CHAPTER SEVEN
7. DATA STRUCTURES AND APPLICATIONS: GRAPHS

7.1. Introduction

In spite of the flexibility of trees and the many different tree applications, trees, by their nature, have one limitation, namely, they can only represent relations of a hierarchical type, such as relations between parent and child. Other relations are only represented indirectly, such as the relation of being a sibling. A generalization of a tree, a graph, is a data structure in which this limitation is lifted. Intuitively, a graph is a collection of vertices (or nodes) and the connections between them. Generally, no restriction is imposed on the number of vertices in the graph or on the number of connections one vertex can have to other vertices. Graphs are versatile data structures that can represent a large number of different situations and events from diverse domains. They provide the ultimate in data structure flexibility. They can model both real-world systems and abstract problems, so they are used in hundreds of applications.

A graph is a way of representing relationships that exist between pairs of objects. That is, a graph is a set of objects, called vertices, together with a collection of pairwise connections between them.

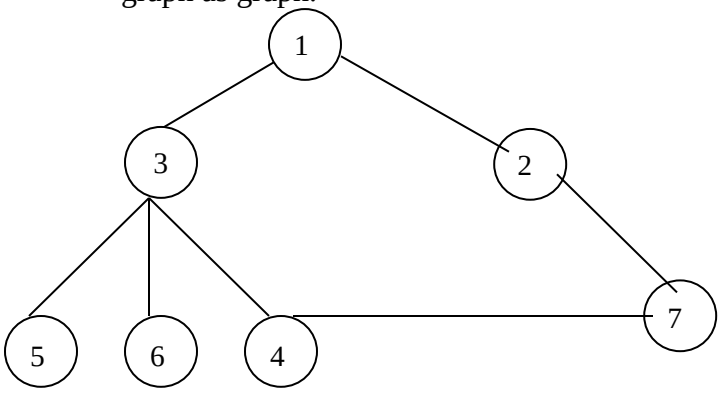
7.2. Graph Terminologies

A graph G is simply a set V of *vertices* and a collection E of pairs of vertices from V, called *edges*. Thus, a graph is a way of representing connections or relationships between pairs of objects from some set V.

That is, G = (V, E), where V is the set of **vertices** and E is the set of **edges**. Each edge is a pair (v, w), where v, w ∈ V. Vertices are sometimes called ***nodes***, and edges are sometimes referred to as ***arcs***. Sometimes an edge has a third component, called the **edge cost** (or **weight**) that measures the cost of traversing the edge.

- Edges in a graph are either **directed** or **undirected**. An edge (u, v) is said to be directed from u to v if the pair (u, v) is ordered, with *u* preceding *v*. An edge (u, v) is said to be undirected if the pair (u, v) is not ordered.
- If all the edges in a graph are undirected, then we say the graph is an **undirected graph**. Likewise, a **directed graph**, is a graph whose edges are all directed. A graph that has both directed and undirected edges is often called a **mixed graph**. In a directed graph each edge is an ordered pairs of vertices, i.e. each edge is represented by a directed pair. If e = (v, w) then *v* is tail or initial vertex and *w* is head of final vertex. Subsequently (v, w) and (w, v) represent two different edges. In an undirected graph, pair of vertices representing any edge is unordered. Thus (v,w)

and (w, v) represent the same edge. Directed graph can be referred as digraph and undirected graph as graph.



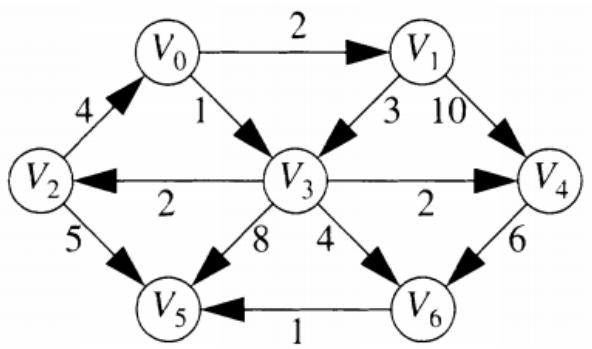


Figure 7.1: (a) Undirected and unweighted graph (b) directed and weighted graph

- ✓ For the above graph (a), the set of vertices and edges are

  V(G) = {1, 2, 3, 4, 5, 6, 7} and **E**(G) = { (1, 2), (1, 3), (2, 7), (3, 4), (5, 3), (3, 6), (7, 4) }

