Chapter 11: Graphs

What are we studying in this chapter?

- Definitions
- Terminologies
- Matrix and Adjacency List Representation of Graphs
- Elementary Graph operations
- Traversal methods:
  - Breadth First Search
  - Depth First Search

11.1 Introduction

In this chapter, let us concentrate another important and non-linear data structure called graph. In this chapter, we discuss basic terminologies and definitions, how to represent graphs and how graphs can be traversed.

11.2 Graph Theory terminology

First, let us see "What is a vertex?"

Definition: A vertex is a synonym for a node. A vertex is normally represented by a circle. For example, consider the following figure:

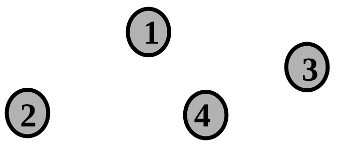


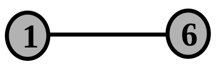
Fig Vertices

In the above figure, there are four nodes identified by 1, 2, 3, 4. They are also called vertices and normally denoted by a set V = {1, 2, 3, 4}.

Now, let us see "What is an edge?"

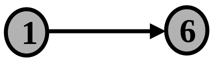
Definition: If *u* and *v* are vertices, then an *arc* or a *line* joining two vertices *u* and *v* is called an *edge*.

Example 1: Consider the figure:



Observe the following points from above figure:

- *There is no direction* for the edge between vertex 1 and vertex 6 and hence it is undirected edge.
- The undirected edge is denoted by an ordered pair (1, 6) where 1 and 6 are called *end points of the edge* (1, 6). In general, if e = (u, v), then the nodes u and v are called *end points of directed edge.*
- In this graph, edge (1, 6) is same as edge (6, 1) since there is no direction associated with that edge. So, (u, v) and (v, u) represent same edge.

Example 2: consider the figure: 

Observe the following points from above figure:

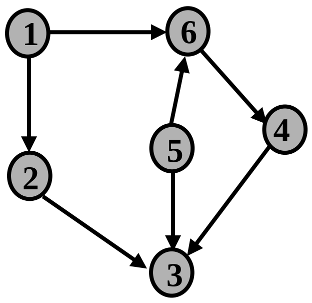
- *There is a direction* for the edge originating at vertex 1 (called tail of the edge) and heading towards vertex 6 (called head of the edge) and hence it is called directed edge.
- The directed edge is denoted by the directed pair <1, 6> where 1 is called *tail of the edge* and 6 is the *head of the edge*. So, the directed pair <1, 6> is not same as directed pair <6, 1>.
- In general, if a directed edge is represented by directed pair <u, v>, *u* is called the *tail of the edge* and *v* is the *head of the edge*. So, the directed pair <u, v> is different from the directed pair <v, u>. So, <u, v> and <v, u> represent two different edges.

Now, let us see "What is a graph?"

Definition: Formally, a graph **G** is defined as a pair of two sets V and E denoted by

G = (V, E)

where V is set of vertices and E is set of edges. For example, consider the graph shown below:



Here, graph G = (V, E) where

- V = {1, 2, 3, 4, 5, 6} is set of vertices
- E = { <1, 6>, <1, 2>, <2, 3>, <4, 3>, <5, 3>, <5, 6>, <6, 4> } is set of directed edges

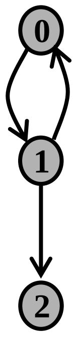
Note:

- |V| = |{1, 2, 3, 4, 5, 6}| = 6 represent the number of vertices in the graph.

- |E| = |{<1, 6>, <1, 2>, <2, 3>, <4, 3>, <5, 3>, <5, 6>, <6, 4> }| = 7 represent the number of edges in the graph.

Now, let us see "What is a directed graph? What is an undirected graph?"

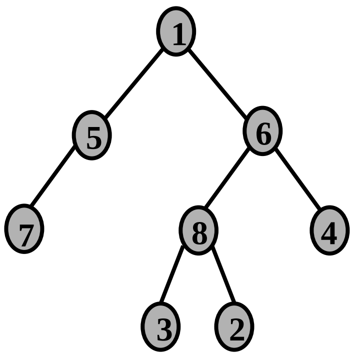
Definition: A graph G = (V, E) in which every edge is directed is called a directed graph. The directed graph is also called digraph. A graph G = (V, E) in which every edge is undirected is called an undirected graph. Consider the following graphs:



Here, graph G = (V, E) where

- V = {0, 1, 2} is set of vertices
- E = {<0, 1>, <1, 0>, <1, 2>} is set of edges

Note: Since all edges are directed it is a directed graph. In directed graph we use angular brackets < and > to represent an edge



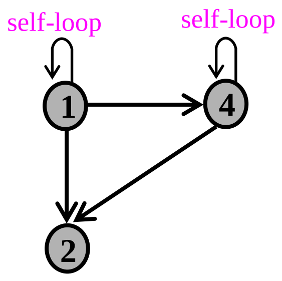
Here, graph G = (V, E) where

- V = {1, 2, 3, 4, 5, 6, 7, 8} is set of vertices
- E = {(1, 5), (1, 6), (5, 7), (6, 8), (6, 4), (8, 3), (8, 2)} is set of edges

Note: Since all edges are undirected, it is an undirected graph. In undirected graph we use parentheses (and) to represent an edge (u, v).

Now let us see "What is a self-loop (or self-edge)?

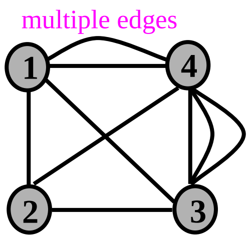
Definition: A loop is an edge which starts and ends on the same vertex. A loop is represented by an ordered pair (i, i). This indicates that the edge originates and ends in the same vertex. A loop is also called self-edge or self-loop. In the given graph shown below, there are two self-loops namely, <1, 1> and <4, 4>.



Now, let us see "What is a multigraph?"

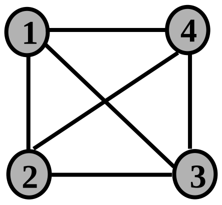
11.4 Graphs

Definition: A graph with multiple occurrence of the same edge between any two vertices is called multigraph. Here, there are two edges between the nodes 1 and 4 and there are three edges between the nodes 4 and 3.

