With path compression the only change to the FIND function is that $S[X]$ is made equal to the value returned by FIND. That means, after the root of the set is found recursively, X is made to point directly to it. This happen recursively to every node on the path to the root.

FIND with path compression

```
def FINDBYSIZE(self, X):
    if( self.S[X] < 0 ):
        return X
    else:
        return self.FINDBYSIZE(self.S[X])
```

Note: Path compression is compatible with UNION by size but not with UNION by height as there is no efficient way to change the height of the tree.

8.10 Summary

Performing m union-find operations on a set of n objects.

Algorithm	Worst-case time
Quick-find	mn
Quick-union	mn
Quick-Union by Size/Height	$n + m \log n$
Path compression	$n + m \log n$
Quick-Union by Size/Height + Path Compression	$(m + n) \log n$

8.11 Disjoint Sets: Problems & Solutions

Problem-1 Consider a list of cities c_1, c_2,...,c_n. Assume that we have a relation R such that, for any i, j, $R(c_i, c_j)$ is 1 if cities c_i and c_j are in the same state, and 0 otherwise. If R is stored as a table, how much space does it require?

Solution: R must have an entry for every pair of cities. There are $\Theta(n^2)$ of these.

Problem-2 For Problem-1, using a Disjoint sets ADT, give an algorithm that puts each city in a set such that c_i and c_j are in the same set if and only if they are in the same state.

Solution:

```
for i in range(0,n-1):
    MAKESET(ci)
    for j in range(1,i-1):
        if(R(cj, ci)):
            UNION(cj, ci)
            break
```

Problem-3 For Problem-1, when the cities are stored in the Disjoint sets ADT, if we are given two cities c_i and c_j, how do we check if they are in the same state?

Solution: Cities c_i and c_j are in the same state if and only if FIND(c_i) = FIND(c_j).

Problem-4 For Problem-1, if we use linked-lists with UNION by size to implement the union-find ADT, how much space do we use to store the cities?

Solution: There is one node per city, so the space is $\Theta(n)$.

Problem-5 For Problem-1, if we use trees with UNION by rank, what is the worst-case running time of the algorithm from 0?

Solution: Whenever we do a UNION in the algorithm from 0, the second argument is a tree of size 1. Therefore, all trees have height 1, so each union takes time O(1). The worst-case running time is then $\Theta(n^2)$.

Problem-6 If we use trees without union-by-rank, what is the worst-case running time of the algorithm from 0? Are there more worst-case scenarios than Problem-5?

Solution: Because of the special case of the unions, union-by-rank does not make a difference for our algorithm. Hence, everything is the same as in Problem-5.

Problem-7 With the quick-union algorithm we know that a sequence of n operations (*unions* and *finds*) can take slightly more than linear time in the worst case. Explain why if all the *finds* are done before all the *unions*, a sequence of n operations is guaranteed to take $O(n)$ time.

Solution: If the *find* operations are performed first, then the *find* operations take O(1) time each because every item is the root of its own tree. No item has a parent, so finding the set an item is in takes a fixed number of operations. Union operations always take O(1) time. Hence, a sequence of n operations with all the *finds* before the *unions* takes $O(n)$ time.

Problem-8 With reference to Problem-7, explain why if all the unions are done before all the finds, a sequence of n operations is guaranteed to take $O(n)$ time.

Solution: This problem requires amortized analysis. *Find* operations can be expensive, but this expensive *find* operation is balanced out by lots of cheap *union* operations.

The accounting is as follows. *Union* operations always take O(1) time, so let's say they have an actual cost of ₹1. Assign each *union* operation an amortized cost of ₹2, so every *union* operation puts ₹1 in the account. Each *union* operation creates a new child. (Some node that was not a child of any other node before is a child now.) When all the union operations are done, there is $1 in the account for every child, or in other words, for every node with a depth of one or greater. Let's say that a *find*(u) operation costs ₹1 if u is a root. For any other node, the *find* operation costs an additional ₹1 for each parent pointer the *find* operation traverses. So the actual cost is ₹($1 + d$), where d is the depth of u. Assign each *find* operation an amortized cost of ₹2. This covers the case where u is a root or a child of a root. For each additional parent pointer traversed, ₹1 is withdrawn from the account to pay for it.

Fortunately, path compression changes the parent pointers of all the nodes we pay ₹1 to traverse, so these nodes become children of the root. All of the traversed nodes whose depths are 2 or greater move up, so their depths are now 1. We will never have to pay to traverse these nodes again. Say that a node is a grandchild if its depth is 2 or greater.

Every time *find*(u) visits a grandchild, ₹1 is withdrawn from the account, but the grandchild is no longer a grandchild. So the maximum number of dollars that can ever be withdrawn from the account is the number of grandchildren. But we initially put $1 in the bank for every child, and every grandchild is a child, so the bank balance will never drop below zero. Therefore, the amortization works out. *Union* and *find* operations both have amortized costs of ₹2, so any sequence of n operations where all the unions are done first takes $O(n)$ time.

CHAPTER

9

GRAPH ALGORITHMS

9.1 Introduction

In the real world, many problems are represented in terms of objects and connections between them. For example, in an airline route map, we might be interested in questions like: "What's the fastest way to go from Hyderabad to New York?" *or* "What is the cheapest way to go from Hyderabad to New York?" To answer these questions we need information about connections (airline routes) between objects (towns). Graphs are data structures used for solving these kinds of problems.

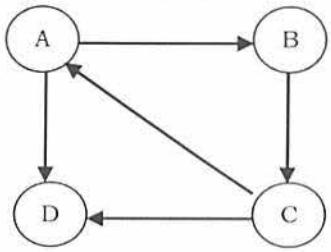
9.2 Glossary

Graph: A graph is a pair (V, E), where V is a set of nodes, called *vertices*, and E is a collection of pairs of vertices, called *edges*.

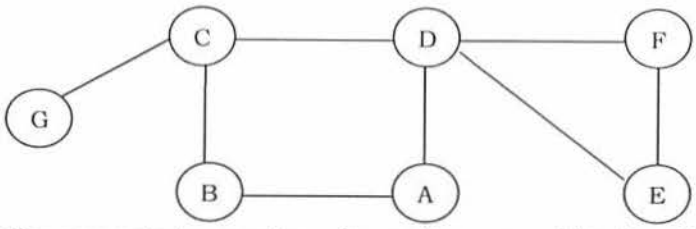
- *Vertices* and *edges* are positions and store elements
- Definitions that we use:
  - *Directed edge*:
    - ordered pair of vertices (u, v)
    - first vertex u is the origin
    - second vertex v is the destination
    - Example: one-way road traffic

  - *Undirected edge*:
    - unordered pair of vertices (u, v)
    - Example: railway lines

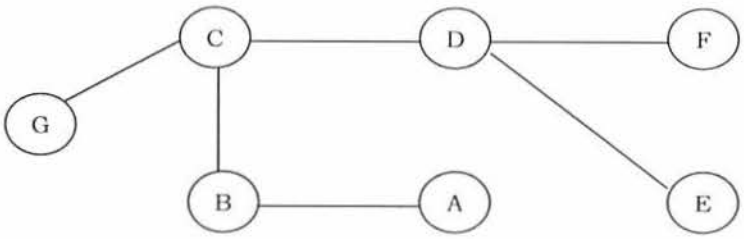
  - *Directed graph*:
    - all the edges are directed
    - Example: route network



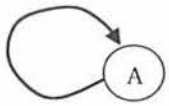
- *Undirected graph*:
  - all the edges are undirected
  - Example: flight network



- When an edge connects two vertices, the vertices are said to be adjacent to each other and the edge is incident on both vertices.
- A graph with no cycles is called a *tree*. A tree is an acyclic connected graph.

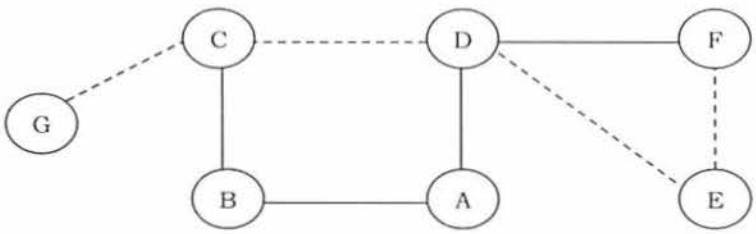


- A self loop is an edge that connects a vertex to itself.

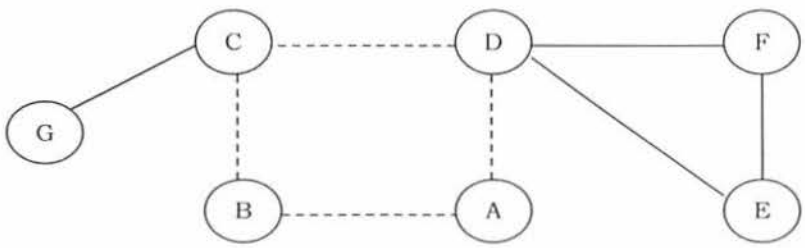


- Two edges are parallel if they connect the same pair of vertices.

- The *D*egree of a vertex is the number of edges incident on it.
- A subgraph is a subset of a graph's edges (with associated vertices) that form a graph.
- A path in a graph is a sequence of adjacent vertices. *S*imple path is a path with no repeated vertices. In the graph below, the dotted lines represent a path from *G* to *E*.

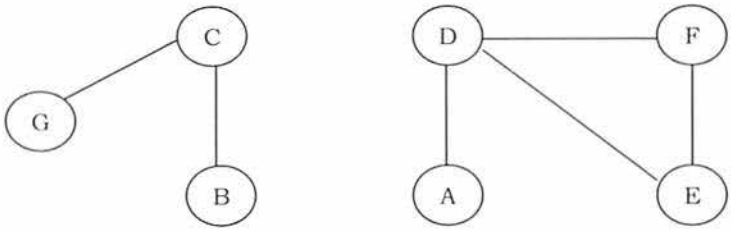


- A cycle is a path where the first and last vertices are the same. A simple cycle is a cycle with no repeated vertices or edges (except the first and last vertices).

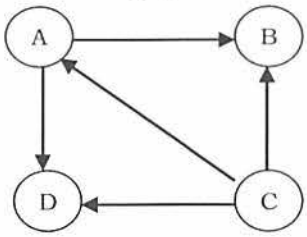


- We say that one vertex is connected to another if there is a path that contains both of them.
- A graph is connected if there is a path from *every* vertex to every other vertex.

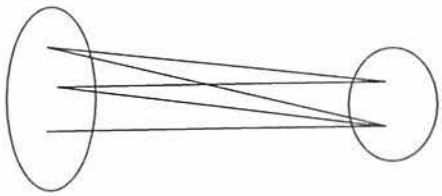
- If a graph is not connected then it consists of a set of connected components.



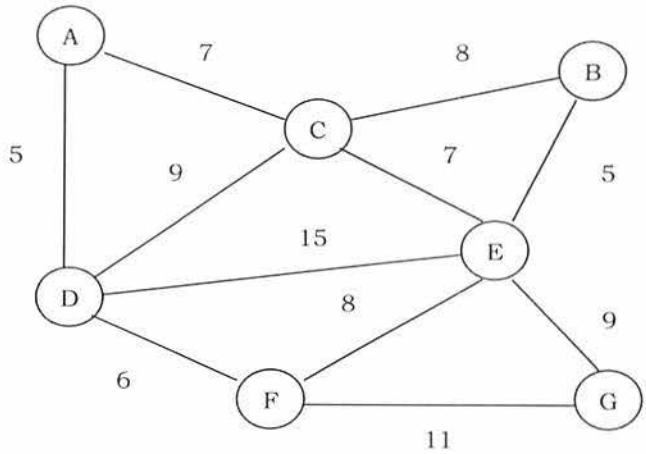
- A *directed* acyclic graph [DAG] is a directed graph with no cycles.