- ✓ For graph (b), the set of vertices and edges are

  V(G) = {V0, V1,V2, V3, V4, V5, V6} and

  E(G) = { (V0, V1, 2), (V0, V3, 1), (V1, V3, 3), (V1, V4, 10), (V2, V0, 4), (V2, V5, 5), (V3,V2, 2), (V3, V4, 2), (V3, V6, 4), (V3, V5, 8), (V4, V6, 6), (V6, V5, 1)}

- A **path** in a graph is a sequence of vertices connected by edges. In other words, it is the sequence of vertices w_1 , w_2, ..., w_N such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < N$. The **path length** is the number of edges on the path; namely N – 1; also called the **unweighted path length**. The **weighted path length** is the sum of the costs of the edges on the path. For example for figure 7.1 (b); V0, V3, V5 is a path from vertex V0 to V5. The path length is two edges, the shortest path between V0 and V5, and the weighted path length is 9. However, if cost is important, the weighted shortest path between these vertices has cost 6 and is V0, V3, V6, V5. A path $w_1$may exist from a vertex to itself. If this path contains no edges, the path length is 0, which is a convenient way to define an otherwise special case. A **simple path** is a path in which all vertices are distinct, except that the first and last vertices can be the same.
- Two vertices *u* and *v* said to be **adjacent** if they are joined by an edge, that is, if there is an edge whose end vertices are u and v. Such vertices are also called **neighbors**. Such an edge is said to be **incident** on vertices *u* and *v*.

For example for the above graph (a) at figure 7.1, adjacent vertices for vertex 3 are 1, 4, 5 and 6. For graph (b), V1, V3 and V6 are adjacent vertices for vertex V4.

- A **cycle** in a directed graph is a path that begins and ends at the same vertex and contains at least one edge. That is, it has a length of at least 1 such that $w_1 = w_N$; this cycle is simple if the path is simple. For undirected graphs, we require that the edges be distinct. The logic of these requirements is that the path u, v, u in an undirected graph should not be considered a cycle, because (u, v) and (v, u) are the same edge. In a directed graph, these are different edges, so it makes sense to call this a cycle. A **directed acyclic graph (DAG)** is a type of directed graph having no cycles.
- A graph with n vertices is called **complete** and is denoted K_n if for each pair of distinct vertices there is exactly one edge connecting them; that is, each vertex can be connected to any other vertex.
- An undirected graph is **connected** if there exists a path from every vertex to every other vertex. A directed graph with this property is called **strongly connected**. If a directed graph is not strongly connected, but the underlying graph (without direction to the arcs) is connected, then the graph is said to be **weakly connected**.

