CHAPTER 9

GRAPH

"Computers make it easier to do a lot of things, but most of the things they make it easier to do don't need to be done." ***-Andy Rooney***

KEY FEATURES

- Categorization of Graph
- Graph Representation
- Graph traversals
- Shortest Path
- Spanning Tree
- Application of Graph

Graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is also used to model networks, complex data structures, scheduling, computation and a variety of other systems, where the relationship between objects in the system plays a key role.

A Graph G = (V, E) consists of finite non-empty set of objects V, where V (G) = $\{v_1, v_2, v_3 \ldots v_n\}$ called vertices and another set E where E (G) = $\{e_1, e_2, e_3 \ldots e_m\}$ whose elements are called edges.

Definition: A Graph G is defined as an ordered set G = (V, E), where V represents a set of elements called vertices (or points or vertices) and E represents a set of edges in G, that connects these vertices.

In the figure 9.1 shows a graph with five vertices, V = {A, B, C, D, E} and six connecting edges, E = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}.

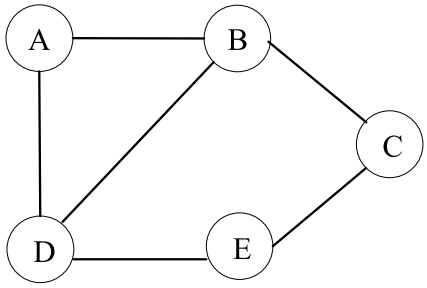


Figure 9.1: Graph with five Vertices and six Edges

Terminology of Graph

Undirected Graph: A Graph can be directed or undirected. In an undirected graph edge, do not have any direction associated with them. That is, if there is an edge between vertex A and B then the vertices can be traversed A to B, as well as B to A. Figure 9.1 shows an undirected graph.

Directed Graph: A directed graph or digraph is a graph, where the vertices are connected together and all the edges are directed from one vertex to another.

In a directed graph, edges form an ordered pair. If there is a directed edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is supposed to initiate from vertex A (initial vertex) and terminate at vertex B (terminal vertex). Figure 9.2 shows a directed graph.

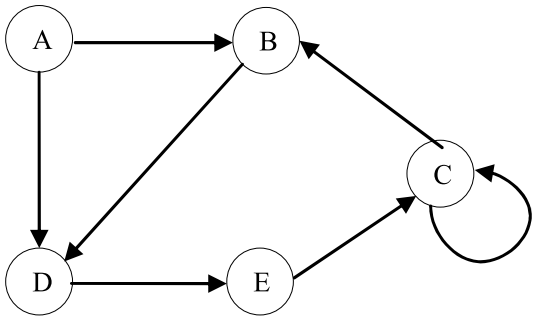


Figure 9.2: Directed Graph with Five Vertices and self-loop

Adjacent vertices or neighbors: For every edge e = (u, v), that connects vertices u and v, the vertices u and v are endpoints and are said to be the adjacent vertices or neighbors.

Degree: The degree of a vertex of a graph is the number of edges incident to the vertex, with self-loops counted twice. The degree of a vertex v is denoted by deg (v) or deg v. If deg (v) = 0, it means that u does not belong to any edge and such a vertex is known as an isolated vertex. In figure 9.2, deg (c) = 4.

The maximum degree of a graph G, denoted by Δ (G) and a minimum degree of a graph, denoted by δ(G), are the maximum and minimum degree of its vertices.

In a digraph, the number of edges coming out of a vertex is called the out-degree of that vertex. A number of edges coming in of a vertex is the in-degree of that vertex.

For a digraph G = (V, E),

$\sum_{v \in V} in\_\deg(v) = \sum_{v \in V} out\_\deg(v) = |E|$, where |E| means the cardinality of the set E (i.e. the number of edges).

For an undirected graph G = (V, E), $\sum_{v \in V} \deg(v) = 2|E|$.

Regular graph: It is a graph, where each vertex has same no of neighbors. That is, every vertex has the same degree. A regular graph with the vertex of degree k is called a k-regular graph. Figure 9.3 shows some regular graphs.

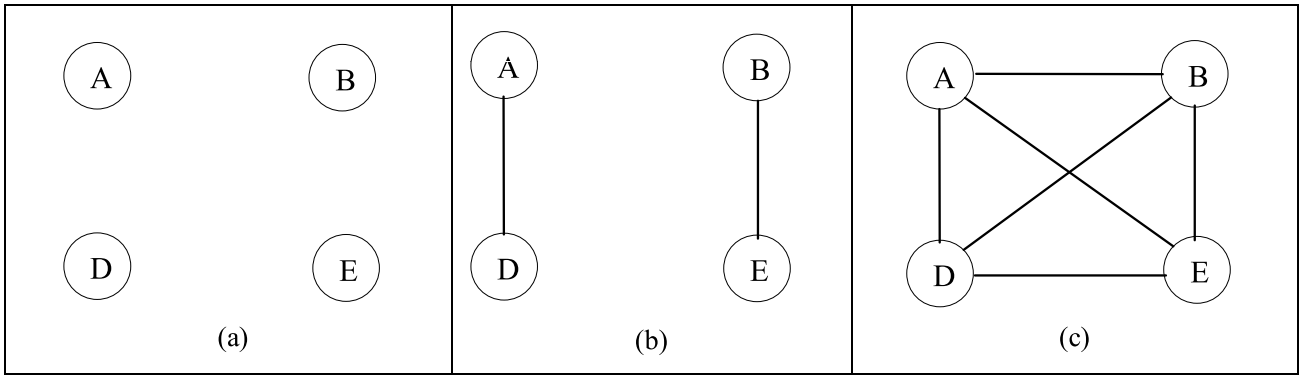


Figure 9.3: Graph with Four Vertices (a) 0-regular graph, (b) 1-regular graph, (c) 3-regular graph

Walk: A walk of a graph is a finite altering sequence of vertices and edges beginning and ending with vertices. In a walk, no edge is traversed more than one.

Open Walk: An open walk is that where no edge is repeated. $V_1\ e_1\ V_2\ e_5\ V_5\ e_7\ V_3\ e_3\ V_4$ in the following graph G is an open walk as no edge is repeated.

Closed walk: A walk having same starting and end point is called closed walk. $V_1\ e_1\ V_2\ e_2\ V_4\ e_{11}\ V_3\ e_4\ V_1$ is a closed walk where both starting and end vertex is V_1.

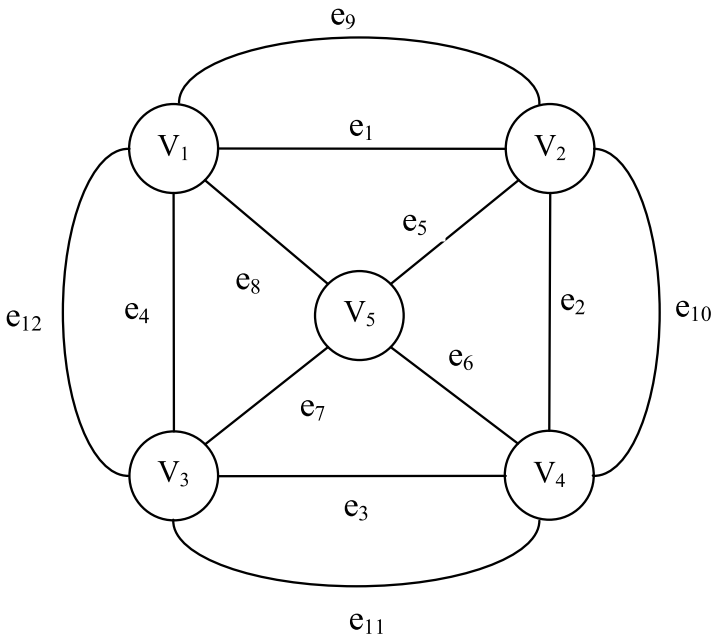


Figure 9.4: Walk of Graph

Path: A path P has written as P = {v_0, v_1, v_2...v_n), of length n from a vertex u to v is defined as a sequence of (n+1) vertices. Here, u = v_0, v = v_n, and v_{i-1} is adjacent to v_i for i = 1, 2, 3… n.

Closed Path: A path is known as a closed path if the edge has same endpoints. In figure 9.4, V_1 V_2 V_4, V_3, V_1 is a closed path with same end point V_1.

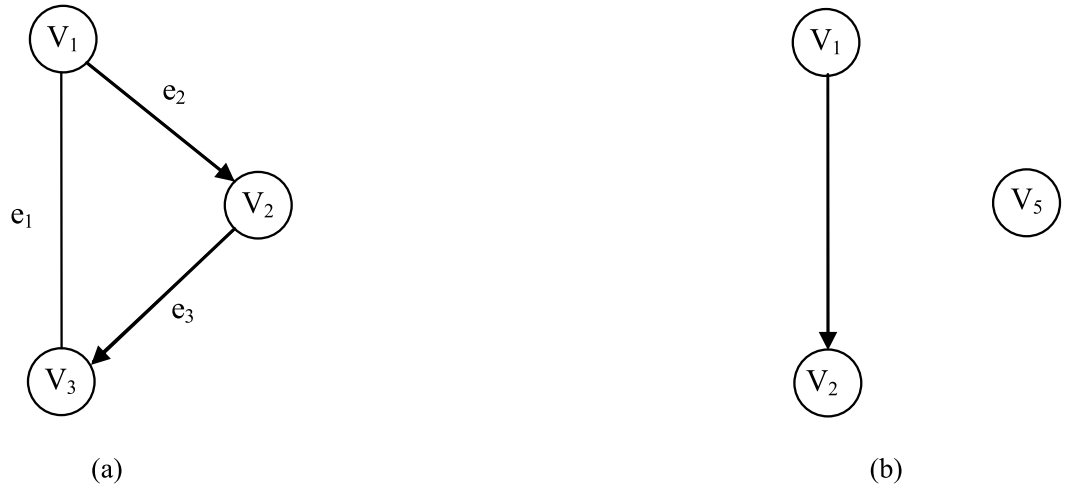


Figure 9.5: (a) connected undirected graph, (b) connected directed graph

Simple/Open Path: A path is known as a simple path if all the vertices in a path are distinct. In figure 9.4, V_1 e_1 V_2 e_5 V_5 e_7 V_3 e_{11} V_4 is an open path as no vertex is repeated.