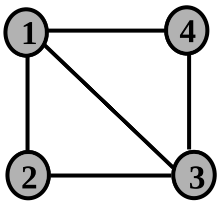


Now, let us see "What is a complete graph?"

Definition: A graph G = (V, E) is said to be a complete graph, if there exists an edge between every pair of vertices. The graph (a) below is complete. Observe that in a complete graph of *n* vertices, there will be n(n-1)/2 edges. Substituting *n* = 4, we get 6 edges. Even if one edge is removed as shown in graph (b) below, it is not complete graph.



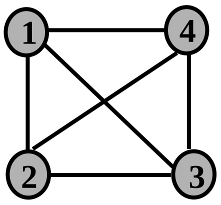
Complete graph



Not a complete graph

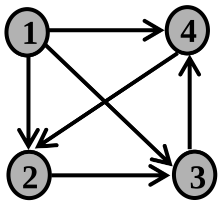
Now, let us see "What is a path?"

Definition: Let G = (V, E) be a graph. A *path* from vertex *u* to vertex *v* in an undirected graph is a sequence of adjacent vertices (u, v_0, v_1, v_2,….v_k, v) such that: (u, v_0), (v_0, v_1),….(v_k, v) are the edges in G. Consider the following graph:



In the graph, the path from vertex 1 to 4 is denoted by: 1, 2, 3, 4 which can also be written as (1, 2), (2, 3), (3, 4).

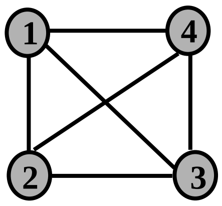
Definition: Let G = (V, E) be a graph. A *path* from vertex *u* to vertex *v* in a directed graph is a sequence of adjacent vertices <u, v_0, v_1, v_2,….v_k, v> such that <u, v_0>, <v_0, v_1>,….<v_k, v> are the edges in G. Consider the following graph:



In the graph, the path from vertex 1 to 3 is denoted by 1, 4, 2, 3 which can also be written as <1, 4>, <4, 2>, <2, 3>

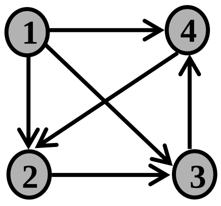
Now, let us see "What is simple path?"

Definition: A *simple path* is a path in which all vertices except possibly the first and last are distinct. Consider the undirected and directed graph shown below:



Ex 1: In the graph, the path 1, 2, 3, 4 is simple path since each node in the sequence is distinct.

Ex2: In the graph, the path 1, 2, 3, 2 is not a simple path since the nodes in sequence are not distinct. The node 2 appears twice in the path



Ex 1: In the graph, the path 1, 4, 2, 3 is simple path since each node in the sequence is distinct.

Ex 2: The sequence 1, 4, 3 is not a path since there is no edge <4, 3> in the graph.

Now, let us see "What is length of the path?"

Definition: The *length* of the path is the number of edges in the path.

Ex 1: In the above undirected graph, the path (1, 2, 3, 4) has length 3 since there are three edges (1, 2), (2, 3), (3, 4). The path 1, 2, 3 has length 2 since there are two edges (1, 2), (2, 3).

Ex 2: In the above directed graph, the path <1, 2, 3, 4> has length 3 since there are three edges <1, 2>, <2, 3>, <3, 4>. The path <1, 4, 2> has length 2 since there are two edges <1, 4>, <4, 2>.

Now, let us "Define the terms cycle (circuit)?"

Definition: A cycle is a path in which the first and last vertices are same.

For example, the path <4, 2, 3, 4> shown in above directed graph is a cycle, since the first node and last node are same. It can also be represented as <4, 2>, <2, 3>, <3, 4> <4, 2>.

Note: A graph with at least one cycle is called a cyclic graph and a graph with no cycles is called ***acyclic*** graph. A tree is an acyclic graph and hence it has no cycle.

Now, let us see "What is a connected graph?"

11.6 Graphs

Definition: In an undirected graph G, two vertices *u* and *v* are said to be connected if there exists a path from *u* to *v*. Since G is undirected, there exists a path from *v* to *u* also. A graph G (directed or undirected) is said to be connected if and only if there exists a path between every pair of vertices.

For example, the graphs shown in figure below are connected graphs.

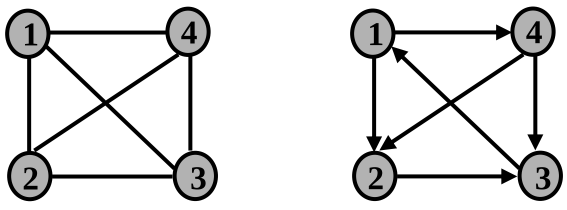
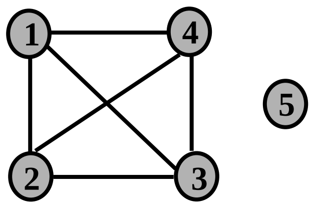


Figure Connected graphs

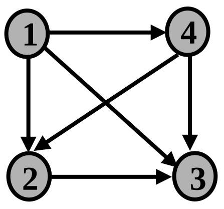
Now, let us see "What is a disconnected graph?"

Definition: Let G = (V, E) be a graph. If there exists at least one vertex in a graph that cannot be reached from other vertices in the graph, then such a graph is called disconnected graph. For example, the graph shown below is a disconnected graph.



Not connected

Since vertex 1 is not reachable from 3, the graph is not connected



11.3 Representation of graph

Now, let us see "What are the different methods of representing a graph?" The graphs can be represented in two different methods:

$$\begin{tikzcd} & \text{Adjacency matrix} \\ \text{Representation of graph} \arrow[ur] \arrow[dr] & \\ & \text{Adjacency linked list} \end{tikzcd}$$

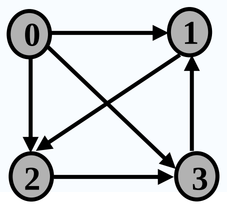
Let us see "What is an adjacency matrix? explain with example"

Definition: Let G = (V, E) be a graph where V is set of vertices and E is set of edges. Let N be the number of vertices in graph G. The adjacency matrix A of a graph G is formally defined as shown below:

$$A[i][j] = \begin{cases} 1 & \text{if there is an edge from vertex } i \text{ to vertex } j. \\ 0 & \text{if there is no edge from vertex } i \text{ to vertex } j. \end{cases}$$

- It is clear from the definition that an adjacency matrix of a graph with *n* vertices is a Boolean square matrix with *n* rows and *n* columns with entries 1's and 0's (bit-matrix)
- In an undirected graph, if there exists an edge (i, j) then a[i][j] and a[j][i] is made 1 since (i, j) is same as (j, i)
- In a directed graph, if there exists an edge <i, j> then a[i][j] is made 1 and a[j][i] will be 0.
- If there is no edge from vertex *i* to vertex *j*, then a[i][j] will be 0.