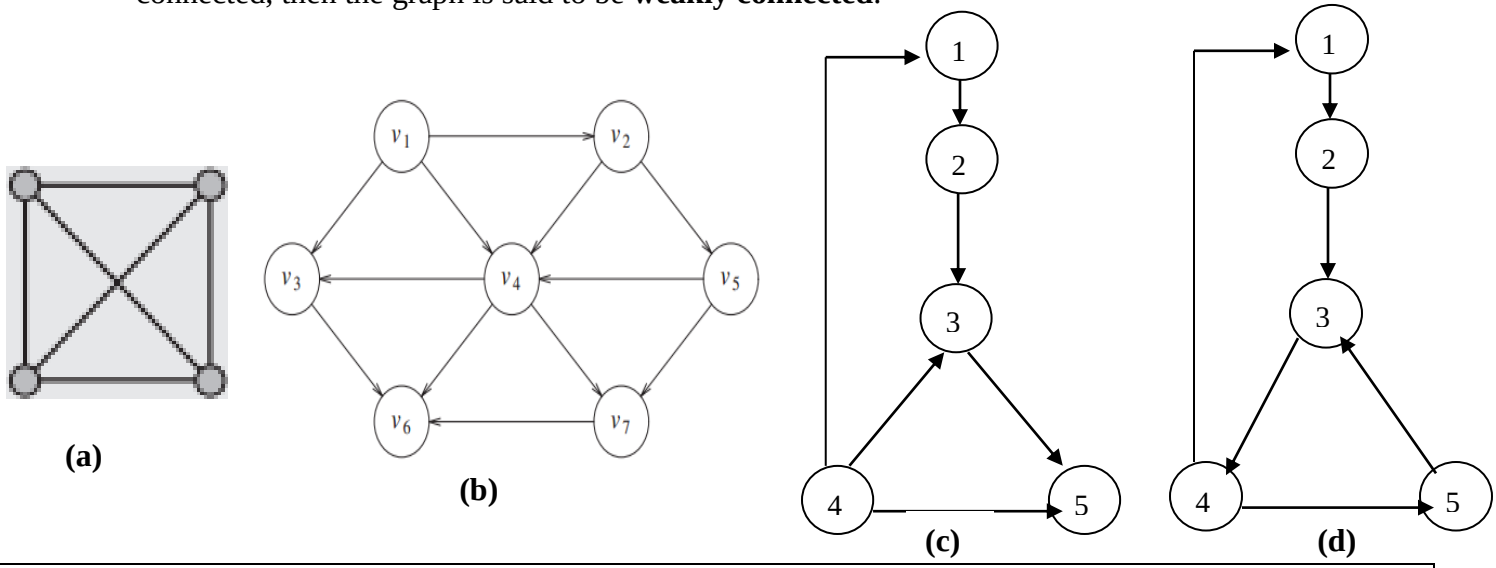


Figure 7.2: (a) complete graph, (b) acyclic directed graph, (c) weakly connected graph, (d) strongly connected graph

From this figure we observe that, in (a), there exists a path from each vertex to every other vertices, so it is called complete graph. In (b), there is no cycle formed in the graph, and hence it is directed acyclic graph. In (c) there does not exist a directed path from vertex 1 to vertex 4 and also from vertex 5 to other vertices and so on. Therefore it is a weakly connected graph. In (d) there is a directed path from any vertex to any other vertex, hence it is strongly connected graph.

- The two vertices joined by an edge are called the **end vertices** (or endpoints) of the edge. If an edge is directed, its first endpoint is its **origin** and the other is the **destination** of the edge. The **outgoing edges** of a vertex are the directed edges whose origin is that vertex. The **incoming edges** of a vertex are the directed edges whose destination is that vertex. The **degree** of a vertex v, denoted deg(v), is the number of incident edges of v. The **in-degree** and **out-degree** of a vertex v are the number of the incoming and outgoing edges of v, and are denoted indeg(v) and outdeg(v), respectively. Thus, the indegree of a vertex v is the number of edges ending at *v* and the outdegree of a vertex v is the number of edges starting from v.
- A vertex *v* is said to be **source** if outdegree(v) > 0 and indegree(v) = 0. A vertex *v* is called a **sink** if indegree(v) > 0 and outdegree(v) = 0.

  For example, from figure 7.1 (a), the degree of vertex 3 is 4. And from figure 7.2 (d) the indegree of vertex 4 is 1 and outdegree of vertex 4 is 2. From figure 7.2 (c), vertex 4 is a source vertex and vertex 5 is a sink vertex.

7.3. Graph Representations

The picture of graph with circles (vertices) and lines (edges) is *only a depiction*. Graph is a mathematical structure and finds its application in many areas of interest in which problems need to be solved using computers. Thus this mathematical structure must be represented as some kind of data structures.