Cycle: A path in which the first and last vertices are same and no repeated edges or vertices. In figure 9.4, V_1 e_1 V_2 e_2 V_4 e_{11} V_3 e_{12} V_1 form a circuit or circle. A graph is said to be **acyclic** if it contains no cycles. A directed graph that is acyclic is called **Directed Acyclic Graph** (DAG).

Connected Graph: A graph is called connected graph, if and only if there is a simple path between

any two vertices of a graph. A connected graph without any cycle is a tree. Figure 9.5 (a) is an example of a connected graph.

Complete Graph: A graph is said to be complete if there is a path between one vertex and every other vertex in the graph. A complete graph with n number of vertices has n(n-1)/2 edges. In the above figure 9.5 graph G is a complete undirected graph.

Clique: In an undirected graph G = (V, E), a clique is a subset of the vertex set, such that for every two vertices in this set, there is an edge that connects two vertices.

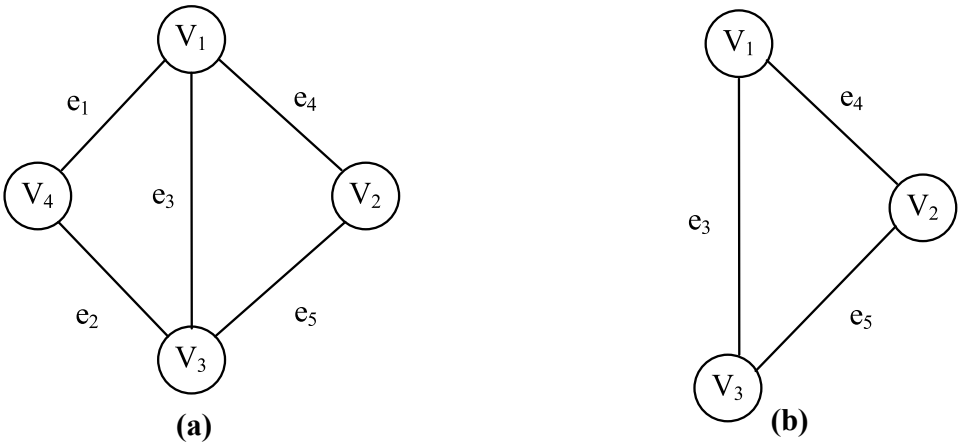


Figure 9.6: (b) Clique of Connected Undirected graph (a)

Weighted Graph: A graph is called weighted if every edge in the graph is assigned some weight data. The edge weight generally denotes by w (e) is a positive value, which indicates the cost of traversing the edge in figure 9.8c.

Parallel or Multiple Edges: Distinct edges, which connect the same endpoints, are called multiple edges. That is, e = (u, v) and e′ = (u, v) are known as multiple edges of a graph.

Loop: An edge has identical endpoints is called a loop. That is e = (u, u) given in figure 9.1.

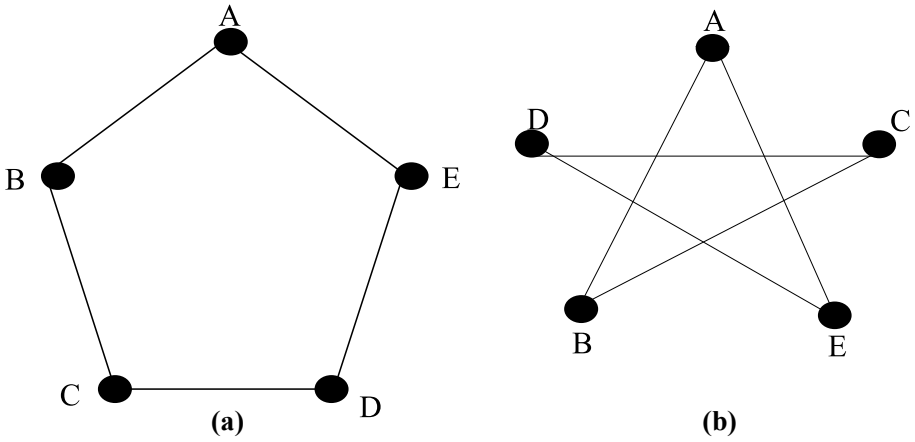


Figure 9.7: (a) and (b) are isomorphic graph

Isomorphism: Two graphs G = (V, E) and G‘ = (V, E) are said to be isomorphism graphs if there exists a one-to-one correspondence between their vertices and their edges such that the incidence relationship preserved, i.e. they contain the same number of vertices connected in the same way, the same number of edges and both have same degree sequences. In figure 9.7 example of isomorphism is given.

A directed graph G, also called as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of vertices in G. For an edge (u, v) .

- The edge begins at u and terminates at v.
- u is known as an initial vertex of e, whereas v is the terminal vertex.
- u is the predecessor of v and correspondingly v is the successor of u.
- u and v vertices are adjacent to each other.

Multigraph: A graph with multiple edges and/or loops is called multi-graph shown in figure 9.8a.

Subgraph: A subgraph H of a graph G, is a graph whose vertices are a subset of the vertex set of G and whose edges are a subset of the edge set of G. H $\subseteq$ G if V′ $\subseteq$ V and E′ $\subseteq$ E. Figure 9.6b is an example of the subgraph.

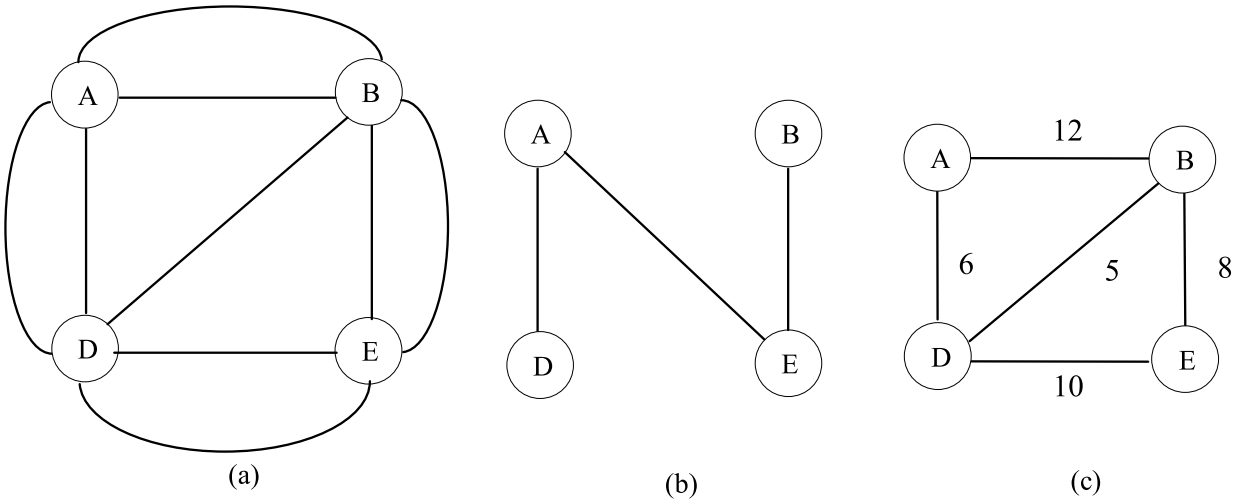


Figure 9.8: (a) Multi Graph, (b) Tree, (c) Weighted Graph

Lemma 1 (Hand Shaking Lemma): In a regular undirected graph the sum of the degree of all the vertices is twice the number of edges. That is for an undirected graph G = (V, E), $\sum_{v \in V} \deg(v) = 2|E|$

Proof: Let P be the proposition ~~"In~~ any graph, the sum of the degrees of all vertices is equal to twice the number of edges" for a graph G = (V, E)

P(n): $\sum_{V \in G} \deg(|V|) = 2n$ where $|E| = n$

Base case: P(0):$2n = 0|n = 0$. Since there is not any edge the number of vertices must be equal to 11or 0.

$\sum_{V \in G} \deg(|V|) = \deg(|V|) = 0$ the number of degrees is equal to 0. Thus P(0) is true.

Induction step: Assuming that P(n) is true for a given natural number. Let show that P(n)⇒P(n+1).

P (n): $\sum_{V \in G} \deg(|V|) = 2n$

$\sum_{V \in G} \deg(|V|) + 2 = 2n + 2$

$\sum_{V \in G} \deg(|V|) + 2 = 2\,(n + 1)$ which yields by adding 2

Vertices of degree 1

$\forall n \in N, \sum_{V \in G} \deg(|V|) = 2|E|$

Terminology of a Directed Graph

Isolated vertex: The vertex with degree zero. Such vertex is not an end point of any edge. Vertex F in figure 9.9 is an isolated vertex.

Pendant vertex: A vertex is said to be pendant vertex if the vertex has degree one. It is also known as leaf vertex. Vertices G and H in figure 9.9 are pendant vertex.

Pendant edge: An edge is said to be pendant edge if the edge incident with the pendant vertex. The edge between G and H in figure 9.9.

Cut vertex: The vertex which when deleted would disconnect the remaining graph. If vertices B and D are deleted, then graph G become disconnected. So B and D are cut vertices in figure 9.9.