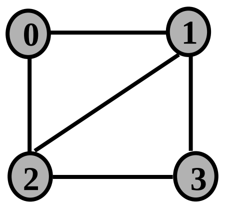
Note: The above definition is true both for directed and undirected graph. For example, following figures shows the directed and undirected graphs along with equivalent adjacency matrices:



(a) Directed graph

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

Adjacency matrix



(b) Undirected graph

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

Adjacency matrix

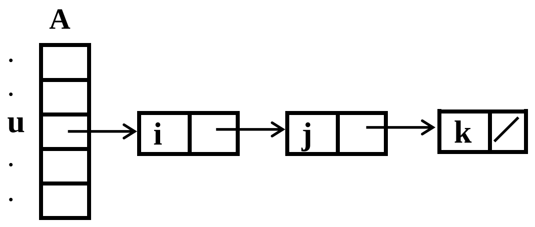
Fig. Graphs and equivalent adjacency matrices

Now, let us see "What is an adjacency list? explain with example"

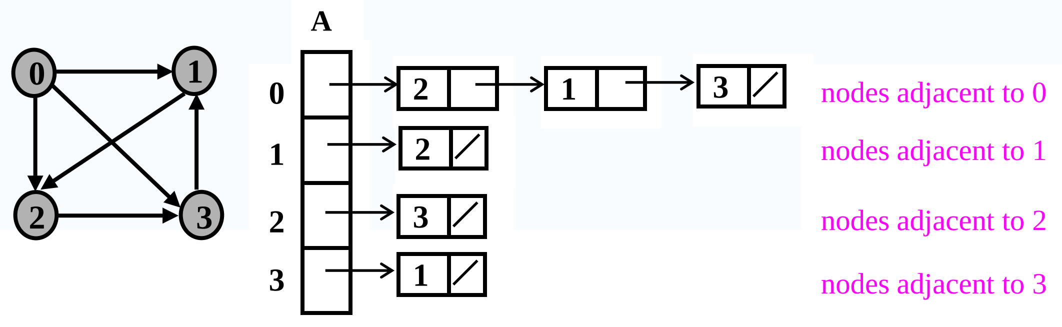
11.8 Graphs

Definition: Let G = (V, E) be a graph. An *adjacency linked list* is an array of *n* linked lists where *n* is the number of vertices in graph G. Each location of the array represents a vertex of the graph. For each vertex $u \in V$, a linked list consisting of all the vertices adjacent to *u* is created and stored in A[u]. The resulting array A is an adjacency list.

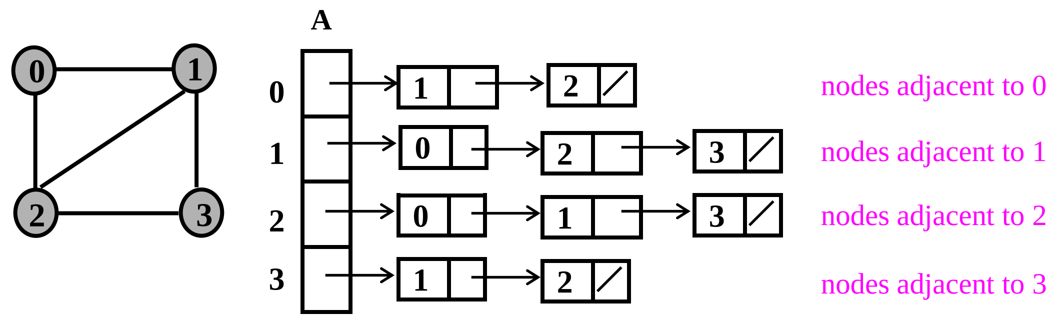
Note: It is clear from the above definition that if *i*, *j* and *k* are the vertices adjacent to the vertex *u*, then *i*, *j* and *k* are stored in a linked list and starting address of linked list is stored in A[u] as shown below:



For example, figures below shows the directed and undirected graphs along with equivalent adjacency linked list:



(a) Directed graph **Adjacency linked list**



(b) Undirected graph **Adjacency linked list**

Fig.: Graphs and equivalent adjacency linked lists

Now, let us see "Which graph representation is best?" The graph representation to be used depends on the following factors:

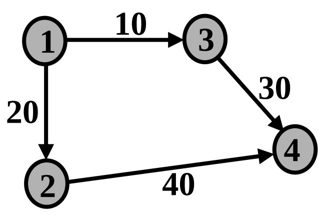
- Nature of the problem
- Algorithm used for solving
- Type of the input.
- Number of vertices and edges:
  - If a graph is *sparse*, less number of edges are present. In such case, the adjacency list has to be used because this representation uses lesser space when compared to adjacency matrix representation, even though extra memory is consumed by the pointers of the linked list.
  - If a graph is *dense*, the adjacency matrix has to be used when compared with adjacency list since the linked list representation takes more memory.

Note: So, based on the nature of the problem and based on whether the graph is *sparse* or *dense*, one of the two representations can be used.

Now, let us see "What is a weighted graph?"

Definition: A graph in which a number is assigned to each edge in a graph is called weighted graph. These numbers are called costs or weights. The weights may represent the cost involved or length or capacity depending on the problem.

For example, in the following graph shown in figure the values 10, 20, 30 and 40 are the weights associated with four edges <1,3>, <1,2>, <3,4> and <2,4>



Let us see "How the weighted graph can be represented?" The weighted graph can be represented using adjacency matrix as well as adjacency linked list. The adjacency matrix consisting of costs (weights) is called cost adjacency matrix. The adjacency linked list consisting of costs (weights) is called cost adjacency linked list. Now, let us see "What is cost adjacency matrix?"

