Q. 4 Explain pre-order, in-order and post-order traversals with example. **(Refer sections 2.5.1, 2.5.2 and 2.5.3)**

Q. 5 Write short note on threaded binary tree traversals. **(Refer section 2.7)**

Q. 6 Define predecessor and successor. **(Refer section 2.7.2)**

Q. 7 Write short note on binary reach trees. **(Refer section 2.9)**

Q. 8 Describe AVL tree. **(Refer section 2.11)**

❑❑❑

CHAPTER 3 UNIT II

Graph and Selection Algorithms

Syllabus

Graph Algorithms : Introduction, Glossary, Applications of Graphs, Graph Representation, Graph Traversals, Topological Sort, Shortest Path Algorithms, Minimal Spanning Tree.

Selection Algorithms : What are Selection Algorithms? Selection by Sorting, Partition-based Selection Algorithm, Linear Selection Algorithm - Median of Medians Algorithm, Finding the K Smallest Elements in Sorted Order.

Syllabus Topic : Graph Algorithm : Introduction

3.1 Introduction

- This chapter introduces an important non-linear data structure called Graph.
- It has applications in various fields like electrical and electronics engineering, computer science, games and puzzles, Geographical Information System etc.

Syllabus Topic : Glossary

3.1.1 Glossary of Graph

- Graph is a non linear data structure. A Graph G consists of finite set of vertices V and finite set of edges E which can be denoted by G = (V, E).
- Where, V represent the entities which has names and other attributes.
- Edges E represent the links that connect the vertices.

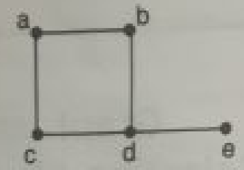


Fig. 3.1.1

In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

3.1.2 Graph Terminology

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 and 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). In the above graph (ab), (ac), (cd), (bd), (de) are edges.

Types of Edges

1. **Undirected Edge :** An undirected edge is a bidirectional edge. If there is a undirected edge between vertices a and b then edge (ab) is equal to edge (ba).
2. **Directed Edge :** A directed edge is a unidirectional edge. If there is a directed edge between vertices a and b then edge (ab) is not equal to edge (ba).
3. **Weighted Edge :** A weighted edge is an edge with cost on it.

Undirected Graph

An undirected graph is a graph in which all the edges are bi-directional i.e. there is no direction associated with the edges.

Directed Graph

A directed graph is a graph in which all the edges are uni-directional i.e. direction is associated with the edges.

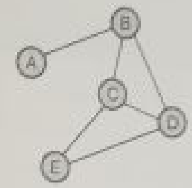


Fig. 3.1.2 : Undirected graph

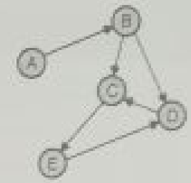


Fig. 3.1.3 : Directed graph

Weighted Graph

The graph in which weight is associate with every edge is a weighted graph.

Fig. 3.1.4

Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent.

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.

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.

Fig. 3.1.5

In the above graph,

Vertex	Number of outgoing edges	Number of incoming edges
1	2	0
2	0	2
3	2	2
4	1	1

Degree of a vertex

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

Indegree of a vertex : Total number of edges terminating at the vertex is said to be indegree of that vertex.

Outdegree of a vertex : Total number of edges starting from the vertex is said to be outdegree of that vertex.

Self Loop

If the starting and terminating vertices of an edge are same, then that edge is termed as self loop.

Parallel edges or multiple edges

If there are multiple edges between a pair of vertices, then such edges are called as parallel or multiple edges.

(a) Self loop (b) Parallel edges

Fig. 3.1.6

Simple Graph

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

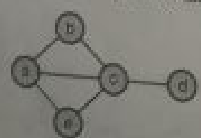


Fig. 3.1.7

Complete Graph

A graph in which every vertex is connected with every other vertex i.e. each vertex is adjacent to all other vertices in the graph, then that graph is called a complete graph.

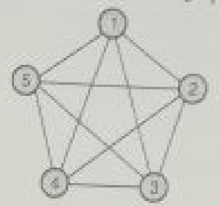


Fig. 3.1.8

Multigraph

The graph which contains multiple edges between a pair of vertices is called a multigraph.

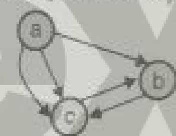


Fig. 3.1.9

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.

Cycle

A cycle is a path where the first and last vertices are the same. A simple cycle is a cycle with no repeated vertices or edges (except the first and last vertices).

Directed Acyclic Graph(DAG)

A directed acyclic graph [DAG] is a directed graph with no cycles.

Syllabus Topic : Applications of Graph

3.1.3 Applications of Graph

1. Representing relationships between components in electronic circuits.
2. Transportation networks: Highway network, Flight network.

3. Computer networks: Local area network, Internet, Web
4. Databases: For representing ER (Entity Relationship) diagrams in databases, for representing dependency of tables in databases

Syllabus Topic : Graph Representation

3.2 Graph Representation

Graph data structure is represented using following representations :

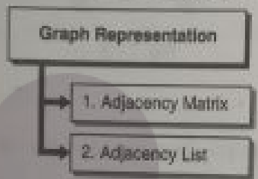
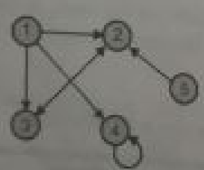


Fig. C3.1: Graph representation

➜ 3.2.1 Adjacency Matrix

- Adjacency Matrix is a 2D array of size V × V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.
- For example, consider the following graph representation.



