Unit 3: Graph Algorithms-1

Structure

3.0 Introduction

Graphs are most widely used mathematical structure. It is widely used in finding shortest path routes, shortest path between every pair of vertices, in computing maximum flow problem which has applications in a large range of problems related to airlines scheduling, maximum bipartite matching and image segmentation.

A graph can be used to model a social network which comprises millions of users or interest groups which can be represented as nodes. There are interdependencies among nodes through mutual interests and common friends. Many algorithms used in social networks are based on graph algorithms like Facebook's friends suggestion algorithm, Google's page ranking algorithm, Linkdn's suggestion to join a group etc.

3.1 Objectives:

After successful completion of this unit, the students will able to:

- Define different types of graphs
- Represent a graph through an adjacency matrix and an adjacency list and calculate complexities of each
- Write pseudo-codes for graph traversal techniques like breadth first search and depth first search and measure their time complexities
- Define directed acyclic graph , topological ordering and connected components
- Write pseudo codes for topological ordering and connected components

3.2 Basic Definition and Terminologies

Graph:

A graph G = (V,E) is a data structure comprising two set of objects, V={v_1,v_2,……} called vertices, and an another set E={e_1,e_2,……} called the edges.

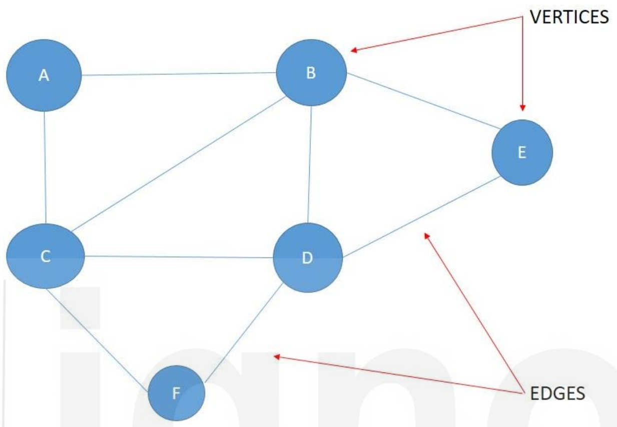


Figure 1: A graph

In the above graph set of vertices V = {A,B,C,D,E,F} and set of edges E = {A-B , A-C, B-E ,B-D,B-C, C-D, C-F, D-F, D-E,}. Vertices are unordered set of nodes V. **Edge** = An edge is identified with an unordered pair of vertices (v_i,v_j), where v_i and v_j are the end vertices of the edge e_k

Graph Types:

Though a graph contains only vertices and edges, but there are many variations in them. Most of the variations are due to edges (directed / undirected, no of edges). In the following such graph types are listed with examples.

1. Simple Graph: A simple graph (figure 2) is a graph in which each edge is connected with two different vertices and no two edges are connected with same vertices. In this there is no self-loop and no parallel edges in a graph. A simple graph may be connected or disconnected. Basically the un-weighted graphs are simple graphs. A simple graph with multiple edges is called multigraph (figure 3).

Example:

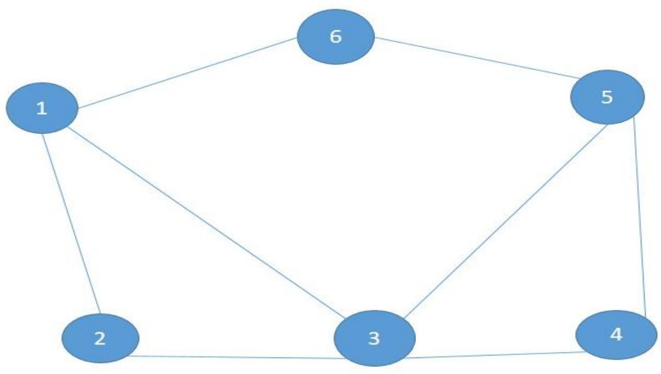


Figure 2: Simple Graph

The above graph is a simple graph, since no vertex has a self-loop and no two vertices have more than one edge connecting them.

Example: Multi graph

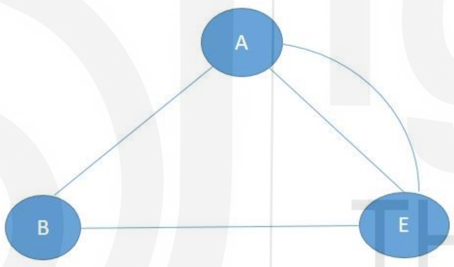


Figure 3: Multi Graph

1. **Undirected Graph:** A graph in which the edges do not have any direction and all the edges are in bi-direction (figure 4). In undirected graph if there is an edge from u to v then we can move from node u to node v and as well as from node v to node u. In this nodes are unordered pairs. We can transform the undirected into directed graph by separating the edges between two edges.