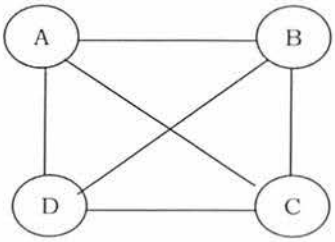
- A forest is a disjoint set of trees.
- A spanning tree of a connected graph is a subgraph that contains all of that graph's vertices and is a single tree. A spanning forest of a graph is the union of spanning trees of its connected components.
- A bipartite graph is a graph whose vertices can be divided into two sets such that all edges connect a vertex in one set with a vertex in the other set.



- In *weighted graphs* integers *(weights)* are assigned to each edge to represent (distances or costs).



- Graphs with all edges present are called *complete* graphs.



- Graphs with relatively few edges (generally if it edges < $|V| \log |V|$) are called *sparse graphs*.
- Graphs with relatively few of the possible edges missing are called *dense*.

- Directed weighted graphs are sometimes called *network*.
- We will denote the number of vertices in a given graph by $|V|$, and the number of edges by $|E|$. Note that E can range anywhere from 0 to $|V|(|V| - 1)/2$ (in undirected graph). This is because each node can connect to every other node.

9.3 Applications of Graphs

- Representing relationships between components in electronic circuits
- Transportation networks: Highway network, Flight network
- Computer networks: Local area network, Internet, Web
- Databases: For representing ER (Entity Relationship) diagrams in databases, for representing dependency of tables in databases

9.4 Graph Representation

As in other ADTs, to manipulate graphs we need to represent them in some useful form. Basically, there are three ways of doing this:

- Adjacency Matrix
- Adjacency List
- Adjacency Set

Adjacency Matrix

Graph Declaration for Adjacency Matrix

First, let us look at the components of the graph data structure. To represent graphs, we need the number of vertices, the number of edges and also their interconnections. So, the graph can be declared as:

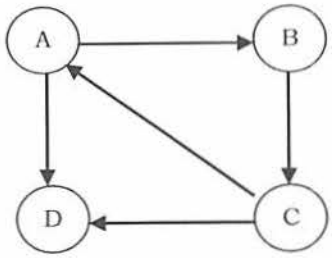
```
class Vertex:
    def __init__(self, node):
        self.id = node
        # Mark all nodes unvisited
        self.visited = False
    def addNeighbor(self, neighbor, G):
            G.addEdge(self.id, neighbor)
    def getConnections(self, G):
        return G.adjMatrix[self.id]
    def getVertexID(self):
        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]*numVertices for _ in range(numVertices)]
        self.numVertices =numVertices
            self.vertices = []
            for i in range(0,numVertices):
                        newVertex = Vertex(i)
                        self.vertices.append(newVertex)
```

Description

In this method, we use a matrix with size $V \times V$. The values of matrix are boolean. Let us assume the matrix is Adj. The value $Adj[u, v]$ is set to 1 if there is an edge from vertex **u** to vertex **v** and **0** otherwise.

In the matrix, each edge is represented by two bits for undirected graphs. That means, an edge from **u** to **v** is represented by 1 value in both $Adj[\mathbf{u}, \mathbf{v}]$ and $Adj[u, v]$. To save time, we can process only half of this symmetric matrix. Also, we can assume that there is an "edge" from each vertex to itself. So, $Adj[\mathbf{u}, \mathbf{u}]$ is set to 1 for all

vertices. If the graph is a directed graph then we need to mark only one entry in the adjacency matrix. As an example, consider the directed graph below.



The adjacency matrix for this graph can be given as:

	A	B	C	D
A	0	1	0	1
B	0	0	1	0
C	1	0	0	1
D	0	0	0	0

Now, let us concentrate on the implementation. To read a graph, one way is to first read the vertex names and then read pairs of vertex names (edges). The code below reads an undirected graph.

```
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]*numVertices for _ in range(numVertices)]
        self.numVertices =numVertices
            self.vertices = []
            for i in range(0,numVertices):
                        newVertex = Vertex(i)
                        self.vertices.append(newVertex)

    def setVertex(self, vtx, id):
            if 0 <= vtx < self.numVertices:
                        self.vertices[vtx].setVertexID(id)

    def getVertex(self, n):
        for vertxin in range(0,self.numVertices):
                        if n == self.vertices[vertxin].getVertexID():
                                    return vertxin
        else:
            return -1

    def addEdge(self, frm, to, cost = 0):
        if self.getVertex(frm) != -1 and self.getVertex(to) != -1:
                        self.adjMatrix[self.getVertex(frm)][self.getVertex(to)] = cost
                        #For directed graph do not add this
                        self.adjMatrix[self.getVertex(to)][self.getVertex(frm)] = cost
```

```
    def getVertices(self):
        vertices = []
        for vertxin in range(0,self.numVertices):
            vertices.append(self.vertices[vertxin].getVertexID())
        return vertices
    def printMatrix(self):
        for u in range(0,self.numVertices):
            row = []
            for v in range(0, self.numVertices):
                row.append(self.adjMatrix[u][v])
            print row
    def getEdges(self):
        edges = []
        for v in range(0,self.numVertices):
            for u in range(0, self.numVertices):
                if self.adjMatrix[u][v] != -1:
                    vid = self.vertices[v].getVertexID()
                    wid = self.vertices[u].getVertexID()
                    edges.append((vid, wid, self.adjMatrix[u][v]))
        return edges
if __name__ == '__main__':
    G = Graph(5)
    G.setVertex(0,'a')
    G.setVertex(1, 'b')
    G.setVertex(2, 'c')
    G.setVertex(3, 'd')
    G.setVertex(4, 'e')
    print 'Graph data:'
    G.addEdge('a', 'e', 10)
    G.addEdge('a', 'c', 20)
    G.addEdge('c', 'b', 30)
    G.addEdge('b', 'e', 40)
    G.addEdge('e', 'd', 50)
    G.addEdge('f', 'e', 60)
    print G.printMatrix()
    print G.getEdges()
```

The adjacency matrix representation is good if the graphs are dense. The matrix requires $O(V^2)$ bits of storage and $O(V^2)$ time for initialization. If the number of edges is proportional to V^2, then there is no problem because V^2 steps are required to read the edges. If the graph is sparse, the initialization of the matrix dominates the running time of the algorithm as it takes takes $O(V^2)$.

Adjacency List

Graph Declaration for Adjacency List

In this representation all the vertices connected to a vertex v are listed on an adjacency list for that vertex v. This can be easily implemented with linked lists. That means, for each vertex v we use a linked list and list nodes represents the connections between v and other vertices to which v has an edge.

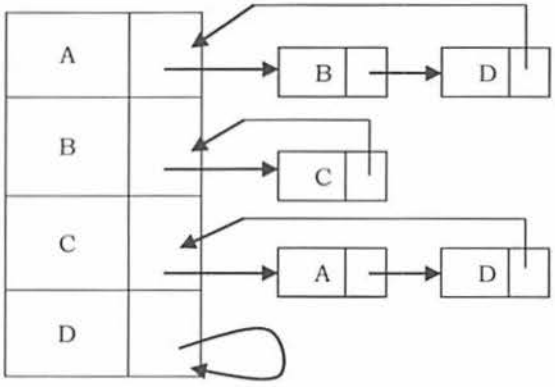
The total number of linked lists is equal to the number of vertices in the graph. The graph ADT can be declared as:

```
    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
```

Description

Considering the same example as that of the adjacency matrix, the adjacency list representation can be given as:



Since vertex A has an edge for B and D, we have added them in the adjacency list for A. The same is the case with other vertices as well.

```
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

    def addNeighbor(self, neighbor, weight=0):
        self.adjacent[neighbor] = weight

    def getConnections(self):
        return self.adjacent.keys()

    def getVertexID(self):
        return self.id

    def getWeight(self, neighbor):
        return self.adjacent[neighbor]

    def setDistance(self, dist):
        self.distance = dist

    def getDistance(self):
        return self.distance

    def setPrevious(self, prev):
        self.previous = prev

    def setVisited(self):
        self.visited = True

    def __str__(self):
        return str(self.id) + ' adjacent: ' + str([x.id for x in self.adjacent])
class Graph:
    def __init__(self):
        self.vertDictionary = {}
        self.numVertices = 0

    def __iter__(self):
        return iter(self.vertDictionary.values())
```

```
    def addVertex(self, node):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(node)
        self.vertDictionary[node] = newVertex
        return newVertex
    def getVertex(self, n):
        if n in self.vertDictionary:
            return self.vertDictionary[n]
        else:
            return None
    def addEdge(self, frm, to, cost = 0):
        if frm not in self.vertDictionary:
            self.addVertex(frm)
        if to not in self.vertDictionary:
            self.addVertex(to)
        self.vertDictionary[frm].addNeighbor(self.vertDictionary[to], cost)
            #For directed graph do not add this
        self.vertDictionary[to].addNeighbor(self.vertDictionary[frm], cost)
    def getVertices(self):
        return self.vertDictionary.keys()
    def setPrevious(self, current):
        self.previous = current
    def getPrevious(self, current):
        return self.previous
    def getEdges(self):
        edges = []
            for v in G:
                    for w in v.getConnections():
                        vid = v.getVertexID()
                        wid = w.getVertexID()
                        edges.append((vid, wid, v.getWeight(w)))
            return edges
if __name__ == '__main__':
    G = Graph()
    G.addVertex('a')
    G.addVertex('b')
    G.addVertex('c')
    G.addVertex('d')
    G.addVertex('e')
    G.addEdge('a', 'b', 4)
    G.addEdge('a', 'c', 1)
    G.addEdge('c', 'b', 2)
    G.addEdge('b', 'e', 4)
    G.addEdge('c', 'd', 4)
    G.addEdge('d', 'e', 4)
    print 'Graph data:'
    print G.getEdges()
```

For this representation, the order of edges in the input is *important*. This is because they determine the order of the vertices on the adjacency lists. The same graph can be represented in many different ways in an adjacency list. The order in which edges appear on the adjacency list affects the order in which edges are processed by algorithms.

Disadvantages of Adjacency Lists

Using adjacency list representation we cannot perform some operations efficiently. As an example, consider the case of deleting a node. . In adjacency list representation, it is not enugh if we simply delete a node from the list representation. if we delete a node from the adjacency list then that is enough. For each node on the adjacency list of that node specifies another vertex. We need to search other nodes linked list also for deleting it. This

problem can be solved by linking the two list nodes that correspond to a particular edge and making the adjacency lists doubly linked. But all these extra links are risky to process.

Adjacency Set

It is very much similar to adjacency list but instead of using Linked lists, Disjoint Sets [Union-Find] are used. For more details refer to the *Disjoint Sets ADT* chapter.

Comparison of Graph Representations

Directed and undirected graphs are represented with the same structures. For directed graphs, everything is the same, except that each edge is represented just once. An edge from x to y is represented by a 1 value in $Adj[x][y]$ in the adjacency matrix, or by adding y on $x's$ adjacency list. For weighted graphs, everything is the same, except fill the adjacency matrix with weights instead of boolean values.