There are two commonly used methods for representing graphs, ***adjacency matrix*** and ***adjacency list.*** The adjacency list structure only store the edges actually present in the graph, while the adjacency matrix stores a placeholder for every pair of vertices (whether there is an edge between them or not). The choice of representation depends on the application and function to be performed on the graph.

i) Adjacency Matrix: Representing a graph by a matrix

An adjacency matrix is a way of representing an n vertex graph G = (V; E) by an n X n matrix, whose entries are boolean values. In the adjacency matrix representation, we think of the vertices as being the integers in the set {0, 1, ... , n − 1} and the edges as being pairs of such integers. This allows us to store references to edges in the cells of a two-dimensional n × n array.

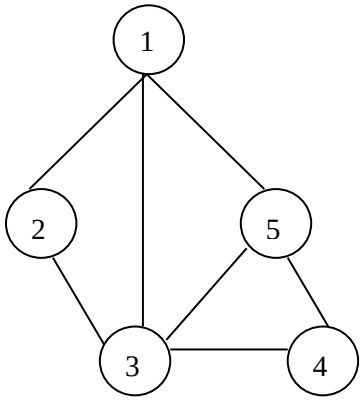
The adjacency matrix for a graph of total vertices |V| is a |V| x |V| array. Assume that |V| = n and that the vertices are labeled from v_0 through v_{n-1}. Row *i* of the adjacency matrix contains entries

for Vertex v_i. Column j in row i is marked if there is an edge from v_i to v_j and is not marked otherwise. Thus, the adjacency matrix requires one bit at each position. Alternatively, if we wish to associate a number with each edge, such as the weight or distance between two vertices, then each matrix position must store that number.

An adjacency matrix of graph G= (V, E) is a binary |V|× |V| matrix such that each entry of this matrix

$$A_{ij} = \begin{cases} 1 & \text{if there exists an edge } (v_i\ v_j) \\ 0 & \text{otherwise} \end{cases}$$

For example, the following undirected graph is represented using the adjacency matrix as follows.

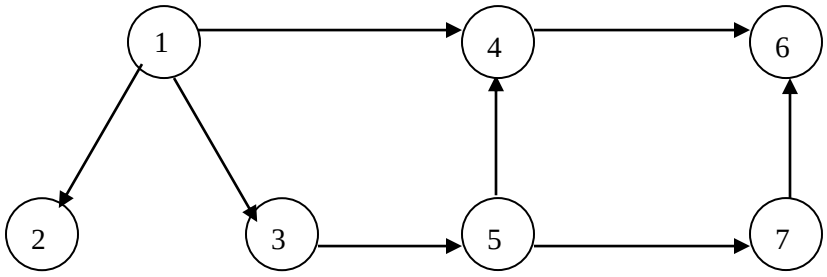


Vertices	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	0	0
3	1	1	0	1	1
4	0	0	1	0	1
5	1	0	1	1	0

Figure 7.3: Adjacency matrix of graph

You may observe that the adjacency matrix for an undirected graph is symmetric, as the lower and upper triangles are same. Also all the diagonal elements are zero.

Consider the following directed graph



Its adjacency matrix looks like the following.

Vertices	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	1	0	0
4	0	0	0	0	0	1	0
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

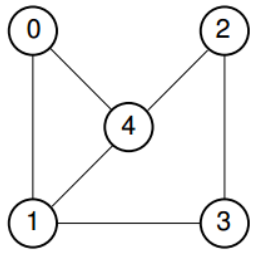
The total number of 1's account for the number of edges in the digraph. The number of 1's in each row tells the *out degree* of the corresponding vertex. The total number of 1's in each column tells the *in degree* of the corresponding vertex.

ii) Adjacency List: A graph as a collection of lists

Adjacency list representations of graphs take a more vertex-centric approach. In an adjacency list representation, the graph G = (V; E) is represented as an array, *adj*, of lists. The list *adj[i]* contains a list of all the vertices adjacent to vertex *i*. That is, it contains every index *j* such that (*i*,j) ∈ E. The adjacency list is an array of linked lists. The array is |V| items long, with position i storing a pointer to the linked list of edges for Vertex v_i. This linked list represents the edges by the vertices that are adjacent to Vertex v_i.

