Data Structures and Algorithms

Graphs and Minimum Spanning Tree Algorithms

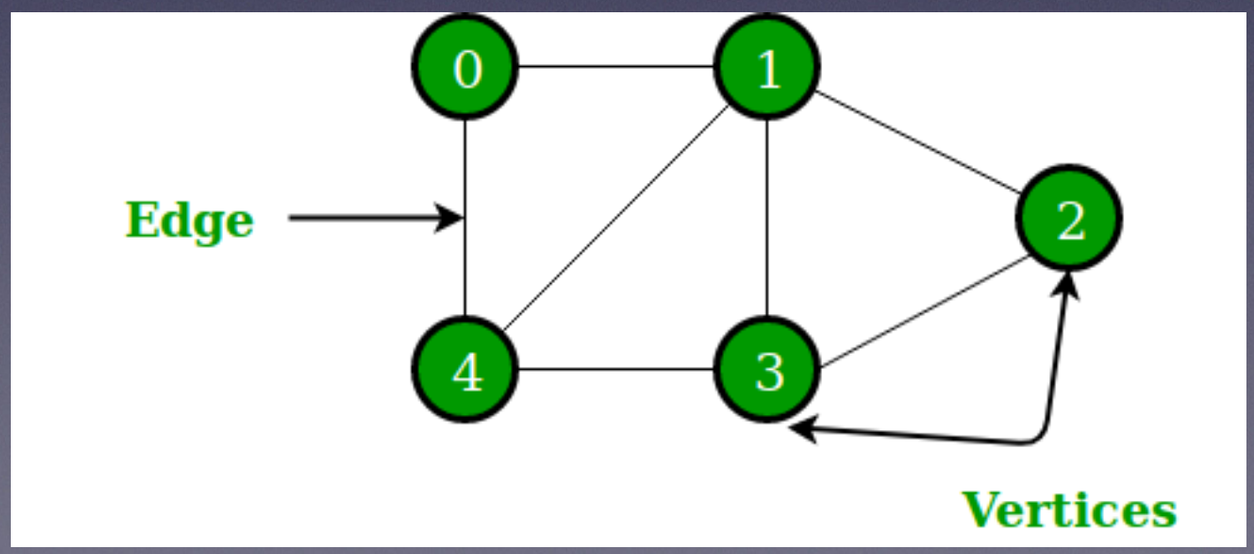
Graphs

- In mathematics, **graph theory** is the study of *graphs*, which are mathematical structures used to model pairwise relations between objects.

- A graph in this context is made up of *vertices* (also called *nodes* or *points*) which are connected by *edges* (also called *links* or *lines*).

Graphs in Computer Science

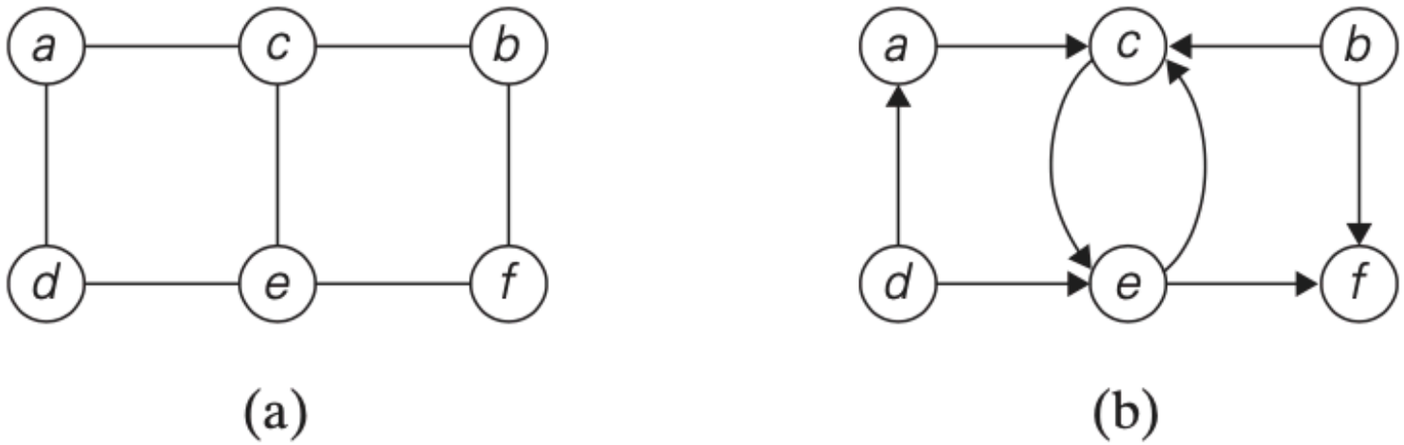
- A Graph is a non-linear data structure consisting of nodes and edges.
- The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.