Representation	Space	Checking edge between v and w?	Iterate over edges incident to v?
List of edges	E	E	E
Adj Matrix	V^2	1	V
Adj List	$E+V$	$Degree(v)$	$Degree(v)$
Adj Set	$E+V$	$log(Degree(v))$	$Degree(v)$

9.5 Graph Traversals

To solve problems on graphs, we need a mechanism for traversing the graphs. Graph traversal algorithms are also called *graph search* algorithms. Like trees traversal algorithms (Inorder, Preorder, Postorder and Level-Order traversals), graph search algorithms can be thought of as starting at some source vertex in a graph and "searching" the graph by going through the edges and marking the vertices. Now, we will discuss two such algorithms for traversing the graphs.

- Depth First Search [DFS]
- Breadth First Search [BFS]

Depth First Search [DFS]

DFS algorithm works in a manner similar to preorder traversal of the trees. Like preorder traversal, internally this algorithm also uses stack.

Let us consider the following example. Suppose a person is trapped inside a maze. To come out from that maze, the person visits each path and each intersection (in the worst case). Let us say the person uses two colors of paint to mark the intersections already passed. When discovering a new intersection, it is marked grey, and he continues to go deeper.

After reaching a "dead end" the person knows that there is no more unexplored path from the grey intersection, which now is completed, and he marks it with black. This "dead end" is either an intersection which has already been marked grey or black, or simply a path that does not lead to an intersection.

The intersections of the maze are the vertices and the paths between the intersections are the edges of the graph. The process of returning from the "dead end" is called *backtracking*. We are trying to go away from the starting vertex into the graph as deep as possible, until we have to backtrack to the preceding grey vertex. In DFS algorithm, we encounter the following types of edges.

Tree edge: encounter new vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

For most algorithms boolean classification, unvisited/visited is enough (for three color implementation refer to problems section). That means, for some problems we need to use three colors, but for our discussion two colors are enough.

false ⟶ Vertex is unvisited

true ⟶ Vertex is visited

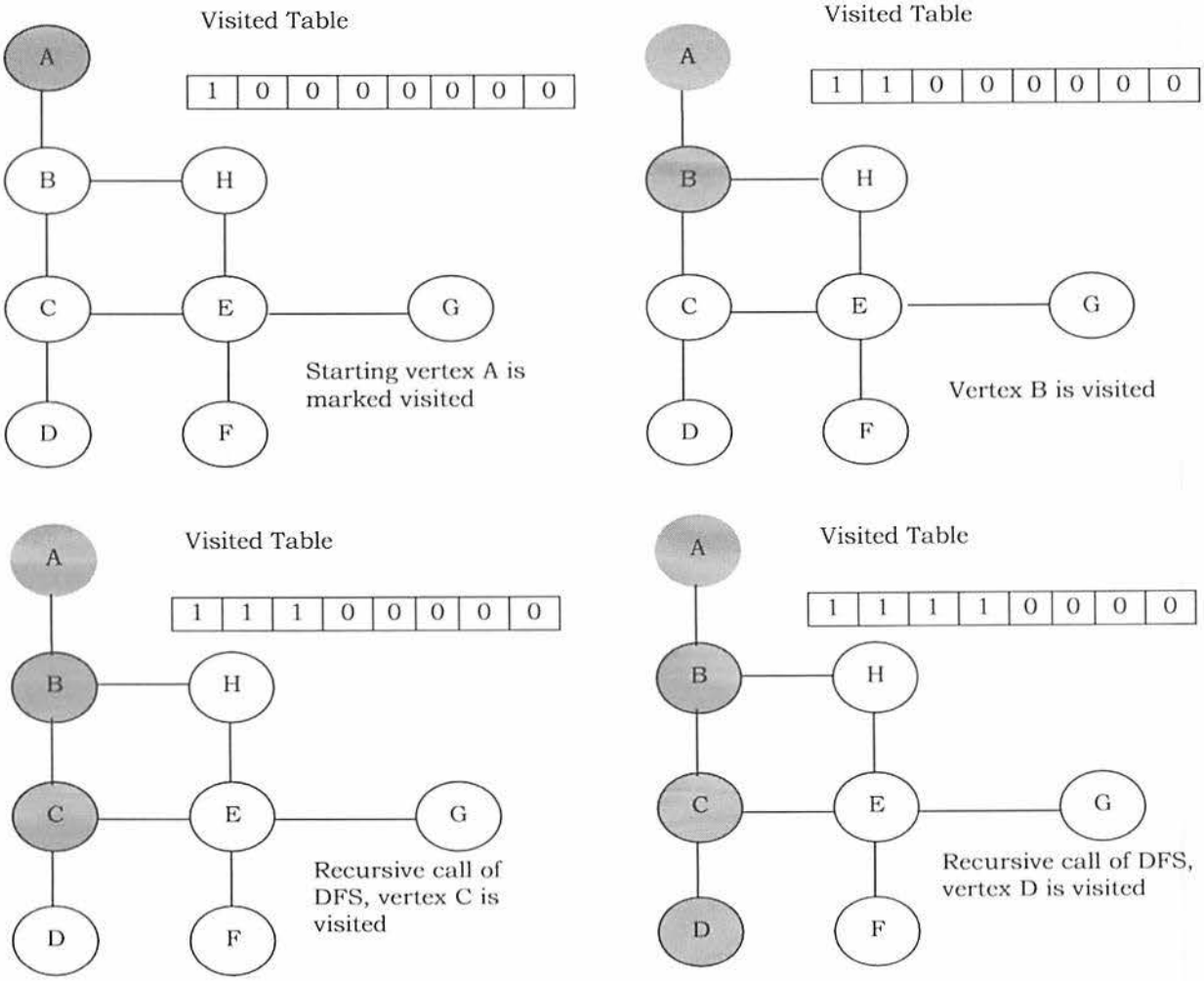
Initially all vertices are marked unvisited (false). The DFS algorithm starts at a vertex u in the graph. By starting at vertex u it considers the edges from u to other vertices. If the edge leads to an already visited vertex, then backtrack to current vertex u. If an edge leads to an unvisited vertex, then go to that vertex and start processing from that vertex. That means the new vertex becomes the current vertex. Follow this process until we reach the dead-end. At this point start *backtracking*.

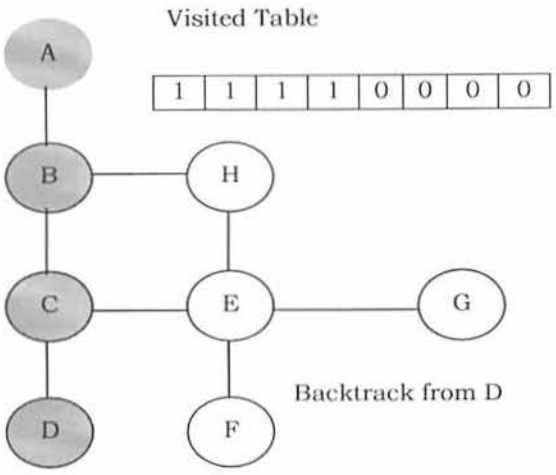
The process terminates when backtracking leads back to the start vertex. The algorithm based on this mechanism is given below: assume Visited[] is a global array.