Adjacency list specifies all vertices adjacent to each vertex of the graph. This list can be implemented as a table, in which case it is called a *star* representation, which can be forward or reverse, as illustrated in Figure 7.4 a, or as a *linked list* (Figure 7.4b).

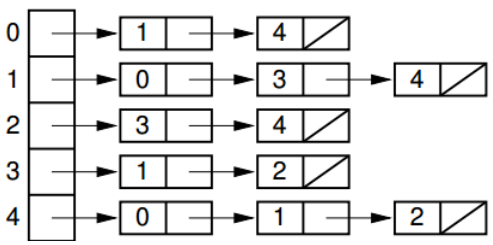
For example, for the following undirected graph, its adjacency list in star and linked list representation is given below.



(a)

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

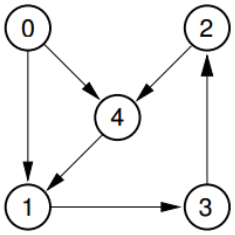
(b)



(c)

Figure 7.4: (a) Undirected graph. Adjacency list of graph (a) in (b) star representation (c) Linked list representation

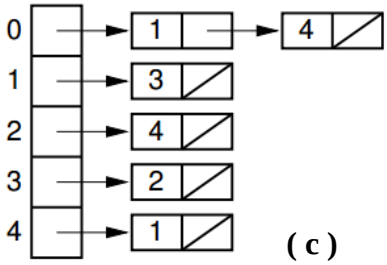
For the following directed graph, its adjacency list in star and linked list representation is given below.



(a)

0	1 4
1	3
2	4
3	2
4	1

(b)



(c)

Figure 7.5: (a) directed graph. Adjacency list of graph (a) in (b) star representation (c) Linked list representation

The entry for Vertex 0 in Figure 7.5(c) stores 1 and 4 because there are two edges in the graph leaving Vertex 0, with one going Vertex 1 and one going to Vertex 4. The list for Vertex 2 stores an entry for Vertex 4 because there is an edge from Vertex 2 to Vertex 4, but no entry for Vertex 3 because this edge comes into Vertex 2 rather than going out.

Both the adjacency matrix and the adjacency list can be used to store directed or undirected graphs. Each edge of an undirected graph connecting Vertices U and V is represented by two directed edges: one from U to V and one from V to U. Which graph representation is more space efficient depends on the number of edges in the graph. The adjacency list stores information only for those edges that actually appear in the graph, while the adjacency matrix requires space for each potential edge, whether it exists or not. However, the adjacency matrix requires no overhead for pointers, which can be a substantial cost, especially if the only information stored for an edge is one bit to indicate its existence. As the graph becomes denser, the adjacency matrix becomes relatively more space efficient. Sparse graphs are likely to have their adjacency list representation be more space efficient.

7.4. Graph Traversal

Often it is useful to visit the vertices of a graph in some specific order based on the graph's topology. This is known as a graph traversal and is similar in concept to a tree traversal. Recall that tree traversals visit every node exactly once, in some specified order such as preorder, inorder, or postorder. Multiple tree traversals exist because various applications require the nodes to be visited in a particular order. For example, to print a binary search tree's nodes in ascending order requires an inorder traversal as opposed to some other traversal. Standard graph traversal orders also exist. Each is appropriate for solving certain problems. For example, many problems in artificial intelligence programming are modeled using graphs. The problem domain may consist of a large collection of states, with connections between various pairs of states. Solving

the problem may require getting from a specified start state to a specified goal state by moving between states only through the connections. Typically, the start and goal states are not directly connected. To solve this problem, the vertices of the graph must be searched in some organized manner.