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

Fig. 3.2.1

Graph Declaration for Adjacency Matrix

To represent graphs, we need the number of vertices, the number of edges and also their interconnections. So, the graph can be declared as:


```
class Vertex:
def init_(self, node):
self.id = node
# Mark all nodes unvisited
self.visited =False
def addNeighbor(self, neighbor, G):
G.addEdge(self.id, neighbor)
def getConnections(self, G):
return G.adjMatrix[self. id]
def getVertexID(self):
return self.id
def setVertexID(self, id):
self.id= id
def setVisited(self):
self.visited =True
def _str_(self):
return str(self. id)
class Graph:
def init (self, numVertices, cost= 0):
self.adjMatrix = [[-1-]*numVerticesfor _ in range(numVertices)]
self.numVertices= numVertices
self.vertices = []
for i in range(O,numVertices):
newVertex = Vertex(i)
self.vertices.append(newVertex)
```

➜ 3.2.2 Adjacency List

- An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[].
- An entry array[i] represents the linked list of vertices adjacent to the i^{th} vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists.

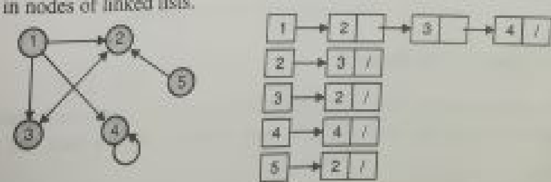


Fig. 3.2.2

Graph Declaration for Adjacency List

```
class Vertex:
def _init_(self, node):
self.id = node
self.adjacent = {}
# Set distance to infinity for all nodes
self.distance = sys.maxint
 Mark all nodes unvisited
self.visited = False
# Predecessor
self.previous= None
    class Graph:
def init_(self):
self.vertDictionary = {}
self.numVertices = 0
```

Syllabus Topic : Graph Traversal

3.3 Graph Traversal

- Graph traversal is technique used for searching a vertex in a graph; Graph traversal means visiting every vertex and edge exactly once in a well-defined order.
- While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. There are two graph traversal techniques and they are as follows,

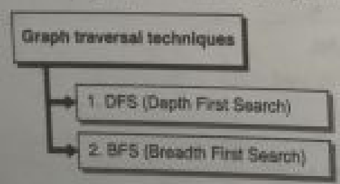


Fig. C3.2 : Graph traversal techniques

3.3.1 DFS (Depth First Search)

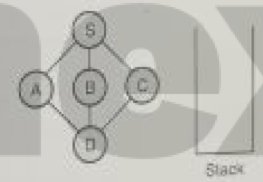
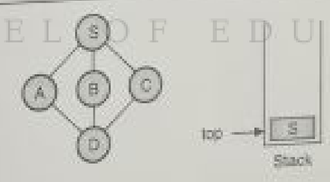
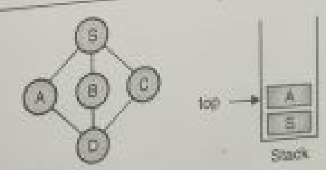
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

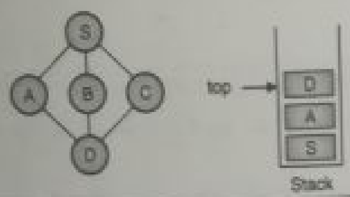
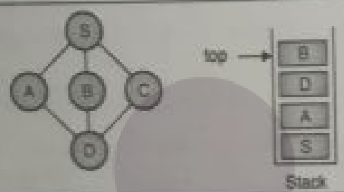
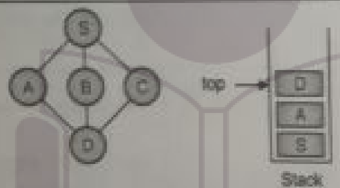
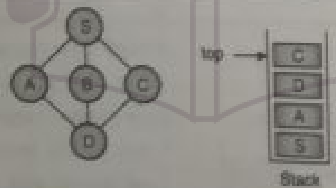
- **Undiscovered state :** The initial state of vertex.

- **Discovered state :** The vertex is encountered but not yet processed.
- **Processed state :** The vertex has been visited.

Steps for DFS

Step 1 :	Initially all the vertices of graph G are set to Undiscovered.
Step 2 :	Change the state of starting vertex of the graph to Discovered, and put it in the stack.
Step 3 :	Repeat steps 4 to 5 while the stack is not empty.
Step 4 :	Remove a vertex say v which is at the top of the stack and change its state to processed.
Step 5 :	Repeat for all undiscovered vertices u of vertex v u is set to the status Discovered and pushed to the stack.

Step	Traversal	Description
1.		Initialize the stack.
2.		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3.		Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.

Step	Traversal	Description
4.		Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.
5.		We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.
6.		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.
7.		Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.