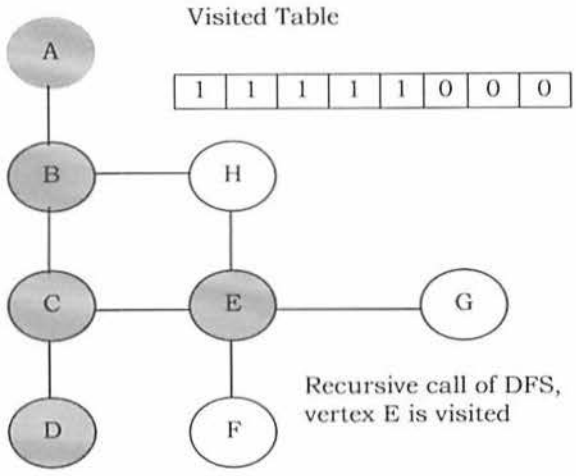
```
def dfs(G, currentVert, visited):
    visited[currentVert]=True                    # Mark the visited node
    print "traversal: " + currentVert.getVertexID()
    for nbr in currentVert.getConnections():     # Take a neighbouring node
        if nbr not in visited:                   # Check whether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

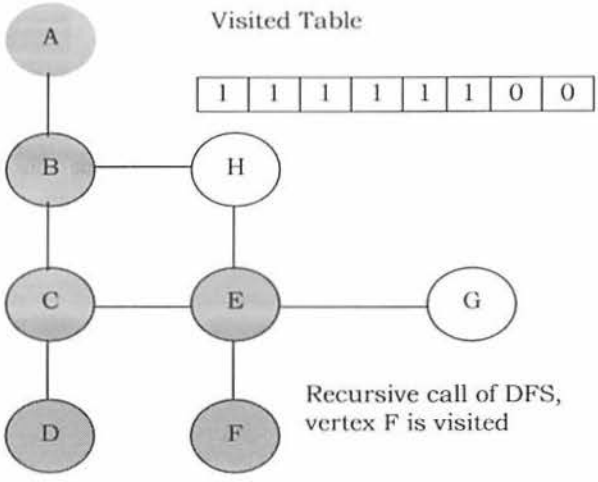
As an example, consider the following graph. We can see that sometimes an edge leads to an already discovered vertex. These edges are called *back edges*, and the other edges are called *tree edges* because deleting the back edges from the graph generates a tree.

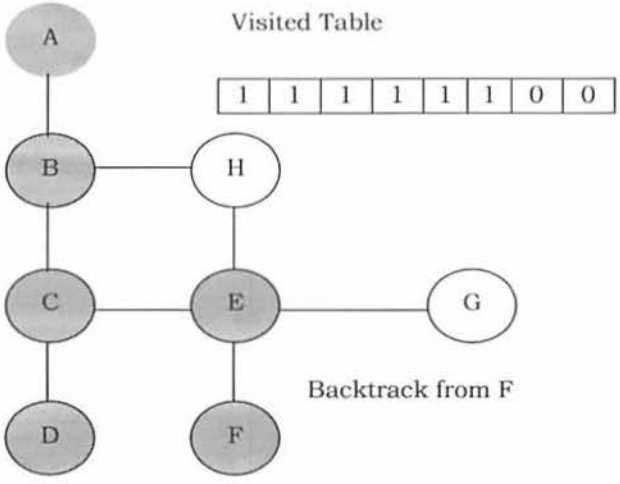
The final generated tree is called the DFS tree and the order in which the vertices are processed is called *DFS numbers* of the vertices. In the graph below, the gray color indicates that the vertex is visited (there is no other significance). We need to see when the Visited table is updated.

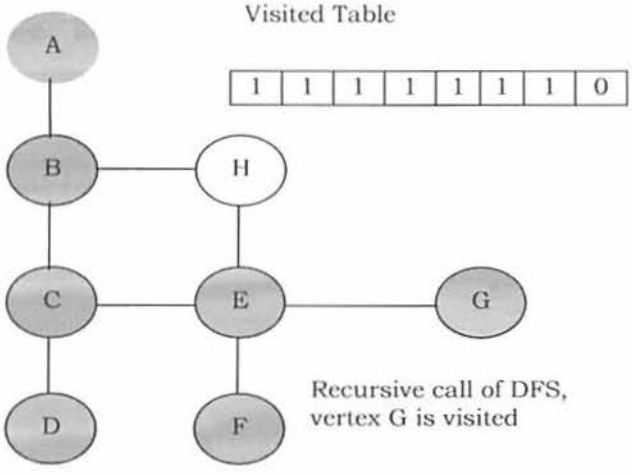


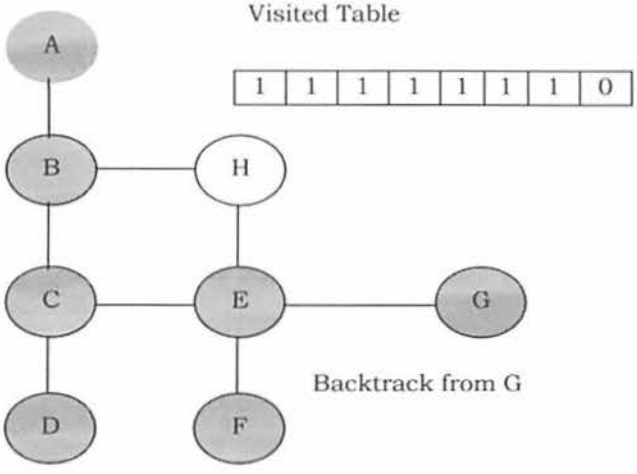

Visited Table
1 1 1 1 0 0 0 0
A
B
H
C
E
G
D
F
Backtrack from D

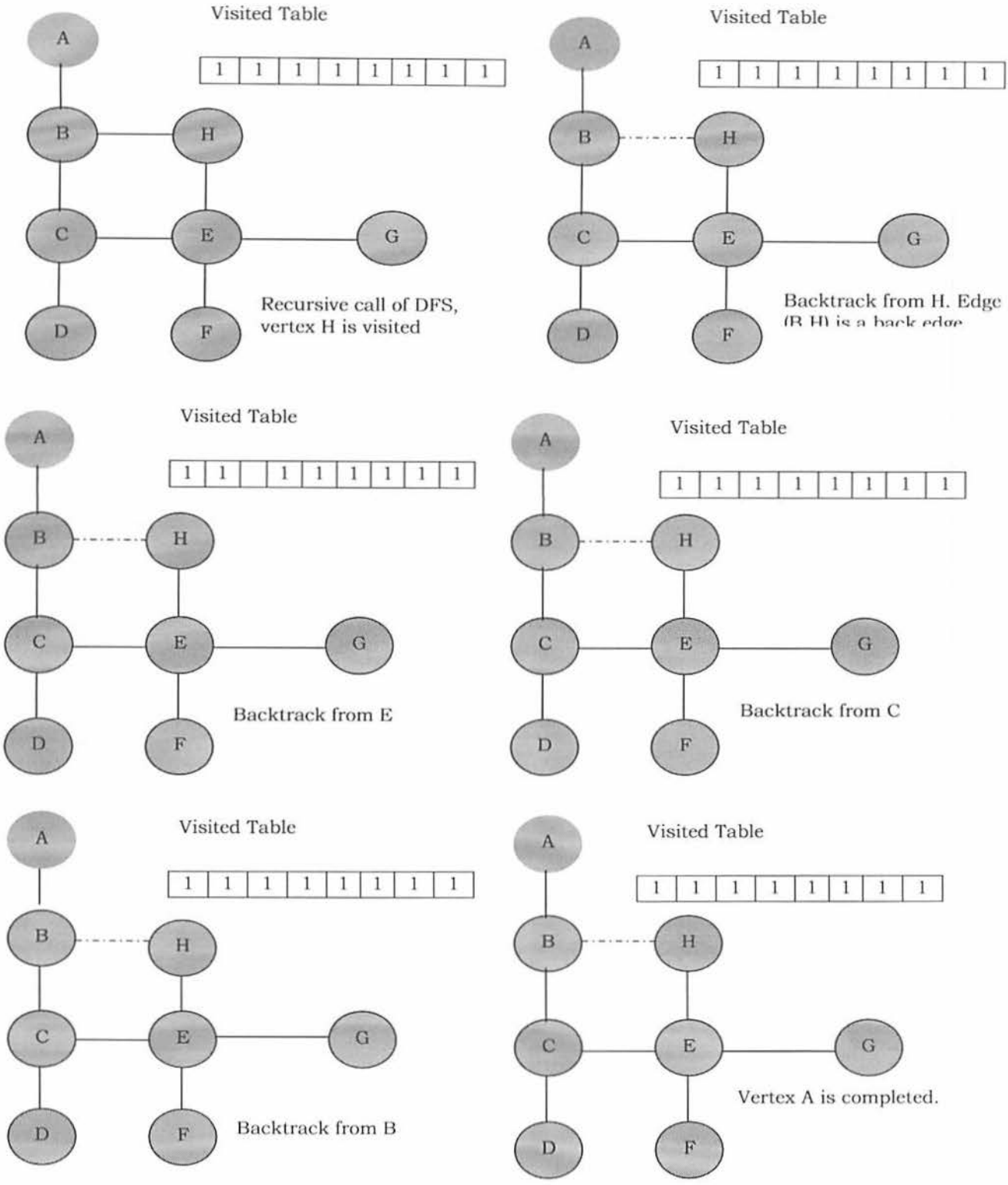

Visited Table
1 1 1 1 1 0 0 0
A
B
H
C
E
G
D
F
Recursive call of DFS,
vertex E is visited


Visited Table
1 1 1 1 1 1 0 0
A
B
H
C
E
G
D
F
Recursive call of DFS,
vertex F is visited


Visited Table
1 1 1 1 1 1 0 0
A
B
H
C
E
G
D
F
Backtrack from F


Visited Table
1 1 1 1 1 1 1 0
A
B
H
C
E
G
D
F
Recursive call of DFS,
vertex G is visited


Visited Table
1 1 1 1 1 1 1 0
A
B
H
C
E
G
D
F
Backtrack from G



From the above diagrams, it can be seen that the DFS traversal creates a tree (without back edges) and we call such tree a *DFS tree*. The above algorithm works even if the given graph has connected components.

The time complexity of DFS is $O(V + E)$, if we use adjacency lists for representing the graphs. This is because we are starting at a vertex and processing the adjacent nodes only if they are not visited. Similarly, if an adjacency matrix is used for a graph representation, then all edges adjacent to a vertex can't be found efficiently, and this gives $O(V^2)$ complexity.

Applications of DFS

- Topological sorting
- Finding connected components
- Finding articulation points (cut vertices) of the graph
- Finding strongly connected components
- Solving puzzles such as mazes

For algorithms refer to *Problems Section*.

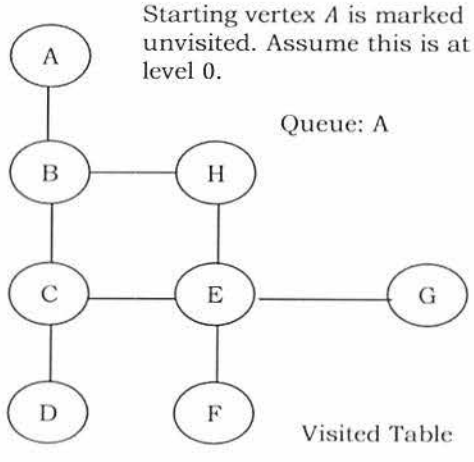
Breadth First Search [BFS]

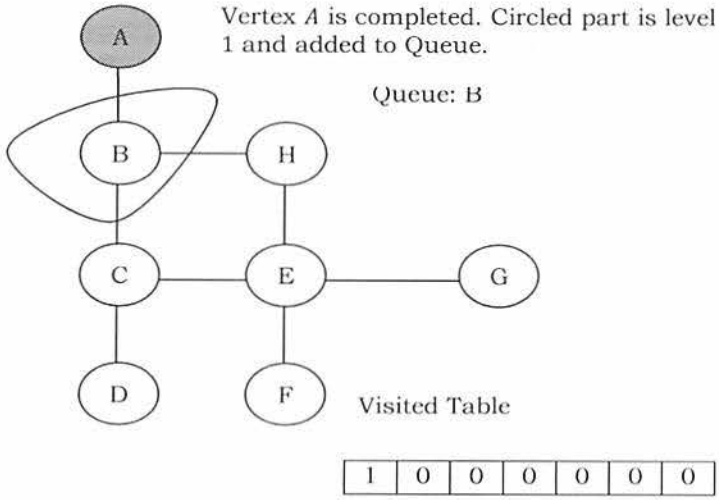
The BFS algorithm works similar to *level – order* traversal of the trees. Like *level – order* traversal, BFS also uses queues. In fact, *level – order* traversal got inspired from BFS. BFS works level by level. Initially, BFS starts at a given vertex, which is at level 0. In the first stage it visits all vertices at level 1 (that means, vertices whose distance is 1 from the start vertex of the graph). In the second stage, it visits all vertices at the second level. These new vertices are the ones which are adjacent to level 1 vertices. BFS continues this process until all the levels of the graph are completed. Generally *queue* data structure is used for storing the vertices of a level.

As similar to DFS, 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 of when they were first visited. The implementation for the above discussion can be given 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 currentVert.getVertexID()
        for nbr in currentVert.getConnections():
            if (nbr.getColor() == 'white'):
                nbr.setColor('gray')
                nbr.setDistance(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.getVertexID())
```

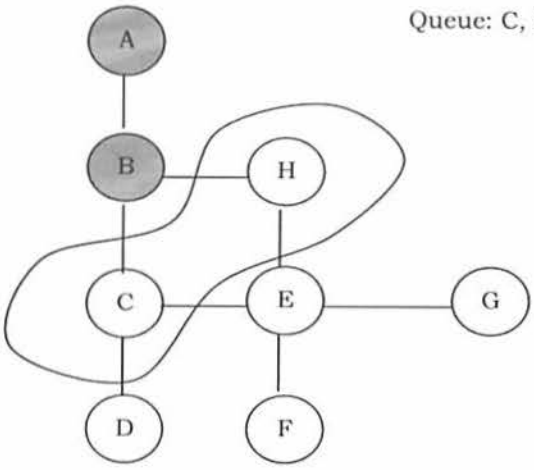
As an example, let us consider the same graph as that of the DFS example. The BFS traversal can be shown as:





B is completed. Selected part is level 2 (add to Queue).

Queue: C, H

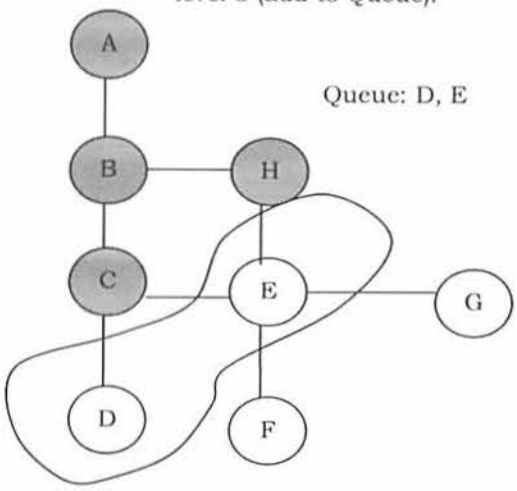


Visited Table

1	1	0	0	0	0	0

Vertices C and H are completed. Circled part is level 3 (add to Queue).