Graph traversal algorithms typically begin with a start vertex and attempt to visit the remaining vertices from there. Graph traversals must deal with a number of troublesome cases. First, it may not be possible to reach all vertices from the start vertex. This occurs when the graph is not connected. Second, the graph may contain cycles, and we must make sure that cycles do not cause the algorithm to go into an infinite loop.

Graph traversal algorithms can solve both of these problems by maintaining a mark bit for each vertex on the graph. At the beginning of the algorithm, the mark bit for all vertices is cleared. The mark bit for a vertex is set when the vertex is first visited during the traversal. If a marked vertex is encountered during traversal, it is not visited a second time. This keeps the program from going into an infinite loop when it encounters a cycle.

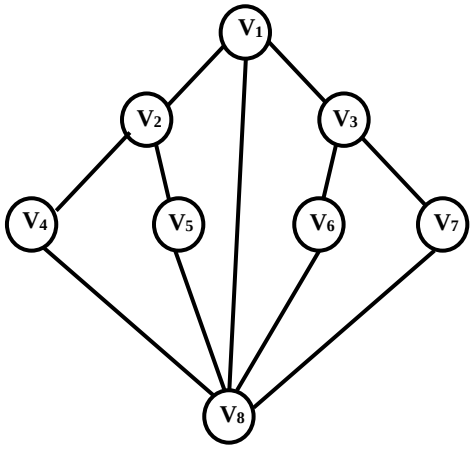
Once the traversal algorithm completes, we can check to see if all vertices have been processed by checking the mark bit array. If not all vertices are marked, we can continue the traversal from another unmarked vertex. Note that this process works regardless of whether the graph is directed or undirected.

In graphs, we do not have any start vertex or any special vertex signaled out to start traversal from. Therefore the traversal may start from any arbitrary vertex. Two graph traversal methods are commonly used. These are Depth First Search (DFS) and Breadth First Search (BFS).

i) Depth – First – Search (DFS) traversal

In this algorithm, each vertex *v* is visited and then each unvisited vertex adjacent to *v* is visited. If a vertex *v* has no adjacent vertices or all of its adjacent vertices have been visited, we backtrack to the predecessor of *v*. The traversal is finished if this visiting and backtracking process leads to the first vertex where the traversal started. If there are still some unvisited vertices in the graph, the traversal continues restarting for one of the unvisited vertices.

For example, consider the following undirected graph. Traversing its node using depth – first – search algorithm looks like as follows.



1) Let us start visiting from vertex V_1.
2) Adjacent vertices for vertex V_1 are V_2, V_8, and V_3.
   Let us take vertex V_2, and visit it.
3) Adjacent vertices for vertex V_2 are V_1, V_4, V_5.
   V_1 is already visited (at step 1). Let us take vertex V_4, and visit it.
4) Adjacent vertices for vertex V_4 are V_2, V_8.
   V_2 is already visited (at step 2). Let us take vertex V_8, and visit it.
5) Adjacent vertices for vertex V_8 are V_4, V_5, V_1, V_6, V_7.
   V_4 (at step 3) and V_1 are already visited. Let us take vertex V_5, and visit it.
6) Adjacent vertices for vertex V_5 are V_2, V_8.
   Both V_2 and V_8 (at step 4) are already visited. Therefore we back track.
7) We have V_6 and V_7 (at step 5) unvisited in the list of V_8. We may visit any. We visit V_6.
8) Adjacent vertices for vertex V_6 are V_8 and V_3.
   Obviously the choice is V_3.
9) Adjacent vertices for vertex V_3 are V_1, V_6 and V_7.
   We visit V_7.

All the adjacent vertices of V_7 are already visited, we backtrack and find that we have visited all the vertices. Therefore the sequence of traversal is

V_1, V_2, V_4, V_8, V_5, V_6, V_3, V_7.

We may implement the depth first search method by using a stack, pushing all unvisited vertices adjacent to the one just visited and popping the stack to find the next vertex to visit.

The algorithm for depth – first – search traversal in iterative and recursive format is stated in the following manner.