DFS algorithm

- Initially all vertices are marked unvisited (false). The DFS algorithm starts at a vertex u in the graph.
- By starting at vertex u it considers the edges from u to other vertices. If the edge leads to an already visited vertex, then backtrack to current vertex u.
- If an edge leads to an unvisited vertex, then go to that vertex and start processing from that vertex. That mea ns the new vertex becomes the current vertex.
- Follow this process until we reach the dead-end. At this point start backtracking. The process terminates when backtracking leads back to the start vertex. The algorithm based on this mechanism is given below: assume visited[] is a global array.


```
def dfs(G, currentVert, visited):
visited[currentVert] =True                      # Mark the visited node
print "traversal: " + currentVert.getVertexID()
for nbr in currentVert.getConnections():         # Take a neighboring node
if nbr not in visited:                           #Check whether the neighbor node is already
visited
dfs(G, nbr, visited)                             # Recursively traverse the neighboring node

def DFSTraversal(G):
visited = {}                                     # Dictionary to mark the visited nodes
for currentVert in G:                            # G contains vertex objects
if currentVert not in visited:                   # Start traversing from the root node only if
                                                 #its not visited
dfs(G, currentVert, visited)                     #For a connected graph this is called only once
```

DFS program

```
# Python program to print DFS traversal from a given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation
class Graph:

    # Constructor
    def _init_(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self,v,visited):

    # Mark the current node as visited and print it
    visited[v]= True
        print v,
```

```
# Recur for all the vertices adjacent to this vertex
    for i in self.graph[v]:
        if visited[i] == False:
            self.DFSUtil(i, visited)

# The function to do DFS traversal. It uses
# recursive DFSUtil()
def DFS(self,v):

    # Mark all the vertices as not visited
    visited = [False]*(len(self.graph))

    # Call the recursive helper function to print
    # DFS traversal
    self.DFSUtil(v,visited)

# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Following is DFS from (starting from vertex 2)"
g.DFS(2)
```

3.3.2 BFS (Breadth First Search)

- BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.
- As the name BFS suggests, you are required to traverse the graph breadthwise as follows :
  1. First move horizontally and visit all the nodes of the current layer
  2. Move to the next layer

- **Undiscovered state :** the initial state of vertex.
- **Discovered state :** the vertex is encountered but not yet processed.
- **Processed state :** the vertex has been visited.

Steps for BFS

Step 1 :	Initially all the vertices of graph G are set to Undiscovered.
Step 2 :	Change the state of starting vertex of the graph to Discovered, and put it in the queue.
Step 3 :	Repeat steps 4 to 5 while the queue is not empty.
Step 4 :	Remove a vertex say v which is at the front of the queue and change its state to processed.
Step 5 :	Repeat for all undiscovered vertices u of vertex v, u is set to the status Discovered and added to the queue.

Example

Consider the following example.

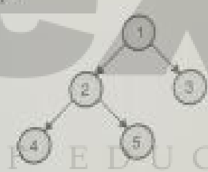


Fig. 3.3.1

Step 1 : Select the vertex 1 as starting point (visit 1). Insert 1 into the queue.

Queue : | 1 | | | | |

Step 2 : Visit all adjacent vertices of 1 which are not visited(2, 3). Insert newly visited vertices in the queue and delete 1 from the queue.

Queue : | | 2 | 3 | | |

Step 3 : Visit all adjacent vertices of 2 which are not visited (4, 5). Insert newly visited vertices in the queue and delete 2 from the queue.

Queue : | | | 3 | 4 | 5 |

Step 4 : Visit all adjacent vertices of 3 which are not visited (there is no vertex). Delete 3 from the queue.

Queue : | | | 3 | 4 | 5 |

Step 5 : Visit all adjacent vertices of 4 which are not visited (there is no vertex). Delete 4 from the queue.

Queue :

			4	5

Step 6 : Visit all adjacent vertices of 5 which are not visited (there is no vertex). Delete 5 from the queue.

Queue :

				5

Queue became empty. So stop the BFS process.

BFS Traversal is: 1 2 3 4 5.

BFS algorithm

- Assume that initially all vertices are marked unvisited (false). Vertices that have been processed and removed from the queue are marked visited (true).
- We use a queue to represent the visited set as it will keep the vertices in the order or when they were first visited. The implementation for the above discussion can be give n as:

```
def BFSTraversal(G,s):
start = G.getVertex(s)
start.setDistance(0)
start.setPrevious(None)
vertQueue = Queue()
vertQueue.enQueue(start)
while (vertQueue.size> 0):
currentVert = vertQueue.deQueue()
print current Vert.getVertexID()
for nbr in currentVert.getConnections():
if (nbr.getColor() "" 'white'):
nbr.setColor('gray')
nbr.set.Distance(currentVert.getDistance() + 1)
nbr.setPrevious(currentVert)
vertQueue.enQueue(nbr)
currentVert.setColor('black')
def BFS(G):
for v in G:
if (v.getColor() == 'white'):
BFSTraversal(G, v .get VertexID())
```

BFS program

```
# Program to print BFS traversal from a given source
# vertex. BFS(int s) traverses vertices reachable
# from s.
```



```
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    # Constructor
    def _init_(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False]*(len(self.graph))

        # Create a queue for BFS
        queue = []

        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True
        while queue:
            # Dequeue a vertex from queue and print it
            s = queue.pop(0)
            print s,
            # Get all adjacent vertices of the dequeued
            # vertex s. If a adjacent has not been visited,
            # then mark it visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
```

```
            visited[i] = True
# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
print "Following is Breadth First Traversal (starting from vertex 0)"
g.BFS(0)
```

Syllabus Topic : Topological Sort

3.4 Topological Sort

- Topological sort is an algorithm for a Directed Acyclic Graph (DAG).
- Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

Working of Topological Sort

- Initially, indegree is computed for all vertices, starting with the vertices which are having indegree 0. To keep track of vertices with indegree zero we can use a queue.
- All vertices of indegree 0 are placed on queue. While the queue is not empty, a vertex v is removed, and all edges adjacent to v have their indegrees decremented. A vertex is put on the queue as soon as its indegree falls to 0.
- The topological ordering is the order in which the vertices DeQueue.
- The time complexity of this algorithm is O(|E| + |V|) if adjacency lists are used.

