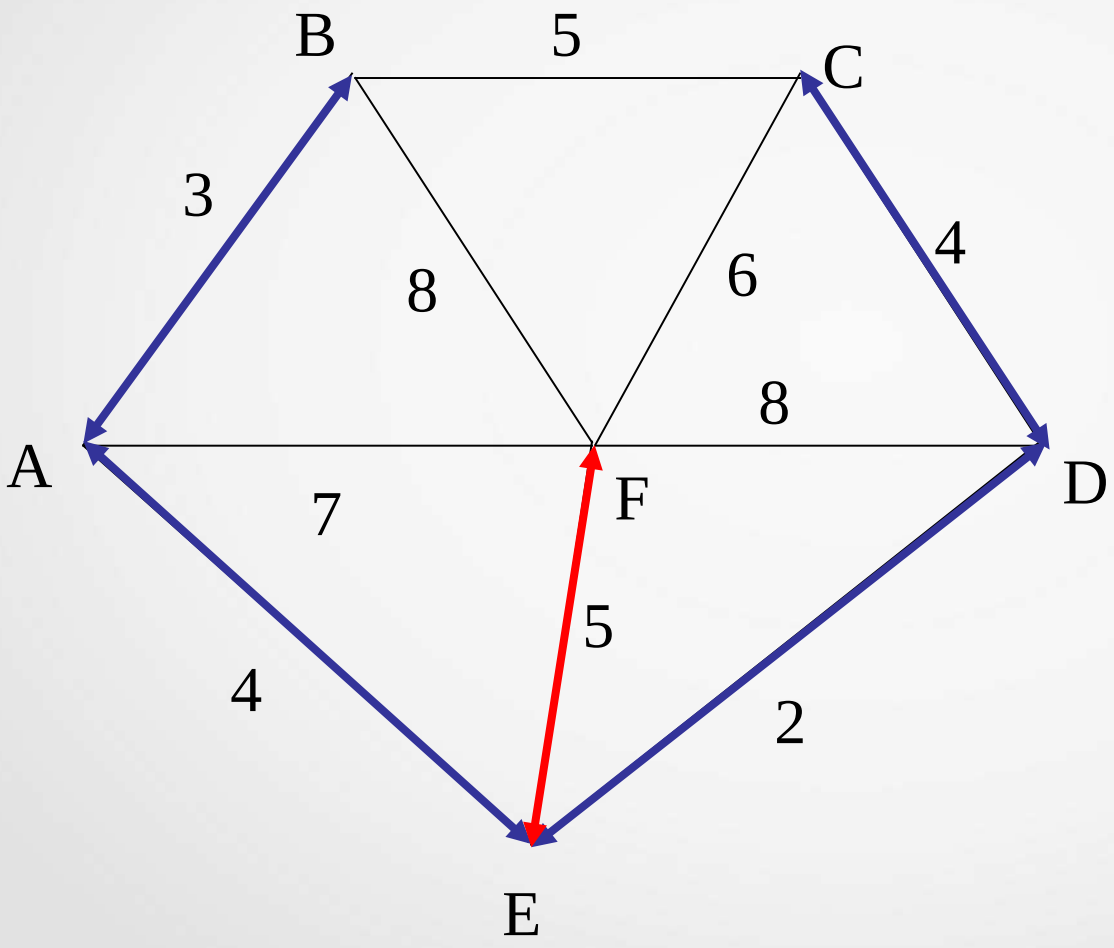
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

DC 4

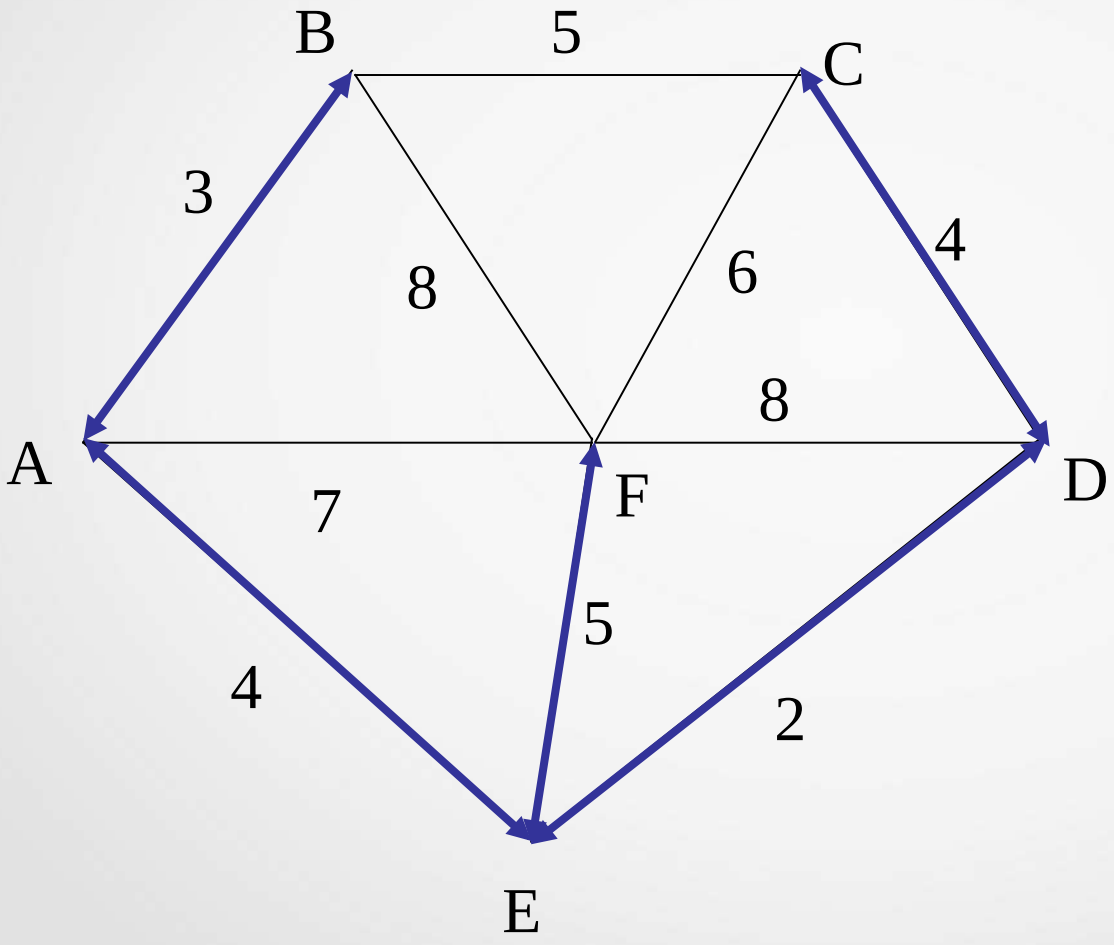
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

EF 5

Prim's Algorithm



All vertices have been connected.