Iterative version:

During the execution of these algorithms, each node will be in one of the following states, called the status of node n.

Status = 1 (Ready state):- Initial state of the node 'n'.

Status = 2 (Waiting state):- 'n' is on stack.

Status = 3(Processed state):- Node 'n' has been processed.

Step1: Initialize all nodes to the ready state i.e. Status =1.

Step 2: Push the starting node onto stack and change its status = 2.

Step 3: Repeat while (stack ! = empty)

a) Pop the top node 'n' of the stack. Process 'n' and change its status to the processed state (i.e, status = 3)

b) Push onto stack all the neighbors of 'n' that are still in the ready state and change their status to the waiting state (i.e. status = 2).

Recursive version:

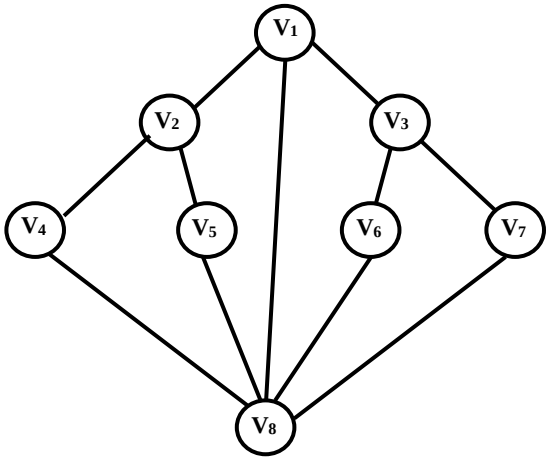
```
dfs (vertex V)
  {
     visited [V] = true;
     for each w adjacent to V
        if (!visited [w])
            dfs(w);
  }
```

ii) Breadth – First – Search (BFS) traversal

The second graph traversal algorithm is known as a breadth-first search (BFS). BFS examines all vertices connected to the start vertex before visiting vertices further away. BFS is implemented similarly to DFS, except that a queue replaces the recursion stack. Note that if the graph is a tree and the start vertex is at the root, BFS is equivalent to visiting vertices level by level from top to bottom.

In DFS we pick on one of the adjacent vertices; visit all of the adjacent vertices and back track to visit the unvisited adjacent vertices. In BFS we first visit all the adjacent vertices of the start vertex and then visit all the unvisited vertices adjacent to these and so on.

Consider the following graph, and let us see how BFS traversal algorithm looks like.



1) We start the traversal from vertex V_1. Its adjacent vertices are V_2, V_8, V_3. We visit all vertices one by one.
2) We take one of the adjacent vertex of V_1, let us take vertex V_2. The unvisited adjacent vertices to V_2 are V_4 and V_5. We visit both.
3) We go back to the remaining unvisited adjacent vertices of V_1 and pick on one of those say V_3. The unvisited adjacent vertices of V_3 are V_6 and V_7.
4) There are no more unvisited adjacent vertices of V_8, V_4, V_5, V_6 and V_7. Thus the sequence so generated is **V_1 V_2 V_8 V_3 V_4 V_5 V_6 V_7**

Here we need a queue instead of a stack to implement it. We add unvisited vertices adjacent to the one just visited, at the rear and read at front to find the next vertex to visit.

Abstract view of BFS algorithm:

Step1: Initialize all nodes to the ready state i.e. Status = 1.

Step 2: Put the starting node onto queue and change its status = 2.

Step 3: Repeat while (queue! = empty)

a) Remove the front node 'n' of the queue. Process 'n' and change its status to the processed state (i.e. Status = 3)
b) Add to the rear of the queue all the neighbors of 'n' that are still in the ready state and change their status to the waiting state (i.e. status = 2).

