

UNIT-V

Introduction to Graphs

Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices. A graph is defined as follows...

Graph is a collection of vertices and arcs which connects vertices in the graph

Graph is a collection of nodes and edges which connects nodes in the graph

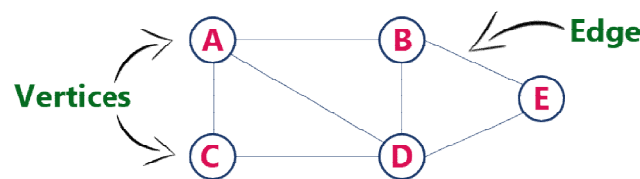
Generally, a graph G is represented as $G = (V, E)$, where V is set of vertices and E is set of edges.

Example

The following is a graph with 5 vertices and 6 edges.

This graph G can be defined as $G = (V, E)$

Where $V = \{A, B, C, D, E\}$ and $E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)\}$.



Graph Terminology

We use the following terms in graph data structure...

Vertex

A individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

Edge

An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

Undirected Edge - An undirected edge is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A, B) is equal to edge (B, A).

Directed Edge - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A, B) is not equal to edge (B, A).

Weighted Edge - A weighted edge is an edge with cost on it.

Undirected Graph

A graph with only undirected edges is said to be undirected graph.

Directed Graph

A graph with only directed edges is said to be directed graph.

Mixed Graph

A graph with undirected and directed edges is said to be mixed graph.

End vertices or Endpoints

The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

Origin

If an edge is directed, its first endpoint is said to be origin of it.

Destination

If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

Incident

An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge

A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Path

A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

Graph Representations

Graph data structure is represented using following representations...

Adjacency Matrix

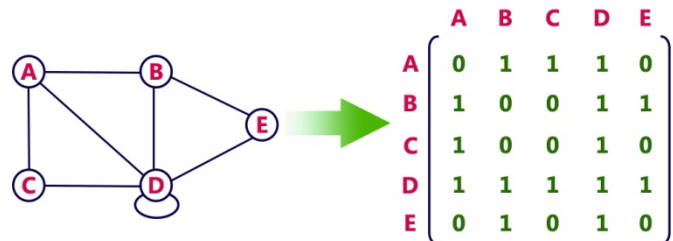
Incidence Matrix

Adjacency List

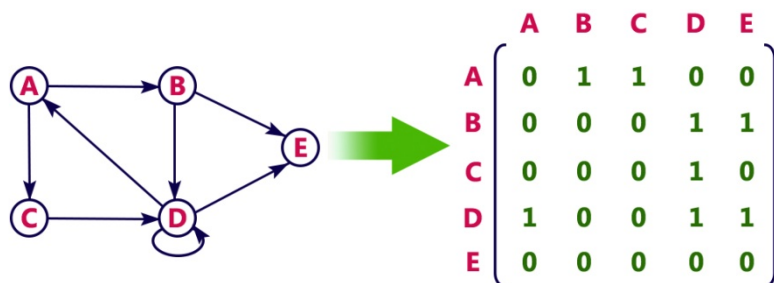
Adjacency Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



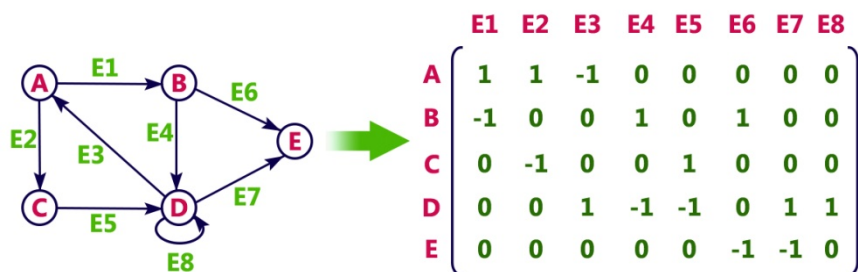
Directed graph representation...



Incidence Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class. In this matrix, rows represents vertices and columns represents edges. This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

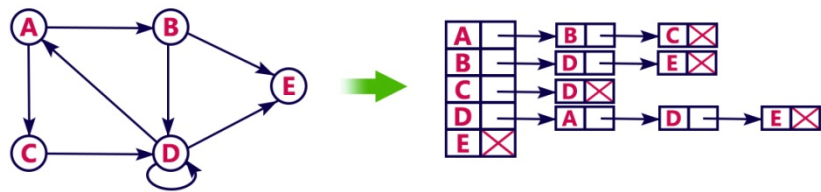
For example, consider the following directed graph representation...



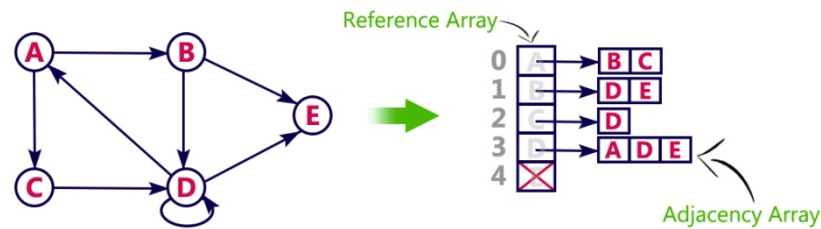
Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using array as follows..



Graph Traversals - DFS

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the egdes to be used in the search process without creating loops that means using graph traversal we visit all verticees of graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

DFS (Depth First Search)

BFS (Breadth First Search)

DFS (Depth First Search)

DFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

We use the following steps to implement DFS traversal...