Consider following example:

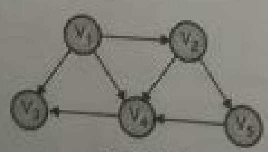


Fig. 3.4.1

1. Compute the indegrees

   V1: 0

   V2: 1

   V3: 2

   V4: 2

   V5: 2

2. Find a vertex with indegree 0: V1. Insert it in the queue.

V1				

3. Output V1 , remove V1 from the queue and update the indegrees.Remove edges: (V1,V2), (V1, V3) and(V1,V4)Updated indegrees:

   V2: 0

   V3: 1

   V4: 1

   V5: 2

4. Now vertex V2 is having indegree 0. Insert V2 in the queue.

V2				

5. Output V2 , remove V2 from the queue and update the indegrees:

   Remove edges: (V2,V4) , (V2, V5)

   Updated indegrees:

   V3: 1

   V4: 0

   V5: 1

6. Now vertex V4 is having indegree 0. Insert V4 in the queue.

V4				

7. Output V4 , remove V4 from the queue and update the indegrees:

   Remove edges: (V4,V5) , (V4, V3)

   Updated indegrees:

   V3: 0

   V5: 0

8. Now vertex V5 and V3 both are having indegree 0. There is no any other remaining vertex or edge. So stop the process.

   Finally, the topological sort is as : V1, V2, V4, V3, V5.

3.4.1 Topological Sort Algorithm

- Initially, indegree is computed for all vertices, starting with the vertices which are having indegree 0. To keep track of vertices with indegree zero we can use a queue. All vertices of indegree 0 are placed on queue.
- While the queue is not empty, a vertex v is removed, and all edges adjacent to v have their indegrees decremented. A vertex is put on the queue as soon as its indegree falls to 0. The topological ordering is the order in which the vertices DeQueue.
- The time complexity of this algorithm is O(E + V) if adjacency lists are used.

```
class Vertex:
def _init_(self, node):
self.id = node
self.adjacent= {}
# Set distance to infinity for all nodes
self.distance= sys.max.int
# Mark all nodes unvisited
self.visited = False
# Predecessor
self.previous = None
#InDegree Count
self.inDegree = 0
#OutDegree Count
self.outDegree =0
#........

class Graph:
    def _init_(self):
self.vertDictionary = {}
self.numVertices = 0
#........

def topologicalSort(G):
"""Perform a topological sort of the nodes. If the graph has a cycle,
throw a Graph Topological Exception with the list or successfully
ordered nodes."""
# Topologically sorted list of the nodes (result)
topologicalList = []
#Queue (fifo list) or the nodes with inDegree 0
```



```
remainingInDegree = {}
nodes= G.getVertices()
for v in G:
indegree = v.getInDegree()
if indegree == 0:
topologicalQueue.append(v)
else:
remainingInDegree[v] = indegree
#Remove nodes with inDegree 0 and decrease the inDegree or their sons
while len(topologicalQueue):
# Remove the first node with degree 0
node= topologicalQueue.pop(0)
topologicalList.append(node)
# Decrease the in Degree or the sons
for son in node.getConnections():
son.setInDegree(son.getInDegree() -1)
if son.getInDegree() == 0:
topologicalQueue.append(son)
#If not all nodes were covered, the graph must have a cycle
# Raise a GraphTopographicalException
if len(topologicalList) != len(nodes):
raise GraphTopologicalException(topologicalList)

#Printing the topological order
while len(topologicalList):
node = topologicalList.pop(0)
print node.getVertexID()
```

Syllabus Topic : Shortest Path Algorithms

3.5 Shortest Path Algorithms

- Finding the shortest path is one of the basic problems encountered in many graph applications. Given a graph G = (V, E) and a distinguished vertex s, we need to find the shortest path from s to every other vertex in G.

- Generally, there can exist more than one path between a particular pair of vertices. There are different variations of the shortest path problem which vary in terms of specifications of the source vertex and the destination vertex of a path.
  1. Shortest path can be found from one particular source vertex to all other vertices. This is known as **Single source shortest path problem(Dijkstra's Algorithm).**
  2. Shortest path can be found from all possible source vertices to all other destination vertices. This is known as **All pairs shortest path problem(Floyd Warshall's Algorithm).**

3.5.1 Single Source Shortest Path Problem (Dijkstra's Algorithm)

- Dijkstra's algorithm finds the shortest path from source vertex v to all other vertices in the graph.
- It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.
- It works in directed or undirected graph and all edges in the graph must have non-negative weight.
- Graph should not contain any cycle.
- Dijkstra's algorithm maintains a distance parameter for each vertex of the graph. These distances are initially set to infinity for all vertices except the source vertex s. This is done not to imply there is an infinite distance, but to note that those vertices have not yet been visited.
- The algorithm involves repeated iteration of the following process. At each iteration, we have a tree T rooted at s. For the first iteration, the tree will be the single vertex s, and the distance to it will be zero.
- For subsequent iterations (after the first), the next vertex v to be added to the tree will be the closest unvisited vertex to s (this will be easy to find). Once we found v, we add v to the tree, and check for each of its neighbors u, if the path from s – v path along with the edge uv gives a shorter distance for u than the one we have stored for u. If yes, the distance of u is updated to the new smaller value.
- We also maintain for each vertex v a parent vertex, which determines the last edge that is used to get to v from s.