The solution is

AB 3
AE 4
ED 2
DC 4
EF 5

Total weight of tree: 18

Some points to note

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

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

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

Minimum Spanning Trees

- Prim's Algorithm

```
// Determines a minimum spanning tree for a weighted,
// connected, undirected graph whose weights are
// nonnegative, beginning with any vertex v.
primsAlgorithm(v: Vertex)

   Mark vertex v as visited and include it in the minimum spanning tree
   while (there are unvisited vertices)
   {
      Find the least-cost edge (v, u) from a visited vertex v to some unvisited vertex u
      Mark u as visited
      Add the vertex u and the edge (v, u) to the minimum spanning tree
   }
```

Minimum spanning tree algorithm

- Kruskal's Algorithm

T (the final spanning tree) is defined to be the empty set;

2. For each vertex v of G, make the empty set out of v;

3. Sort the edges of G in ascending (non-decreasing) order;

4. For each edge (u, v) from the sored list of step 3.

    If u and v belong to different sets

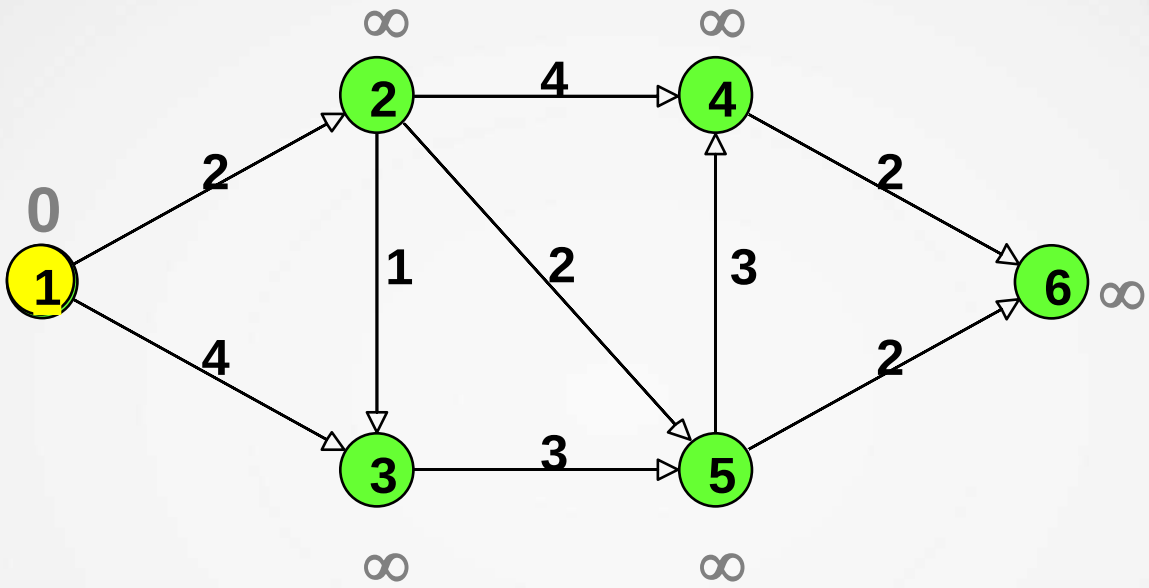
        Add (u,v) to T;

        Get together u and v in one single set;

5. Return T

Dijkstra's shortest-path algorithm

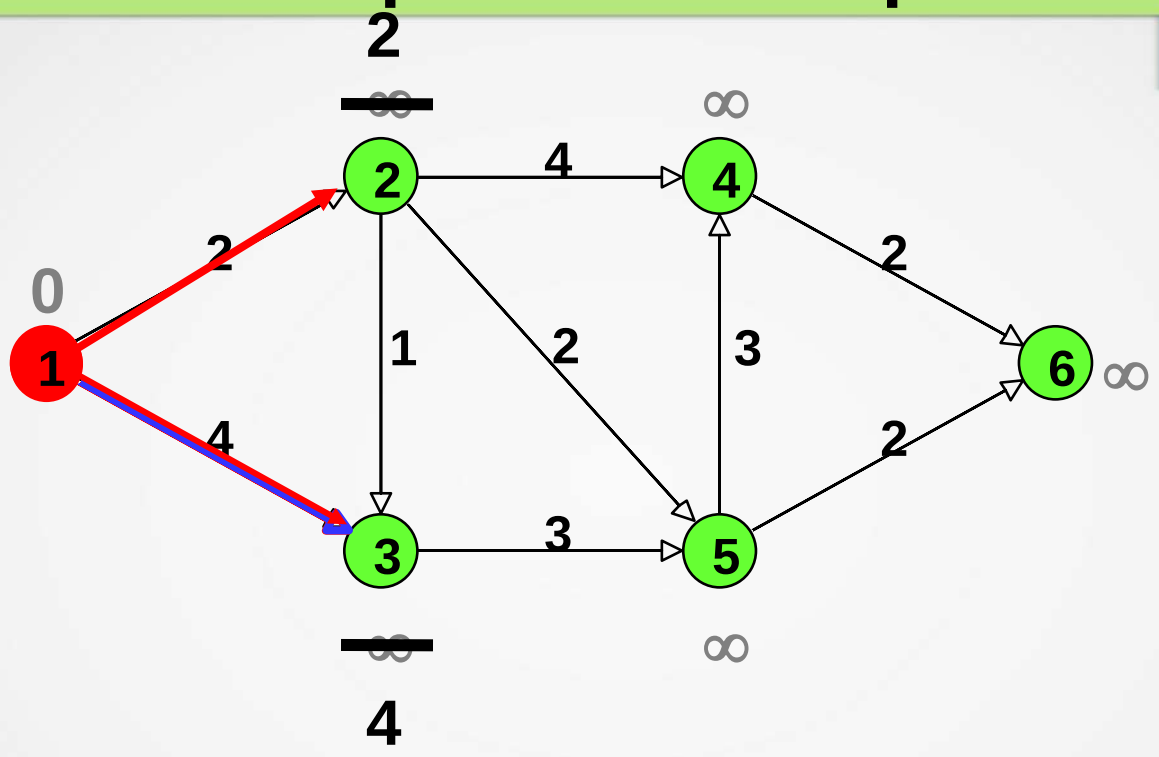
An Example



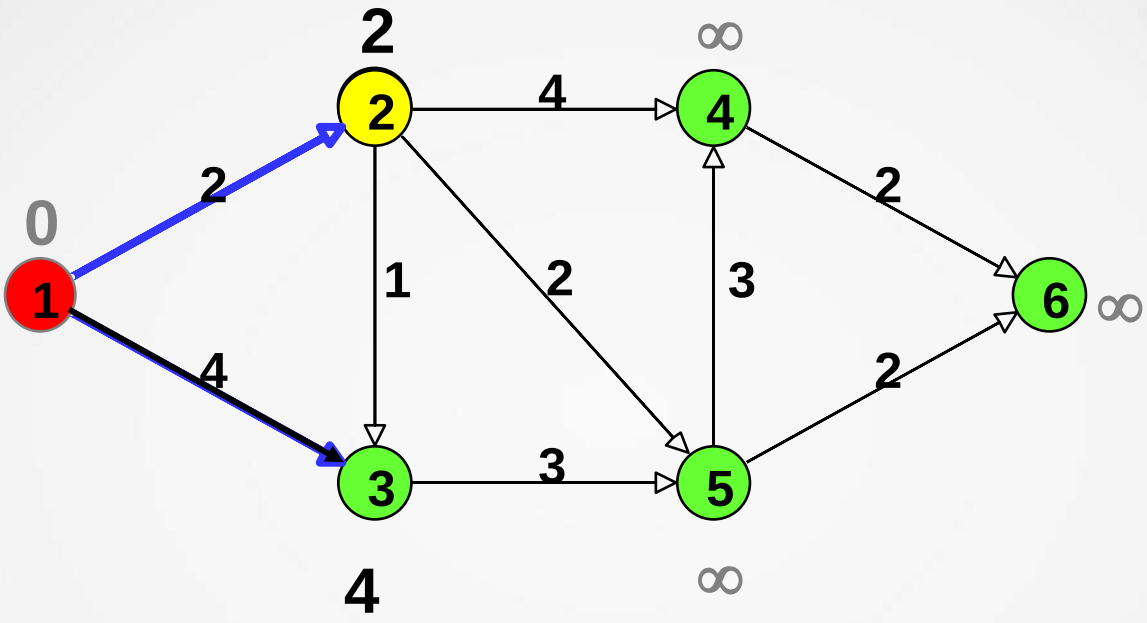
Initialize

Select the node with the minimum temporary distance label.