Real-life Problems

- Graphs are used to represent networks.
- Paths in a city
- Telephone Network
- Social Networks (Linkedin,Facebook)

Undirected and Directed Graphs



$$V = \{a, b, c, d, e, f\},\ E = \{(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)\}.$$

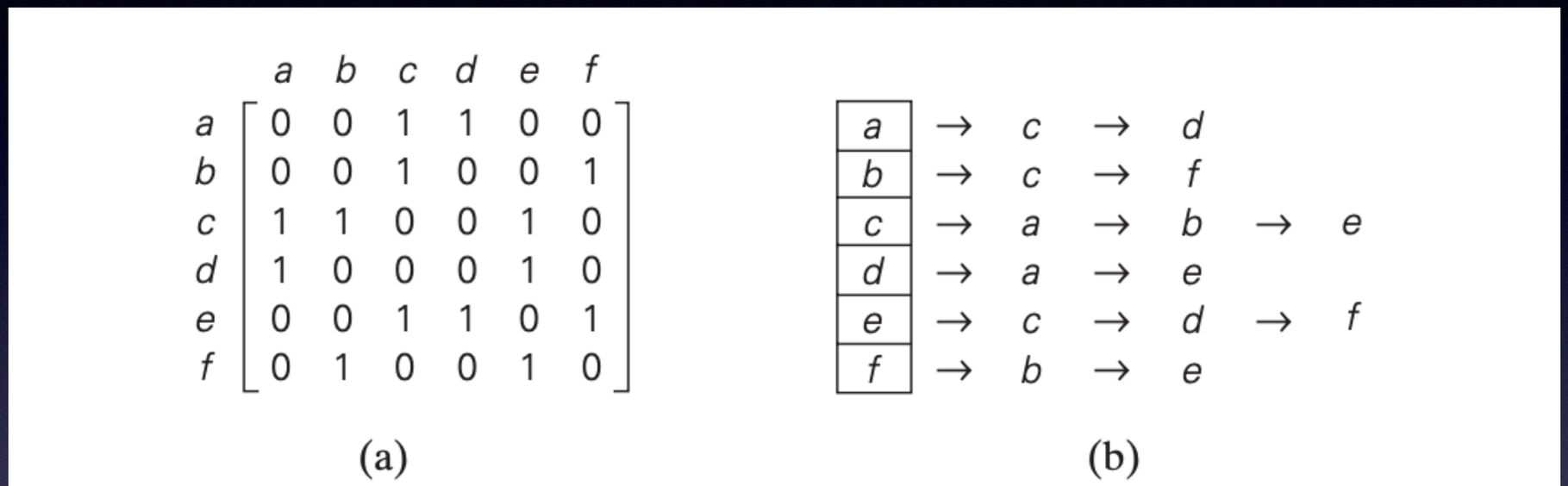
$$V = \{a, b, c, d, e, f\},\ E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}.$$

If a pair of vertices (u, v) is not the same as the pair (v, u), we say that the edge (u, v) is ***directed*** from the vertex u, called the edge's ***tail***, to the vertex v, called the edge's ***head***. We also say that the edge (u, v) leaves u and enters v. A graph whose every edge is directed is called ***directed***. Directed graphs are also called ***digraphs***.

Complete, Dense and Sparse Graphs

A graph with every pair of its vertices connected by an edge is called ***complete***. A standard notation for the complete graph with $|V|$ vertices is $K_{|V|}$. A graph with relatively few possible edges missing is called ***dense***; a graph with few edges relative to the number of its vertices is called ***sparse***. Whether we are dealing with a dense or sparse graph may influence how we choose to represent the graph and, consequently, the running time of an algorithm being designed or used.

Graph Representations



	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

(a)

a	→	c	→	d		
b	→	c	→	f		
c	→	a	→	b	→	e
d	→	a	→	e		
e	→	c	→	d	→	f
f	→	b	→	e		

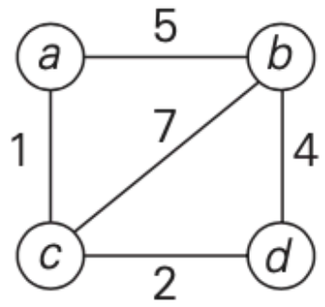
(b)

Graph Representations Graphs for computer algorithms are usually represented in one of two ways: the adjacency matrix and adjacency lists. The ***adjacency matrix*** of a graph with n vertices is an $n \times n$ boolean matrix with one row and one column for each of the graph's vertices, in which the element in the ith row and the jth column is equal to 1 if there is an edge from the ith vertex to the jth vertex, and equal to 0 if there is no such edge. For example, the adjacency matrix for the graph of Figure 1.6a is given in Figure 1.7a.