Dijkstra's algorithm

```
import heap q
def dijkstrastra(G, source):
print "Dijkstra's shortest path"
# Set the distance for the source node to zero
source.setDistance(0)
```



```
# Put tuple pair into the priority queue
unvisitedQueue = [(v.getDistance(),v) for v in G]
heapq.heapify(unvisitedQueue)
while len(unvisitedQueue):
    #Pops a vertex with the smallest distance
uv = heapq.heappop(unvisitedQueue)
current = uv[1]
current.setVisited()
#for next in v.adjacent:
for next in current.adjacent:
# if visited, skip
if next.visited:
continue
newDist = current.getDistance() + current.getWeight(next)
if newDist < next.getDistance():
next.setDistance(newDist)
next.setPrevious(current)
print 'Updated : current = %s next= %s newDist = %s' \
%(current.getVertexID(), next.getVertexID(), next.getDistance())
else:
print 'Not updated : current = %s next = %s newDist = %s' \
%(current.getVertexID(), next.getVertexID(), next.getDistance())
#Rebuild heap
# 1. Pop every item
while len(unvisitedQueue):
heapq.heappop(unvisitedQueue)
# 2. Put all vertices not visited into the queue
unvisitedQueue = [(v.getDistance(),v) for v in G if not v.visited]
heapq. heapify(unvisitedQueue)
```

Example

- Consider the following example:

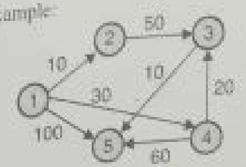


Fig. 3.5.1

Initially

S = {1}, D[2] = 10, D[3] = ∞, D[4] = 30, D[5] = 100

Iteration 1

Select w = 2, so that S = {1, 2}

D[3] = min(∞, D[2] + C[2, 3]) = 60

D[4] = min(30, D[2] + C[2, 4]) = 30

D[5] = min(100, D[2] + C[2, 5]) = 100

Iteration 2

Select w = 4, so that S = {1, 2, 4}

D[3] = min(60, D[4] + C[4, 3]) = 50

D[5] = min(100, D[4] + C[4, 5]) = 90

Iteration 3

Select w = 3, so that S = {1, 2, 4, 3}

D[5] = min(90, D[3] + C[3, 5]) = 60

Iteration 4

Select w = 5, so that S = {1, 2, 4, 3, 5}

D[2] = 10

D[3] = 50

D[4] = 30

D[5] = 60

3.5.2 All Pairs Shortest Path Problem (Bellman-Fordalgorithm)

- Dijkstra algorithm doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs.
- Bellman-Ford algorithm is used to find all shortest path in a graph from one source to all other nodes.
- This algorithm works if there are no negative-cost cycles. Each vertex can DeQueue at most |V| times, so the running time is O(|E|.|V|) if adjacency lists are used.
- Bellman-Ford algorithm has more running time than Dijkstra's algorithm.
- This algorithm takes two nodes as arguments and an edge connecting these nodes.

- If the distance from the source to the first node (A) plus the edge length is less than distance to the second node, than the first node is denoted as the predecessor of the second node and the distance to the second node is recalculated (distance (A) + edge.length). Otherwise no changes are applied.

Bellman-Ford Algorithm

```
import sys
def BellmanFord(G, source):
destination = {}
predecessor = {}
for node in G:
destination[node] = sys.maxint # We start admiting that the rest of nodes are very very far
predecessor[node] =None
destination[source] = 0 # For the source we know how to reach
for i in range(len(G)- 1):
  for u in G:
    for v in G[u]: #For each neighbour of u
    # If the distance between the node and the neighbour is lower than the one I have now
    if destination[v] > destination[u] + G[u][v]
    # Record this lower distance
    Destination[v] = destination[u] + G[u][v]
    Predecessor[v] = u
# Step 3: check for negative-weight cycles
for u in G:
for v in G[u]:
assert destination[v] <= destination[u] + G[u][v]
return destination, predecessor
if _name_ == '_main_':
G = {
'A': {'B': - 1, 'C': 4},
'B': {'C': 3, 'D': 2, 'E': 2},
'C': {},
'D': {'B': 1, 'C': 5},
'E': {'D': -3}
}
print BellmanFord(G, 'A')
```

Example

Consider following example.

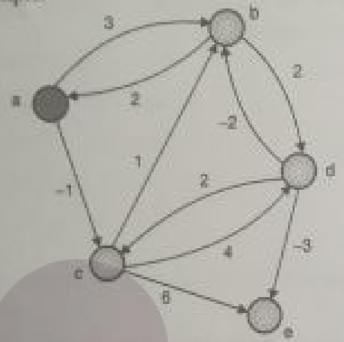


Fig. 3.5.2

Step 1 : Take the 0^{th} iteration:

Let the given source vertex be a. Initialize all distances as infinite, except the distance to source itself.