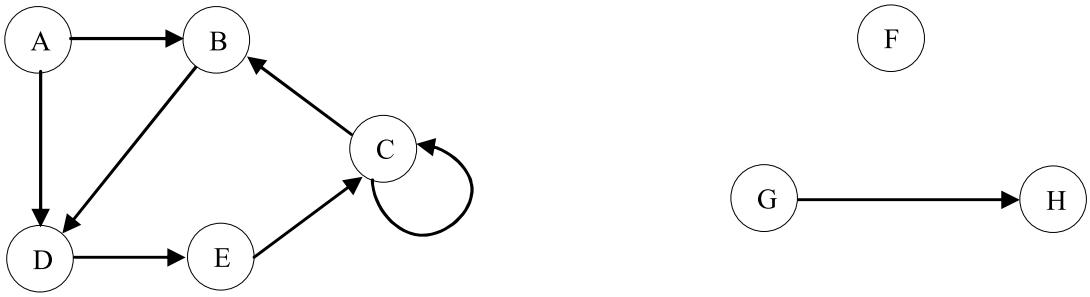


Figure 9.9: Directed Graph G

Source: The vertex with positive out-degree, but no in-degree. Vertex A and G in graph G has out-degree 2 and 1 respectively and in-degree 0. Therefore, they are source vertices in figure 9.9.

Sink: The vertex with positive in-degree, but no out-degree. Vertex H has I in degree and 0 out degree. Therefore, it is sink vertex in figure 9.9.

Reachability: A vertex v is said to be reachable from vertex u, if and only if there exists a path from vertex u to v. Vertex B is reachable from C but vertex G is not reachable from vertex C in figure 9.9.

Connected Graph: A graph G is a connected graph if, between every pair of vertices in G, there always exists a path in G.

Strongly connected directed graph: A digraph is said to be strongly connected, if and only if there exists a path between every pair of vertices. That is, if there is a path from vertex u to v then there must be a path from vertex v to u. Figure 9.10a is an example of a strongly connected graph.

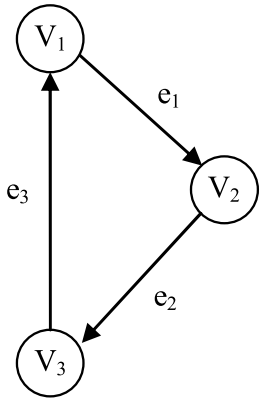


Figure 9.10a: Strongly Connected Directed Graph

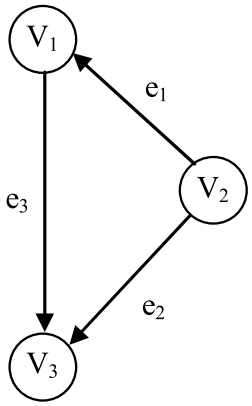


Figure 9.10b: Weakly Connected Directed Graph

Unilaterally connected graph: A digraph is said to be unilaterally connected, if there exists a path between any pair of vertices u, v in such that there is a path from u to v or a path from v to u, but not both. Figure 9.10a also is a unilaterally connected graph.

Weakly connected digraph: A digraph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any vertex from any other vertex by traversing edges in any direction. The vertices in a weakly connected digraph must have either out-degree or in-degree of at least 1. Figure 9.10b is an example of a weakly connected graph.

Lemma 2: Prove that every connected graph with n vertices has at least n − 1 edges.

Proof: This can be proved by induction

Base case: n=1

In such situation, only one vertex is there in a graph i.e. isolated vertex. There is no edge, so n-1=0.

Induction step: n≥ 2. Assume that there are n-1 number of vertices.

Let G is a graph with n vertices. Choose a vertex v from the set V and edge e from the set E where e= {v, w} a unique edge.

Remove v and e from the graph G and a subgraph G‘ is formed where

(i) G‘ is connected with only one path went to/ from v.

(ii) G‘ has no cycles

(iii) So G‘ is a graph with n-1 vertices.

By the induction hypothesis G‘ has n - 2 edges.

Then G has (n - 2) + 1 = (n -1) edges.

Simple directed graph: A directed graph G is said to be a simple directed graph if and only if it has no parallel edges. However, a simple directed graph may contain cycles with an exception that is it cannot have more than one loop at a given vertex.

The graph G is a directed graph in which there are four vertices and eight edges. Note that, the edges e3 & e5 are parallel since both begin with C and end at D. The edge e_8 is a loop since both originates at single vertex B.

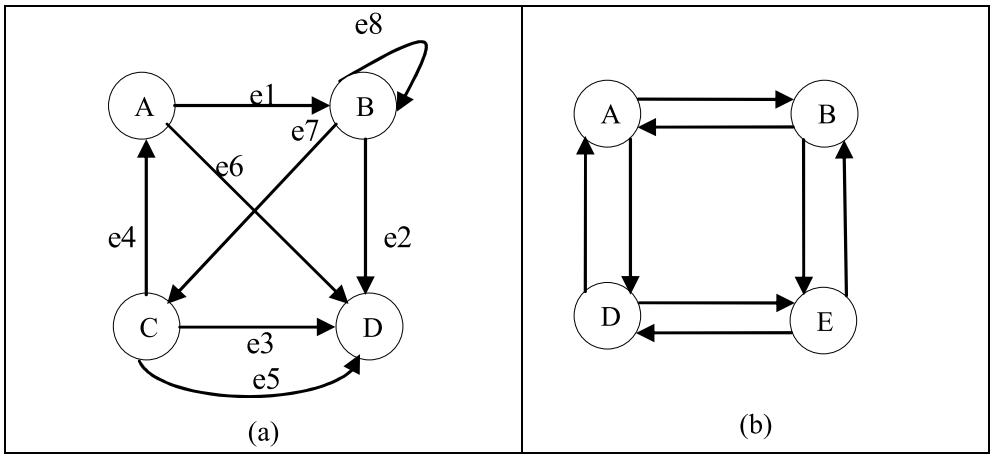


Figure 9.11: (a) Directed acyclic graph (b) Strongly connected directed acyclic graph

Biconnected Graphs

In graph theory, a **biconnected graph** is a connected and "non-separable" graph, meaning that if any vertex were to be removed, the graph will remain connected. Therefore, a biconnected graph has no articulation vertices.

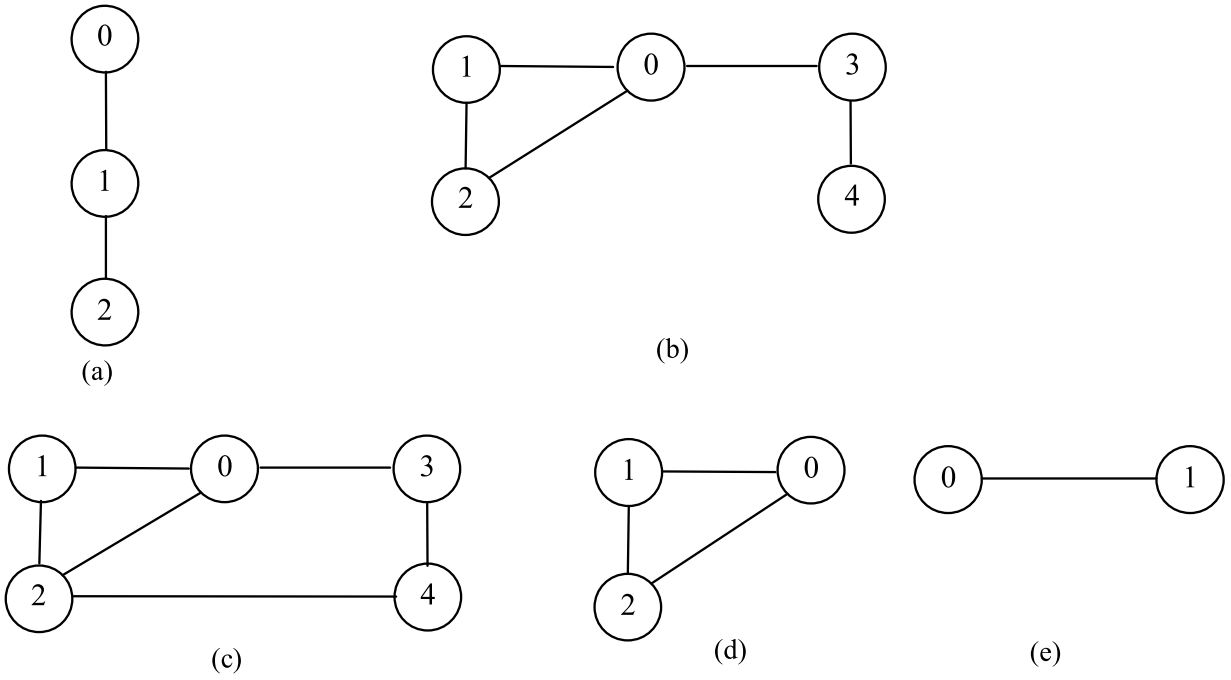


Figure 9.12: (a, b) non-Bi Connected Graph, (c, d, e) Bi-Connected

Lemma 3: There are even numbers of vertices of odd degree.

Proof: From Hand Shaking Lemma, it is proved that for a graph G = (V, E)

$\sum_{v \in G} \deg(|V|) = 2|E|$

Partitioning the vertices into those of even degree and those of odd degree, we know

$$\sum_{v \in V} \deg(v) = \sum_{d(v) \text{is even}} \deg(v) + \sum_{d(v) \text{is odd}} \deg(v)$$

By the Handshaking Lemma, the value of the left-hand side of this equation equals twice the number of edges, and so is even. On the right-hand side, the first summand is even since it is a sum of even

values. Therefore, the second summand on the right-hand side must also be even. However, since it is given that the degree of the vertices must be odd, so a number of vertices should be even.

Operations on Graph

Operations supported by a graph are as follows:

Table 9.1: Different operations on graph

Operation	Description
Create	This operation representation/storing a graph in memory.
Traverse	This operation traverse/visit all the vertices of the graph exactly once.
Insertion	This operation inserts a vertex to the graph.
Deletion	This operation removes a vertex from the graph.
Searching	This operation performs searching for a key value in the graph.

Representation of Graphs

There are three common ways of representing a graph or storing a graph in computer memory. They are:

- *Sequential representation* by using an adjacency matrix.
- *Linked representation* by using adjacency lists using a linked list.
- *The adjacency multi - list, which* is an extension of linked representation.

Adjacency Matrix

Let G (V, E) be a graph with ‗n' vertices where n >= 1. The adjacency matrix of G is a two-dimensional (n × n) array say A, with the property that A[i][j] = 1, if the edge for undirected graph (Vi, Vj) is in E(G) or for a directed graph an edge from V_i to Vj exist in E(G). A[i][j] = 0, if there is no such edges in G. Adjacency matrix represents vertex to vertex relation. The adjacency matrix is also known as bit matrix or Boolean matrix.