Graph Representations - II

If a graph is sparse, the adjacency list representation may use less space than the corresponding adjacency matrix despite the extra storage consumed by pointers of the linked lists; the situation is exactly opposite for dense graphs. In general, which of the two representations is more convenient depends on the nature of the problem, on the algorithm used for solving it, and, possibly, on the type of input graph (sparse or dense).

Weighted Graphs



(a)

	a	b	c	d
a	∞	5	1	∞
b	5	∞	7	4
c	1	7	∞	2
d	∞	4	2	∞

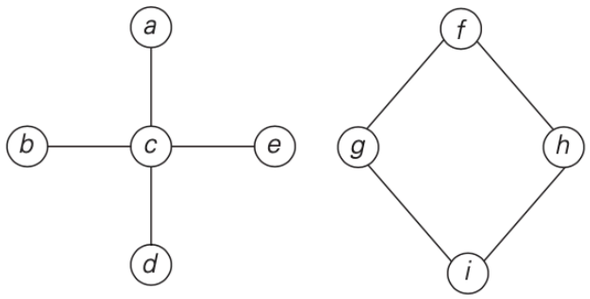
(b)

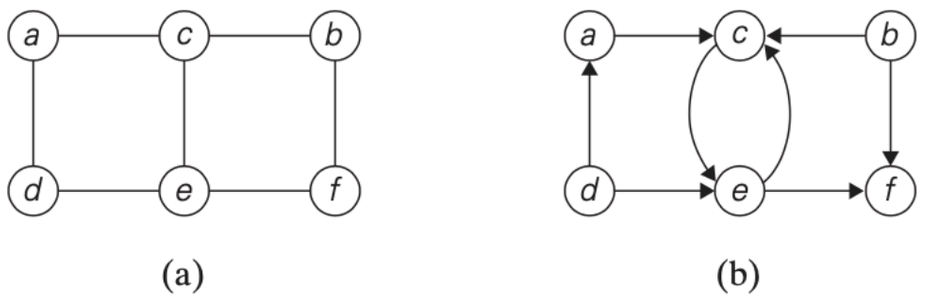
a	→ b, 5 → c, 1
b	→ a, 5 → c, 7 → d, 4
c	→ a, 1 → b, 7 → d, 2
d	→ b, 4 → c, 2

(c)

Weighted Graphs A ***weighted graph*** (or weighted digraph) is a graph (or digraph) with numbers assigned to its edges. These numbers are called ***weights*** or ***costs***. An interest in such graphs is motivated by numerous real-world applications, such as finding the shortest path between two points in a transportation or communication network or the traveling salesman problem mentioned earlier.