x	u	Distance d[u] ← min {d[u], d[x] + c(x, u)}	a	b	c	d	e
			0	∞	∞	∞	0
a,	b	d[b] ← min {∞, 0 + 3} = 3	0	3	∞	∞	∞
a,	c	d[c] ← min {∞, 0 − 1} = −1	0	3	−1	∞	∞
b,	a	d[a] ← min {0, 3 + 2} = 0	0	3	−1	∞	∞
b,	d	d[d] ← min {∞, 3 + 2} = 5	0	3	−1	5	∞
c,	b	d[b] ← min {3, −1 + 1} = 0	0	0	−1	5	∞
c,	d	d[d] ← min {5, −1 + 4} = 3	0	0	−1	3	∞
c,	e	d[e] ← min {∞, −1 + 6} = 5	0	0	−1	3	5
d,	b	d[b] ← min {0, −3 + 2} = 0	0	0	−1	3	5
d,	c	d[c] ← min {− 1, 3 + 2} = −1	0	0	−1	3	5
d	e	d[e] ← min {5, 3 − 3} = 0	0	0	−1	3	0

Iteration	Dist(x)				
	a	b	c	d	e
0	0	∞	∞	∞	∞

Step 2 : Take the 1^{st} iteration:

Take one vertex at a time say A and edges which are outgoing from the vertex A.

Iteration	Dist(x)				
	a	b	c	d	e
0	0	∞	∞	∞	∞
1	0	3	-1	∞	∞

x	u	Distance d[u] ← min {d[u], d[x] + c(x, u)}	a	b	c	d	e
			0	0	−1	3	0
a,	b	d[b] ← min {0, 0 + 3} = 0	0	0	∞	3	0
a,	c	d[c] ← min {−1, 0 − 1} = −1	0	0	−1	3	0
b,	a	d[a] ← min {0, 0 + 2} = 0	0	0	−1	3	0
b,	d	d[d] ← min {3, 0 + 2} = 2	0	0	−1	2	0
c,	b	d[b] ← min {0, −1 + 1} = 0	0	0	−1	2	0
c,	d	d[d] ← min {2, −1 + 4} = 2	0	0	−1	2	0
c,	e	d[e] ← min {0, −1 + 6} = 0	0	0	−1	2	0
d,	b	d[b] ← min {0, −2 + 2} = 0	0	0	−1	2	0
d,	c	d[c] ← min {− 1, 2 + 2} = −1	0	0	−1	2	0
d	e	d[e] ← min {0, 2 − 3} = −1	0	0	−1	2	−1

Step 3 : Take the next iteration and perform the same way.

Iteration	Dist(x)				
	a	b	c	d	e
0	0	∞	∞	∞	∞
1	0	3	−1	∞	∞
2	0	0	−1	2	0
3	0	0	−1	2	−1

Syllabus Topic : Minimal Spanning Tree

3.6 Minimal Spanning Tree

- In a connected undirected graph G = (V, E), if we cover all the vertices but not all edges, also no cycle is formed while covering all the vertices then such a component of a graph is called a spanning tree.

- We can also define a spanning tree of a connected graph G as a subgraph of G that contains all the vertices.

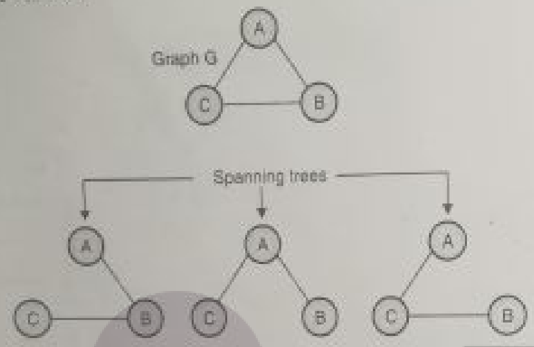


Fig. 3.6.1

- There are two famous algorithms for this problem:
  1. Kruskal's algorithm 2. Prim's algorithm

3.6.1 Kruskal's Algorithm

- Kruskal's algorithm uses the greedy approachto find the minimum cost spanning tree.
- In this algorithm, all the edges of the graph G are ordered in increasing order of their weight.
- Add all the edges one by one in increasing order skipping those edges whose addition would create a cycle.

Algorithm

```
def kruskal(G):
edges = []
for v in G:
makeSet(v.getVertexID())
for w in v.getConnections():
vid = v.getVertexID()
wid = w.getVertexID()
edges.append((v.getWeight(w),vid, wid))
edges.sort()
minimumSpanningTree = set()
for edge in edges:
weight, vertice1, vertice2 = edge
if find(vertice1) != find(vertice2):
union(vertice1, vertice2)
```



```
minimumSpanningTree.add(edge)
return minimumSpanningTree
```

Example

- Let us consider the following example :

Fig. 3.6.2

Step 1 : Remove all loops and Parallel Edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

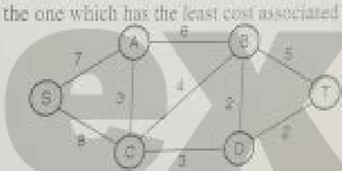


Fig. 3.6.3

Step 2 : Arrange all the edges in increasing order.

Edge	B, D	D, T	A, C	C, D	B, C	B, T	A, B	A, S	S, C
weight	2	2	3	3	4	5	6	7	8

Step 3 : Add edges one by one in increasing order but avoid those edges which can create a cycle. So, we will add the following edges.