Adjacency Matrix representation of graphs

An adjacency matrix is used to represent which vertices are adjacent to one another. By definition, to vertices are said to be adjacent, if there is an edge connecting them.

An adjacency matrix is a way of representing n vertex graph G= (V, E) by a n×n matrix, a, whose entries are Boolean values.

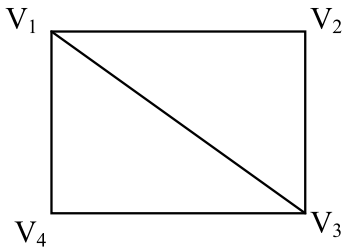
The matrix entry a[i][j] is defined as

$$a[i][j] = \begin{cases} \text{true} & \text{if } (i, j) \in E \\ \text{false} & \text{otherwise} \end{cases}$$

Adjacency Matrix Representation for an Undirected Graph (G)

In an undirected graph, the adjacency matrix A of graph G will be symmetric matrix, in which A[i][j] = A[j][i] for every i and j.

Example:



	V_1	V_2	V_3	V_4
V_1	0	1	1	1
V_2	1	0	1	0
V_3	1	1	0	1
V_4	1	0	1	0

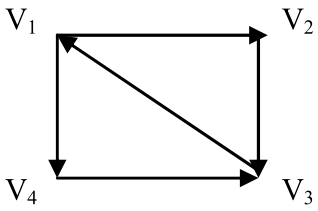
Figure 9.13: Adjacency Matrix representation of undirected graphs

Adjacency Matrix Representation for a Directed Graph (G')

The adjacency matrix of a directed graph G is defined as

$$A[i][j] = \begin{cases} 1, & \text{If Vi is adjacent to Vj} \\ & \text{i.e. if there is an edge (Vi, Vj)} \\ 0, & \text{Otherwise} \end{cases}$$

Example:



	V_1	V_2	V_3	V_4
V_1	0	1	0	1
V_2	0	0	1	0
V_3	1	0	0	0
V_4	0	0	1	0

Figure 9.14: Adjacency Matrix representation of directed graphs G'

Table 9.2: Difference between Adjacency Matrix of Directed Graph (G') and Undirected Graph (G)

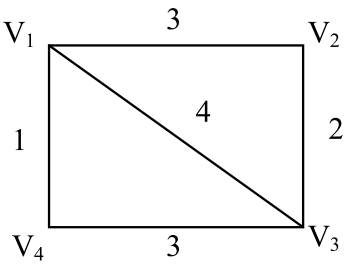
Feature	Directed Graph	Undirected Graph
Symmetric	May or may not be symmetric. If all the vertices are both ways connected, then only the adjacency matrix will be symmetric	Adjacency matrix is always symmetric
Degree of vertex	Row sum is out-degree of a vertex Column sum is in-degree of a vertex.	Both row sum and column sum are same for a vertex. Row sum is the degree of a vertex. No question of in-degree and out-degree.

Adjacency Matrix Representation for an Undirected Weighted Graph (G)

The adjacency matrix of a weighted graph G is defined as

$$A[i][j] = \begin{cases} w, & \text{If there is an edge (Vi, Vj) of weight w} \\ 0, & \text{Otherwise} \end{cases}$$

Example:



	V_1	V_2	V_3	V_4
V_1	0	3	4	1
V_2	3	0	2	0
V_3	4	2	0	3
V_4	1	0	3	0

Figure 9.15: Adjacency Matrix representation of undirected weighted graph

Advantages of adjacency matrix

The adjacency matrix is very convenient to work with. Add or remove an edge can be done in O (1) time, the same time is required to check if there is an edge between two vertices. It is very easy to implement adjacency matrix of a graph.

Disadvantages of adjacency matrix

i) Adjacency matrix consumes a huge volume of memory for storing big graph. For dense graph adjacency matrix is optimal, but for a sparse graph where not much edges are connected this matrix is not essential.

ii) The complexity of scanning an adjacency matrix during the implementation of graph searching algorithm like DFS is very high O (V2), which can be reduced to O (|V|+|E|).

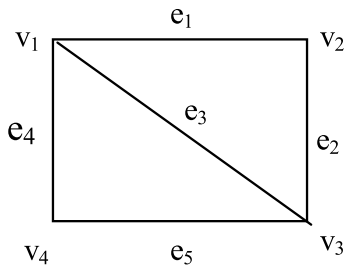
iii) To analyses, a graph using adjacency matrix is time-consuming, as we need to go through all the rows and columns.

Incident Matrix

Let G (V, E) be a graph with ‗n' vertices where n >= 1 and m edges where m >= 1.The incident matrix of G is the two-dimensional (n×m) array say A, with the property that A[i][j] = 1 if there is a relation between the edge ej and vertex vi in E(G).A[i][j] = 0, if there is no relation between the edge e_i and vertex v_i. The relation between edge and vertex is known as ―Incident" relation.

Incident Matrix Representation for an Undirected Graph (G)

Example:



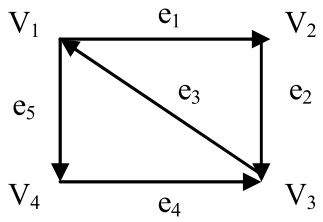
	e_1	e_2	e_3	e_4	e_5
V_1	1	0	1	1	0
V_2	1	1	0	0	0
V_3	0	1	1	0	1
V_4	0	0	0	1	1

Figure 9.16: Incident Matrix representation of undirected graphs

Incident Matrix Representation for a Directed Graph (G')

For a directed graph, the relation between vertex and edge is represented in two ways. This is because in the case of directed graphs the edges are either outgoing edge that are coming out from a vertex or incoming edge that is the edge incident on the vertex. Depending on the relation, there are two types of incident matrix.

Example:



	e_1	e_2	e_3	e_4	e_5
V_1	1	0	0	0	1
V_2	0	1	0	0	0
V_3	0	0	1	0	0
V_4	0	0	0	1	0

(a) Out-degree

	e_1	e_2	e_3	e_4	e_5
V_1	0	0	1	0	0
V_2	1	0	0	0	0
V_3	0	1	0	1	0
V_4	0	0	0	0	1

(b) In degree

Figure 9.17: Incident Matrix representation of directed graphs (a) in degree & (b) out degree

Advantage

It is easy to draw and analyze a undirected graph by the use of Incident graph.

Disadvantage

i) Require more memory space for storing a graph.
ii) For a directed graph in degree and out degree, the matrix should be computed during implementation.
iii) Time-consuming.

Adjacency Lists

In this representation, the ‗n' rows of the adjacency matrix are represented as ‗n' linked list. There is one list for each vertex in G. The vertex in the list I represent the vertex that is adjacent to vertex i. Each vertex has at least two fields, one for vertex and another for the link with next vertex. The vertex field contains the index of the vertex adjacent to vertex i.

Structure definition of a vertex in Adjacency list is:

```
struct vertex
{
    int vertex;
    struct vertex *link;
};
```

Adjacency list representation for an Undirected Graph (G)

In the case of the undirected graph with ‗n' vertices and ‗e' edges, this representation requires ‗n' head vertices and ‗2e' list vertices. The degree of any vertex in an undirected graph may be determined by just counting the number of a vertex in the adjacency list. The total number of edges in graph G may be determined in time O (n+2e). If n<<e then time complexity is O (2e) = O(e).

Example:

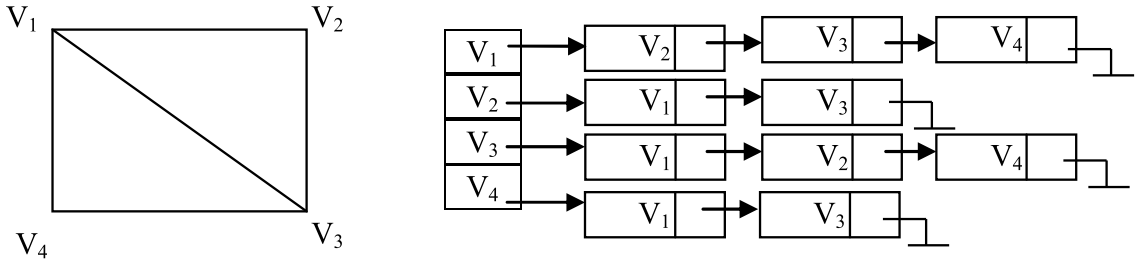


Figure 9.18: Adjacency list representation for an Undirected Graph (G)

Adjacency list representation for a Directed Graph (G')

In the case of directed graph number of head vertices _n' and a number of list vertices _e'. The out-degree of a directed graph of any vertex may be determined by counting the number of vertices on its adjacency list. The total amount of time can, therefore, be determined O (n + e). If n << e, then complexity is O(e).

Example:

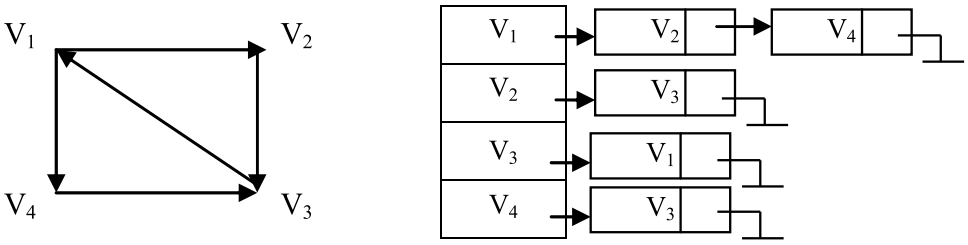


Figure 9.19: Adjacency list representation for a Directed Graph (G)

Inverse Adjacency list representation for a Directed Graph (G')

To determine the in-degree of a vertex of a directed graph another list is considered which is termed as an inverse adjacency list of a directed graph.