Paths and Cycles

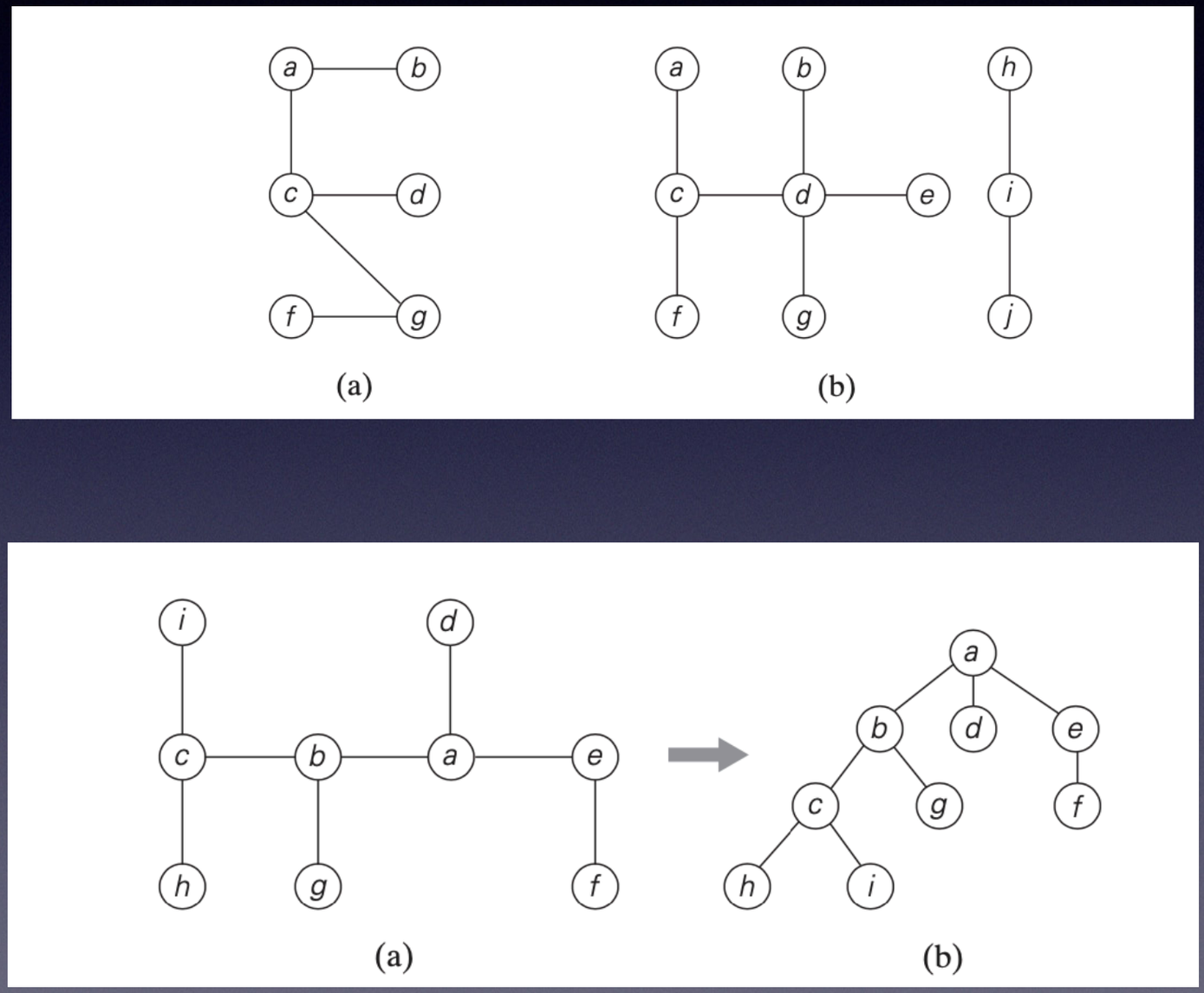




Paths and Cycles Among the many properties of graphs, two are important for a great number of applications: ***connectivity*** and ***acyclicity***. Both are based on the notion of a path. A ***path*** from vertex u to vertex v of a graph G can be defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v. If all vertices of a path are distinct, the path is said to be ***simple***. The ***length*** of a path is the total number of vertices in the vertex sequence defining the path minus 1, which is the same as the number of edges in the path. For example, a, c, b, f is a simple path of length 3 from a to f in the graph in Figure 1.6a, whereas a, c, e, c, b, f is a path (not simple) of length 5 from a to f.

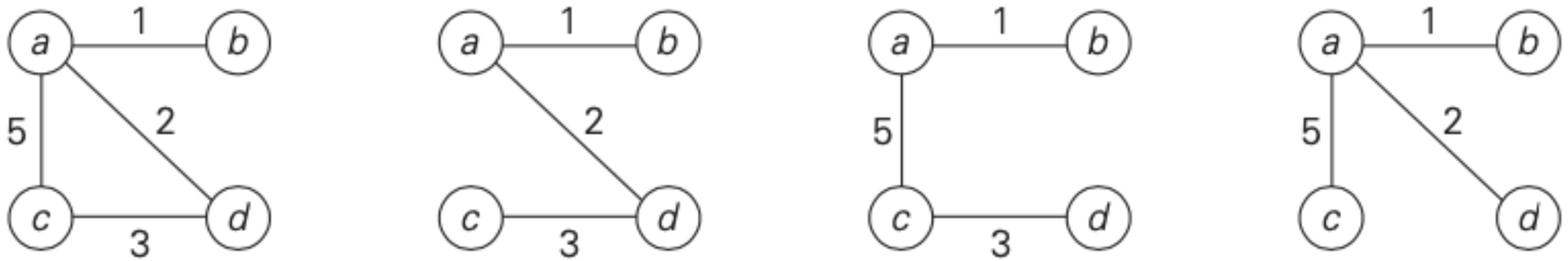
It is important to know for many applications whether or not a graph under consideration has cycles. A ***cycle*** is a path of a positive length that starts and ends at the same vertex and does not traverse the same edge more than once. For example, f, h, i, g, f is a cycle in the graph in Figure 1.9. A graph with no cycles is said to be ***acyclic***. We discuss acyclic graphs in the next subsection.

Special Graphs - Trees



A ***tree*** (more accurately, a ***free tree***) is a connected acyclic graph (Figure 1.10a). A graph that has no cycles but is not necessarily connected is called a ***forest***: each of its connected components is a tree (Figure 1.10b).

Minimum Spanning Tree



graph | $w(T_1) = 6$ | $w(T_2) = 9$ | $w(T_3) = 8$

DEFINITION A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

Greedy Algorithms

- Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.

- Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.

Greedy Algorithms

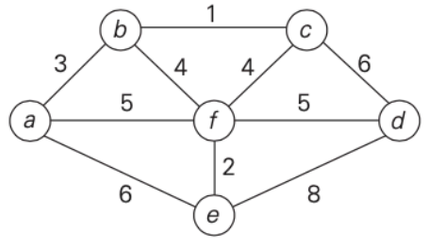
1) Kruskal's Minimum Spanning Tree (MST): In Kruskal's algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.