Edge	weight
(B, D)	2
(D, T)	2
(A, C)	3
(C, D)	3
(A, S)	7

We have not added the edges (B, C), (B, T), (A, B) and (S, C) because if we try to add these edges, it will create a cycle. So, ignore such edges.

we now have minimum cost spanning tree having cost as follows:

$$2 + 2 + 3 + 3 + 7 = 17$$

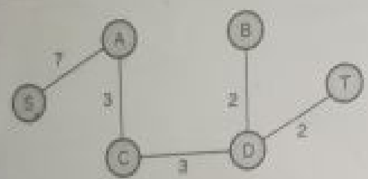


Fig. 3.6.4

3.6.2 Prim's Algorithm

- This algorithm is used to find minimum spanning tree for connected weighted undirected graph. Prim's algorithm starts by choosing an arbitrary vertex node of the graph.
- At each step, a new node will be added to the tree.
- This algorithm stops when all the nodes from the graph are added to the tree.

Algorithm

```
def Prims(G, source):
print "Dijkstra Modified for Prim"
# Set the distance for the source node to zero
source.setDistance(0)
# Put tuple pair into the priority queue
unvisitedQueue = [(v.getDistance(),v) for v in G]
heapq.heapify(unvisitedQueue)
while len(unvisitedQueue):
  #Pops a vertex with the smallest distance
  uv = heapq.heappop(unvisitedQueue)
  current = uv[1]
  current.setVisited()
  #for next in v.adjacent:
  for next in current.adjacent:
    # if visited, skip
    if next. visited:
      continue
    newCost = current.getWeight(next)
    if newCost < next.getDistance():
      next.setDistance(current.getWeight(next))
      next.setPrevious(current)
      print 'Updated : current = %s next = %s newCost = %s' \
          %(current.getVertexID(), next.getVertexID(), next.getDistance())
```



```
      else:
        print 'Not updated : current = %s next = %s newCost = %s' \
          % (current.getVertexID(), next.getVertexID(), next.getDistance())
  # Rebuild heap
  # 1. Pop every item
  while len(unvisitedQueue):
    heapq.heappop(unvisitedQueue)
  # 2. Put all vertices not visited into the queue
  unvisitedQueue = [(v.getDistance(),v) for v in G if not v.visited]
  heapq.heapify(unvisitedQueue)
```

Example

- To understand the prim's algorithm consider the following example:

Fig. 3.6.5

First choose any arbitrary node as root node. In this case, we choose **S** node as the root node of Prim's spanning tree. Now,

Step 1 : S = { NULL } , G = {S, A, B, C, D, T}

Now check which are the outgoing edges of S and chose the one which have less cost. s = {S}.

G = {A, B, C, D, T }

= min{(S, A), (S, B)} = min{7, 8} = 7

We choose the edge (S,A) as it has lesser value.

Step 2 : s = {S, A},

G = {B, C, D, T }

= min{(S, C), (A, C), (A, B)} = min{8, 3, 6} = 3

We choose the edge (A, C) as it has lesser value.

Step 3 : s = {S, A, C},

G = { B, D, T}

= min{(S, C), (B, C), (A, B), (C, D)}

= min{8, 4, 6, 3} = 3

We choose the edge (C, D) as it has lesser value.

Step 4 : = {S, A, C, D},

G = { B, T}

= min{(S, C), (C, B), (A, B), (D, B), (D, T), (C, D)}

= min{8, 4, 6, 2, 2, 3} = 2

We choose the edge (D, B) as it has lesser value.

Step-5: s = {S, A, C, D, B},

G = {T}

= min{(S, C), (C, B), (A, B), (D, T), (C, D), (B, T)}

= min{8, 4, 6, 2, 3, 5} = 2

We choose the edge (D, T) as it has lesser value.

Fig. 3.6.6

Syllabus Topic : Algorithms : What are selection algorithms?

3.7 Selection Algorithms

- Selection algorithm is an algorithm for finding the k^{th} smallest/largest number in a list (also called as k^{th} order statistics).
- This includes finding the minimum, maximum, and median elements. For finding the k^{th} order statistic, there are multiple solutions which provide different complexities, and in this chapter we will enumerate those possibilities.

Algorithm 3.7.1

Write an algorithm to find the largest element in an array A of size n.

Solution :

```
def FindLargestInArray(A):
max = 0
for number in A:
if number> max:
```



```
max= number
return max
print(FindLargestInArray([2, 1,5,234,3,44,7,6,4,5,9, 11, 12, 14, 13]))
Time Complexity - O(n). Space Complexity - O(1).
```

Algorithm 3.7.2

Write an algorithm to find the smallest and largest elements in an array II of size n.

Solution :

```
def FindSmallestAndLargestInArray(A):
max = 0
min = 0
for number in A:
if number > max:
max= number
elif number < min:
min = number
print("Smallest: %d", min)
print(" Largest: %d", max)
FindSmallestAndLargestInArray([2, 1,5,234,3,44, 7,6,4,5,9, 11, 12, 14, 13])
Time Complexity - O(n). Space Complexity - O(1).
The worst-case number of comparisons is 2(n - 1).
```

Syllabus Topic : Selection by Sorting

3.8 Selection by Sorting

- A selection problem can be converted to a sorting problem. In this method, we first sort the input elements and then get the desired element. It is efficient if we want to perform many selections.
- For example, let us say we want to get the minimum element. After sorting the input elements we can simply return the first element (assuming the array is sorted in ascending order). Now, if we want to find the second smallest element, we can simply return the second element from the sorted list.
- That means, for the second smallest element we are not performing the sorting again. The same is also the case with subsequent queries.