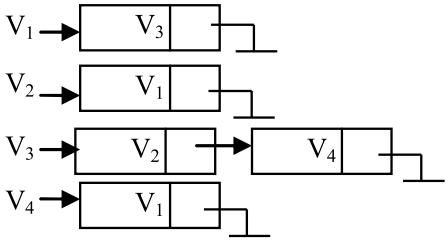


Figure 9.20: Inverse Adjacency list representation for a Directed Graph (G)

Advantages

The adjacency list allows us to store the graph in more compact form, than adjacency matrix, but the difference decreasing as a graph becomes denser. Next advantage is that adjacent list allows getting the list of adjacent vertices in O(1) time, which is a big advantage for some algorithms.

Disadvantages

i) Adding/removing an edge to/from the adjacent list is not as easy as for adjacency matrix. It

requires, on the average (|E| / |V|) time, which may result in cubical complexity for dense graphs to add all edges.

ii) The adjacency list does not allow us to make an efficient implementation if a dynamic change of vertex number is required. Adding a new vertex can be done in O (V), but removal results in O (E) complexity.

Orthogonal List

Alternatively, one could adopt a simplified version of the list structure called orthogonal list used for sparse matrix representation. Each vertex would now have four fields and would represent one edge.

The vertex structure be would

```
struct vertex
{
   int head, tail;
   struct vertex *hp, *tp;
};
```

Example:

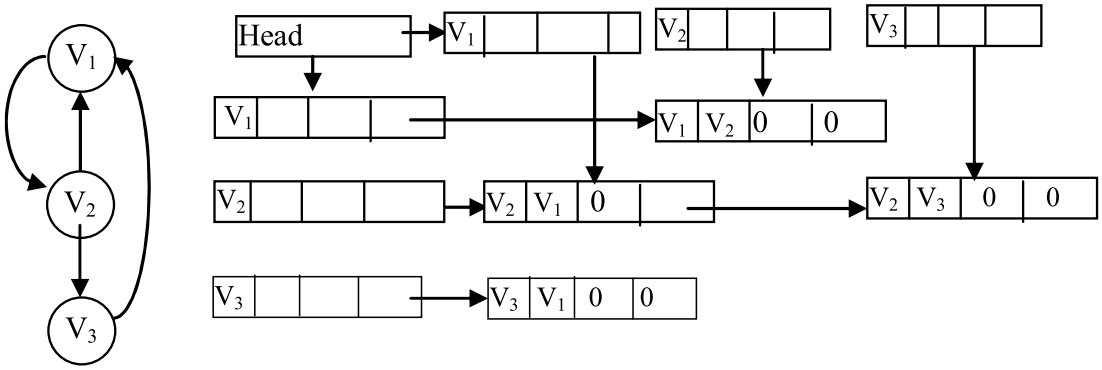


Figure 9.21: Orthogonal list representation for a Directed Graph (G)

Adjacency Multi-lists

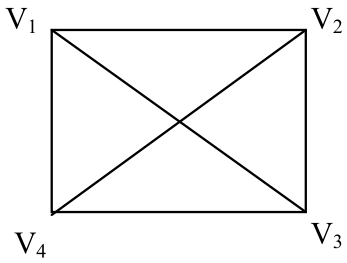
Adjacency multi-lists are an edge based, rather than vertex based in the graph representation. In the adjacency list representation of an undirected graph, each edge (v_i, v_j) is represented by two entries. One on the list for v_i and the other on the list v_j. As it is observed in some situation, it is necessary to be able to determine the second entry for a particular edge and mark that edge as already having been examined. This can be accomplished easily if the adjacency lists are actually maintained as multi-list.

For each edge, there will be exactly one vertex, but this vertex will be in two lists. The vertex structure is

```
struct vertex
{
   int m;
   int  v1, v2;
   struct vertex *p1, *p2;
};
```

Mark(m)	1st vertex(v_1)	2nd vertex(v_2)	1st path(p_1)	2nd path(p_2)

Example:



List of edges:

Edge (1, 2) → N1

Edge (1, 3) → N2

Edge (1, 4) → N3

Edge (2, 3) → N4

Edge (2, 4) → N5

Edge (3, 4) → N6

List of vertices\paths:

Vertex 1: N1 → N2 → N3

Vertex 2: N1 → N4 → N5

Vertex 3: N2 → N4 → N6

Vertex 4: N3 → N5 → N6

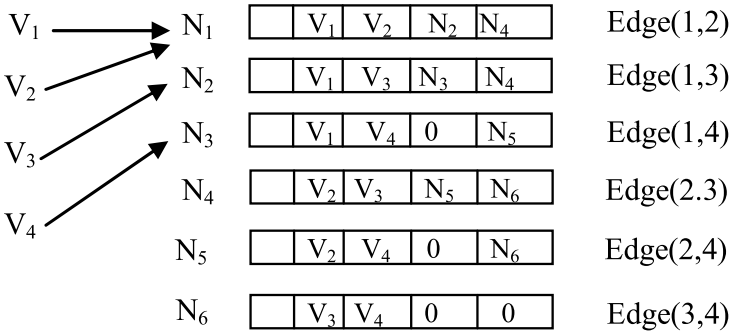


Figure 9.22: Adjacency Multi-list representation for a Directed Graph (G)

Algorithm for Adjacency Matrix Creation for A graph G

In the following algorithm, N represents a number of vertices and Adj[][] is the adjacency matrix of the graph. Adj[i][j] is set to 1 if there is an edge between the vertices i and j else it is set to 0.

Algorithm to create an adjacency matrix of a graph

```
Algorithm: CREATE (G)
[G is a given graph of N vertices]
```

```
1. Set N = number of vertices
2. Repeat For I = 1 to N
3.     Repeat For J = 1 to N
4.         If I = J then   [avoid self-loop]
                Adj[I][J] = 0
                Else
                Adj[i][j] = 1
           [End of If]
       [End of Loop]
   [End of Loop]
5. Return
```

Algorithm for Deletion of a vertex for A graph G

Following algorithm describes the procedure for deletion of a vertex from a graph. If the vertex V is to be deleted, the status of Adj[I][V] is set to 0 and the status of Adj[V][V] is set to 0.

Algorithm to delete a vertex from a graph

```
Algorithm: Delete (G)
[G is a given graph of N vertices]
1. Set N = number of vertices
2. Set V = vertex to be deleted
3. Repeat For I = 1 to N
4.     Adj[I][V]=0
       Adj[v][V]=0
   [End of Loop]
5. Return
```

Graph Traversal

Graph traversal technique is used to reach to a vertex u from a vertex v of a graph G. There are two ways of traversing the graph.

Depth First Search

In a depth first search the vertex (v) explored first when a new vertex (u) is found the exploration of vertex v is stopped and the new vertex is then explored. After an exploration of all child vertices, the parent vortex again explored.

Given an undirected graph G = (V, E) with n vertices and an array `VISITED[n]` initially set to zero, this algorithm visits all vertices reachable from s. `G` and `VISITED` are global.

Algorithm to traverse a graph using depth first search

```
Algorithm: DFS (G, s)
[G is a given graph of N vertices, s is the source node]
1. Set VISITED [s] = 1
2. Visit s
3. Repeat for each vertex v adjacent to s do
```

```
4.    If VISITED [v] = 0 then
5.        Call DFS (v)
      [End of if]
   [End of For loop]
6. Return
```

Algorithm to traverse a graph using depth first search in non-recursive/iterative process.

```
Algorithm: DFS (G, s)
[G is a given graph of N vertices, s is the source node]
1. Repeat for all vertices v of graph G do
2.    Set VISITED [v] = 0
   [End of loop]
3. Initialize Stack STK to be empty
4. Set VISITED [s] = 1
5. Call PUSH (STK, s)
6. While ISEMPTY(STK) = false do
7.    Call POP(STK, s)
8.    Visit s
9.    Repeat for all vertices v adjacent to s do
10.       If VISITED[v] = 0 then
11.          Call PUSH (STK, v)
12.          Set VISITED[v] = 1
      [End of For loop]
   [End of While loop]
13. Return
```

In case G is represented by its adjacency lists, then the vertices, w adjacent to v can be determined by following a chain of links. Since the algorithm, DFS would examine each vertex in the adjacency lists at most once and there are 2e list vertices, the time to complete the search is O (e). If G is represented by its adjacency matrix, then the time to determine all vertices adjacent to v is O (n). Since at most n vertices are visited, the total time is O (n^2).

Example:

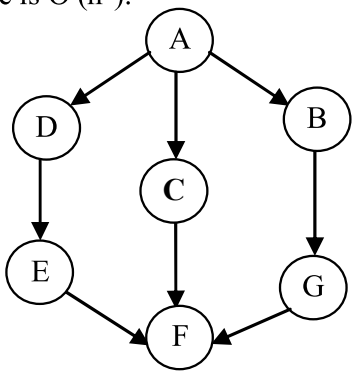


Figure 9.23: Depth First Search for a Directed Graph (G)

In the above example, the graph is traversed starting from vertex A to vertex F. In DFS recursive call is used, and recursion always uses stack data structure. So here, stack position is also given.

Table 9.2: Depth First Search

Operation	Stack	Display
PUSH A	A	-
POP A PUSH B, C, D	B,C,D	A
POP D PUSH E	B,C,E	A, D
POP E PUSH F	B,C,F	A, D, E
POP F	B,C	A, D, E, F
POP C	B	A, D, E, F, C
POP B PUSH G	G	A, D, E, F, C, B
POP G	-	A, D, E, F, C, B, G

During DFS, directed graph edges can be classified into the following:

Back edges: An edge (u, v) in which v is assumed as an ancestor of in the tree (though it may not be in proper order). Thus, a self-loop is considered as back edge.

Forward edges: An edge (u, v) in which v is a proper descendant of u in the tree. These types of edges are known as forward edges.

Cross edges: An edge (u, v) in which u and v are not an ancestor or a descendant of one another. Such type of edges is known as cross edges.

Spanning Tree of Depth First Search

In depth-first search sequence in which vertices of a graph G are visited can form a spanning tree of that graph G. Following is te DFS spanning tree of the above-mentioned graph G.