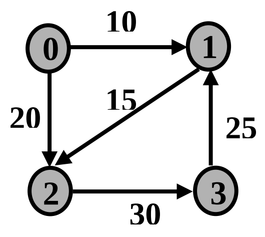
Definition: Let G = (V, E) be the graph where V is set of vertices and E is set of edges with *n* number of vertices. The cost adjacency matrix A of a graph G is formally defined as shown below:

$$A[i][j] = \begin{cases} w & \text{if there is a weight associated with edge from vertex } i \text{ to vertex } j. \\ \infty & \text{if there is no edge from vertex } i \text{ to vertex } j. \end{cases}$$

It is clear from the above definition that

- The element in ith row and jth column is weight *w* provided there exist an edge from ith vertex to jth vertex with cost *w*
- ∞ if there is no edge from vertex *i* to vertex *j*.
- The cost from vertex *i* to vertex *i* is ∞ (assuming there is no loop).

For example, the weighted graph and its *cost adjacency matrix* is shown below:



(a) Weighted graph

$$\begin{array}{c|cccc} & \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \\ \hline \mathbf{0} & \infty & 10 & 20 & \infty \\ \mathbf{1} & \infty & \infty & 15 & \infty \\ \mathbf{2} & \infty & \infty & \infty & 30 \\ \mathbf{3} & \infty & 25 & \infty & \infty \end{array}$$

Note: Diagonal values can be replaced by 0's

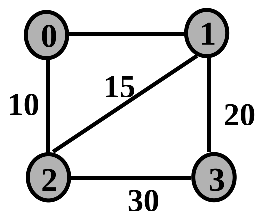
(b) Adjacency matrix

Figure A weighted digraph and the cost adjacency matrix

For the undirected graph, the elements of the cost adjacency matrix are obtained using the following definition:

$$A[i][j] = \begin{cases} w & \text{if there is a weight associated with edge (i, j) or (j, i)} \\ \infty & \text{if there is no edge from vertex } i \text{ to vertex } j. \end{cases}$$

The undirected graph and its equivalent adjacency matrix is shown below:



(a)

$$\begin{array}{c|cccc} & \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \\ \hline \mathbf{0} & \infty & 25 & 10 & \infty \\ \mathbf{1} & 25 & \infty & 15 & 20 \\ \mathbf{2} & 10 & 15 & \infty & 30 \\ \mathbf{3} & \infty & 20 & 30 & \infty \end{array}$$

Diagonal values can be replaced by 0's

(b)

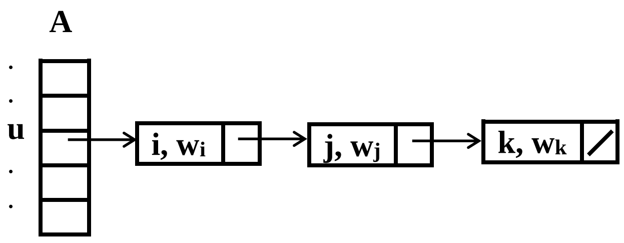
Figure: Weighted undirected graph and the adjacency matrix

Note: The cost adjacency matrix for the undirected graph is symmetric (i.e., a[i, j] is same as a[j, i]) whereas the cost adjacency matrix for a directed graph may not be symmetric.

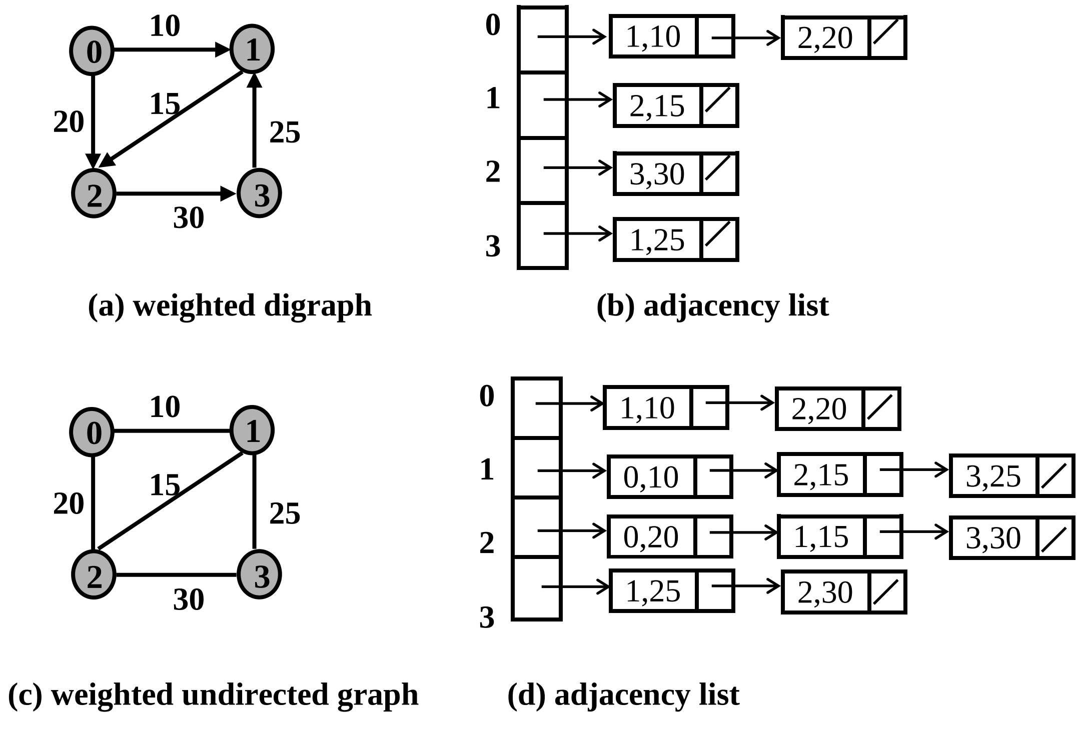
Note: For some of the problems, it is more convenient to store 0's in the main diagonal of cost adjacency matrix instead of ∞.