Example:

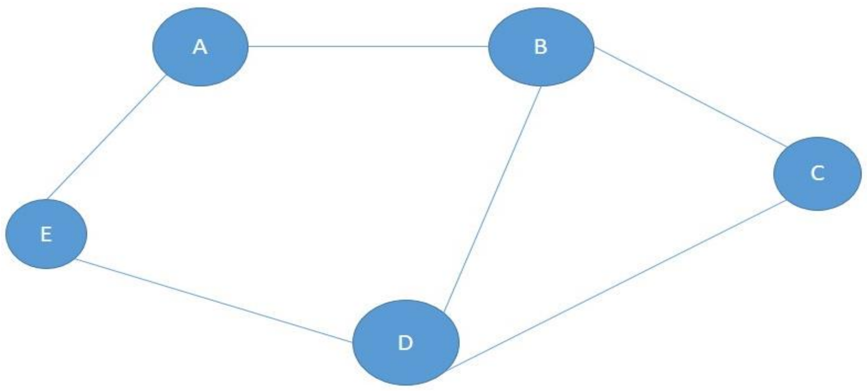


Figure 4: Undirected Graph

In the above undirected graph we can traverse through following paths:

From A: A – E, A – B
From B: B – A, B – D, B – C
From C: C – B, C –D
From D: D – E, D –B,D –C
From E: E –A, E –D

1. **Directed Graph:** A graph in which the edges have direction (figure 5). It is also called digraph. This is usually indicated with an arrow on the edge. In a directed graph if there is an edge from u to v then we can move from a node u to a node v only .With the help of directed graph we can represent asymmetrical relationships between nodes , roads network , hyperlinks connecting web pages etc.

Example:

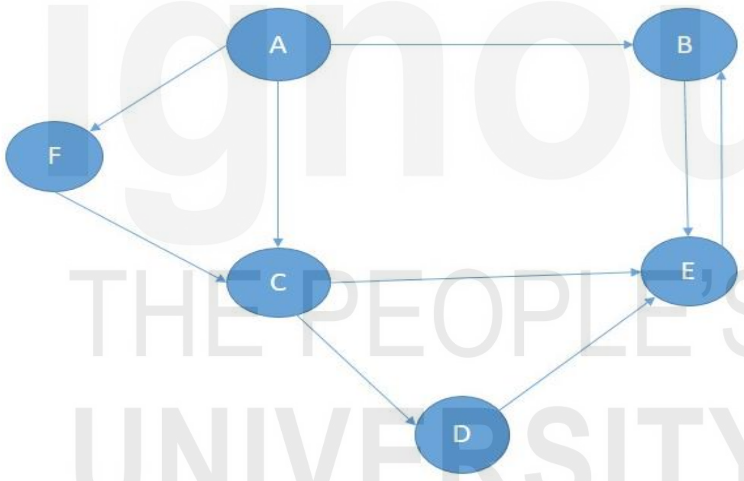


Figure 5: Directed Graph

In the above directed graph we can traverse through following paths:

From A: A – F, A – C, A –B
From B: B – E
From C: C – E, C – D
From D: D – E
From E: E – B
From F: F – C

2. **Subgraph:** A graph whose vertices and edges are subsets of a another graph. It is not necessary that a subgraph will have all the edges of graph.

Example:

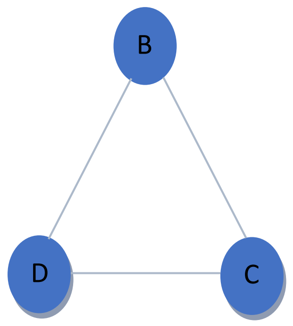


Figure 6: A sub graph of a figure-4

This is a subgraph of the graph which has the nodes A, B, C, D. In this there are C, B, D nodes.

1. **Connected Graph:** A directed graph in which there is a path between each pair of vertices in a subset. An undirected graph is said to be connected if for every pair of two different vertices v_i , v_j, there is a path between these two vertices. The graph G is connected whereas G_2 is not connected

Example:

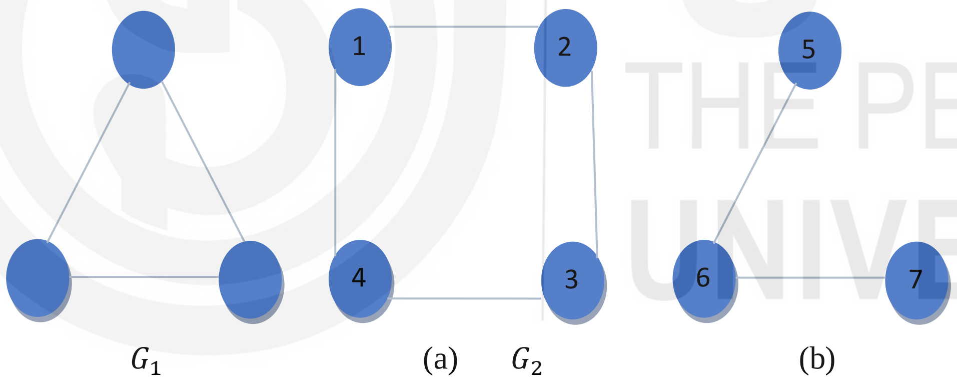


Figure 7: Connected graph

3.3 Graph Representation

The purpose of graph representation is typically to search a graph most systematically such that the edges of the graphs can be used to effectively visit all the vertices of it. In order to have an effective search algorithm, the logical representation of the graph plays a very critical role. When it comes to representations of graphs, two most standard and common computational representations are in practice such as Adjacency Matrix and Adjacency List. Apart from these other additional representations such as Linked lists, contiguous lists and combinations are also used in fewer cases. The selection of a particular

representation depends on applications and functions one wants to perform on these graphs.
In case the graph is sparse for which the number of edges also written as |E| is quite less than $|v|^2$, adjacency list is preferred but if the graph is dense where $|E|$ is close to $|v|^2$ an adjacency matrix is selected.

3.3.1 Adjacency Matrix:

Adjacency matrix representation is typically used to represent both directed and undirected graphs, where the concentration of nodes is a graph is dense in nature. Usually in such graphs |E| is much close to $|V|^2$ These adjacency matrix are typically drawn for Directed Acyclic Graph (*DAG)*, where we have no loops or double directions.

- *Adjacency matrix A is a square matrix which is used to represent a finite graph. The matrix elements shows whether pairs of vertices are adjacent (connected) or not in the graph. It is 2 dimensional array , which have the size V x V , where V are the numbers of vertices present in graph* otherwise it
  $A[i,j] = 1$ if there is edges between A_i & A_j
  $A[i,j] = 0$, otherwise

Adjacency Matrix of the following graph( ) is given below.

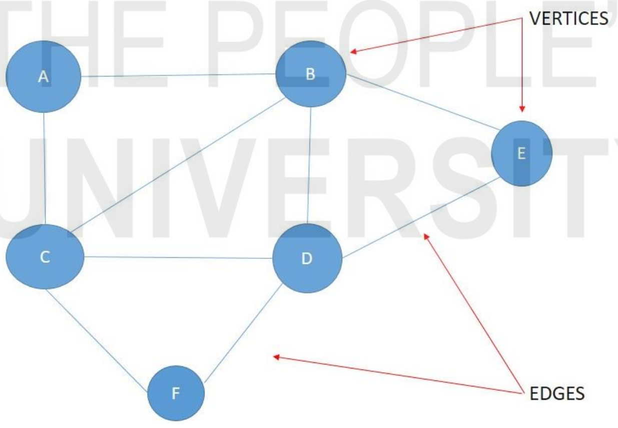


Figure 8: Example graph for matrix representation

In the given graph, if there is a link between the vertices we mark as‘1’ in the adjacency matrix, else we mark as ‘0’.

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	1	1	0
C	1	1	0	1	0	1
D	0	1	1	0	1	1
E	0	1	0	1	0	0
F	0	0	1	1	0	0

Figure 9: Adjacency matrix representation of a graph given at figure 8

It is easy to determine if there is an edge between two vertices in a graph if it is represented through adjacency matrix. To find out how many edges are in a graph, it will require at least $O(v^2)$ time as there are v^2 entries of matrix have to be examined except the diagonal element which are zeroes. But when the most of the enteries are O_s (in case of a sparse graph) then it might take $O(e + v)$ where $e << v^2$ which is significantly less time.
Adjacency matrix for a directed graph may or may not be symmetric but in case of undirected graph. It is always symmetric
Space complexity $= O(v^2)$ time complexity. Where V is the number of vertiex is independent of a number of edges.

3.3.2 Adjacency List

Adjacency list representation is typically used to represent graphs, where the number edges |E| is much less than $|V|^2$. Adjacency list is represented as an array of |V| linked lists. There is one linked list for every vertex node in a linked list ί Each Node in this linked list is a reference to the other vertices which share an edge with the current vertex. The following figure,(b) shows adjacency list of a given graph.