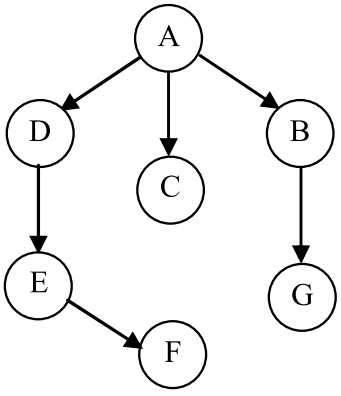


Figure 9.24: Spanning tree of Depth First Search for a Directed Graph (G)

Breadth First Search (BFS)

In breadth first search we visit a vertex (v) first and then explore all its adjacent vertices and make them visited. A vertex (v) is said to be explored when all its adjacent vertices are visited. The newly visited vertex is put into the list of unexplored vertices.

In BFS, a queue Q is used to implement unexplored vertices. A breadth first search of G is carried out beginning at vertex s. All vertices visited are marked as `VISlTED[i] = 1`. The graph G and array `VISITED` are global and `VISITED` is initialized to zero.

Algorithm of Breadth First Search (BFS)

```
Algorithm: BFS (G, s)
[G is a given graph of N vertices, s is the source node]
1. Repeat for all vertices v of graph G do
2.    Set VISITED [v] = 0
   [End of loop]
3. Initialize queue Q to be empty
4. Set VISITED [s] = 1
5. Call ENQUE (Q, s)
6. While ISEMPTY(Q) = false do
7.    Call DEQUE(Q, s)
8.    visit s
9.    Repeat for all vertices v adjacent to s do
10.       If VISITED[v] = 0 then
11.          Call ENQUE (Q, v)
12.          Set VISITED[v] = 1
      [End of For loop]
   [End of While loop]
13. Return
```

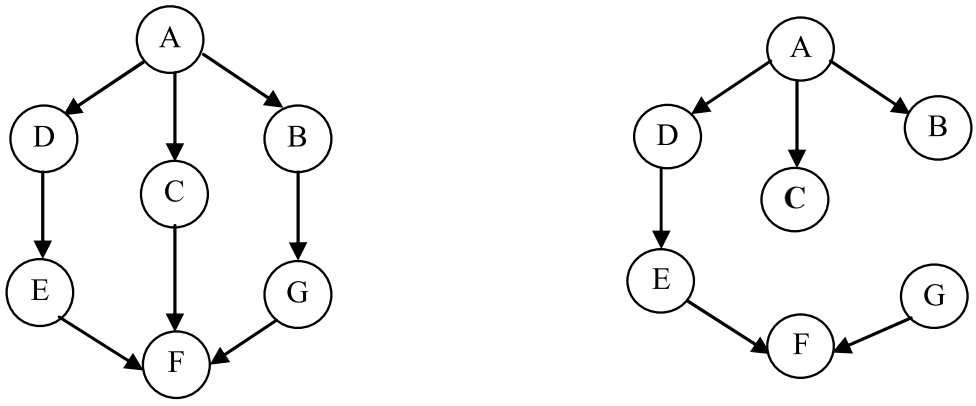


Figure 9.25: Traversal for Breadth First Search for a Directed Graph (G)

In BFS, the queue data structure is used so here in the above example, the position of the queue is given for BFS traversal.

Table 9.3: Breadth First Search

Operation	Queue(q)	Print
ENQUEUE A	A	-
DEQUEUE A ENQUEUE B, C, D	B, C, D	A
DEQUEUE B ENQUEUE G	C, D, G	A, B
DEQUEUE C ENQUEUE F	D, G, F	A, B, C
DEQUEUE D ENQUEUE E	G, F, E	A, B, C, D
DEQUEUE G	F, E	A, B, C, D, G
DEQUEUE F	E	A, B, C, D, G, F
DEQUEUE E	-	A, B, C, D, G, F, E

Spanning Tree of Breadth First Search

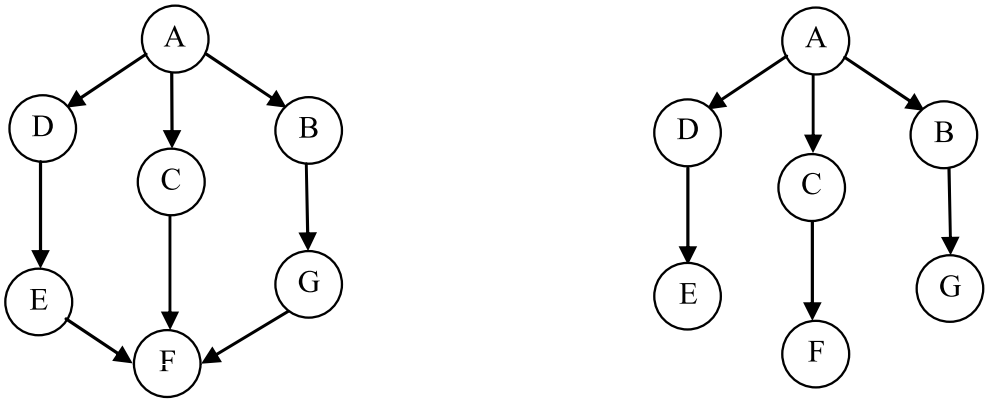


Figure 9.26: Spanning tree for Breadth First Search for a Directed Graph (G)

Lemma 4: The length of the shortest path from S to V is denoted by $\delta(S,V)$. Then, on completion of BFS dist[V] = $\delta(S,V)$

Proof: The proof is based on induction taking the length of the shortest path.

Consider the shortest path from S
to V, u be the predecessor of V and BFS processes it first among all such vertices.

Thus, $\delta(S,V) = \delta(S,u)+1$ (i)

When u is processed, then by induction we have dist[u] = $\delta(S,u)$ (ii)

Since V is a neighbour of u we set

dist[V] = dist[u]+1 (iii)

thus from equation (ii)

dist[V]= $\delta(S,u)+1$ (iv)

from equation (i) we have

dist[V]= $\delta(S,V)$

Table 9.4 Difference between Breadth First Search and Depth First Search

Feature	Breadth First Search	Depth First Search
Memory Space	More space requires	Less space requires
Backtrack	Backtracking is not required	If the wrong path is followed, then backtracking is required.
Speed	Slow searches	Fast searches
Time	We can reach goal early.	If we follow, a wrong path then more time is required to reach the goal. Else, we can reach the goal faster than BFS.
Complexity	O(V+E)	O(V+E)

Spanning Trees and Minimum Spanning Trees

A subgraph T of an undirected graph G = (V, E) is a spanning tree if it is a tree that contains all the vertices of the graph G and has no cycle.

There will be more than one Spanning tree of a graph. This is shown in the following diagram.

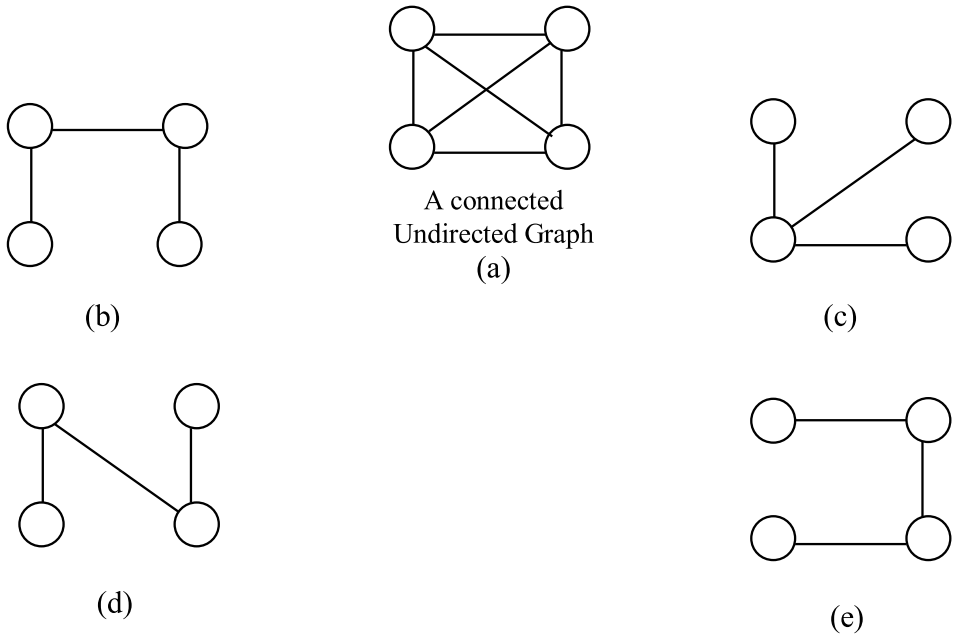


Figure 9.27: (a) An undirected graph. (b, c, d, e) different spanning tree of the graph

If the edges of a graph are weighted the graph is said to be ~~"~~weighted graph". For a weighted graph, cost of a spanning tree is measured by computing the sum of all the edges of that spanning tree.

Lemma 5: The number of spanning trees of a graph is the value of any cofactor of the Laplacian matrix of G.

Proof: Let L = L (G) be the Laplacian matrix of G with eigen values $0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$.

The (i, j)-cofactor of a matrix M is by definition

$l_{xy} = (-1)^{i+j} \det M(i, j)$,

Where M (I, j) is the matrix obtained from M by deleting row i and column j. Let l_{xy} be the (x, y)-cofactor of L. Note that l_{xy} does not depend on an ordering of the vertices of G. We set N = t(G) and show that

$N = l_{xy} = \det(L + 1/n^2 . J) = 1/n . \lambda_2 \ldots \lambda_n$ for any x, y ∈ V (G).