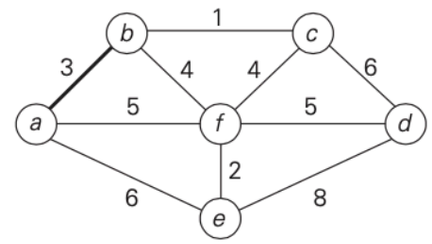
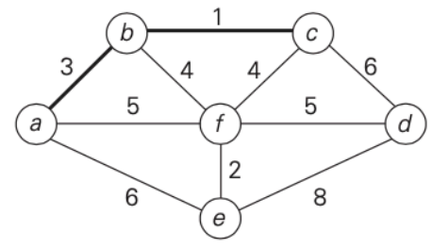
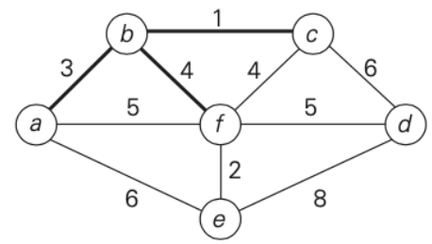
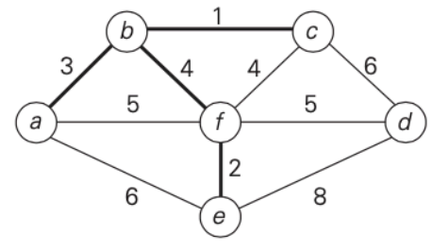
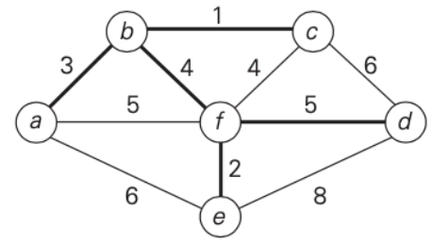
2) Prim's Minimum Spanning Tree: In Prim's algorithm also, we create a MST by picking edges one by one. We maintain two sets: a set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.

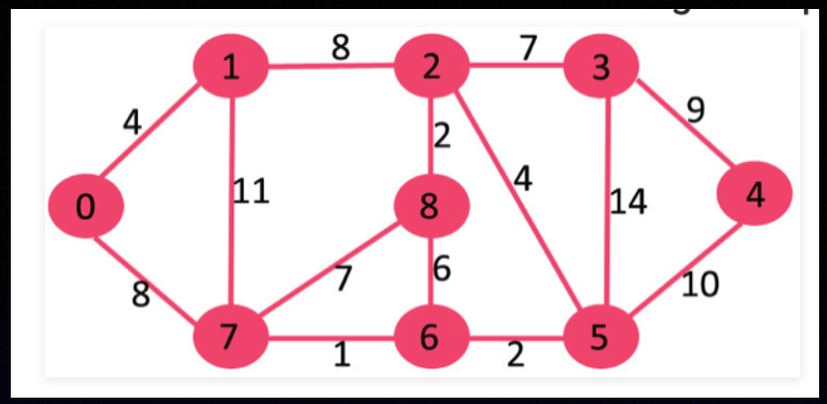
3) Dijkstra's Shortest Path: The Dijkstra's algorithm is very similar to Prim's algorithm. The shortest path tree is built up, edge by edge. We maintain two sets: a set of the vertices already included in the tree and the set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.

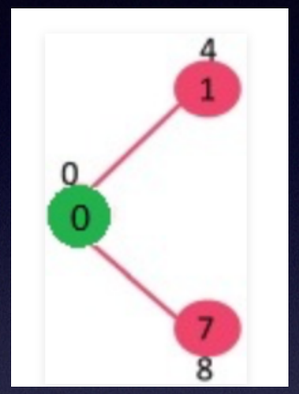
4) Huffman Coding: Huffman Coding is a loss-less compression technique. It assigns variable-length bit codes to different characters. The Greedy Choice is to assign least bit length code to the most frequent character.