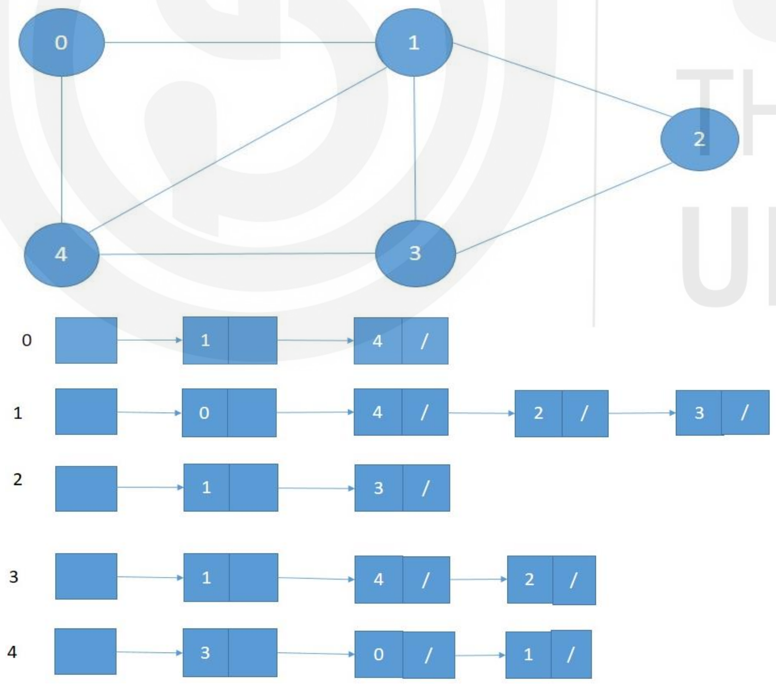


Figure 10 : An adjacency list of the above graph

If there is undirected with V vertices and E edges adjacency list representation requires |V| head nodes and 2|E| adjacency list nodes. In an undirected edges (ί, ȷ), ί appear in ȷ's adjacency list and ȷ appears in ί's adjacency list. In terms of memory requires it is O$(V + E)$ for both types of graphs representations: adjacency matrix and adjacency list.

3.4 Graph Traversal Algorithms

Traversal algorithms are used to navigate across a given graph among all the nodes using all possible vertices. These algorithms will help us in finding the nodes, making paths that are shortest or feasible or prioritized in nature. Graph traversal can be carried out using breadth or depth as a criteria. In the data structure course we have learnt to traverse a tree in preorder, inorder and postorder. One encounters a similar problem of traversing a graph: given a graph $G = (V, E)$ one wants to visit all the vertices in G from a given vertex. There are two key graphs traversal Depth First Search (DFS) & Breadth First Search (BFS) algorithms. In the next section we will examine both algorithms.

3.4.1 Depth First Search (DFS)

DFS starts with any arbitrary vertex in a graph and traverses to the deepest node as far as possible and then backtracks. The algorithm proceeds as follows: it starts with selecting any arbitrary vertex as start vertex then traverses the next node w adjacent to the starting vertex v. The process of traversing the unexplored nodes adjacent to the previous node w and continues till it finds that no adjacent node is left to be examined. If the last vertex is u which does not have any adjacent vertices, it backtracks to w and explores the unvisited adjacent nodes.

The traversal process terminates when all the vertices are traversed. Recursive implementation of DFS pseudo-code is given below.

```
Function DFS( V)   // V is a starting vertex of a graph
Input $G(V,E)$, visited [$n$], visited[n]    // visited[n] = 0
{
visited [$v$] = 1
for each vertex $w$ adjacent to $v$ do
{
If visited [$w$] = 0 // if vertex $w$ is not yet visited
                DFS(w);             // Recursive call to DFS
}
} - // end of DFS (  )
```

Figure 11: pseudo-code of DFS

Let us apply the pseudo-code of DFS to the following example graph

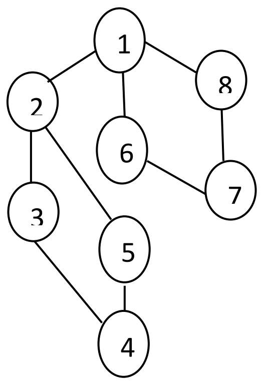


Figure 12 : An example graph for DFS traversal

Output of DFS traversal of a graph at figure 12: 1, 2, 3, 4, 5 ,6, 7, 8

What is the running time of DFS ?
Suppose $G(V, E)$ is an undirected graph with v number of vertices and E edges and represented through adjacency list, then a sum of the length of all the adjacency list is $2|E|$.
DFS visits each node only once in the adjacency list. Therefore the time to complete visiting all edges and the associated vertices is O(V+E). If a graph is represented through its adjacency matrix, the time to determine all vertices which are adjacent to the starting vertex v is V. Since at most V vertices are visited, the total running time is O(V^2)