Queue: D, E

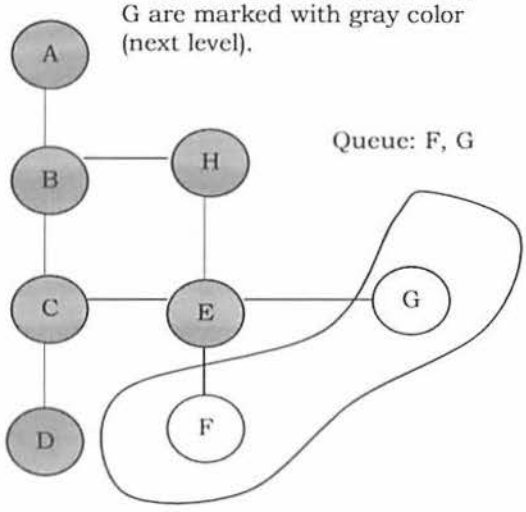


Visited Table

1	1	0	0	0	0	1

D and E are completed. F and G are marked with gray color (next level).

Queue: F, G

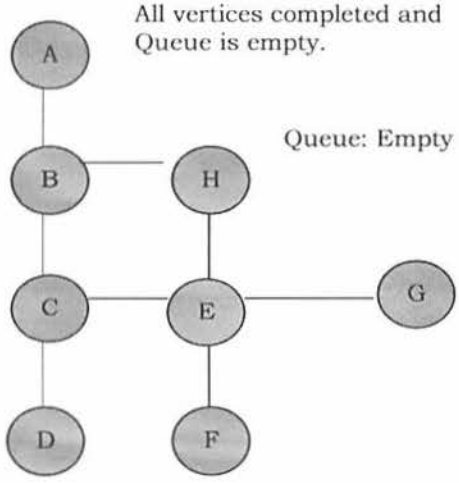


Visited Table

1	1	1	1	0	0	1

All vertices completed and Queue is empty.

Queue: Empty



Visited Table

1	1	1	1	1	1	1

Time complexity of BFS is $O(V + E)$, if we use adjacency lists for representing the graphs, and $O(V^2)$ for adjacency matrix representation.

Applications of BFS

- Finding all connected components in a graph
- Finding all nodes within one connected component
- Finding the shortest path between two nodes
- Testing a graph for bipartiteness

Comparing DFS and BFS

Comparing BFS and DFS, the big advantage of DFS is that it has much lower memory requirements than BFS because it's not required to store all of the child pointers at each level. Depending on the data and what we are looking for, either DFS or BFS can be advantageous. For example, in a family tree if we are looking for someone who's still alive and if we assume that person would be at the bottom of the tree, then DFS is a better choice. BFS would take a very long time to reach that last level.

The DFS algorithm finds the goal faster. Now, if we were looking for a family member who died a very long time ago, then that person would be closer to the top of the tree. In this case, BFS finds faster than DFS. So, the advantages of either vary depending on the data and what we are looking for.

DFS is related to preorder traversal of a tree. Like *preorder* traversal, DFS visits each node before its children. The BFS algorithm works similar to *level – order* traversal of the trees.

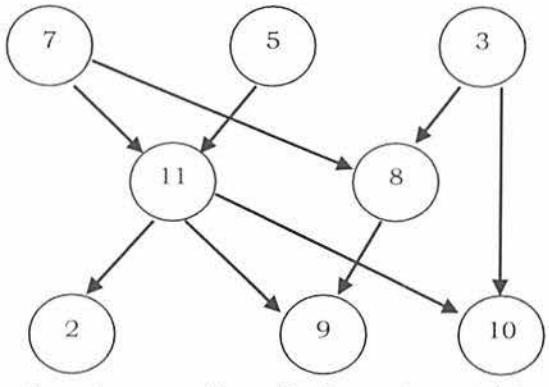
If someone asks whether DFS is better or BFS is better, the answer depends on the type of the problem that we are trying to solve. BFS visits each level one at a time, and if we know the solution we are searching for is at a low depth, then BFS is good. DFS is a better choice if the solution is at maximum depth. The below table shows the differences between DFS and BFS in terms of their applications.

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	Yes	Yes
Shortest paths		Yes
Minimal use of memory space	Yes	

9.6 Topological Sort

Topological sort is an ordering of vertices in a directed acyclic graph [DAG] in which each node comes before all nodes to which it has outgoing edges. As an example, consider the course prerequisite structure at universities. A directed *edge* (v,w) indicates that course v must be completed before course w. Topological ordering for this example is the sequence which does not violate the prerequisite requirement. Every DAG may have one or more topological orderings. Topological sort is not possible if the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.

Topological sort has an interesting property. All pairs of consecutive vertices in the sorted order are connected by edges; then these edges form a directed Hamiltonian path [refer to *Problems Section*] in the DAG. If a Hamiltonian path exists, the topological sort order is unique. If a topological sort does not form a Hamiltonian path, DAG can have two or more topological orderings. In the graph below: 7, 5, 3, 11, 8, 2, 9, 10 and 3, 5, 7, 8, 11, 2, 9, 10 are both topological orderings.



Initially, *indegree* is computed for all vertices, starting with the vertices which are having indegree 0. That means consider the vertices which do not have any prerequisite. 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.maxint
        # 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 GraphTopologicalException with the list of successfully
        ordered nodes."""
        # Topologically sorted list of the nodes (result)
        topologicalList = []
        # Queue (fifo list) of the nodes with inDegree 0
        topologicalQueue = []
        # {node: inDegree} for the remaining nodes (those 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 of their sons
        while len(topologicalQueue):
            # Remove the first node with degree 0
            node = topologicalQueue.pop(0)
            topologicalList.append(node)
            # Decrease the inDegree of 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()
```

Total running time of topological sort is $O(V + E)$.

Note: The Topological sorting problem can be solved with DFS. Refer to the *Problems Section* for the algorithm.

Applications of Topological Sorting

- Representing course prerequisites
- Detecting deadlocks
- Pipeline of computing jobs

- Checking for symbolic link loop
- Evaluating formulae in spreadsheet

9.7 Shortest Path Algorithms

Let us consider the other important problem of a graph. 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. There are variations in the shortest path algorithms which depend on the type of the input graph and are given below.

Variations of Shortest Path Algorithms

Shortest path in unweighted graph
Shortest path in weighted graph
Shortest path in weighted graph with negative edges

Applications of Shortest Path Algorithms

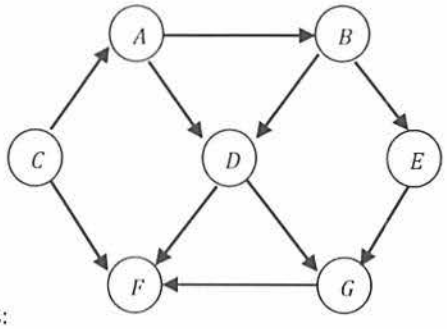
- Finding fastest way to go from one place to another
- Finding cheapest way to fly/send data from one city to another

Shortest Path in Unweighted Graph

Let s be the input vertex from which we want to find the shortest path to all other vertices. Unweighted graph is a special case of the weighted shortest-path problem, with all edges a weight of 1. The algorithm is similar to BFS and we need to use the following data structures:

- A distance table with three columns (each row corresponds to a vertex):
  - Distance from source vertex.
  - Path - contains the name of the vertex through which we get the shortest distance.
- A queue is used to implement breadth-first search. It contains vertices whose distance from the source node has been computed and their adjacent vertices are to be examined.

As an example, consider the following graph and its adjacency list representation.



The adjacency list for this graph is:

$A: B \rightarrow D$
$B: D \rightarrow E$
$C: A \rightarrow F$
$D: F \rightarrow G$
$E: G$
$F: -$
$G: F$

Let $s = C$. The distance from C to C is 0. Initially, distances to all other nodes are not computed, and we initialize the second column in the distance table for all vertices (except C) with -1 as below.

Vertex	Distance[v]	Previous vertex which gave Distance[v]
A	-1	-
B	-1	-
C	0	-
D	-1	-
E	-1	-
F	-1	-
G	-1	-

Algorithm

```
class Vertex:
    def __init__(self, node):
        self.id = node
            self.adjacent = {}
            # Set distance to infinity for all nodes
            self.distance = -1
            # Mark all nodes unvisited
            self.visited = False
            # Predecessor
            self.previous = None
            # .....
class Graph:
    def __init__(self):
        self.vertDictionary = {}
        self.numVertices = 0
            # .....
def UnweightedShortestPath(G,s):
    source = G.getVertex(s)
    source.setDistance(0)
    source.setPrevious(None)
    vertQueue = Queue()
    vertQueue.enQueue(source)
    while (vertQueue.size > 0):
            currentVert = vertQueue.deQueue()
            for nbr in currentVert.getConnections():
                    if nbr.getDistance() == -1:
                            nbr.setDistance(currentVert.getDistance() + 1)
                            nbr.setPrevious(currentVert)
                            vertQueue.enQueue(nbr)
    for v in G.vertDictionary.values():
            print source.getVertexID(), " to ",v.getVertexID(), "-->",v.getDistance()
```

Running time: $O(|E| + |V|)$, if adjacency lists are used. In for loop, we are checking the outgoing edges for a given vertex and the sum of all examined edges in the while loop is equal to the number of edges which gives $O(|E|)$.

If we use matrix representation the complexity is $O(|V|^2)$, because we need to read an entire row in the matrix of length $|V|$ in order to find the adjacent vertices for a given vertex.

Shortest path in Weighted Graph [Dijkstra's]

A famous solution for the shortest path problem was developed by *Dijkstra*. *Dijkstra's* algorithm is a generalization of the BFS algorithm. The regular BFS algorithm cannot solve the shortest path problem as it cannot guarantee that the vertex at the front of the queue is the vertex closest to source s.

Before going to code let us understand how the algorithm works. As in unweighted shortest path algorithm, here too we use the distance table. The algorithm works by keeping the shortest distance of vertex v from the source in the *Distance* table. The value *Distance*[v] holds the distance from s to v. The shortest distance of the source to itself is zero. The *Distance* table for all other vertices is set to -1 to indicate that those vertices are not already processed.

Vertex	Distance[v]	Previous vertex which gave Distance[v]
A	-1	-
B	-1	-
C	0	-
D	-1	-
E	-1	-
F	-1	-
G	-1	-

After the algorithm finishes, the *Distance* table will have the shortest distance from source s to each other vertex v. To simplify the understanding of *Dijkstra's* algorithm, let us assume that the given vertices are maintained in

two sets. Initially the first set contains only the source element and the second set contains all the remaining elements. After the k^{th} iteration, the first set contains k vertices which are closest to the source. These k vertices are the ones for which we have already computed the shortest distances from source.

Notes on Dijkstra's Algorithm

- It uses greedy method: Always pick the next closest vertex to the source.
- It uses priority queue to store unvisited vertices by distance from s.
- It does not work with negative weights.

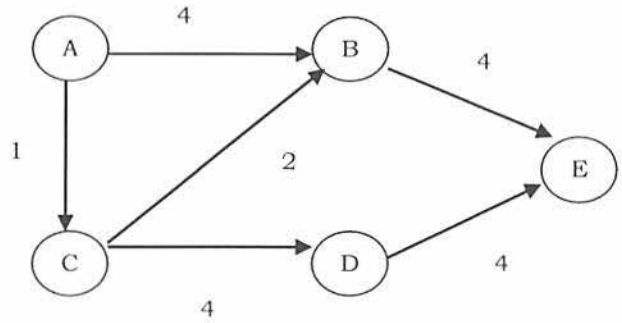
Difference between Unweighted Shortest Path and Dijkstra's Algorithm

1) To represent weights in the adjacency list, each vertex contains the weights of the edges (in addition to their identifier).
2) Instead of ordinary queue we use priority queue [distances are the priorities] and the vertex with the smallest distance is selected for processing.
3) The distance to a vertex is calculated by the sum of the weights of the edges on the path from the source to that vertex.
4) We update the distances in case the newly computed distance is smaller than the old distance which we have already computed.