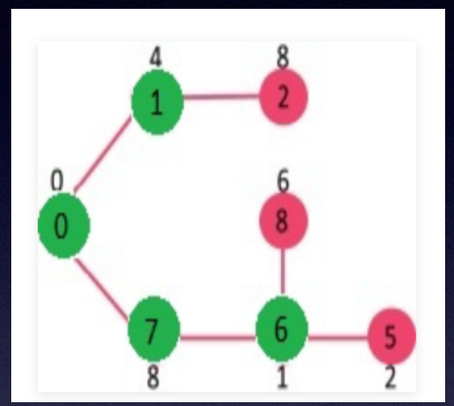
Prim's Algorithm

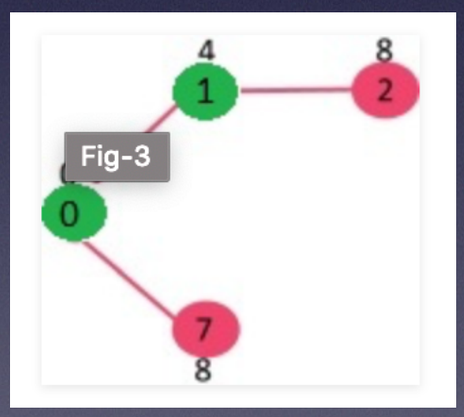


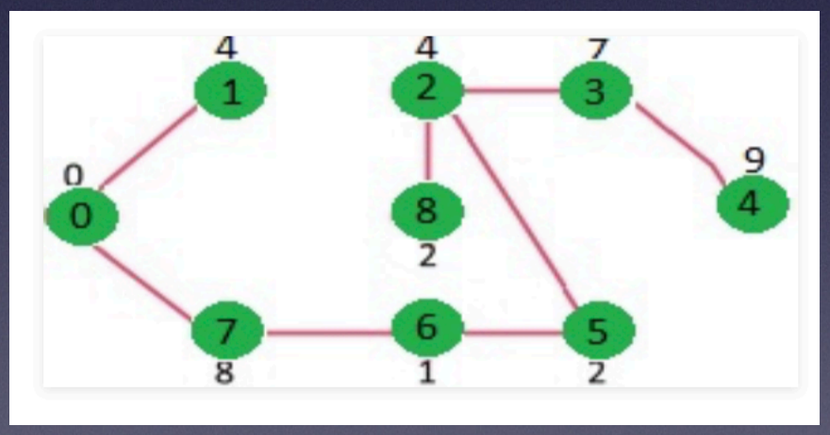
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b**(**a**, **3**) c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c**(**b**, **1**) d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f**(**b**, **4**)	
f(b, 4)	d(f, 5) **e**(**f**, **2**)	
e(f, 2)	**d**(**f**, **5**)	
d(f, 5)		

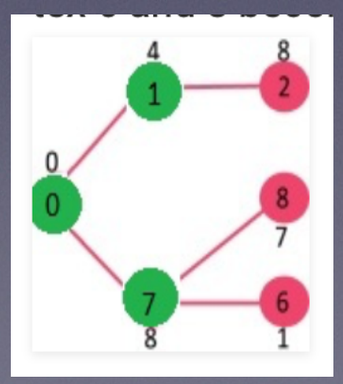

0
1
2
3
4
5
6
7
8
4
8
7
9
4
2
11
4
14
8
7
6
10
1
2


4
1
0
0
7
8


4
8
1
2
0
0
6
8
7
6
5
8
1
2


4
8
1
2
Fig-3
0
0
7
8

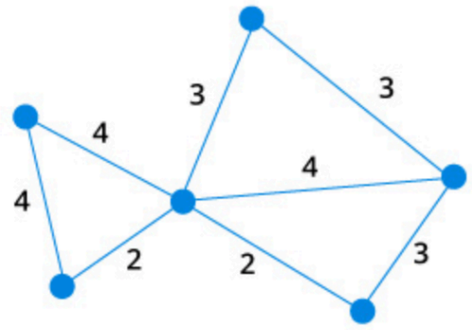

4
4
7
1
2
3
0
9
0
8
4
2
7
6
5
8
1
2


4
8
1
2
0
0
8
7
7
6
8
1

Example of Prim's algorithm

1

Start with a weighted graph



2

Choose a vertex

3

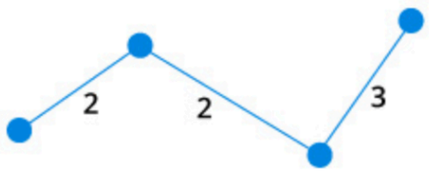
Choose the shortest edge from this vertex and add it

4

Choose the nearest vertex not yet in the solution

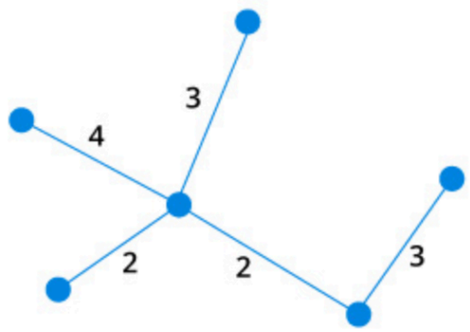
5

Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



6

Repeat until you have a spanning tree



C Code for Prim's Algorithm

Code

```c
#include<stdio.h>
#include<stdlib.h>

#define infinity 9999
#define MAX 20

int G[MAX][MAX],spanning[MAX][MAX],n;

int prims();
```

```c
int main()
{
    int i,j,total_cost;
    printf("Enter no. of vertices:");
    scanf("%d",&n);

    printf("\nEnter the adjacency matrix:\n");

    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%d",&G[i][j]);

    total_cost=prims();
    printf("\nspanning tree matrix:\n");

    for(i=0;i<n;i++)
    {
        printf("\n");
        for(j=0;j<n;j++)
            printf("%d\t",spanning[i][j]);
    }

    printf("\n\nTotal cost of spanning tree=%d",total_cost);
    return 0;
}
```