3.4.2 Breadth First Search (BFS)

BFS is a graph searching algorithm which start from any arbitrary vertex as a starting vertex and visits all its adjacent vertices first and then moves at the second level to visit all the unvisited vertices which are adjacent to vertices at the first level vertices and so on. BFS has become a basis for the development of many other graph algorithms such as Dijkstra's single source shortest path algorithm and Prim's minimum cost spanning tree problem. BFS algorithm starts with any arbitrary vertex 1 and then visits all its adjacency vertices 2,6,8(i.e nodes at the first level) first instead of discovering the deepest node as in DFS. At the next level (i.e the second level) it will visit the adjacent vertices of 2,6 and 8 respectively. The adjacent vertices of 2 are 3 and 5, the adjacent vertex of 6 and 8 is 7 respectively. Finally it will visit 4 which is a common adjacent vertex of 3 and 5 vertices.
The final output of BFS traversal of a graph is: 1, 2,6,8,3,5,7,4
The following algorithm describes the implementation details of BFS. Queue is a main data structure used to implement the algorithm

Pseudo code:

```
function BFS (v)     // v is a starting  node of a graph
1.    let Q be queue data structure .
2.     Q.insert(v)//Insert s in queue until all its neighbour vertices are marked.
3.     Visited[v] =1  // Visited[ ] will mark a vertex '1' if it is visited, otherwise
it is 0
4      print Q
4.    while( Q is not empty)
5.    //Remove that vertex from queue,whose neighbour will be visited now
6.        Q.remove()

      //processing all the neighbours of v
7.    for all neighbours w of v in a Graph G
8.    if w is not visited
9.    Q.insert(w)//Stores w in Q to further visit its neighbor and  mark as
visited.
10     Visited[ w] = 1
11        Print Q.
```

Figure 13: BFS Pseudo-code

Step wise explanation of Algorithm:

1. Initially visited [ ] and Q(which represents a queue data structure)are empty because all vertices are unvisited
   visited[ ] =
   Q = Nil
2. All visited vertices are marked as ' 1'
3. BFS is initiated with a starting vertex 1 in the graph (figure) and marked as '1' in visited array and entered in Q.
   visited [] =
   Q =
4. Print the visited vertex '1'(starting vertex) and remove it from Q
   visited[ ] =
   print: 1
   Q :
5. Insert non-visited adjacent vertices 2, 6,8 of the starting vertex 1 in Q and mark them as '1' in visited array
   Visited[ ] =
   Q :
6. Insert adjacent vertices 3 and 5 of 2 in Q and mark them as '1'in visited[ ]
   visited [ ] =
   Q :
7. Print 2 and remove it from Q
   visited [ ] =

Print: 2
Q : 6,8,3,5

1. Insert a common vertex 7 of vertices 6,8 in Q and mark them as ‘1’ in visited [ ].
   visited[ ]=
   Q : 6,8,3,5,7
2. Print 6 and 8 and remove 6 and 8 from Q
   Visited[ ] =
   Q : 3,5,7
   Print : 6,8
3. Finally we insert 4 in Q which is adjacent to vertices 3 and 5 and mark it as ‘1’ in
   visited[ ].
   visited[ ] =
   Q : 3,5,7,4
4. Print 3,5,7 and remove it from Q
   Visited[ ]=
   Q : 4
   Print: 3,5,7
5. All vertices have been visited and vertex 4 does not have any adjacent node. So print 4 and remove it from Q
   Visited [ ]=
   Q : Nil
   Print: 4

The final output of a BFS traversal is 1,2,6,8,3,5,7,4

Complexities:

Time complexity: O(V + E), where $O(V)$ is a total time taken to complete queue operations(insertion and deletion of vertices . Insertion and deletion of a single vertex takes O(1) unit of time . Since there are V number of vertices in the graph, it will take O(V) time. The time taken to traverse each adjacency list only once is O(E) . Therefore the total time is O (V + E)

Check Your progress -1

Q1 Which is the most suitable graph representation scheme for a sparse graph?
Q2 What is a simple graph?
Q3 How does BFS graph traversal scheme work?

3.5 Directed Acyclic Graph and Topological Ordering

A directed graph without a cycle is called a directed acyclic graph (or a (DAG (for short) which is a frequently used graph structure to represent precedence relation or dependence is a network. The following is a example of a directed acyclic task graph

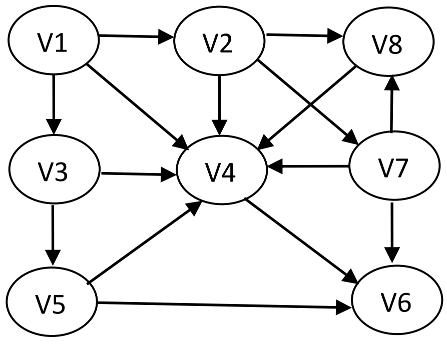


Figure 14: Directed Acyclic Graph

In this graph except vertex V1, all other vertices are dependent upon other vertices.