```
import heapq
def dijkstra(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)
```

The above algorithm can be better understood through an example, which will explain each step that is taken and how *Distance* is calculated. The weighted graph below has 5 vertices from $A - E$.

The value between the two vertices is known as the edge cost between two vertices. For example, the edge cost between A and C is 1. Dijkstra's algorithm can be used to find the shortest path from source A to the remaining vertices in the graph.

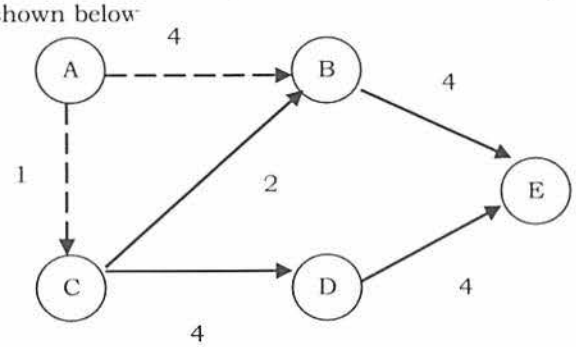


Initially the *Distance* table is:

Vertex	Distance[v]	Previous vertex which gave Distance[v]
A	0	-
B	-1	-
C	-1	-
D	-1	-
E	-1	-

After the first step, from vertex A, we can reach B and C. So, in the *Distance* table we update the reachability of B and C with their costs and the same is shown below

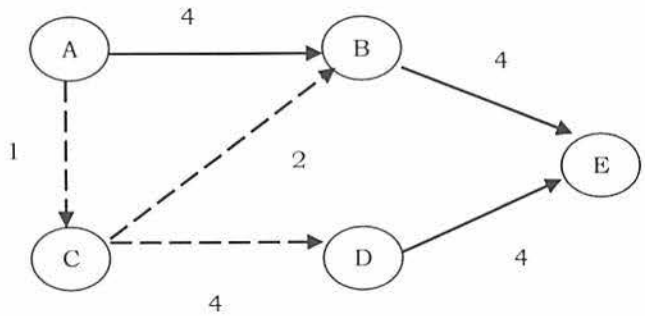
A	0	-
B	4	A
C	1	A
D	-1	-
E	-1	-



Shortest path from B, C from A

Now, let us select the minimum distance among all. The minimum distance vertex is C. That means, we have to reach other vertices from these two vertices (A and C). For example, B can be reached from A and also from C. In this case we have to select the one which gives the lowest cost. Since reaching B through C is giving the minimum cost (1 + 2), we update the *Distance* table for vertex B with cost 3 and the vertex from which we got this cost as C.

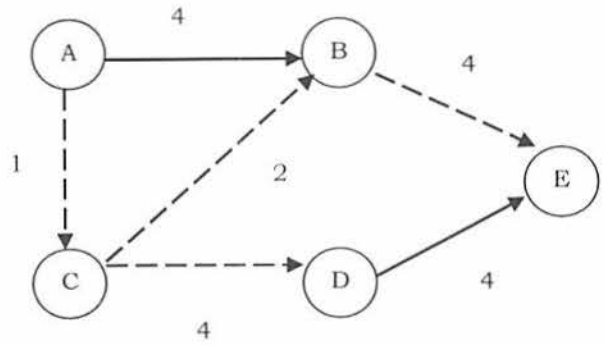
A	0	-
B	3	C
C	1	A
D	5	C
E	-1	-



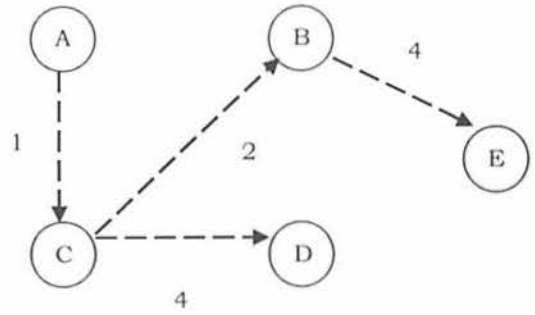
Shortest path to B, D using C as intermediate vertex

The only vertex remaining is E. To reach E, we have to see all the paths through which we can reach E and select the one which gives the minimum cost. We can see that if we use B as the intermediate vertex through C we get the minimum cost.

A	0	-
B	3	C
C	1	A
D	5	C
E	7	B



The final minimum cost tree which Dijkstra's algorithm generates is:



Performance

In Dijkstra's algorithm, the efficiency depends on the number of DeleteMins (V DeleteMins) and updates for priority queues (E updates) that are used. If a *standard binary heap* is used then the complexity is $O(ElogV)$.
The term $ElogV$ comes from E updates (each update takes $logV$) for the standard heap. If the set used is an array then the complexity is $O(E + V^2)$.

Disadvantages of Dijkstra's Algorithm

- As discussed above, the major disadvantage of the algorithm is that it does a blind search, thereby wasting time and necessary resources.
- Another disadvantage is that it cannot handle negative edges. This leads to acyclic graphs and most often cannot obtain the right shortest path.

Relatives of Dijkstra's Algorithm

- The *Bellman-Ford* algorithm computes single-source shortest paths in a weighted digraph. It uses the same concept as that of *Dijkstra's* algorithm but can handle negative edges as well. It has more running time than *Dijkstra's* algorithm.
- Prim's algorithm finds a minimum spanning tree for a connected weighted graph. It implies that a subset of edges that form a tree where the total weight of all the edges in the tree is minimized.

Bellman-Ford Algorithm

If the graph has negative edge costs, then *Dijkstra's* algorithm does not work. The problem is that once a vertex u is declared known, it is possible that from some other, unknown vertex v there is a path back to u that is very negative. In such a case, taking a path from s to v back to u is better than going from s to u without using v. A combination of Dijkstra's algorithm and unweighted algorithms will solve the problem. Initialize the queue with s. Then, at each stage, we *DeQueue* a vertex v. We find all vertices w adjacent to v such that,

$$distance\ to\ v\ +\ weight(v,w) <\ \text{old distance to w}$$

We update w old distance and path, and place w on a queue if it is not already there. A bit can be set for each vertex to indicate presence in the queue. We repeat the process until the queue is empty.

```
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')
```

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.

Overview of Shortest Path Algorithms

Shortest path in unweighted graph $[Modified\ BFS]$	$O(\|E\| + \|V\|)$
Shortest path in weighted graph $[Dijkstra's]$	$O(\|E\|\ log\ \|V\|)$
Shortest path in weighted graph with negative edges $[Bellman - Ford]$	$O(\|E\|.\|V\|)$
Shortest path in weighted acyclic graph	$O(\|E\| + \|V\|)$

9.8 Minimal Spanning Tree

The *Spanning tree* of a graph is a subgraph that contains all the vertices and is also a tree. A graph may have many spanning trees. As an example, consider a graph with 4 vertices as shown below. Let us assume that the corners of the graph are vertices.

For this simple graph, we can have multiple spanning trees as shown below.

The algorithm we will discuss now is *minimum spanning tree* in an undirected graph. We assume that the given graphs are weighted graphs. If the graphs are unweighted graphs then we can still use the weighted graph algorithms by treating all weights as equal. A *minimum spanning tree* of an undirected graph G is a tree formed from graph edges that connect all the vertices of G with minimum total cost (weights). A minimum spanning tree exists only if the graph is connected. There are two famous algorithms for this problem:

- *Prim's* Algorithm
- *Kruskal's* Algorithm

Prim's Algorithm

Prim's algorithm is almost the same as Dijkstra's algorithm. As in Dijkstra's algorithm, in Prim's algorithm we keep the values *distance* and *paths* in the distance table. The only exception is that since the definition of *distance* is different, the updating statement also changes a little. The update statement is simpler than before.