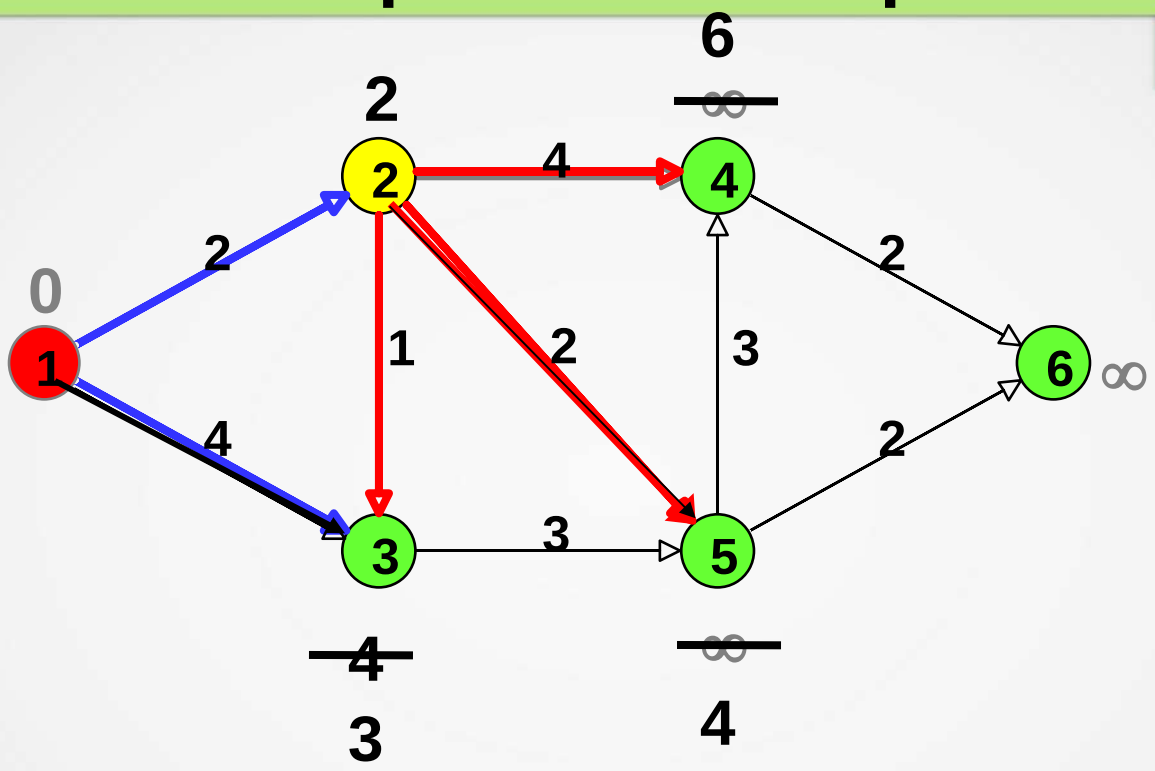
Update Step



Choose Minimum Temporary Label

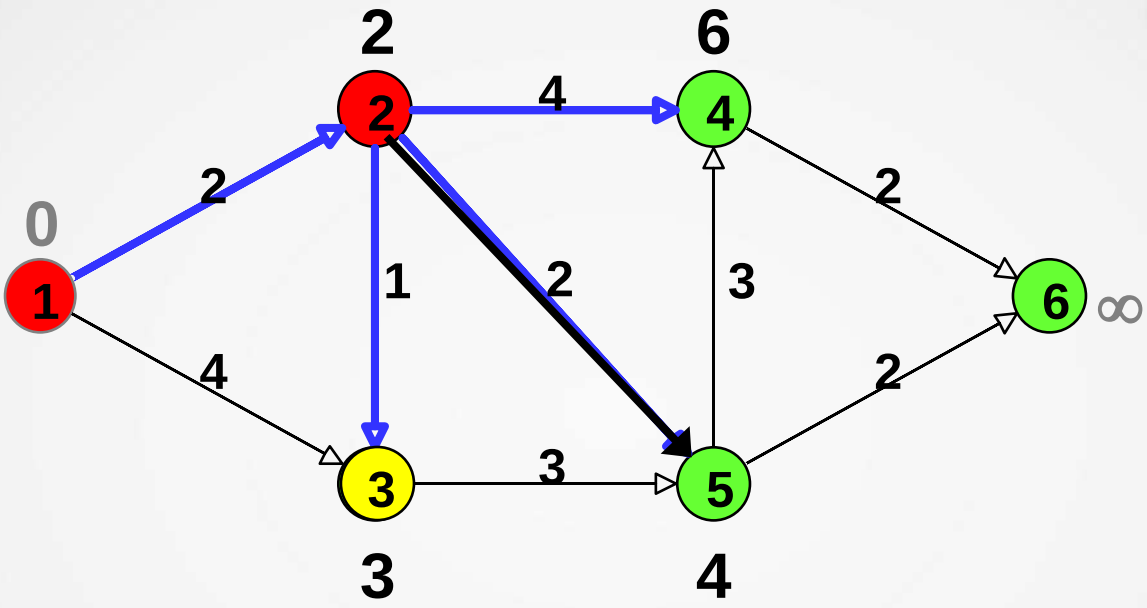


Update Step

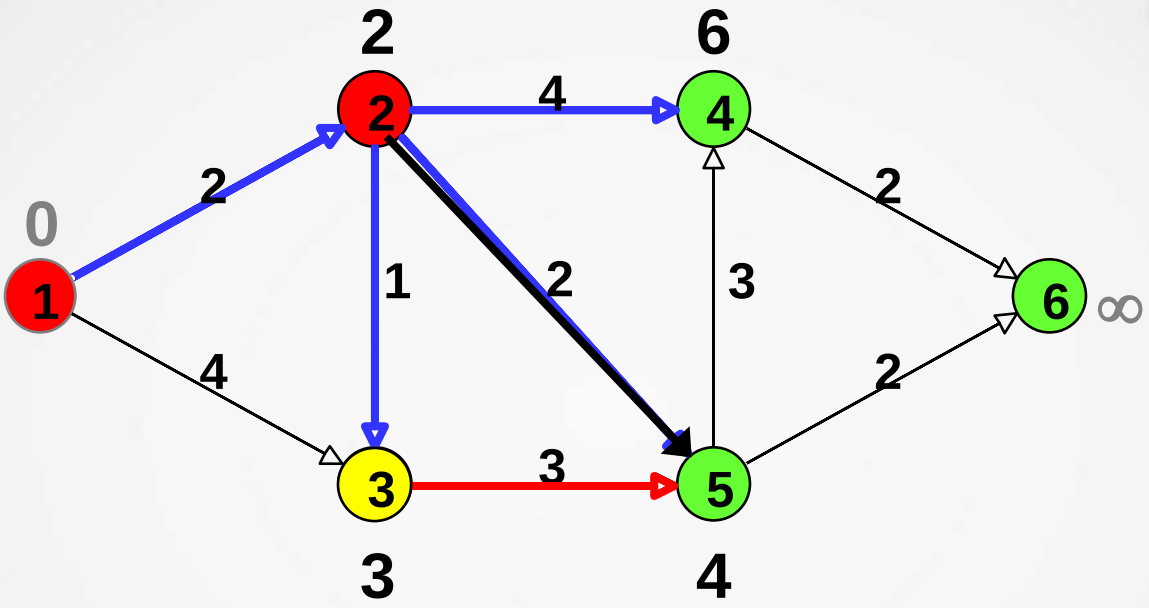