Any major task can be broken down into several subtasks. The successful completion of the task is possible only when all the subtasks are completed successfully. Dependencies among subtasks are represented through a directed graph. In such representation , subtasks are represented through vertices and an edge between two vertices define a precedence relation. After showing dependency, the next task is to ordering is to ordering of these subtasks for execution . This is also called topological sorting or topological ordering

Topological sort or ordering of a $DAG\ G = (V, E)$ is a liner ordering of its vertices $V_1, V_2 \dots . V_n$ so that if a graph G contains an edge $from\ v_i\ to\ v_j$ then v_i comes before v_j in the ordering. In other words, all edges between vertices representing tasks show forward direction in ordering or sorting. It the graph G contains a cycle then no topological order is possible.

Therefore, if a graph G has a topological sort, then G is a DAG.

Proof- Applying contradiction that G has a typological sorting of all its vertices (tasks): $V_1, V_2 \dots . V_n$ and G is also having a cycle. let all the vertices be indexed : $V_1, V_2 \dots . V_n$. let V_i be the lowest index in topological ordering. let there be an edges $(V_j, V_i) in$ a cyclic graph G. If $V_j > V_i$ i.e. V_j, comes before V_i which contradicts that all the vertices $V_1, V_2 \dots . V_n$ are topological sorted is G.

The following is a pseudo-code topological for computing a DAG.
Pseudo-code to compute topological sorting

```
Function topological_sort(G)
Input  $G = (V, E)$ //$G$ is a DAG.
{
search for a node $V$ with $zero$ in-degree (no incoming edges) and order it first in topological sorting
remove V from G
topological_sort(G-{V})  // recursively compute topological sorting
append the ordering
}
```

Figure 15: Pseudo-code for topological sorting

Topological sorting time complexity

Finding out a vertex with no incoming edge, deleting it from a graph and appending it in the linear ordering would take O(n) time. Since there are n number of vertices is G, there will be n times loop, the total time will be $O(n^2)$. But if the graph is sparse where the $|E| << n^2\ and$ the graph is represented through an adjacency list it is possible to have $O(m+n)$ time complexity, where m is |E|.

Illustration of an example

The following figure show the application of the algorithm to the example given in the figure 15

Step 1: V_1 - does not have incoming edges so it is deleted first.

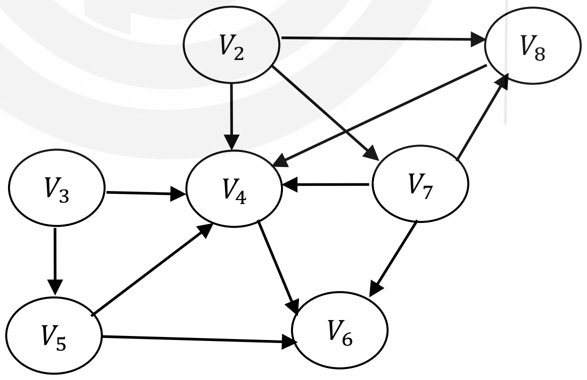


Figure 16 (a) : Deletion of V_1

Step 2 : In the graph there are the two nodes V_2 & V_3 with no incoming edges. One can pick up either of the two vertices. Let us remove V_2 and append it after V_1, i.e., V_1 , V_2

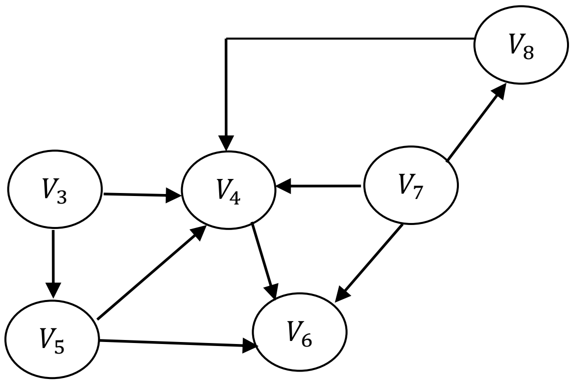


Figure 16(b): Deletion of V_2

Step 3: - V_3 is the only node with no incoming edge. Therefore V_3 will be removed and appended after V_2 , $i.e.$, $V_1V_2V_3$

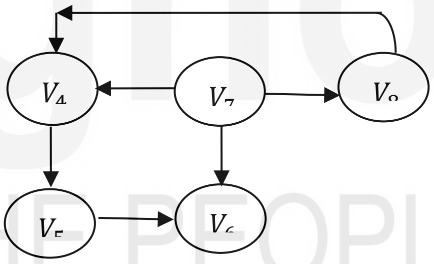


Figure 16(c): Deletion of V_3

Step 4 : V_5 is the only node with no incoming edge. V_5 will be removed and appended to the list ,i.e.,