C Code for Prim's Algorithm

```c
int prims()
{
    int cost[MAX][MAX];
    int u,v,min_distance,distance[MAX],from[MAX];
    int visited[MAX],no_of_edges,i,min_cost,j;

    //create cost[][] matrix,spanning[][]
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
        {
            if(G[i][j]==0)
                cost[i][j]=infinity;
            else
                cost[i][j]=G[i][j];
                spanning[i][j]=0;
        }

    //initialise visited[],distance[] and from[]
    distance[0]=0;
    visited[0]=1;

    for(i=1;i<n;i++)
    {
        distance[i]=cost[0][i];
        from[i]=0;
        visited[i]=0;
    }

    min_cost=0;         //cost of spanning tree
    no_of_edges=n-1;        //no. of edges to be added
    while(no_of_edges>0)
    {
        //find the vertex at minimum distance from the tree
        min_distance=infinity;
        for(i=1;i<n;i++)
            if(visited[i]==0&&distance[i]<min_distance)
            {
                v=i;
                min_distance=distance[i];
            }

        u=from[v];

        //insert the edge in spanning tree
        spanning[u][v]=distance[v];
        spanning[v][u]=distance[v];
        no_of_edges--;
        visited[v]=1;

        //updated the distance[] array
        for(i=1;i<n;i++)
            if(visited[i]==0&&cost[i][v]<distance[i])
            {
                distance[i]=cost[i][v];
                from[i]=v;
            }

        min_cost=min_cost+cost[u][v];
    }

    return(min_cost);
}
```

C Code for Prim's Algorithm

Output

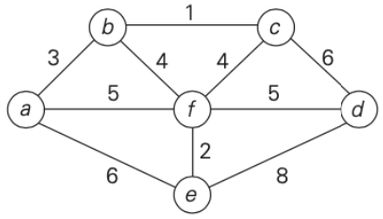
Enter no. of vertices:6

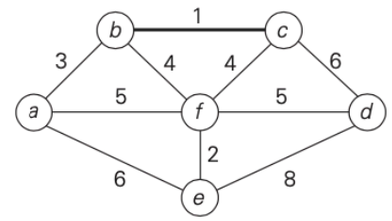
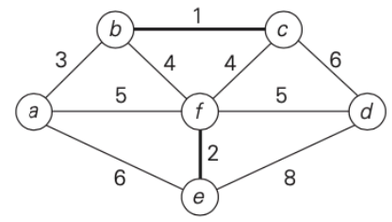
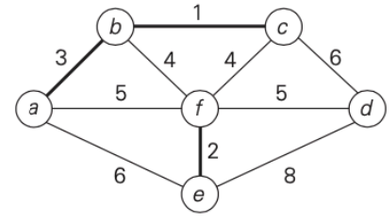
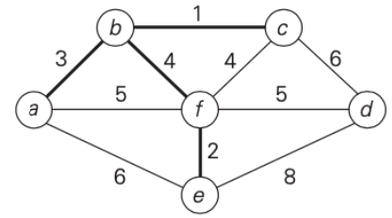
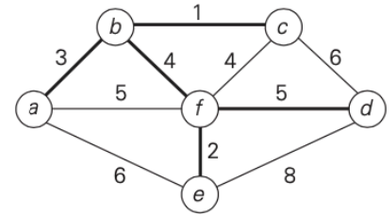
Enter the adjacency matrix:
0 3 1 6 0 0
3 0 5 0 3 0
1 5 0 5 6 4
6 0 5 0 0 2
0 3 6 0 0 6
0 0 4 2 6 0

spanning tree matrix:
0 3 1 0 0 0
3 0 0 0 3 0
1 0 0 0 0 4
0 0 0 0 0 2
0 3 0 0 0 0
0 0 4 2 0 0