Detailed view of BFS algorithm:

```
bfs (vertex v)
{   vertex w, queue q;
    visited [v] = true;
    initialise (q);
    addqueue (q, v)
    while (! Emptyqueue ( q ) )
    {   deletequeue (q, v);
        for all vertices w adjacent to v
            if ( ! visited [w] )
            {  addqueue (q, w);
               visited[w] = true;
            }
     }
}
```

7.5. Application of graphs

An example of a real-life situation that can be modeled by a graph is the airport system. Each airport is a vertex, and two vertices are connected by an edge if there is a nonstop flight from the airports that are represented by the vertices. The edge could have a weight, representing the time, distance, or cost of the flight. It is reasonable to assume that such a graph is directed, since it might take longer or cost more (depending on local taxes, for example) to fly in different directions. We would probably like to make sure that the airport system is strongly connected, so that it is always possible to fly from any airport to any other airport. We might also like to quickly determine the best flight between any two airports. "Best" could mean the path with the fewest number of edges or could be taken with respect to one, or all, of the weight measures.

Traffic flow can be modeled by a graph. Each street intersection represents a vertex, and each street is an edge. The edge costs could represent, among other things, a speed limit or a capacity (number of lanes). We could then ask for the shortest route or use this information to find the most likely location for bottlenecks.

Here is a small sampling of the range of problems that graphs are routinely applied to.

- Modeling connectivity in computer and communications networks.
- Representing a map as a set of locations with distances between locations; used to compute shortest routes between locations.
- Modeling flow capacities in transportation networks.
- Finding a path from a starting condition to a goal condition; for example, in artificial intelligence problem solving.
- Modeling computer algorithms, showing transitions from one program state to another.
- Finding an acceptable order for finishing subtasks in a complex activity, such as constructing large buildings.
- Modeling relationships such as family trees, business or military organizations, and scientific taxonomies.

7.6. Shortest Path problems

Finding the shortest path is a classical problem in graph theory, and a large number of different solutions have been proposed. Edges are assigned certain weights representing, for example, distances between cities, times separating the execution of certain tasks, costs of transmitting information between locations, amounts of some substance transported from one place to

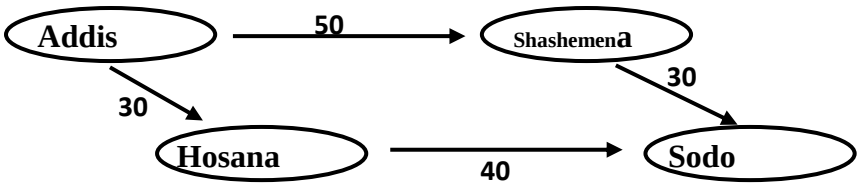
another, and so on. When determining the shortest path from vertex v to vertex u, information about distances between intermediate vertices w has to be recorded. This information can be recorded as a label associated with these vertices, where the label is only the distance from v to w or the distance along with the predecessor of w in this path. The methods of finding the shortest path rely on these labels.

Let G be a weighted graph. The length (or weight) of a path is the sum of the weights of the edges of P. That is, if P = ((v0, v1), (v1,v2), ..., (vk−1,vk)), then the length of P, denoted w(P), is defined as

$$w(P) = \sum_{i=0}^{k-1} w((v_i, v_{i+1})).$$

The distance from a vertex v to a vertex U in G, denoted d(v, U), is the length of a minimum length path (also called shortest path)from v to u, if such a path exists.

For example, in the graph we can go from Addis Ababa to Sodo through Shashemena at a cost of 80 or through Hosana at a cost of 70. (These numbers may reflect the bus ticket in birr.) In these and many other applications, we are often required to find a shortest path. i.e., a path having minimum weight between two vertices.



Length of the path is the sum of weights of the edges on that path. The starting vertex of the path is called the source vertex and the last vertex of the path is called the destination vertex.

Single source shortest path problem

Given Vertex S in Graph G, find a shortest path from S to every other vertex in G. We might want only the shortest path between two vertices, S and T. How-ever in the worst case, while finding the shortest path from S to T, we might find the shortest paths from S to every other vertex as well. So there is no better algorithm (in the worst case) for finding the shortest path to a single vertex than to find shortest paths to all vertices.