Let L S , for S ⊂ V(G), denote the matrix obtained from L by deleting the rows and columns indexed by S, so that $l_{xx} = \det L^{\{x\}}$. The equality $N = l_{xx}$ follows by induction on n, and for fixed n > 1 on the number of edges incident with x. Indeed, if n = 1 then $l_{xx} = 1$. Otherwise, if x has degree 0, then $l_{xx} = 0$ since $L^{\{x\}}$ has zero row sums. Now, if xy is an edge, then deleting this edge from G decreases l_{xx} by

$\det L^{\{x,y\}}$, which by induction is the number of spanning trees of G with edge xy collapsing to a point, which is the number of spanning trees containing the edge xy. This shows $N = l_{xx}$. Since the sum of the columns of L is zero, so that one column is minus the sum of the other columns, we have $l_{xx} = l_{xy}$ for any x, y.

Now, we consider the Laplacian polynomial

$\mu(G, t) = \det(tI-L) = t\Pi^{n}_{j=2} (t - \lambda_i)$ for graph G then

$(-1)^{n-1} \lambda_2 \ldots \lambda_n$ is the coefficient of t , that is

$d/dt(\det(tI-L)) = \Sigma_x \det(tI-L^{\{x\}})$

Putting, t = 0 it is obtained that $\lambda 2 \ldots \lambda n = \Sigma_x l_{xx} = nN$

Finally, the eigenvalues of $L+1/n^2.J$ are 1/n and $\lambda_2 \ldots \lambda_n$, so

$\det(L + 1/n^2 . J) = 1/ n(\lambda_2 \ldots \lambda_n.)$

Case Study:

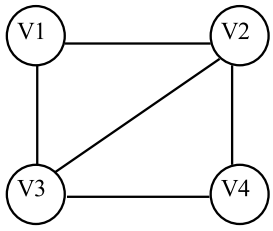


Figure 9.28: Undirected Graph G

Degree Matrix $D = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$

Adjacency Matrix $A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$ Laplacian Matrix $L = D - A = \begin{bmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ 0 & -1 & -1 & 2 \end{bmatrix}$

Compute (2. 2) cofactor of matrix D-A

(i, j) co-factor = $(-1)^{i+j}$ [det of (n-1) × (n-1) matrix obtained by removing i th row and j^{th} Column]

$l_{2,2} = (-1)^{2+2} \begin{bmatrix} 2 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 2 \end{bmatrix} = 2 \begin{bmatrix} 3 & -1 \\ -1 & 2 \end{bmatrix} - (-1) \begin{bmatrix} -1 & -1 \\ 0 & 2 \end{bmatrix} = 2 (6-1) + (-2) = 10 - 2 = 8$

A number of possible spanning tree for the above-mentioned graph G is 8.

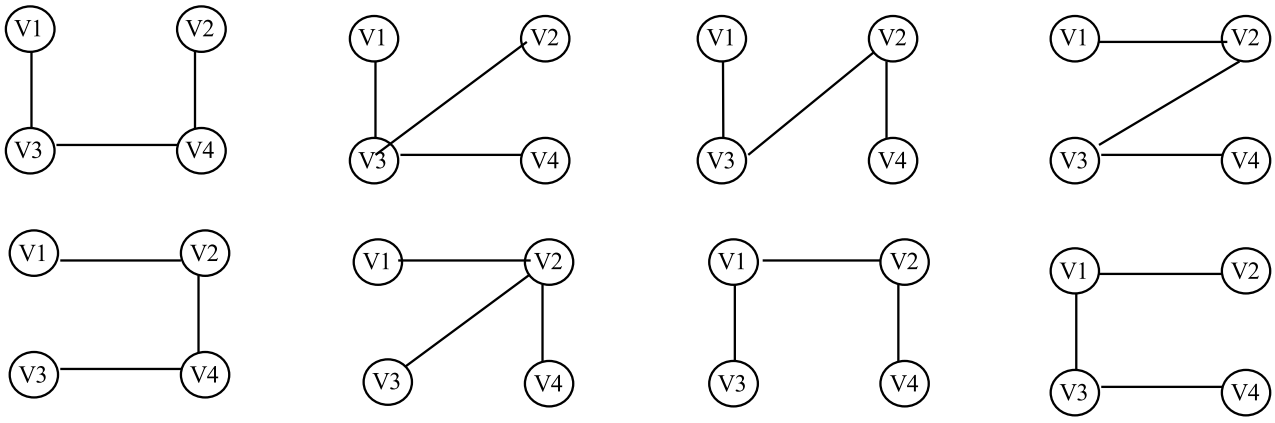


Figure 9.29: Possible Spanning Tree of Undirected Graph G

Minimal Spanning Tree

A Minimal spanning tree of a weighted graph is the spanning tree that has minimum cost. That implies the sum of the cost of all the edges is guaranteed to be a minimum of all possible spanning trees in the graph.

There are several different ways to construct a minimum spanning tree.

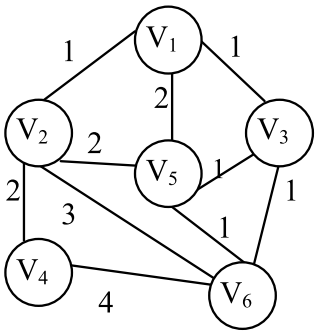
Kruskal's Algorithm

Let G = (V, E) be a graph with |v| = n. T is the set of minimum cost edges which is initially empty or null. N is the number of vertices of a graph G. Assume v and w be any two adjacent vertexes of G.

Kruskal's Algorithm for finding minimum spanning tree

```
Algorithm: MST(G)
[G is a given graph of N number of edges and V number of vertices]
1. T = NULL
2. Repeat step 3 to 5 while T contains less than (N-1) edges and
   E not empty
3.     Choose an edge (v, w) from E of lowest cost.
4.     Delete (v, w) from E
5.     If the edge (v, w) does not create a cycle in T then
            Add(v, w) to T
       Else
            Discard (v, w)
   [End of loop]
6. If T contains fewer than (N-1) edges, then there is no spanning
tree.
7. Return
```

Example :



E = {(v_1, v_2), (v_1, v_3), (v_1, v_5), (v_2, v_4), (v_2, v_5), (v_2, v_6), (v_3, v_5), (v_3, v_6)}

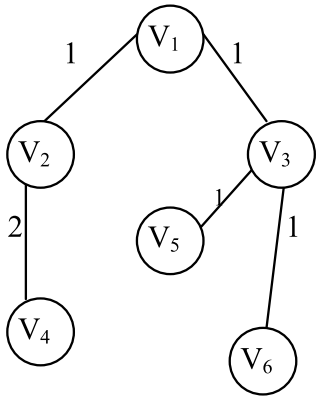


Figure 9.30: Minimal Spanning Tree of Undirected Weighted Graph G by Kruskal's Algorithm

T={(v_1, v_3), (v_1, v_2), (v_2, v_4), (v_3, v_5), (v_3, v_6)}
Minimum cost=6

Complexity of Kruskal's Algorithm is O(|E|log|E|)

Prim's Algorithm

Suppose V = {v_1, v_2, v_3, v_4…v_n} of a weighted undirected graph G = (V, E). The Prim's algorithm begins with a set V′ initialize to you, i.e. V′ = {u}. It then grows a spanning tree one edge at a time. At each step it finds the shortest edge (u, v) that connects W and (V−V′) and then adds v, the vertex in (V−V′) to V′. It repeats this step until V′= V.

E is the set of edges, which are to be extracted to obtain the minimum cost spanning tree.

Prim's Algorithm for finding minimum spanning tree

```
Algorithm: MST(G)
[G is a given graph of N number of edges and V number of vertices]
1. Initialization E′ to a null set i.e. E′= NULL
2. Select minimum cost edge (u, v) from E
3. V′ = {u}
4. Repeat step 5 to 7 until V′= V
5.   Select lowest cost edge (u, v) such that u is in V′ and
     v is in (V- V′)
6.   Add edge to set E′ i.e. E′ = E′ U {(u, v)}
```

```
7.   Add v to V′ i.e. V′ = V′ U {v}
   [End of loop]
8: Return
```

Example:

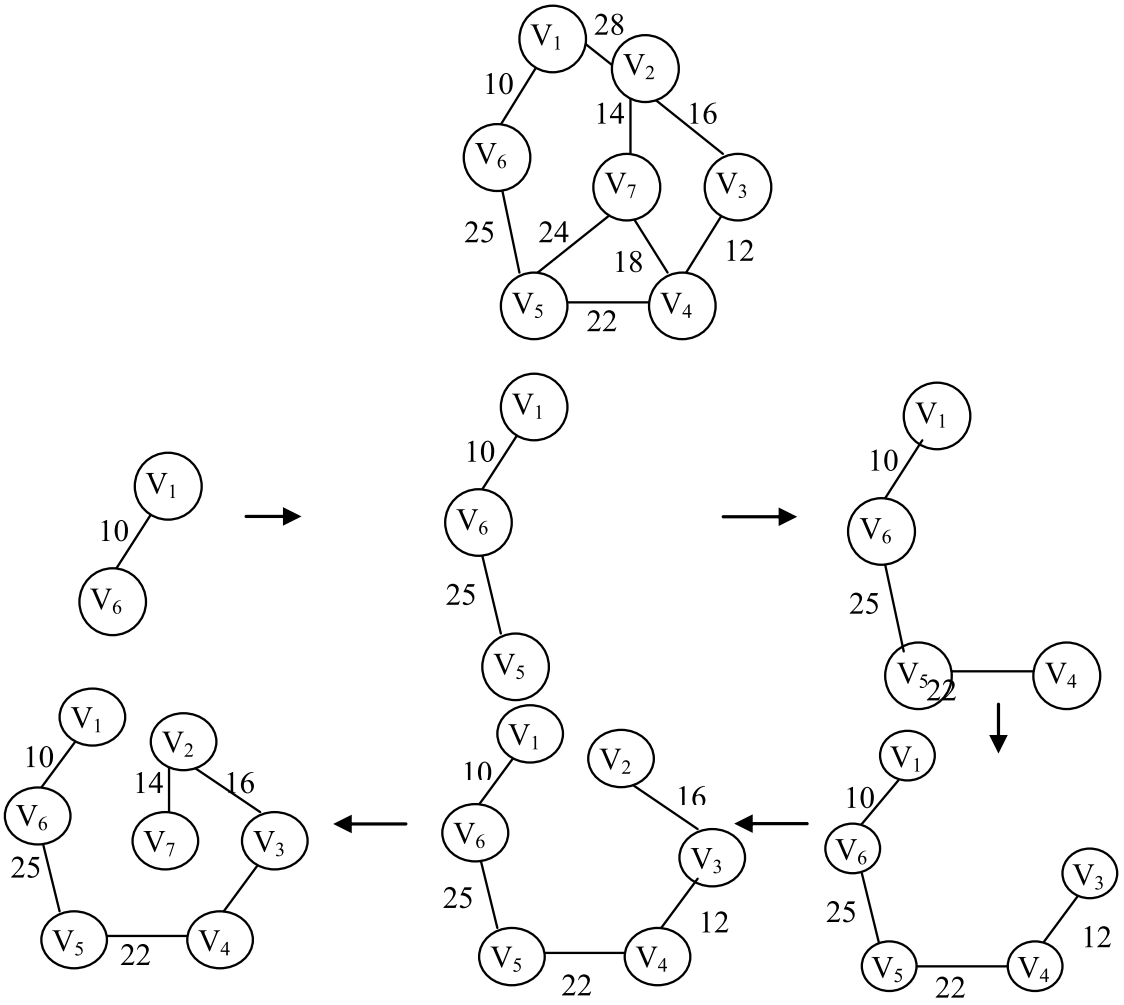


Figure 9.31: Minimal Spanning Tree of Undirected Weighted Graph G by Prim's Algorithm