The predecessor of node 3 is now node 2

Choose Minimum Temporary Label

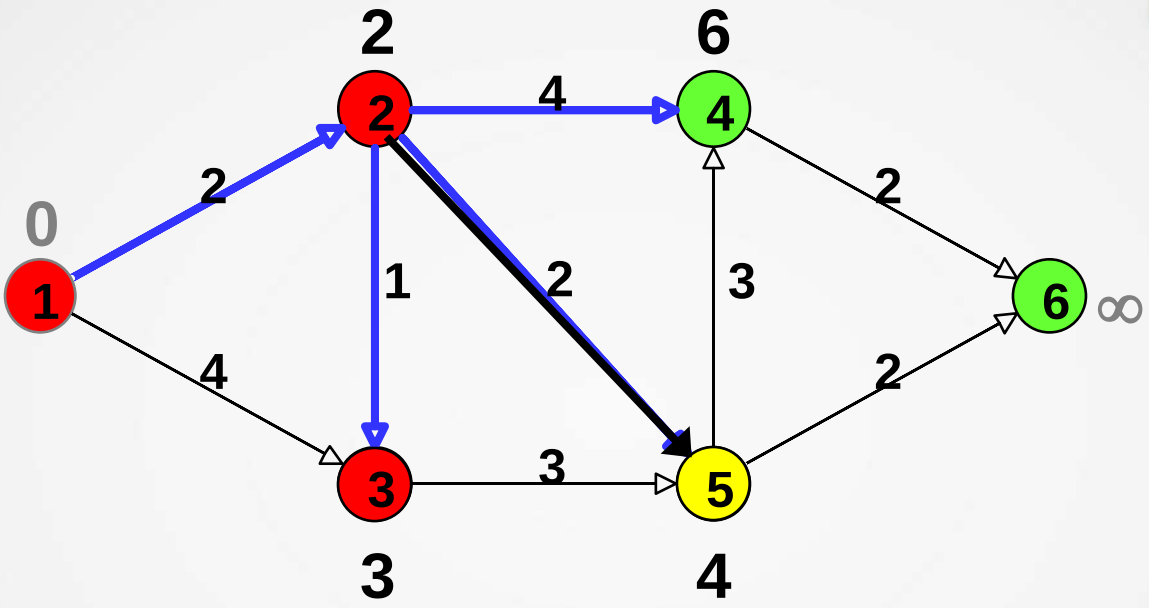


Update

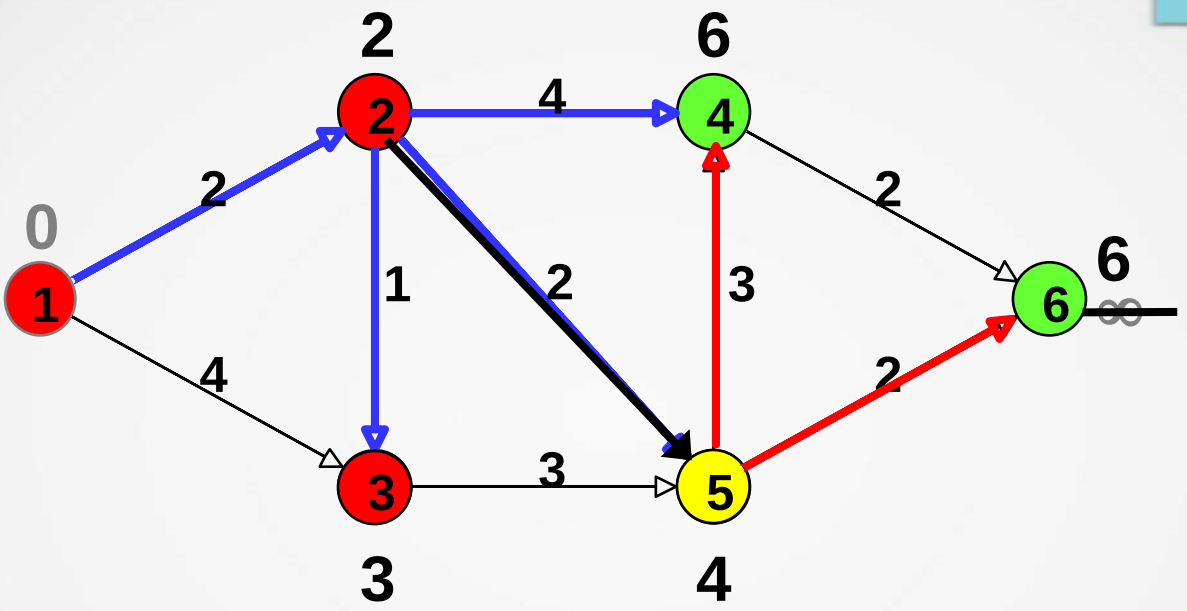


d(5) is not changed.