V_1 V_2 V_3 V_5

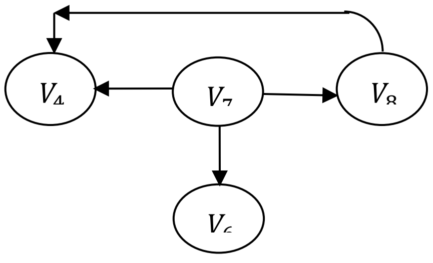


Figure 16(d): Deletion of V_5

Step 5 : V_7 is the only node with no incoming edge. Therefore V_7 will be removed and appended, i.e.,
V_1 V_2 V_3 V_5 V_7

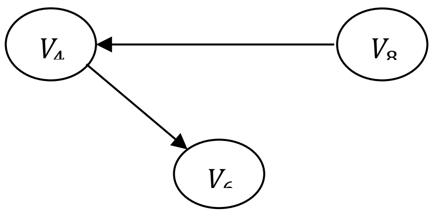


Figure 16(e): Deletion of V_7

Step 6: - V_8 will be removed and appended, i.e., V_1 V_2 V_3 V_5 V_7 V_8

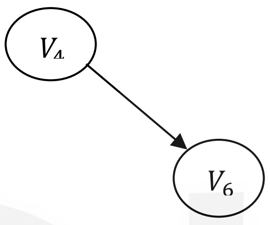
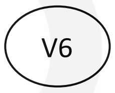


Figure 16(f) : Deletion of V_8

Step 7: V_4 will be removed and appended as: V_1, V_2, V_3, V_5, V_7 , V_8, V_4



Step 8: Finally V_6 will be appended: V_1, V_2, V_3, V_5, V_7 , V_8, V_4, V_6

The linear ordering of vertices is shown in the following figure:

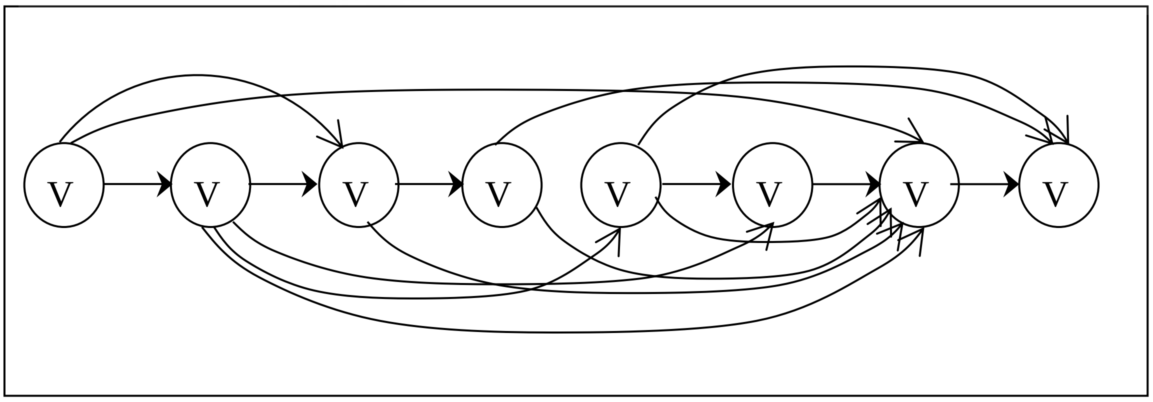


Figure 17: Topological Sorted order of vertices

3.6 Strongly Connected Components (SCC):

In this section we define two terms: strongly connected and strongly connected components of a directed graph then we apply an algorithm to find out whether a given directed graph is strongly connected component or not. A directed graph $G = (V, E)$ is strongly connected if for every two vertices ν_i and ν_j there is a pair of edges from ν_i to ν_j and ν_j to ν_i

Strongly connected components is a maximal set of vertices $M \subseteq V$ such that every pair of vertices in M are mutually reachable. The following figure is an example of strongly connected components of a graph *G*.