- Even if we want to get k^{th} smallest element, just one scan of the sorted list is enough to find the element (or we can return the k^{th}-indexed value if the elements are in the array).
- From the above discussion what we can say is, with the initial sorting we can answer any query in one scan, O(n). In general, this method requires O(n log n) time (for sorting) where n is the length of the input list.

Syllabus Topic : Partition based Selection Algorithm

3.9 Partition based Selection Algorithm

- **Quick select** is a selection algorithm to find the k^{th} smallest element in an unordered list. It is related to the quicksort sorting algorithm.
- Quickselect uses the same approach as quicksort, it chooses one element as a pivot and partition the data in two based on the pivot, accordingly as less than or greater than the pivot.
- However, instead of recursing into both sides, as in quicksort, quickselect only recourses into one side – the side with the element it is searching for. This reduces the average complexity from O(n log n) to O(n), with a worst case of $O(n^2)$.

Algorithm for Quickselect

- Suppose we are given an unsorted sequence S of n comparable elements together with an integer $k \in [1,n]$. At a high level, the quick-select algorithm for finding the k^{th} smallest element in S. We pick a "pivot" element from S at random and use this to subdivide S into three subsequences L, E, and G, storing the elements of S less than, equal to, and greater than the pivot, respectively.
- We determine which of these subsets contains the desired element, based on the value of k and the sizes of those subsets. We then recur on the appropriate subset, noting that the desired element's rank in the subset may differ from its rank in the full set.

```
def quick select(S, k):
""" Return the kth smallest element of list S, for k from 1 to len(S)."""
if len(S) == 1:
   return S[0]
pivot = random.choice(S)          # pick random pivot element from S
L = [x for x in S if x < pivot]   # elements less than pivot
E = [x for x in S if x == pivot]  # elements equal to pivot
G = [x for x in S if pivot < x]   # elements greater than pivot
if k <= len(L):
return quick select(L, k)         # kth smallest lies in L
```



```
elif k <= len(L) + len(E):
return pivot                      # kth smallest equal to pivot
else:
  j = k - len(L) - len(E)         # new selection parameter
return quick select(G, j)         # kth smallest is jth in G
```

Syllabus Topic : Linear Selection Algorithm : Median of Medians Algorithm

3.10 Linear Selection Algorithm - Median of Medians Algorithm

- The median of medians is an approximate selection algorithm, frequently used to supply a good pivot for an exact selection algorithm.
- Median of medians finds an approximate median in linear time only.
- The **median-of-medians algorithm** chooses the pivot in the following way :
  1. Divide the list into sublists of length five. (Note that the last sublist may have length less than five.)
  2. Sort each sublist and determine its median directly.
  3. Use the median of medians algorithm to recursively determine the median of the set of all medians from the previous step.
  4. Use the median of the medians from step 3 as the pivot.

Median of Medians algorithm

```
CHUNK_SIZE = 5
def kthByMedianOfMedian(unsortedList, k):
if len(unsortedList) <= CHUNK_SIZE:
return get_kth(unsortedList, k)
chunks = splitIntoChunks(unsortedList, CHUNK_SIZE)
medians_list = []
for chunk in chunks:
median_chunk = get_median(chunk)
medians_list.append(median_chunk)
size = len(medians_list)
mom = kthByMedianOfMedian(medians_list, size / 2 + (size % 2))
smaller, larger = splitListByPivot(unsortedList, mom)
```

```
valuesBeforeMom = len(smaller)
if valuesBeforeMom == (k - 1):
return mom
elif valuesBeforeMom > (k - 1 ):
return kthByMedianOfMedian(smaller, k)
else:
return kthByMedianOfMedian(larger, k - valuesBeforeMom - 1)
```

Syllabus Topic : Finding The K Smallest Elements in Sorted Order

3.11 Finding the K Smallest Elements in Sorted Order

Given a set of n elements from a totally-ordered domain, find the k smallest elements, and list them in sorted order.

```
class Solution
def findMedianSortedArrays(self, A, B):
#comparing middle elements of A and B, which we identify as Ai and Bj. If Ai is between Bj and Bj-1, we have just found the i+j+1 smallest element. Therefore, if we choose i and j such that i+j = k-1, we are able to find the k-th smallest element.
def findkth(a,b,k):
        lena = len(a)
        lenb = len(b)
if lena > lenb:
return findkth(b,a,k)
if a == []: return b[k-1]
if k == 1: return min(a[0],b[0])
        parta = min(k/2, lena)
        partb = k - parta
if a[parta-1] < b[partb-1]:  #delete impossible value from a
return findkth(a[parta:], b, k-parta)
else:                #delete impossible value from b
return findkth(a, b[partb:], k-partb)
        length = len(A)+len(B)
if length % 2 == 0:
```



```
return (findkth(A,B,length/2)+findkth(A,B,length/2+1))*0.5
else:
return findkth(A,B,length/2+1)*1.0
```

Review Questions

Q. 1 List the applications of graph. (Refer section 3.1.3)

Q. 2 Explain adjacency matrix and adjacency list. (Refer sections 3.2.1 and 3.2.2)

Q. 3 Write short note on : DFS and BFS (Refer sections 3.3.1 and 3.3.2)

Q. 4 Write DFS algorithm. (Refer section 3.3.1)

Q. 5 Explain working of topological sort. (Refer section 3.4)

Q. 6 Write Dijkstra's algorithm. (Refer section 3.5.1)

Q. 7 Explain Kruskal and Prim's algorithms. (Refer sections 3.6.1 and 3.6.2)

❑❑❑