Choose Minimum Temporary Label

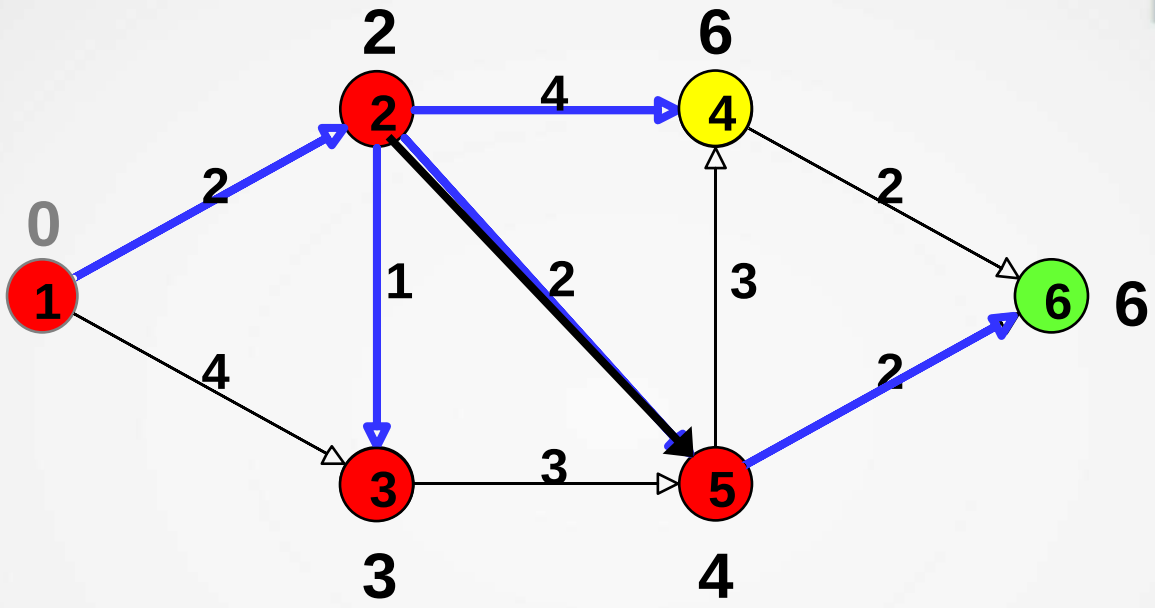


Update

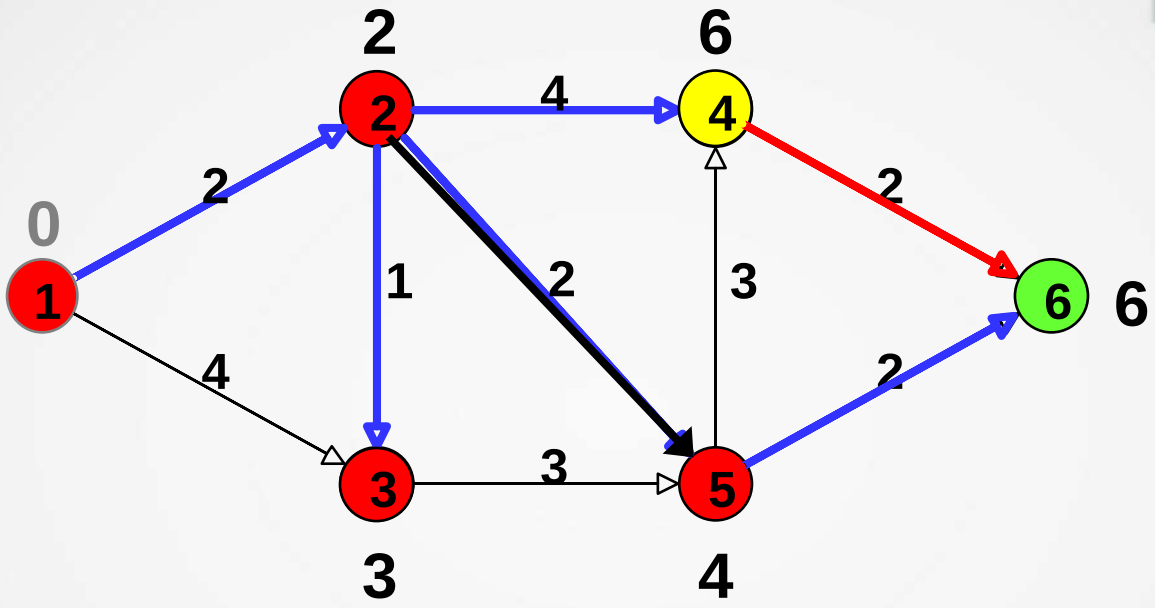


d(4) is not changed

Choose Minimum Temporary Label

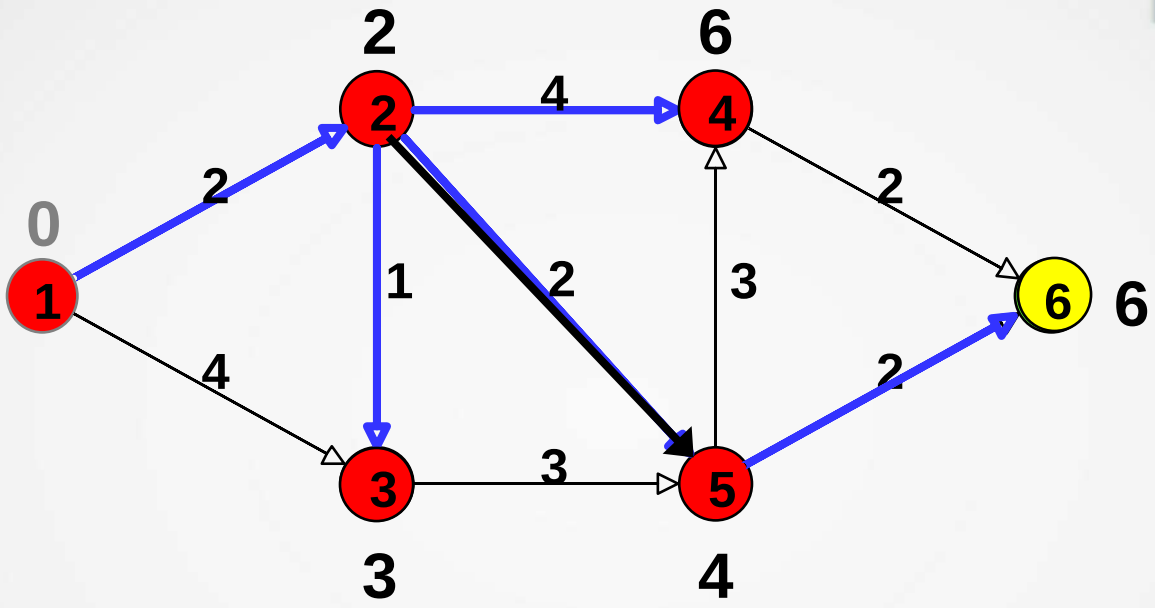


Update



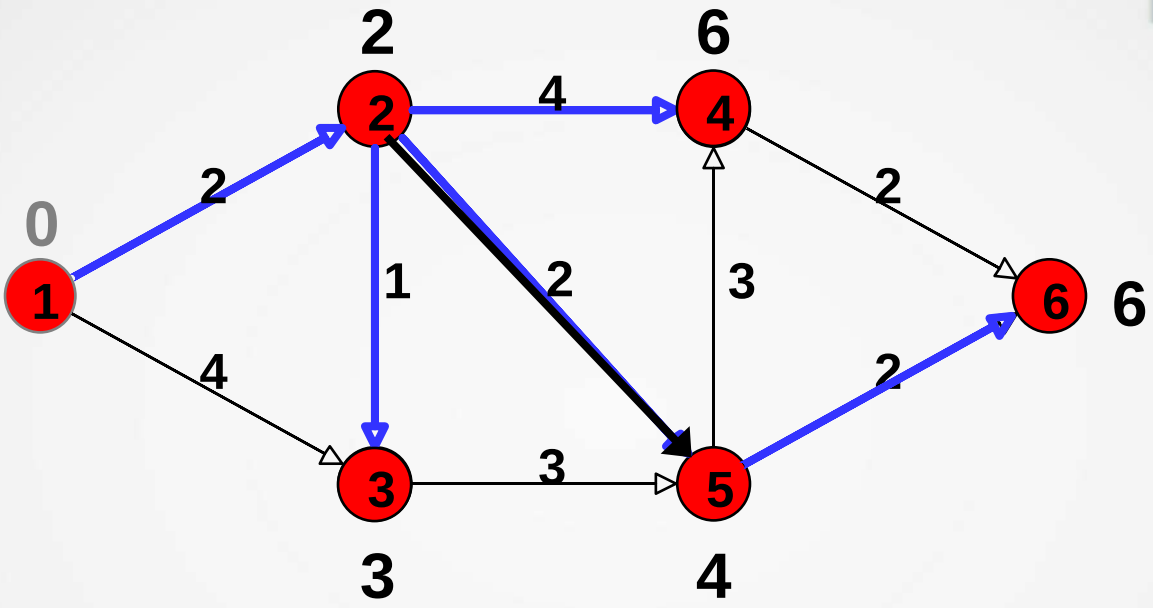
d(6) is not updated

Choose Minimum Temporary Label



There is nothing to update

End of Algorithm



All nodes are now permanent

The predecessors form a tree

The shortest path from node 1 to node 6 can be found by tracing back predecessors

Shortest Paths

- Shortest path between two vertices in a weighted graph has smallest edge-weight sum

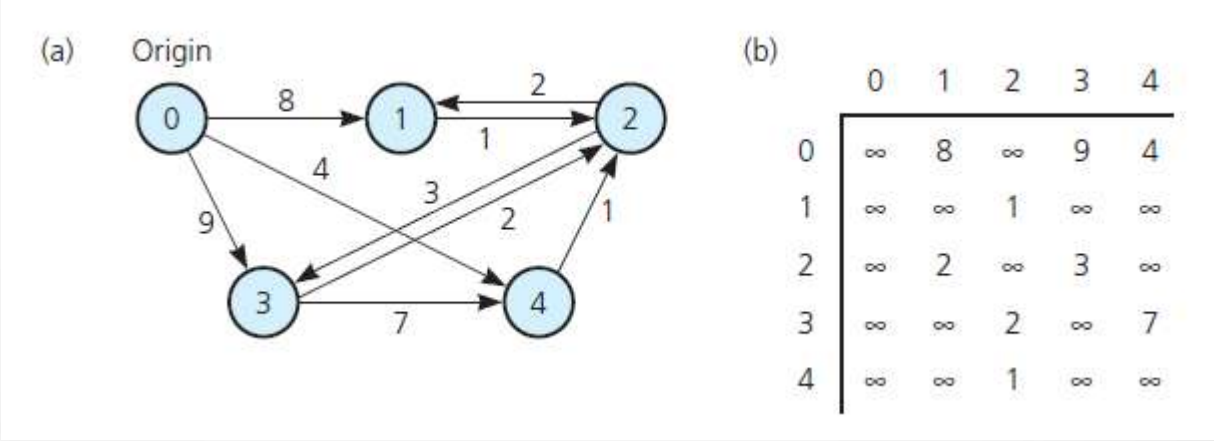


FIGURE (a) A weighted directed graph and (b) its adjacency matrix

Shortest Paths

			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

FIGURE A trace of the shortest-path algorithm applied to the graph in Figure 20-24 a