Now, let us see "What is cost adjacency linked list?"

Definition: Let G = (V, E) be a graph where V is set of vertices and E is set of edges with *n* number of vertices. A *cost adjacency linked list* is an array of *n* linked lists. For each vertex u ∈ V, A[u] contains the address of a linked list. All the vertices which are adjacent from vertex *u* are stored in the form of a linked list (in an arbitrary manner) and the starting address of first node is stored in A[u]. If *i, j* and *k* are the vertices adjacent to the vertex u, then *i, j* and *k* are stored in a linked list along with the weights in A[u] as shown below:



For example, the figure below shows the weighted diagraph and undirected graph along with equivalent adjacency list.



(a) weighted digraph (b) adjacency list

(c) weighted undirected graph (d) adjacency list

Figure weighted graph and equivalent adjacency list

Now, the function to read an adjacency matrix can be written as shown below:

Example 11.1: Function to read adjacency matrix

```
void read_adjacency_matrix(int a[10][10], int n)
{
        int     i, j;

        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n; j++)
                {
                        scanf(“%d”, &a[i][j]);
                }
        }
}
```

The function to read adjacency list can be written as shown below:

Example 11.2: Function to read adjacency list

```
void read_adjacency_list (NODE a[], int n)
{
        int     i, j, m, item;

        for (i = 0; i < n; i++)
        {
                printf("Enter the number of nodes adjacent to %d:", i);
                scanf("%d", &m);

                if (m ==0) continue;

                printf("Enter nodes adjacent to %d : ", i);

                for (j = 0; j < m; j++)
                {
                        scanf("%d", &item);

                        a[i] = insert_rear(item, a[i]);
                }
        }
}
```

11.4 Graph traversals

Now, we concentrate on a very important topic namely graph traversal techniques and see "What is graph traversal? Explain different graph traversal techniques"

Definition: The process of visiting each node of a graph systematically in some order is called graph traversal. The two important graph traversal techniques are:

- **B**readth **F**irst **S**earch (**BFS**)
- **D**epth **F**irst **S**earch (**DFS**)

11.4.1 Breadth First Search (BFS)

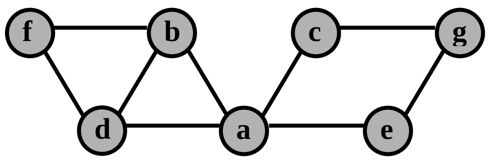
Now, let us see "What is breadth first search (BFS)?"

Definition: The breadth first search is a method of traversing the graph from an arbitrary vertex say *u*. First, visit the node *u*. Then we visit all neighbors of *u*. Then we visit the neighbors of neighbors of *u* and so on. That is, we visit all the neighboring nodes first before moving to next level neighbors. The search will terminate when all the vertices have been visited.

BFS traversal can be implemented using a queue. As we visit a node, it is inserted into queue. Now, delete a node from a queue and see the adjacent nodes which have not been visited. The unvisited nodes are inserted into queue and marked as visited. Deleting and inserting operations as discussed are continued until queue is empty.

Now, let us take an example and see how BFS traversal can be used to see what are all the nodes which are reachable from a given source vertex.

Example 11.3: Traverse the following graph by breadth-first search and print all the vertices reachable from start vertex *a*. Resolve ties by the vertex alphabetical order.



Solution: It is given that source vertex is *a*. Perform the following activities:

Initialization: Insert source *vertex a into queue* and *add a to S* as shown below:

	(i) u = del(Q)	(ii) v = adj. to u	(iii) Nodes visited S	(iii) queue
Initialization	-	-	a	a

Step 1: **i):** Delete an element *a* from queue
ii): Find the nodes adjacent to *a* but not in S: i.e., *b, c, d* and *e*
iii): Add *b, c, d* and *e* to S, insert into queue as shown in the table:

	(i) u = del(Q)	(ii) v = adj. to u	(iii) Nodes visited S	(iii) queue
	-	-	a	a
Step 1	a	b, c, d, e	a, b, c, d, e	b, c, d, e

Step 2: **i):** Delete *b* from queue
ii): Find nodes adjacent to *b* but not in S: i.e., f
iii): Add *f* to S and insert *f* into queue as shown in table:

	(i) u = del(Q)	(ii) v = adj. to u	(iii) Nodes visited S	(iii) queue
	-	-	a	a
Step 1	a	b, c, d, e	a, b, c, d, e	b, c, d, e
Step 2	b	f	a, b, c, d, e, f	c, d, e, f

Stage 3: **i) :** Delete *c* from queue
ii): Find nodes adjacent to *c* but not in S: i.e., g
iii): Add *g* to S, insert *g* into queue as shown in table

	(i) u = del(Q)	(ii) v = adj. to u	(iii) Nodes visited S	(iii) queue
	-	-	a	a
Step 1	a	b, c, d, e	a, b, c, d, e	b, c, d, e
Step 2	b	f	a, b, c, d, e, f	c, d, e, f
Step 3	c	g	a, b, c, d, e, f, g	d, e, f

The remaining steps are shown in the following table:

(i) (ii) ⟵ (iii) ⟶

	u = del(Q)	v = adj. to u	Nodes visited S	queue
Initialization	-	-	a	a
Step 1	a	b, c, d, e	a, b, c, d, e	b, c, d, e
Step 2	**b**	a, d, **f**	a, b, c, d, e, f	c, d, e, f
Step 3	c	a, **g**	a, b, c, d, e, f, g	d, e, f, g
Step 4	d	a, b, f	a, b, c, d, e, f, g	e, f, g
Step 5	e	a,g	a, b, c, d, e, f, g	f, g
Step 6	f	b, d	a, b, c, d, e, f, g	g
Step 7	g	c. e	a, b, c, d, e, f, g	empty

Thus, the nodes that are reachable from source *a*: **a, b, c, d, e, f, g**

11.4.1.1 Breadth First Search (BFS) using adjacency matrix

The above activities are shown below in the form of an algorithm along with pseudocode in C when graph is represented as an adjacency matrix.

```
no node is visited to start with              // int s [10] = {0};

insert source u to q                          // f = 0, r = -1, q[++r] = u
print u                                       // print u
mark u as visited i.e., add u to S            // s[u] = 1
while queue is not empty                      // while f <= r

        Delete a vertex u from q              //      u = q[f++]
        For every v adjacent to u             //      for each v, if a[u][v] == 1
        If v is not visited                   //              if s[v] == 0
                print v                       //                      print v
                mark v as visited             //                      s[v] = 1
                Insert v to queue             //                      q[++r] = v
        end if                                //              endif
                                              //      endif
end while                                     // end while
```