Complexity of Prim's Algorithm is O (n + Elog n), where n is a number of vertices and E implies a number of edges.

Shortest Paths

Shortest path problem states a way to find the path in a weighted graph connecting two given vertices u and v with the property that the sum of the weights of all the edges is minimized over all such paths.

There are a few variants of the shortest path problem:

1. Single-source shortest-path problem
2. Single-destination shortest paths problem
3. Single-pair shortest-path problem
4. All-pairs shortest paths problem

Single-source shortest-path

The starting vertex of the path is considered to be the source(S) vertex and the last vertex is the destination vertex. There are two ways for solving single source shortest path problem.

Dijkstra's Algorithm

This algorithm finds the shortest path to a vertex to the rest of the vertices in a graph. First, it explores the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on. The algorithm both directed and undirected graphs. The one constraint for this algorithm is that all the edges must be non-negative edges.

G = (V, E) is a weighted connected graph,

W = weight matrix

S = set of visited vertices from source vertex to destination vertex, initially it is empty.

s = source vertex, dist[v] = distance of vertex v from s

Dijkstra's Algorithm for single source shortest path

```
Algorithm: Shortest_path(G, s)
[G is a given graph of N number of edges and V number of vertices, s
is the sourse vertex]
1. Initialize S = {s}
2. Initialize dist[s] = 0
3. Repeat for all v € V-{s}
     dist[v] = α
   [End of loop]
4. Repeat while S ≠ V
5.  Find a vertex w € V-S such that dist[w] is a minimum distance
6.  S=S U {w}
7.      Repeat for all v € V-S
8.            dist[v] = min(dist[v],dist[w] + W(w,v))
     [End of loop)
   [End of loop]
9. Return
```

Example:

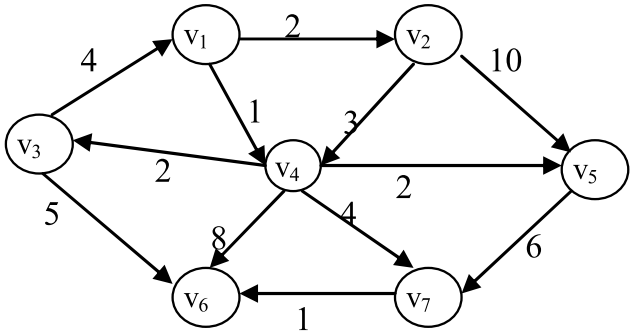


Figure 9.32: Shortest Path Using Dijkstra Algorithm

Table 9.5: Shortest Path By Dijkstra

Iteration	S	W	dist[w]	dist[v2]	dist[v3]	dist[v4]	dist[v5]	dist[v6]	dist[v7]
	{ v1 }	-	-	∞	∞	∞	∞	∞	∞
	{ v1}	-	-	2	∞	**1**	∞	∞	∞
1	{ v1, v4}	v4	1	**2**	3	1	3	9	5
2	{ v1, v4, v2}	v2	2	2	**3**	1	3	9	5
3	{ v1, v4, v2, v3}	v3	3	2	3	1	**3**	8	5
4	{ v1, v4, v2, v3, v5}	v5	3	2	3	1	3	8	**5**
5	{ v1, v4, v2, v3, v5, v7}	v7	5	2	3	1	3	**6**	5
6	{ v1, v4, v2, v3, v5, v7, v6 }	v6	6	2	3	1	3	6	5

Bellman Ford Algorithm

In this algorithm authors Richard Bellman, Samuel End and Lester Ford. Jr. have considered negative weighted edge that was not focused in Dijkstra's Algorithm. The algorithm finds the shortest path repeating a vertex of a graph. G=(V, E) is a directed graph with source S and weight function W:E->R. This algorithm returns a Boolean value TRUE if and only if the graph contains a negative weight cycle that is reachable from the source.

Bellman Ford algorithm for single source shortest path

```
Algorithm: Shortest_path()
[G is a given graph of N number of edges and V number of vertices, s
is the sourse vertex]
1. Repeat step 2 for all vertices
2.    dist[i] = cost[v, i]
   [End of loop]
3. Repeat step 4-6 for k = 2 to n-1 do
4.  For each u such that u≠v and
    u has atleast one incoming edge do
5.       For each (i,u) in graph do
6.          if dist[u]>dist[i]+cost[i,u] then
                dist[u]=dist[i]+cost[i,u]
7. Return
```

Example:

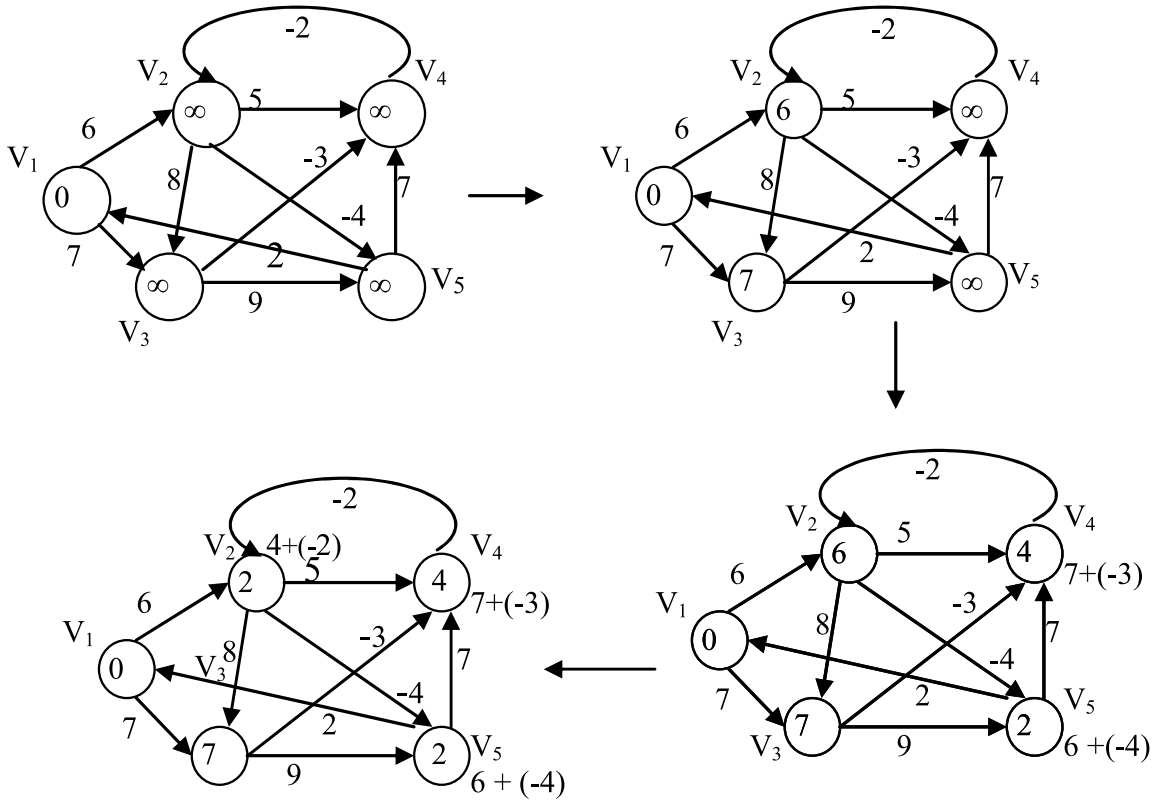


Figure 9.33: Shortest Path Using Bellman Ford Algorithm

The shortest path from source vertex v1 is v1 → v3 → v4 → v2

Another shortest path from source vertex v1 is v1 → v2 → v5

The complexity of this algorithm is O (VE). Since initialization takes O(V) and traversing each edge associated with the vertex O(E). So, total complexity O(VE)

All Pairs Shortest Paths

Consider a weighted digraph G = (V, E). Solving All Pair Shortest Path problem consists of computing the shortest distance or path in terms of cost/weight between every pair of vertices(u, v) where u, v∈V in the graph G. It is assumed that the graph does not contain any negative or zero weight cycle.

Floyd-Warshall Algorithm

Robert Floyd and Stephen Warshall had invented this algorithm. In a single execution of the algorithm, the shortest path between all pairs of vertices is discovered. This algorithm is based on dynamic programming.

Cost[1:n,1:n] is the cost adjacency matrix of a graph with n vertices. A[i, j] is the cost of a shortest path from vertex i to vertex j. cost[i,j]=0, for 1<=i<=n.

Floyd-Warshall Algorithm for all pairs shortest path

```
Algorithm: Shortest_path(A, cost)
[A[i, j] is the cost of a shortest path from vertex i to vertex j]
```