Shortest Paths

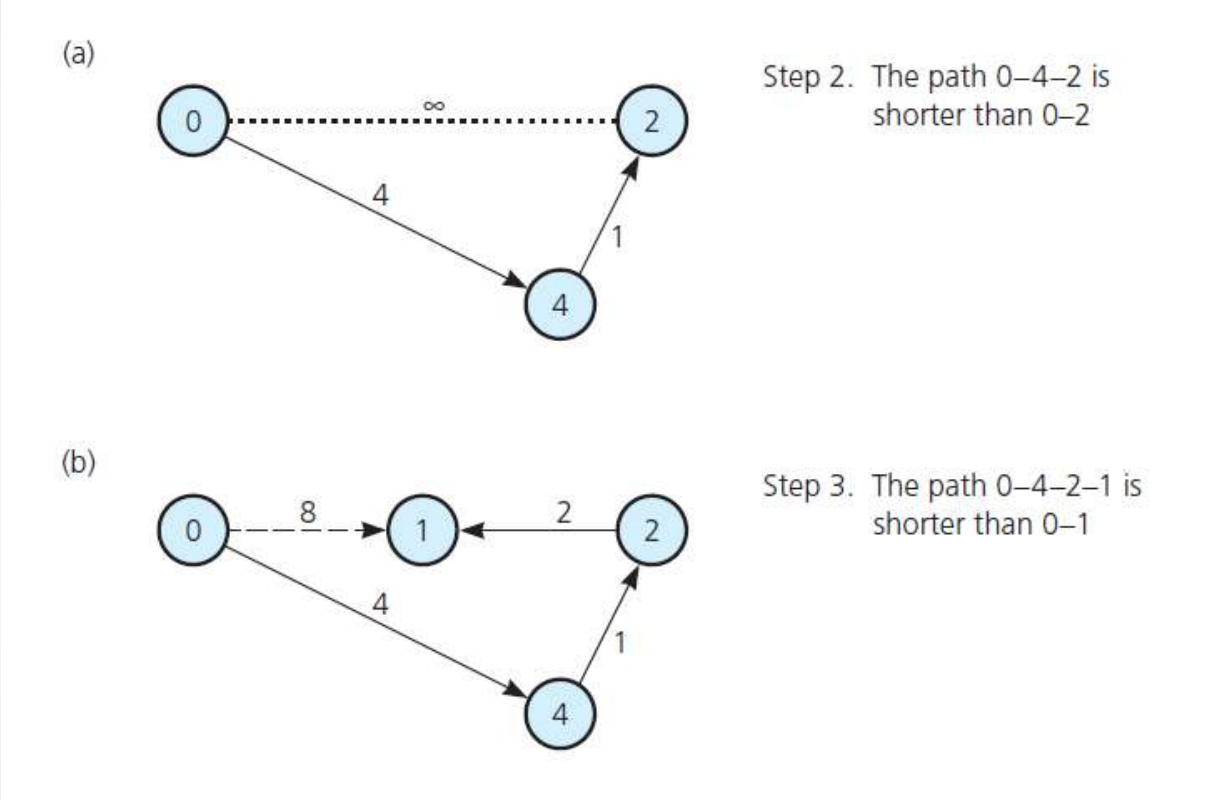


FIGURE 26 Checking weight [u] by examining the graph: (a) weight [2] in step 2; (b) weight [1] in step 3;

Shortest Paths

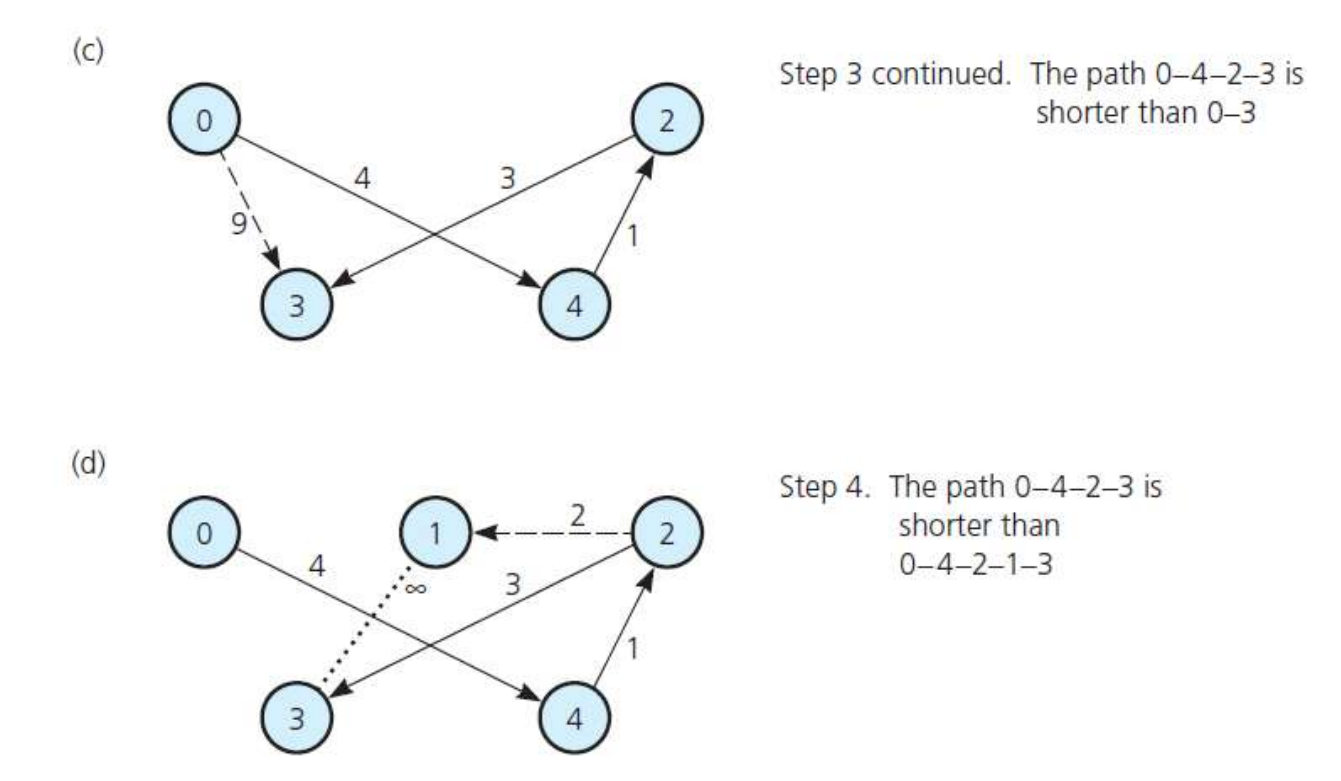


FIGURE Checking weight [u] by examining the graph: (c) weight [3] in step 3; (d) weight [3] in step 4