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3: Visit any one of the adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.

Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

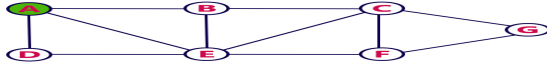
Back tracking is coming back to the vertex from which we came to current vertex.

example

Consider the following example graph to perform DFS traversal



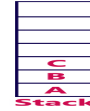
Step 1:
 - Select the vertex **A** as starting point (visit **A**).
 - Push **A** on to the Stack.



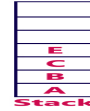
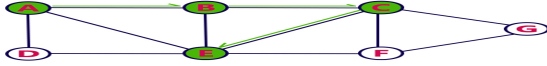
Step 2:
 - Visit any adjacent vertex of **A** which is not visited (**B**).
 - Push newly visited vertex **B** on to the Stack.



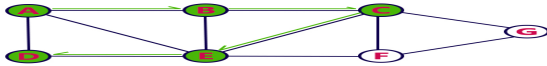
Step 3:
 - Visit any adjacent vertex of **B** which is not visited (**C**).
 - Push **C** on to the Stack.



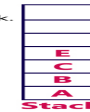
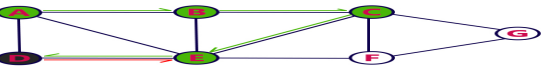
Step 4:
 - Visit any adjacent vertex of **C** which is not visited (**E**).
 - Push **E** on to the Stack.



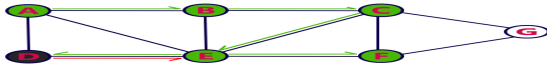
Step 5:
 - Visit any adjacent vertex of **E** which is not visited (**D**).
 - Push **D** on to the Stack.



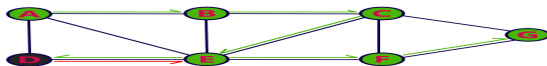
Step 6:
 - There is no new vertex to be visited from **D**. So use back track.
 - Pop **D** from the Stack.



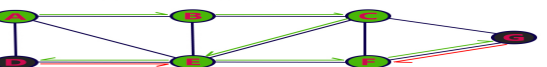
Step 7:
 - Visit any adjacent vertex of **E** which is not visited (**F**).
 - Push **F** on to the Stack.



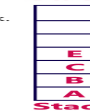
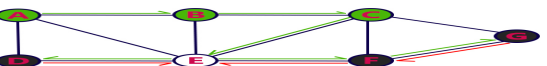
Step 8:
 - Visit any adjacent vertex of **F** which is not visited (**G**).
 - Push **G** on to the Stack.



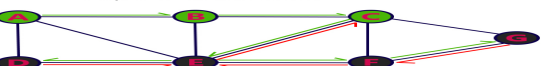
Step 9:
 - There is no new vertex to be visited from **G**. So use back track.
 - Pop **G** from the Stack.



Step 10:
 - There is no new vertex to be visited from **F**. So use back track.
 - Pop **F** from the Stack.



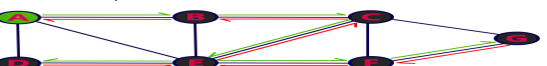
Step 11:
 - There is no new vertex to be visited from **E**. So use back track.
 - Pop **E** from the Stack.



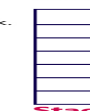
Step 12:
 - There is no new vertex to be visited from **C**. So use back track.
 - Pop **C** from the Stack.



Step 13:
 - There is no new vertex to be visited from **B**. So use back track.
 - Pop **B** from the Stack.



Step 14:
 - There is no new vertex to be visited from **A**. So use back track.
 - Pop **A** from the Stack.



- Stack became Empty. So stop DFS Traversal.
 - Final result of DFS Traversal is following spanning tree.



Graph Traversals - BFS

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the egdes to be

used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

DFS (Depth First Search)

BFS (Breadth First Search)

BFS (Breadth First Search)

BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

Step 1: Define a Queue of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3: Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

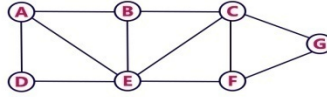
Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5: Repeat step 3 and 4 until queue becomes empty.

Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

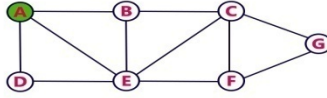
Example

Consider the following example graph to perform BFS traversal



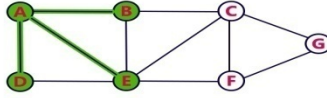
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



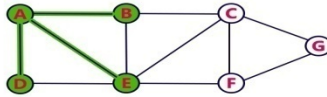
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D, E, B**).
- Insert newly visited vertices into the Queue and delete **A** from the Queue.



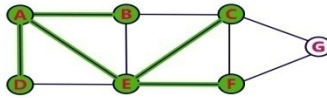
Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete **D** from the Queue.



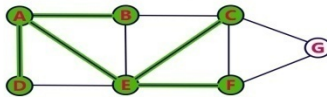
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C, F**).
- Insert newly visited vertices into the Queue and delete **E** from the Queue.



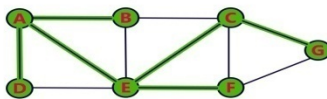
Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



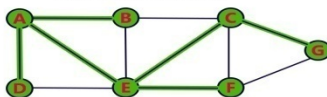
Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



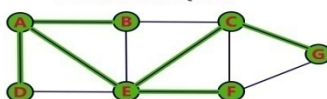
Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...