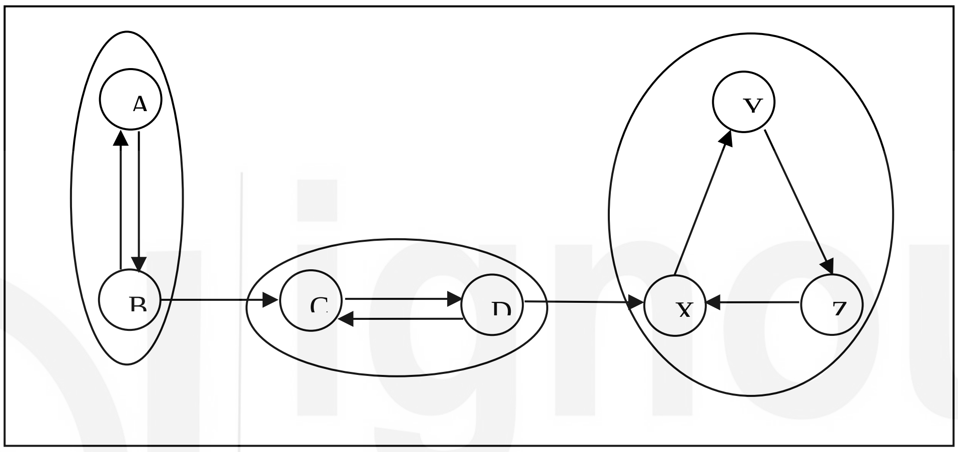


Figure 18:

In the above graph *G* there are 3 subsets of vertices, i.e., AB and CD and XYZ which are mutually reachable.

Pseudocode of Strongly Connected Components

- Perform DFS (Depth First Search) on the whole graph *G*
- Transpose of the original *G* (i.e. G^T)
- Perform DFS (Depth First Search) on $G^T = (V, E^T)$
- Print the final result.

Figure 19 : Strongly connected components

The prospered pseudocode for finding strongly connected components of a $G = (V, E)$ uses the graph traversal scheme DFS first and then performs transpose of a graph G, i.e.,$G^T (V, E^T)$ with all the edges of *G* reversed. E^T consists of all the edges in reverse direction of *G*.

It is to be observed that a graph *G* and its transpose G^T have exactly the same strongly connected components.

Time Complexity :- It a graph is represented through adjacency lists, time to create G^T is O(V + E)

Check your progress -2

Q1. Define topological ordering.

Q2 If a graph G has a topological sort, then G is a DAG. Write a proof.

3.7 Summary

A graph is a very frequently used data structure for many basic and significant algorithms. There are two standard approaches to represent a graph: Adjacency matrix and adjacency lists which can be used to represent both directed as well as undirected graph. When the graph is a sparse, adjacency list is preferred because it is a more compact way to represent it. Many graph algorithms are based on the assumption that the graph is represented through an adjacency list. BFS and DFS are two simplest searching algorithms. BFS is a basis of Prim's minimum cost spanning tree problem and Dijkstra's single source shortest path algorithm. The unit also describes two applications of DFS: (i) directed acyclic graph and topological sorting (ii) decomposing a graph into strongly connected components.

3.8 Solution to Check Your Progress

Check Your progress -1

Q1 . Which is the most suitable graph representation scheme for a sparse graph?
Ans. Adjacency list provides a compact way to represent a sparse graph
Q2 What is a simple graph?
Ans. A simple graph is a graph in which each edge is connected with two different vertices and no two edges are connected with same vertices. In this there is no self-loop and no parallel edges in a graph. A simple graph may be connected or disconnected. Basically the un-weighted graphs are simple graphs. A simple graph with multiple edges is called multigraph
Q3 How does BFS graph traversal scheme work?
Ans. BFS is a graph searching algorithm which start from any arbitrary vertex as a starting vertex and visits all its adjacent vertices first and then moves at the second level to visit all the unvisited vertices which are adjacent to vertices at the first level vertices and so on

Check your progress -2

Q1. Define topological ordering.

Topological sort or ordering of a $DAG\ G = (V, E)$ is a liner ordering of its vertices $V_1, V_2 \ldots . V_n$ so that if a graph G contains an edge $from\ \ v_i\ to\ v_j$

then v_i comes before v_j in the ordering. In other words, all edges between vertices representing tasks show forward direction in ordering or sorting. It the graph G contains a cycle then no topological order is possible. Therefore, if a graph G has a topological sort, then G is a DAG.

Q2 If a graph G has a topological sort, then G is a DAG. Write a proof.

Proof- Applying contradiction that G has a topological sorting of all its vertices (tasks): $V_1, V_2 \ldots. V_n$ and G is also having a cycle. let all the vertices be indexed : $V_1, V_2 \ldots. V_n$. let V_i be the lowest index in topological ordering. let there be an edges $(V_j, V_i) in$ a cyclic graph G. If $V_j > V_i$ i.e. V_j, comes before V_i which contradicts that all the vertices $V_1, V_2 \ldots. V_n$ are topological sorted is G.
The following is a pseudo-code topological for computing a DAG.

3.9 Further Readings

1. Thomas H. Cormen, et el, Introduction to Algorithms, 3rd edition, prentice Hall of India, 2012
2. John Kleinberg and Eva Tardos, Algorithm Design, Pearson, 2012