The above algorithm can be written using C function as shown below:

Example 11.4: C function to show the nodes visited using BFS traversal

```
void bfs(int a[10][10], int n, int u)
{
        int     f, r, q[10], v;
        int     s[10] = {0};  /* initialize all elements in s to 0 i.e, no node is visited */

        printf("The nodes visited from %d : ", u);
        f = 0, r = -1;              // queue is empty
        q[++r] = u;                 // Insert u into queue

        s[u] = 1;                   // insert u to s
        printf("%d  ", u);          // print the node visited

        while ( f <= r )
        {
                u = q[f++];                         // delete an element from q
                for (v = 0; v < n; v++)
                {
                        if (a[u][v] == 1)           // If v is adjacent to u
                        {
                                if (s[v] == 0)  // If v is not in S i.e., v has not been visited
                                {
                                        printf("%d  ", v);   // print the node visited
                                        s[v] = 1;            // add v to s, mark it as visited
                                        q[++r] = v;          // Insert v into queue
                                }
                        }
                }
        }
        printf("\n");
}
```

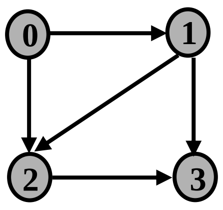
Now, the C program that prints all the nodes that are reachable from a given source vertex is shown below:

Example 11.5: Algorithm to traverse the graph using BFS

```
#include <stdio.h>
/* Insert: Example  11.1: Function to read an adjacency  matrix*/
/* Insert: Example  11.4: Function to traverse the graph in BFS */
```

```
void main()
{
        int     n, a[10][10], source, i, j;

        printf("Enter the number of nodes : ");
        scanf("%d", &n);

        printf("Enter the adjacency matrix:\n");
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n; j++) scanf("%d", &a[i][j]);
        }

        for (source = 0; source < n; source++)
                bfs(a, n, source);
}
```

Now, let us see how to obtain the nodes reachable from each node of the following graph using the above program:



Given graph

$$\begin{array}{c|c|c|c|c|}
 & 0 & 1 & 2 & 3 \\ \hline
0 & 0 & 1 & 1 & 0 \\ \hline
1 & 0 & 0 & 1 & 1 \\ \hline
2 & 0 & 0 & 0 & 1 \\ \hline
3 & 0 & 0 & 0 & 0 \\ \hline
\end{array}$$

Adjacency matrix

Output

```
Enter the number of nodes: 4
Enter the adjacency matrix:
0 1 1 0
0 0 1 1
0 0 0 1
0 0 0 0
The nodes visited from 0: 0  1  2  3
The nodes visited from 1: 1  2  3
The nodes visited from 2: 2  3
The nodes visited from 3: 3
```

11.4.1.2 Breadth First Search (BFS) using adjacency list

We know that BFS traversal uses queue data structure which require insert rear function and delete front function. We can use insert rear function given in example 8.6. But, the delete front function shown in example 8.5 is modified after deleting the printf() function.

Example 11.6: C function to delete an item from the front end of singly linked list

```
NODE delete_front(NODE first)
{
        NODE temp;

        if ( first == NULL )  return NULL;

        temp = first;               /* Retain address of the node to be deleted */
        temp = temp->link;          /* Obtain address of the second node */
        free(first);                /* delete the front node */
        return temp;                /* return address of the first node */
}
```

The algorithm for BFS along with pseudocode when a graph is represented as an adjacency list can be written as shown below:

```
        no node is visited to start with        // int s [10] = {0}

        insert source u to q                    // q = NULL, q = insert_rear(u, q);

        mark u as visited i.e., add u to S      // s[u] = 1, printf("%d  ", u);

        while queue is not empty                // while q != NULL
                                                        u = q->info;
                Delete a vertex u from q        //      q = delete_front(q),
                                                        list = a[u]; // list of vertices adj. to u
                Find vertices v adjacent to u   //      while (list != NULL)
                                                                v = list->info;
                If v is not visited             //              if (s[v] == 0)
                        print v                 //                      print v
                        mark v as visited       //                      s[v] = 1
                        Insert v to queue       //                      q = insert_rear(v, q);
                end if                          //              endif
                                                                list = list->link
                                                //      end while
        end while                               // end while
```

Now, the complete C function to traverse the graph using BFS when a graph is represented as adjacency list can be written as shown below:

Example 11.7: C function to show the nodes visited using BFS traversal

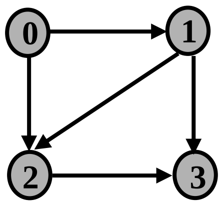
```
void bfs(NODE a[], int n, int u)
{
        NODE        q, list;
        int         v;

        int     s[10] = {0};   /* initialize all elements in s to 0 i.e, no node is visited */

        printf("The nodes visited from %d : ", u);

        q = NULL;                       // queue is empty
        q = insert_rear(u, q);          // Insert u into queue

        s[u] = 1;                       // insert u to s
        printf("%d  ", u);              // print the node visited
        while ( q != NULL )                             // as long as queue is not empty
        {
                u = q->info;                            // delete a node from queue
                q = delete_front(q);

                list = a[u];                            // obtain nodes adjacent to u
                while (list != NULL)                    // as long as adjacent nodes exist
                {
                        v = list->info;                 // v is the node adjacent to u
                        if (s[v] == 0)  // If v is not in S i.e., v has not been visited
                        {
                                printf("%d  ", v);      // print the node visited

                                s[v] = 1;               // add v to s, mark it as visited
                                q = insert_rear(v, q);  // Insert v into queue
                        }
                        list = list->link;
                }
        }
        printf("\n");
}
```

Now, the complete C program to see the nodes reachable from each of the nodes in the graph can be written as shown below:

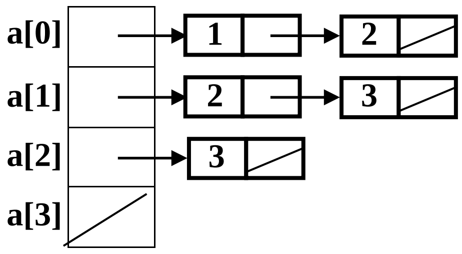
Example 11.8: Program to print nodes reachable from a vertex (bfs using adjacency list)

```
#include <stdio.h>

#include <stdlib.h>

struct node
{
        int             info;
        struct node     *link;
};

typedef struct node *NODE;

/* Insert: Example  8.2: Function to get a node */

/* Insert: Example  8.6: Function to insert an element into queue */

/* Insert: Example  11.2: Function to read adjacency list */

/* Insert: Example  11.6: Function to delete an element from front end of queue */

/* Insert: Example  11.7: Function to traverse the graph in BFS (adjacency list) */

void main()
{
        int             n, i, source;
        NODE            a[10];

        printf("Enter the number of nodes : ");
        scanf("%d", &n);

        for (i = 0; i < n; i++) a[i] = NULL;          // Graph is empty to start with

        read_adjacency_list(a, n);

        for (source = 0; source < n; source++)
                bfs(a, n, source);
}
```

Now, let us see how to obtain the nodes reachable from each node of the following graph using the above program:



Given graph



Adjacency linked list

Input

```
Enter the number of nodes: 4
Enter the number of nodes adjacent 0: 2
Enter nodes adjacent to 0: 1  2

Enter the number of nodes adjacent 1: 2
Enter nodes adjacent to 1: 2  3

Enter the number of nodes adjacent 2: 1
Enter nodes adjacent to 2: 3

Enter the number of nodes adjacent 3: 0
Enter nodes adjacent to 3:
```

Output

```
The nodes visited from 0: 0  1  2  3
The nodes visited from 1: 1  2  3
The nodes visited from 2: 2  3
The nodes visited from 3: 3
```

11.4.2 Depth First Search (DFS)

The depth first search is a method of traversing the graph by visiting each node of the graph in a systematic order. As the name implies *depth-first-search* means "to search deeper in the graph". Now, let us see "What is depth first search (DFS)?"

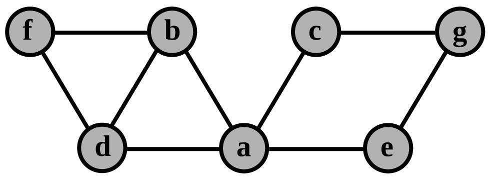
Definition: In DFS, a vertex *u* is picked as source vertex and is visited. The vertex *u* at this point is said to be unexplored. The exploration of the vertex *u* is postponed and a vertex *v* adjacent to *u* is picked and is visited. Now, the search begins at the vertex *v*. There may be still some nodes which are adjacent to *u* but not visited. When the vertex *v* is completely examined, then only *u* is examined. The search will terminate when all the vertices have been examined.

Note: The search continues deeper and deeper in the graph until no vertex is adjacent or all the vertices are visited. Hence, the name DFS. Here, the exploration of a node is postponed as soon as a new unexplored node is reached and the examination of the new node begins immediately.

Design methodology The iterative procedure to traverse the graph in DFS is shown below:

```
Step 1: Select node u as the start vertex (select in alphabetical order), push u onto
        stack and mark it as visited.  We add u to  S for marking

Step 2: While stack is not empty
            For vertex u on top of the stack, find the next immediate adjacent vertex.
            if v is adjacent
                if a vertex v not visited then
                    push it on to stack and number it in the order it is pushed.
                    mark it as visited by adding v to S
                else
                    ignore the vertex
                end if
            else
                remove the vertex from the stack
                number it in the order it is popped.
            end if
        end while

Step 3: Repeat step 1 and step 2 until all the vertices in the graph are considered
```

Example 11.9: Traverse the following graph using DFS and display the nodes reachable from a given source vertex



Solution: Since vertex *a* is the least in alphabetical order, it is selected as the start vertex. Follow the same procedure as we did in BFS. But, there are two changes:

- Instead of using a queue, we use stack
- In BFS, all the nodes adjacent and which are not visited are considered. In DFS, only one adjacent which is not visited earlier is considered. Rest of the procedure remains same.

Now, the graph can be traversed using DFS as shown in following table

	Stack	**v = adj(s[top])**	**Nodes visited S**	**pop(stack)**
Initial step	a	-	a	
Stage 1	a	b	a, b	-
Stage 2	a, b	d	a, b, d	-
Stage 3	a, b, d	f	a, b, d, f	-
Stage 4	**a, b, d, f**	-	a, b, d, f	f
Stage 5	a, b, d	-	a, b, d, f	d
Stage 6	a, b	-	a, b, d, f	b
Stage 7	a	c	a, b, d, f	-
Stage 8	a, c	g	a, b, d, f, g	-
Stage 9	a, c, g	e	a, b, d, f, g, e	-
Stage 10	a, **c, g, e**	-	a, b, d, f, g, e	e
Stage 11	a, c, g	-	a, b, d, f, g, e	g
Stage 12	a, c	-	a, b, d, f, g, e	c
Stage 13	a_1	-	a, b, d, f, g, e	$a_{1,7}$

11.4.2.1 Depth First Search (DFS) using adjacency matrix

It is clear from the above example that the *stack* is the most suitable data structure to implement DFS. Whenever a vertex is visited for the first time, that vertex is pushed on to the stack and the vertex is deleted from the stack when a dead end is reached and the search resumes from the vertex that is deleted most recently. If there are no vertices adjacent to the most recently deleted vertex, the next node is deleted from the stack and the process is repeated till all the vertices are reached or till the stack is empty.