Shortest Paths

- Dijkstra's shortest-path algorithm

```
// Finds the minimum-cost paths between an origin vertex
// (vertex 0) and all other vertices in a weighted directed
// graph theGraph; theGraph's weights are nonnegative.
shortestPath(theGraph: Graph, weight: WeightArray)

   // Step 1: initialization
   Create a set vertexSet that contains only vertex 0
   n = number of vertices in theGraph
   for (v = 0 through n - 1)
      weight[v] = matrix[0][v]

   // Steps 2 through n
   // Invariant: For v not in vertexSet, weight[v] is the
   // smallest weight of all paths from 0 to v that pass
   // through only vertices in vertexSet before reaching
   // v. For v in vertexSet, weight[v] is the smallest
   // weight of all paths from 0 to v (including paths
   // outside vertexSet) and the shortest path
```

Shortest Paths

- Dijkstra's shortest-path algorithm, ctd.

```
// smallest weight of all paths from 0 to v that pass
// through only vertices in vertexSet before reaching
// v. For v in vertexSet, weight[v] is the smallest
// weight of all paths from 0 to v (including paths
// outside vertexSet), and the shortest path
// from 0 to v lies entirely in vertexSet.
for (step = 2 through n)
{
   Find the smallest weight[v] such that v is not in vertexSet
   Add v to vertexSet

   // Check weight[u] for all u not in vertexSet
   for (all vertices u not in vertexSet)
      if (weight[u] > weight[v] + matrix[v][u])
         weight[u] = weight[v] + matrix[v][u]
}
```

Applications of Graphs

- Topological Sorting

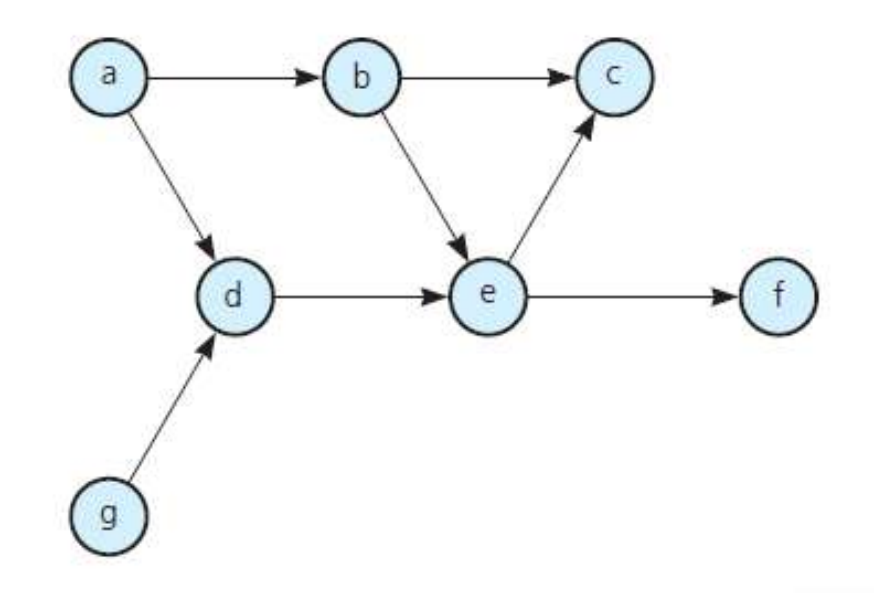


FIGURE A directed graph without cycles

Applications of Graphs

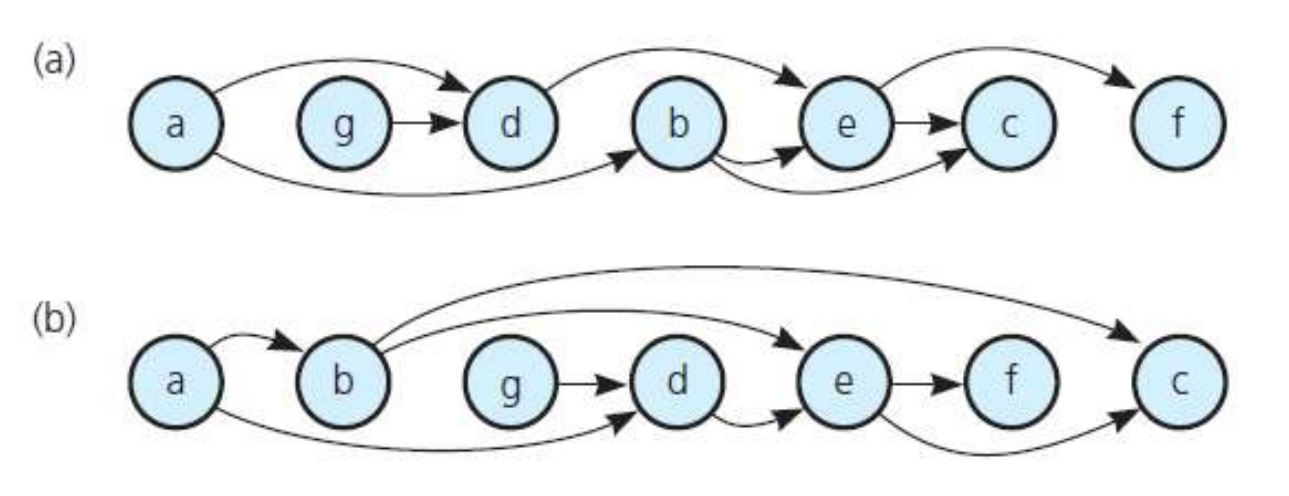


FIGURE The graph in Figure 14 arranged according to the topological orders (a) *a, g, d, b, e, c, f* and (b) *a, b, g, d, e, f, c*

Applications of Graphs

- Topological sorting algorithm

```
// Arranges the vertices in graph theGraph into a
// topological order and places them in list aList.
topSort1(theGraph: Graph, aList: List)

   n = number of vertices in theGraph
   for (step = 1 through n)
   {
      Select a vertex v that has no successors
      aList.insert(1, v)
      Remove from theGraph vertex v and its edges
   }
```