Total cost of spanning tree=13

Kruskal Algorithm



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

Kruskal Algorithm

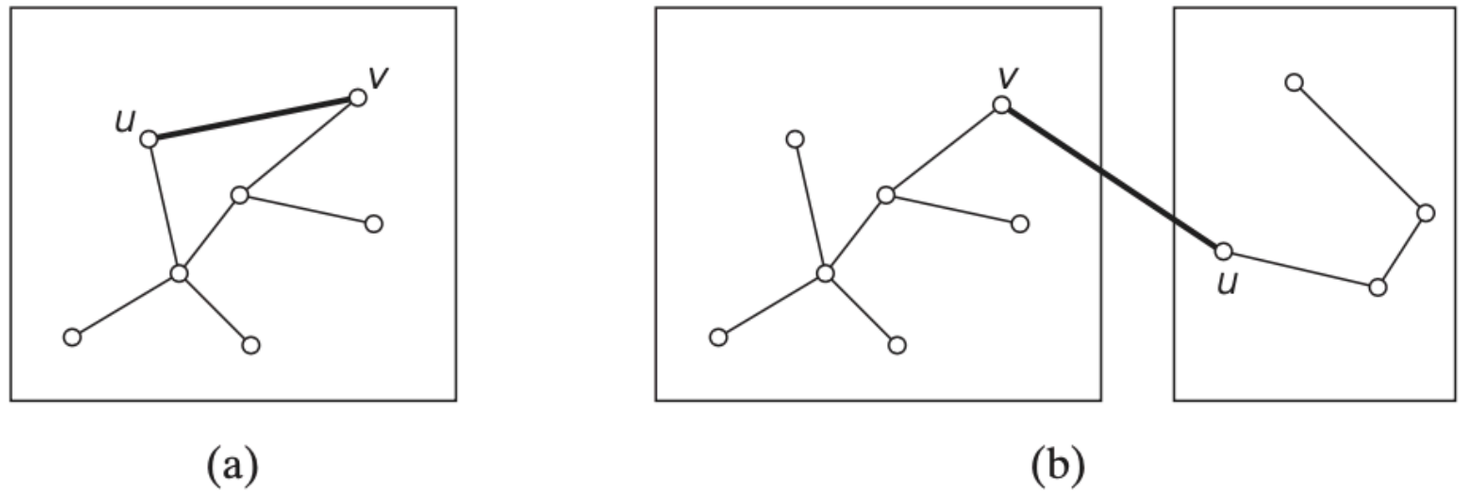


FIGURE 9.6 New edge connecting two vertices may (a) or may not (b) create a cycle.

C code for Kruskal's Algorithm

```c
#include<stdio.h>

#define MAX 30

typedef struct edge
{
    int u,v,w;
}edge;

typedef struct edgelist
{
    edge data[MAX];
    int n;
}edgelist;

edgelist elist;

int G[MAX][MAX],n;
edgelist spanlist;

void kruskal();
int find(int belongs[],int vertexno);
void union1(int belongs[],int c1,int c2);
void sort();
void print();
```

```c
void main()
{
    int i,j,total_cost;

    printf("\nEnter number of vertices:");

    scanf("%d",&n);

    printf("\nEnter the adjacency matrix:\n");

    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%d",&G[i][j]);

    kruskal();
    print();
}
```

```c
void kruskal()
{
    int belongs[MAX],i,j,cno1,cno2;
    elist.n=0;

    for(i=1;i<n;i++)
        for(j=0;j<i;j++)
        {
            if(G[i][j]!=0)
            {
                elist.data[elist.n].u=i;
                elist.data[elist.n].v=j;
                elist.data[elist.n].w=G[i][j];
                elist.n++;
            }
        }

    sort();

    for(i=0;i<n;i++)
        belongs[i]=i;

    spanlist.n=0;

    for(i=0;i<elist.n;i++)
    {
        cno1=find(belongs,elist.data[i].u);
        cno2=find(belongs,elist.data[i].v);

        if(cno1!=cno2)
        {
            spanlist.data[spanlist.n]=elist.data[i];
            spanlist.n=spanlist.n+1;
            union1(belongs,cno1,cno2);
        }
    }
}
```

```c
int find(int belongs[],int vertexno)
{
    return(belongs[vertexno]);
}
```

C code for Kruskal's Algorithm

```c
void union1(int belongs[],int c1,int c2)
{
    int i;

    for(i=0;i<n;i++)
        if(belongs[i]==c2)
            belongs[i]=c1;
}
```

```c
void sort()
{
    int i,j;
    edge temp;

    for(i=1;i<elist.n;i++)
        for(j=0;j<elist.n-1;j++)
            if(elist.data[j].w>elist.data[j+1].w)
            {
                temp=elist.data[j];
                elist.data[j]=elist.data[j+1];
                elist.data[j+1]=temp;
            }
}
```

```c
void print()
{
    int i,cost=0;

    for(i=0;i<spanlist.n;i++)
    {
        printf("\n%d\t%d\t%d",spanlist.data[i].u,spanlist.data[i].v,spanlist.data[i].w);
        cost=cost+spanlist.data[i].w;
    }

    printf("\n\nCost of the spanning tree=%d",cost);
}
```

Prim vs Kruskal

PRIM'S ALGORITHM	KRUSKAL'S ALGORITHM
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of O(V^2), V being the number of vertices.	Kruskal's algorithm's time complexity is O(logV), V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.