The recursive function can be written as shown below: (Assuming adjacency matrix *a*, number of vertices *n* and array *s* as global variables)

Example 11.10: Program to print nodes reachable from a vertex (dfs - adjacency matrix)

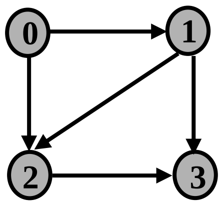
```
void dfs(int u)
{
        int v;

        s[u] = 1;

        printf("%d ", u);

        for (v = 0; v < n; v++)
        {
                if (a[u][v] == 1 && s[v] == 0) dfs(v);
        }
}
```

The complete program that prints the nodes reachable from each of the vertex given in the graph can be written as shown below:

Example 11.11: Program to print nodes reachable from a vertex (dfs - adjacency matrix)

```
#include <stdio.h>

int     a[10][10], s[10], n;            // Global variables

/* Insert: Example  11.1: Function to read an adjacency  matrix*/
/* Insert: Example  11.10: Function to traverse the graph in DFS */

void main()
{
        int     i, source;

        printf("Enter the number of nodes in the graph : ");
        scanf("%d", &n);
        printf("Enter the adjacency matrix:\n");
        read_adjacency_matrix(a, n);

        for (source = 0; source < n; source++)
        {
                for (i = 0; i < n; i++) s[i] = 0;

                printf("\nThe nodes reachable from %d: ", source);
                dfs(source);
        }
}
```

Now, let us see how to obtain the nodes reachable from each node of the following graph using the above program:



Given graph

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

Adjacency matrix

Output

```
Enter the number of nodes: 4
Enter the adjacency matrix:
0 1 1 0
0 0 1 1
0 0 0 1
0 0 0 0
The nodes visited from 0: 0  1  2  3
The nodes visited from 1: 1  2  3
The nodes visited from 2: 2  3
The nodes visited from 3: 3
```

11.4.2.2 Depth First Search (DFS) using adjacency linked list

The procedure remains same. But, instead of using adjacency matrix, we use adjacency list. The recursive function can be written as shown below: (Assuming adjacency list *a*, number of vertices *n* and array *s* as global variables.)

Example 11.12: Program to print nodes reachable from a vertex (dfs - adjacency list)

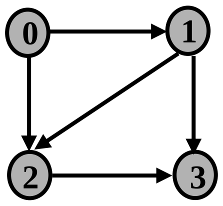
```
void dfs(int u)
{
        int v;
        NODE temp;

        s[u] = 1;

        printf("%d ", u);

        for (temp = a[u];   temp != NULL;   temp = temp->link)
                if (s[temp->info] == 0) dfs(temp->info);
}
```

The complete program that prints the nodes reachable from each of the vertex given in the graph using DFS represented using adjacency list can be written as shown below:

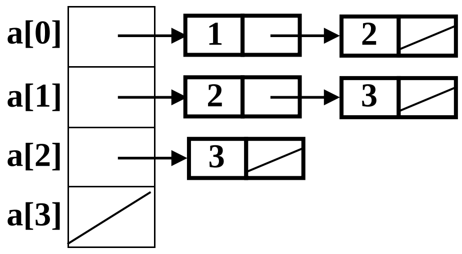
Example 11.13: Program to print nodes reachable from a vertex (dfs - adjacency matrix)

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

struct node
{
        int             info;
        struct node    *link;
};

typedef struct node *NODE;

NODE        a[10];
int         s[10], n;                // Global variables

/* Insert: Example  8.2: Function to get a node */

/* Insert: Example  8.6: Function to insert an element into queue */

/* Insert: Example  11.2: Function to read adjacency list */

/* Insert: Example  11.12: Function to traverse the graph in DFS */
void main()
{
        int     i, source;

        printf("Enter the number of nodes in the graph : ");
        scanf("%d", &n);

        printf("Enter the adjacency list:\n");
        read_adjacency_list(a, n);

        for (source = 0; source < n; source++)
        {
                for (i = 0; i < n; i++) s[i] = 0;

                printf("\nThe nodes reachable from %d: ", source);
                dfs(source);
        }
}
```

Now, let us see how to obtain the nodes reachable from each node of the following graph using the above program:



Given graph



Adjacency linked list

Input

```
Enter the number of nodes: 4
Enter the number of nodes adjacent 0: 2
Enter nodes adjacent to 0: 1  2

Enter the number of nodes adjacent 1: 2
Enter nodes adjacent to 1: 2  3

Enter the number of nodes adjacent 2: 1
Enter nodes adjacent to 2: 3

Enter the number of nodes adjacent 3: 0
Enter nodes adjacent to 3:
```

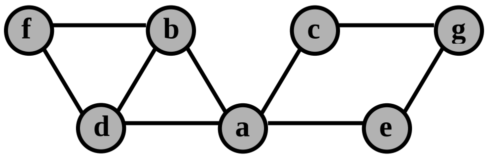
Output

```
The nodes visited from 0: 0  1  2  3
The nodes visited from 1: 1  2  3
The nodes visited from 2: 2  3
The nodes visited from 3: 3
```

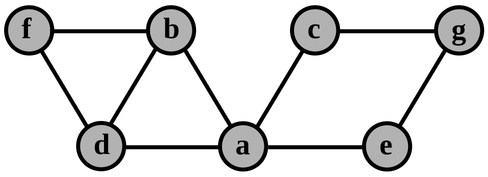
Exercises

1) Define the terms: a) vertex b) edge c) graph d) directed graph
   e) undirected graph
2) Define the terms: a) self-loop (or self-edge) b) multigraph c) complete graph
3) Define the terms: a) path b) simple path c) length of the path
4) Define the terms: a) cycle (circuit) b) Connected graph c) disconnected graph
5) What are the different methods of representing a graph?
6) What is an adjacency matrix? explain with example

7) What is an adjacency list? Explain with example
8) What is a weighted graph?
9) How the weighted graph can be represented?
10) What is cost adjacency matrix? What is cost adjacency linked list?
11) What is graph traversal? Explain different graph traversal techniques
12) What is breadth first search (BFS)?"
13) Traverse the following graph by breadth-first search and print all the vertices reachable from start vertex *a*. Resolve ties by the vertex alphabetical order.



14) Write a C function to show the nodes visited using BFS traversal (adjacency matrix)
15) Write a C function to show the nodes visited using BFS traversal (adjacency list)
16) What is depth first search (DFS)?"
17) Traverse the following graph using DFS and display the nodes reachable from a given source vertex



18) Write a program to print nodes reachable from a vertex (dfs - adjacency matrix)
19) Write a program to print nodes reachable from a vertex (dfs - adjacency matrix)