Applications of Graphs

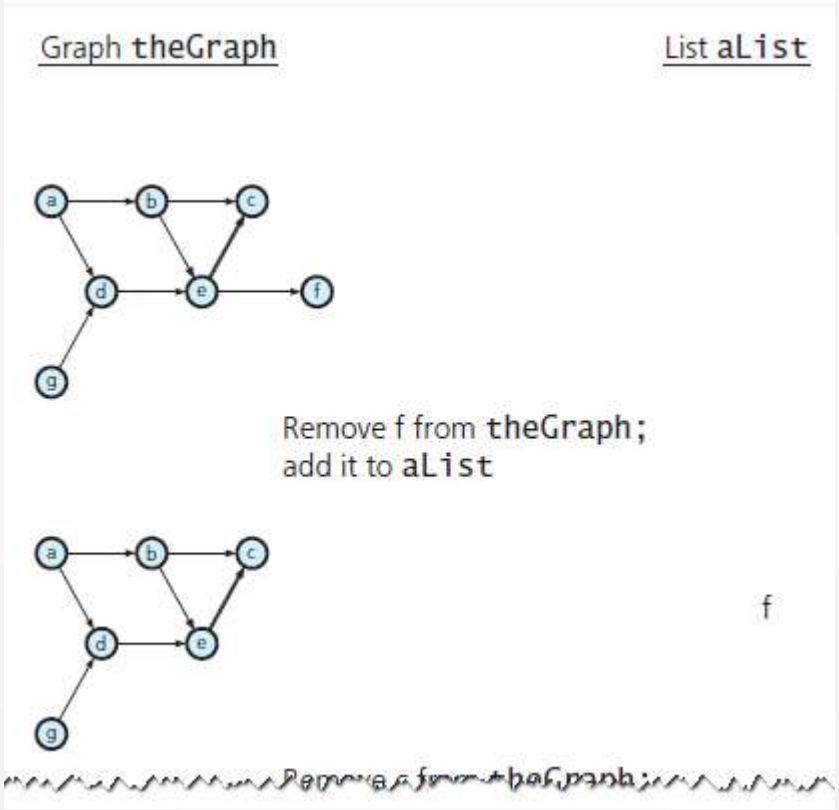


FIGURE A trace of topSort1 for the graph in Figure 14

Applications of Graphs

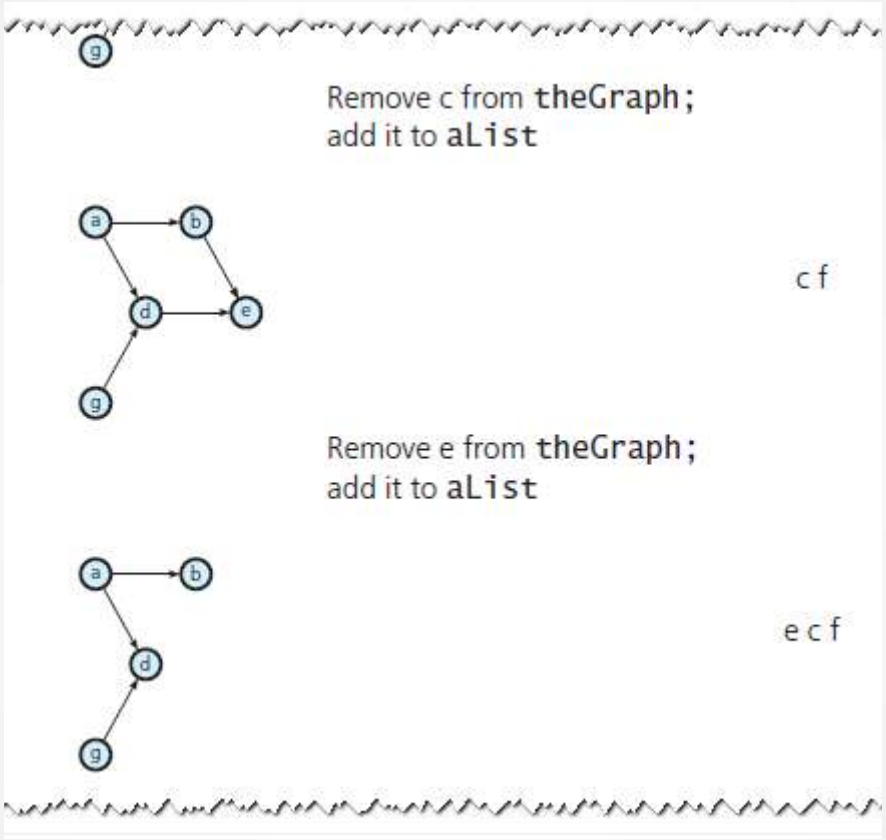


FIGURE A trace of topSort1 for the graph in Figure 20-14

Applications of Graphs

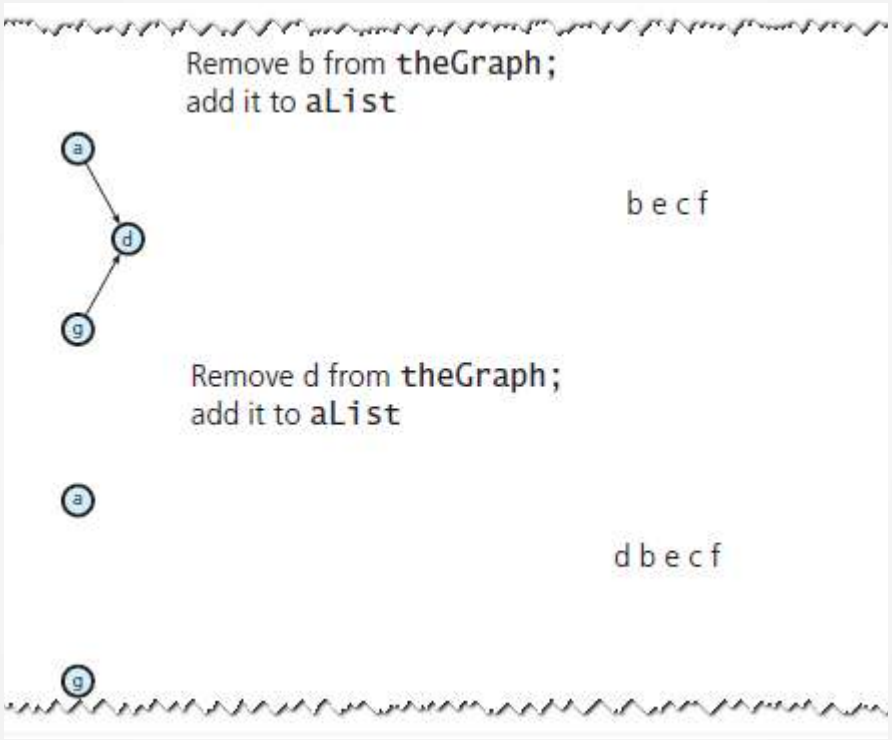


FIGURE A trace of topSort1 for the graph in Figure 14

Applications of Graphs

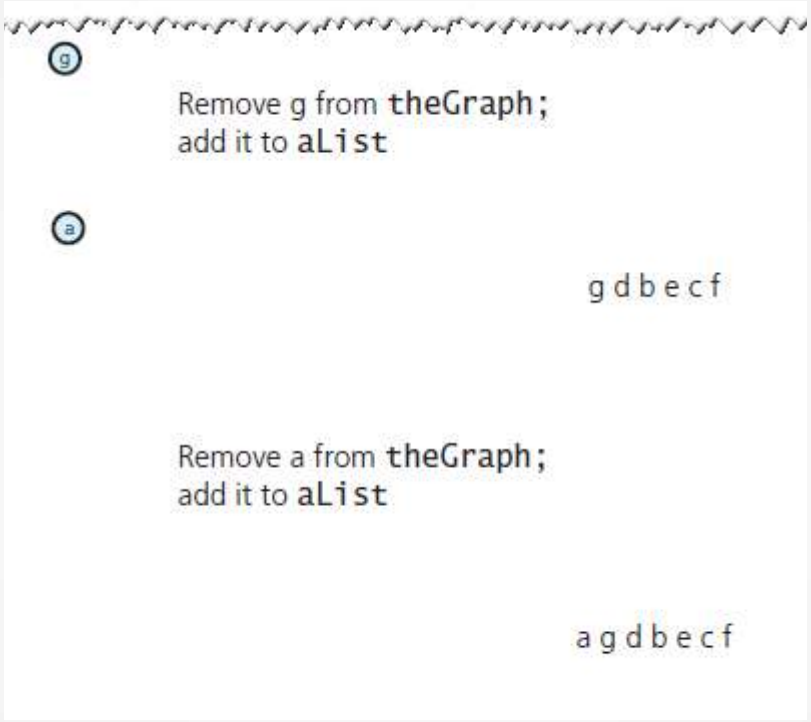


FIGURE A trace of topSort1 for the graph in Figure 20-14

Case Studies-

Webgraph-

It is a directed graph whose vertices are nothing but the web pages and directed edges between any two vertices are nothing but the web pages and the directed edges between any two vertices V1 and V2 exists if there is a hyperlink present on web page V1 referring to page V2.

Application-

- Webgraph is used to calculate PageRank.The PageRank is an algorithm used for measuring the importance of website pages.
- For determining the web pages of similar topics, the webgraph is used.
- Used to identify hubs and authorities of web pages.

Case Studies-

Google Maps-

It is a service developed by Google.
It offers services for satellite imagery ,street maps,360 views of streets and real time traffic conditions
Technologies Used-

- Makes use of Javascripts.
- Adobe Flash
- .JPG,.PNG,.PDG,.GIF or .BMP for floor plan.

connected components

In graph, a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.