```
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)
```

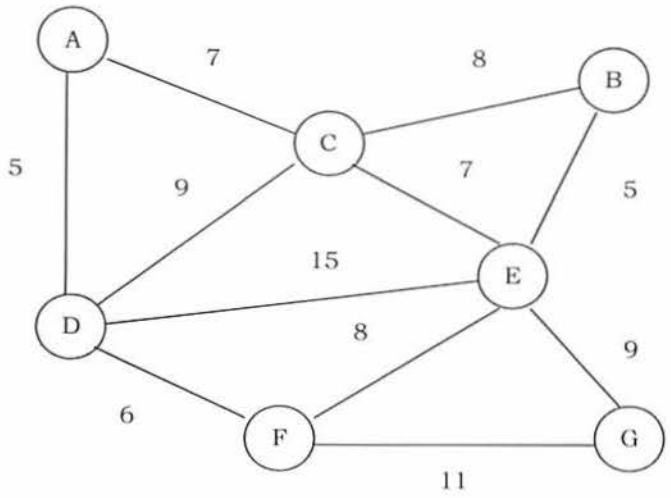
The entire implementation of this algorithm is identical to that of Dijkstra's algorithm. The running time is $O(|V|^2)$ without heaps [good for dense graphs], and $O(ElogV)$ using binary heaps [good for sparse graphs].

Kruskal's Algorithm

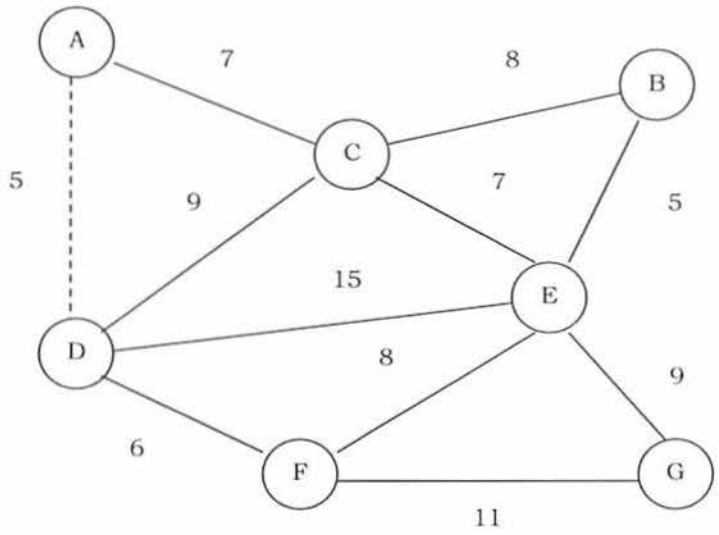
The algorithm starts with V different trees (V is the vertices in the graph). While constructing the minimum spanning tree, every time Kruskal's alorithm selects an edge that has minimum weight and then adds that edge if it doesn't create a cycle. So, initially, there are |V| single-node trees in the forest. Adding an edge merges two trees into one. When the algorithm is completed, there will be only one tree, and that is the minimum spanning tree. There are two ways of implementing Kruskal's algorithm:

- By using Disjoint Sets: Using UNION and FIND operations
- By using Priority Queues: Maintains weights in priority queue

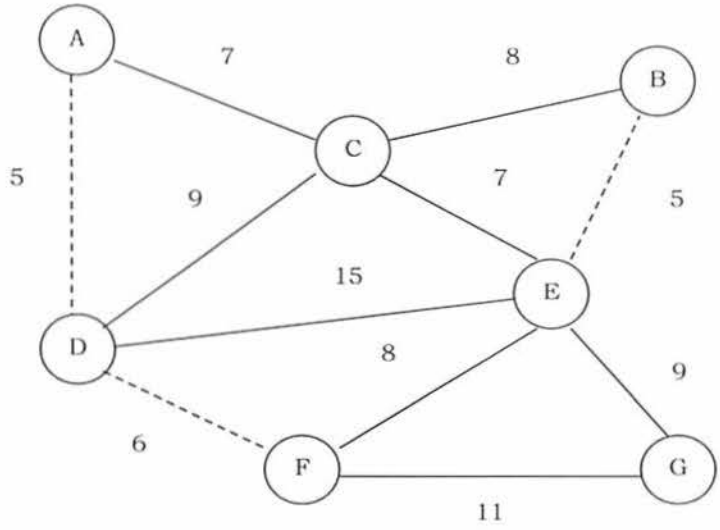
The appropriate data structure is the UNION/FIND algorithm [for implementing forests]. Two vertices belong to the same set if and only if they are connected in the current spanning forest. Each vertex is initially in its own set. If u and v are in the same set, the edge is rejected because it forms a cycle. Otherwise, the edge is accepted, and a UNION is performed on the two sets containing u and v. As an example, consider the following graph (the edges show the weights).



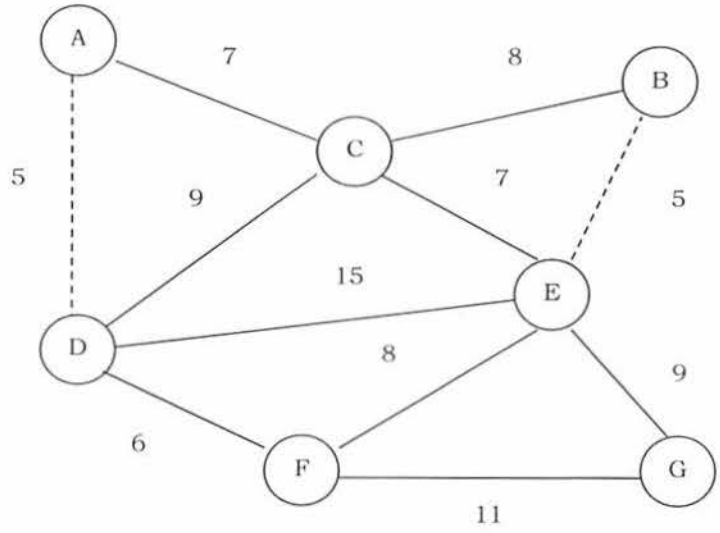
Now let us perform Kruskal's algorithm on this graph. We always select the edge which has minimum weight.



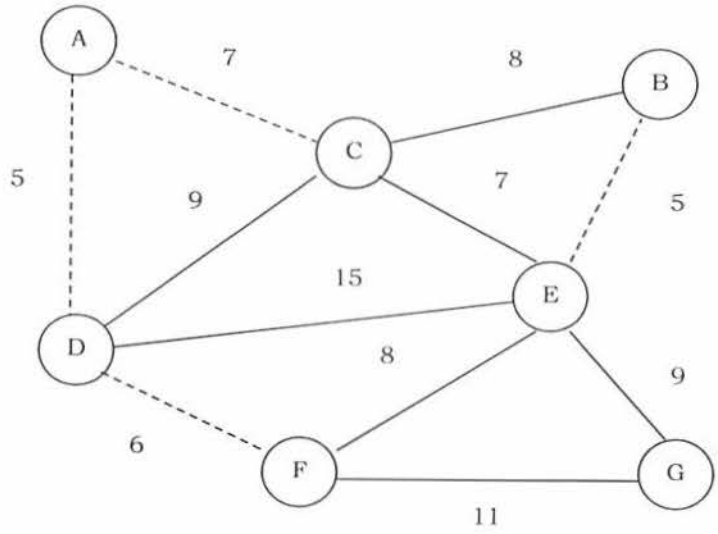
From the above graph, the edges which have minimum weight (cost) are: AD and BE. From these two we can select one of them and let us assume that we select AD (dotted line).



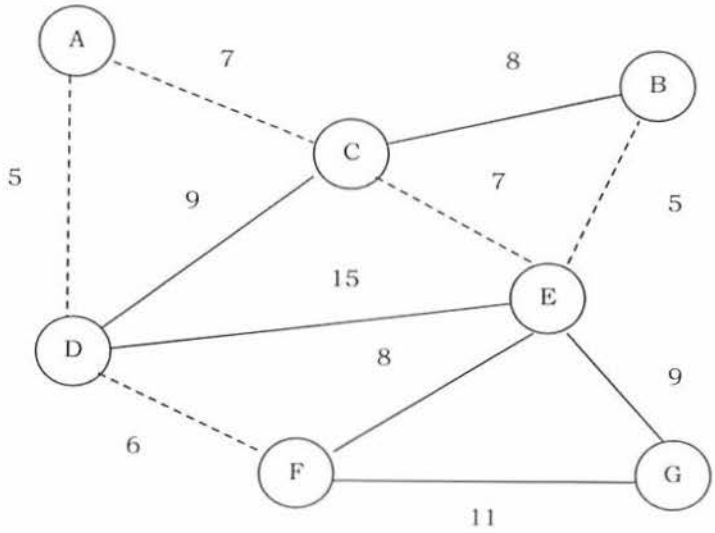
DF is the next edge that has the lowest cost (6).



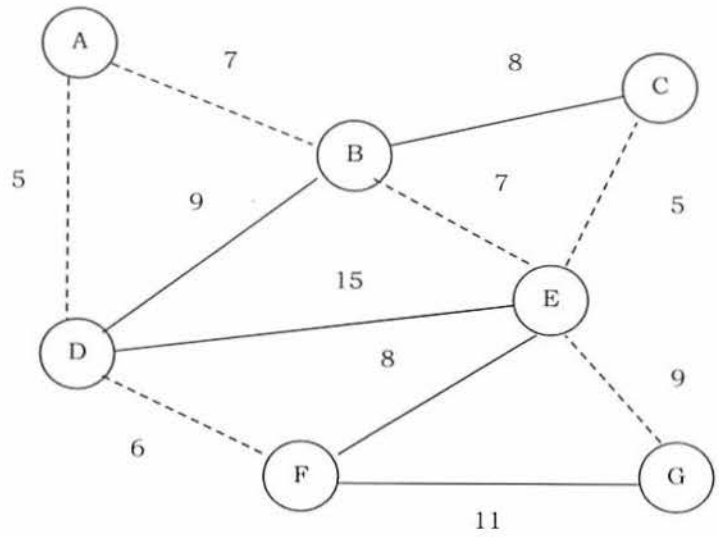
BE now has the lowest cost and we select it (dotted lines indicate selected edges).



Next, AC and CE have the low cost of 7 and we select AC.



Then we select CE as its cost is 7 and it does not form a cycle.



The next low cost edges are CB and EF. But if we select CB, then it forms a cycle. So we discard it. This is also the case with EF. So we should not select those two. And the next low cost is 9 (BD and EG). Selecting BD forms a cycle so we discard it. Adding EG will not form a cycle and therefore with this edge we complete all vertices of the graph.

```
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
```

Note: For implementation of UNION and FIND operations, refer to the *Disjoint Sets ADT* chapter.
The worst-case running time of this algorithm is O(*ElogE*), which is dominated by the heap operations. That means, since we are constructing the heap with E edges, we need O(*ElogE*) time to do that.

9.9 Graph Algorithms: Problems & Solutions

Problem-1 In an undirected simple graph with n vertices, what is the maximum number of edges? Self-loops are not allowed.

Solution: Since every node can connect to all other nodes, the first node can connect to $n-1$ nodes. The second node can connect to $n-2$ nodes [since one edge is already there from the first node]. The total number of edges is: $1+2+3+\cdots+n-1=\frac{n(n-1)}{2}$ edges.

Problem-2 How many different adjacency matrices does a graph with n vertices and E edges have?

Solution: It's equal to the number of permutations of n elements. i.e., $n!$.

Problem-3 How many different adjacency lists does a graph with n vertices have?

Solution: It's equal to the number of permutations of edges. i.e., $E!$.

Problem-4 Which undirected graph representation is most appropriate for determining whether or not a vertex is isolated (is not connected to any other vertex)?

Solution: Adjacency List. If we use the adjacency matrix, then we need to check the complete row to determine whether that vertex has edges or not. By using the adjacency list, it is very easy to check, and it can be done just by checking whether that vertex has NULL for next pointer or not [NULL indicates that the vertex is not connected to any other vertex].

Problem-5 For checking whether there is a path from source s to target t, which one is best between disjoint sets and DFS?

Solution: The table below shows the comparison between disjoint sets and DFS. The entries in the table represent the case for any pair of nodes (for s and t).

Method	Processing Time	Query Time	Space
Union-Find	$V + E \log V$	$\log V$	V
DFS	$E + V$	1	$E + V$

Problem-6 What is the maximum number of edges a directed graph with n vertices can have and still not contain a directed cycle?

Solution: The number is $V(V-1)/2$. Any directed graph can have at most n^2 edges. However, since the graph has no cycles it cannot contain a self loop, and for any pair x, y of vertices, at most one edge from (x, y) and (y, x) can be included. Therefore the number of edges can be at most $(V^2 - V)/2$ as desired. It is possible to achieve $V(V-1)/2$ edges. Label n nodes $1, 2 \ldots n$ and add an edge (x, y) if and only if $x < y$. This graph has the appropriate number of edges and cannot contain a cycle (any path visits an increasing sequence of nodes).

Problem-7 How many simple directed graphs with no parallel edges and self-loops are possible in terms of V?

Solution: $(V) \times (V-1)$. Since, each vertex can connect to $V-1$ vertices without self-loops.

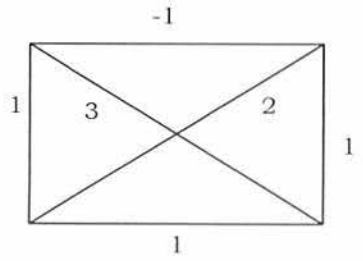
Problem-8 What are the differences between DFS and BFS?

Solution:

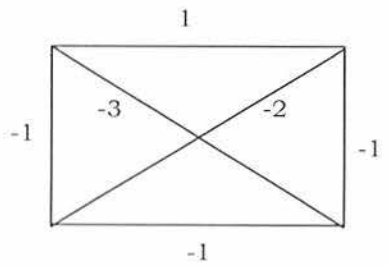
DFS	BFS
Backtracking is possible from a dead end.	Backtracking is not possible.
Vertices from which exploration is incomplete are processed in a LIFO order	The vertices to be explored are organized as a FIFO queue.
The search is done in one particular direction	The vertices at the same level are maintained in parallel.

Problem-9 Earlier in this chapter, we discussed minimum spanning tree algorithms. Now, give an algorithm for finding the maximum-weight spanning tree in a graph.

Solution:



Given graph



Transformed graph with negative edge weights

Using the given graph, construct a new graph with the same nodes and edges. But instead of using the same weights, take the negative of their weights. That means, weight of an edge = negative of weight of the corresponding edge in the given graph. Now, we can use existing *minimum spanning tree* algorithms on this new graph. As a result, we will get the maximum-weight spanning tree in the original one.

Problem-10 Give an algorithm for checking whether a given graph G has simple path from source s to destination d. Assume the graph G is represented using the adjacent matrix.

Solution: Let us assume that the structure for the graph is:

```
class Graph(object):
    def __init__(self, graph_dict={}):
        """ initializes a graph object """
        self.graphDictionary = graph_dict
    def vertices(self):
        """ returns the vertices of a graph """
        return list(self.graphDictionary.keys())
    def edges(self):
```

```
        """ returns the edges of a graph """
        return self.generateEdges()
    def addVertex(self, vertex):
        """ If the vertex "vertex" is not in
            self.graphDictionary, a key "vertex" with an empty
            list as a value is added to the dictionary.
            Otherwise nothing has to be done.
        """
        if vertex not in self.graphDictionary:
            self.graphDictionary[vertex] = []
    def addEdge(self, edge):
        """ assumes that edge is of type set, tuple or list;
            between two vertices can be multiple edges!
        """
        edge = set(edge)
        (vertex1, vertex2) = tuple(edge)
        if vertex1 in self.graphDictionary:
            self.graphDictionary[vertex1].append(vertex2)
        else:
            self.graphDictionary[vertex1] = [vertex2]
```

The following method finds a path from a start vertex to an end vertex:

```
def checkForPath(self, source, destination, path=[]):
    """ find a path from source to destination
        in graph """
    graph = self.graphDictionary
    path = path + [source]
    if source == destination:
        return path
    if source not in graph:
        return None
    for vertex in graph[source]:
        if vertex not in path:
            extendedPath = self.checkForPath(vertex, destination, path)
            if extendedPath:
                return extendedPath
    return None
if __name__ == "__main__":
    g = { "a" : ["b", "c"],
          "b" : ["d", "e"],
          "c" : ["d", "e"],
          "d" : ["e"],
          "e" : ["a"],
          "f" : []
        }
    graph = Graph(g)
    print("Vertices of graph:")
    print(graph.vertices())
    print("Edges of graph:")
    print(graph.edges())
    pathResult = graph.checkForPath("a", "e")
    if(pathResult == None):
        print "No path between source and destination"
    else:
        print pathResult
    pathResult = graph.checkForPath("a", "f")
    if(pathResult == None):
        print "No path between source and destination"
    else:
        print pathResult
```

Time Complexity: O(E). In the above algorithm, for each node, since we are not calling *DFS* on all of its neighbors (discarding through *if* condition), Space Complexity: O(V).

Problem-11 Count simple paths for a given graph G has simple path from source s to destination d? Assume the graph is represented using the adjacent matrix.

Solution: Similar to the discussion in Problem-10, start at one node and call DFS on that node. As a result of this call, it visits all the nodes that it can reach in the given graph. That means it visits all the nodes of the connected component of that node. If there are any nodes that have not been visited, then again start at one of those nodes and call DFS.

Before the first DFS in each connected component, increment the connected components *count*. Continue this process until all of the graph nodes are visited. As a result, at the end we will get the total number of connected components. The implementation based on this logic is given below:

```
def countSimplePathsFromSourceToDestination(self, source, destination, path=[]):
    """ find all paths from source to destination in graph """
    graph = self.graphDictionary
    path = path + [source]
    if source == destination:
        return [path]
    if source not in graph:
        return []
    paths = []
    for vertex in graph[source]:
        if vertex not in path:
            extended_paths = self.countSimplePathsFromSourceToDestination(vertex, destination, path)
            for p in extended_paths:
                paths.append(p)
    return paths
if __name__ == "__main__":
    g = { "a" : ["b", "c"],
          "b" : ["d", "e"],
          "c" : ["d", "e"],
          "d" : ["e"],
          "e" : ["a"],
          "f" : []
        }
    graph = Graph(g)
    print("Vertices of graph:")
    print(graph.vertices())
    print("Edges of graph:")
    print(graph.edges())
    pathResult = graph.countSimplePathsFromSourceToDestination("a", "e")
    if(len(pathResult) == 0):
        print "No path between source and destination"
    else:
        print pathResult
    pathResult = graph.countSimplePathsFromSourceToDestination("a", "f")
    if(len(pathResult) == 0):
        print "No path between source and destination"
    else:
        print pathResult
```

Problem-12 **All pairs shortest path problem:** Find the shortest graph distances between every pair of vertices in a given graph. Let us assume that the given graph does not have negative edges.

Solution: The problem can be solved using n applications of *Dijkstra's* algorithm. That means we apply *Dijkstra's* algorithm on each vertex of the given graph. This algorithm does not work if the graph has edges with negative weights.

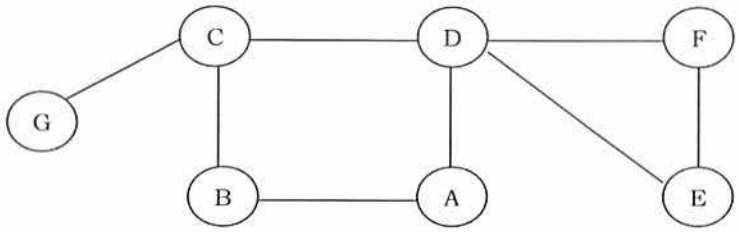
Problem-13 In Problem-12, how do we solve the all pairs shortest path problem if the graph has edges with negative weights?

Solution: This can be solved by using the *Floyd – Warshall algorithm*. This algorithm also works in the case of a weighted graph where the edges have negative weights. This algorithm is an example of Dynamic Programming – refer to the *Dynamic Programming* chapter.

Problem-14 **DFS Application:** *Cut Vertex* or *Articulation Points*

Solution: In an undirected graph, a *cut vertex* (or articulation point) is a vertex, and if we remove it, then the graph splits into two disconnected components. As an example, consider the following figure. Removal of the "*D*" vertex divides the graph into two connected components ($\{E, F\}$ and $\{A, B, C, G\}$).

Similarly, removal of the "*C*" vertex divides the graph into ($\{G\}$ and $\{A, B, D, E, F\}$). For this graph, *A* and *C* are the cut vertices.

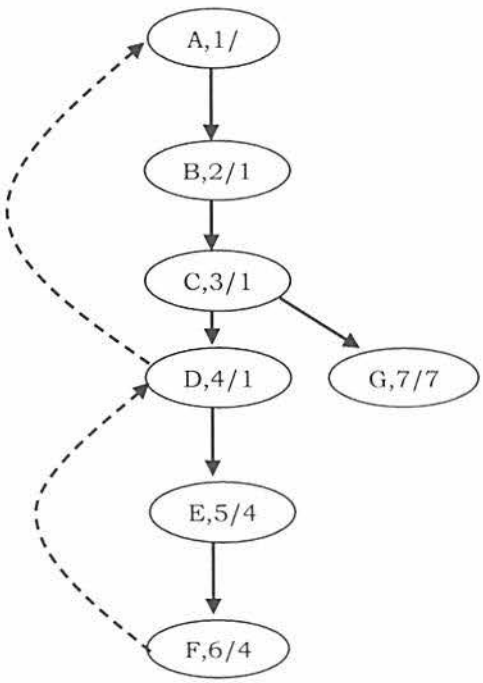


Note: A connected, undirected graph is called $bi-connected$ if the graph is still connected after removing any vertex.

DFS provides a linear-time algorithm (O(n)) to find all cut vertices in a connected graph. Starting at any vertex, call a *DFS* and number the nodes as they are visited. For each vertex v, we call this DFS number $dfsnum$(v). The tree generated with DFS traversal is called *DFS spanning tree*. Then, for every vertex v in the *DFS* spanning tree, we compute the lowest-numbered vertex, which we call $low(v)$, that is reachable from v by taking zero or more tree edges and then possibly one back edge (in that order).

Based on the above discussion, we need the following information for this algorithm: the $dfsnum$ of each vertex in the *DFS* tree (once it gets visited), and for each vertex v, the lowest depth of neighbors of all descendants of v in the *DFS* tree, called the *low*.

The $dfsnum$ can be computed during DFS. The low of v can be computed after visiting all descendants of v (i.e., just before v gets popped off the *DFS* stack) as the minimum of the $dfsnum$ of all neighbors of v (other than the parent of v in the *DFS* tree) and the *low* of all children of v in the *DFS* tree.



The root vertex is a cut vertex if and only if it has at least two children. A non-root vertex u is a cut vertex if and only if there is a son v of u such that $low(v) \geq dfsnum(u)$. This property can be tested once the *DFS* is returned from every child of u (that means, just before u gets popped off the DFS stack), and if true, u separates the

graph into different bi-connected components. This can be represented by computing one bi-connected component out of every such v (a component which contains v will contain the sub-tree of v, plus u), and then erasing the sub-tree of v from the tree.

For the given graph, the *DFS* tree with *dfsnum/low* can be given as shown in the figure below. The implementation for the above discussion is:

```
import math
dfsnum = [0] * G.numVertices
num = 0
low = [0] * G.numVertices
def CutVertices( G, u ) :
    low[u] = num
    dfsnum[u] = num
    num = num +1
    for v in range(0,G.numVertices):
        if(G.adjMatrix[u][v] and dfsnum[v] == -1):
            CutVertices( v )
            if(low[v] > dfsnum[u]):
                print "Cut Vetex:",u
            low[u] = min ( low[u] , low[v] )
        else:   # (u,v) is a back edge
            low[u ] = min(low[u] , dfsnum[v])
```

Problem-15 Let G be a connected graph of order n. What is the maximum number of cut-vertices that G can contain?

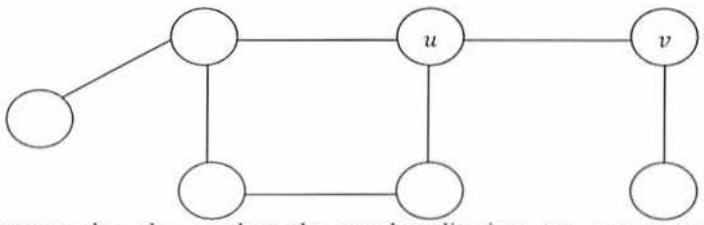
Solution: $n - 2$. As an example, consider the following graph. In the graph below, except for the vertices 1 and n, all the remaining vertices are cut vertices. This is because removing 1 and n vertices does not split the graph into two. This is a case where we can get the maximum number of cut vertices.

Problem-16 **DFS Application:** *Cut Bridges* or *Cut Edges*

Solution:

Definition: Let G be a connected graph. An edge uv in G is called a *bridge* of G if $G - uv$ is disconnected.

As an example, consider the following graph.



In the above graph, if we remove the edge uv then the graph splits into two components. For this graph, uv is a bridge. The discussion we had for cut vertices holds good for bridges also. The only change is, instead of printing the vertex, we give the edge. The main observation is that an edge (u, v) cannot be a bridge if it is part of a cycle. If (u, v) is not part of a cycle, then it is a bridge.

We can detect cycles in *DFS* by the presence of back edges. (u, v) is a bridge if and only if none of v or v's children has a back edge to u or any of u's ancestors. To detect whether any of v's children has a back edge to u's parent, we can use a similar idea as above to see what is the smallest *dfsnum* reachable from the subtree rooted at v.

```
import math
dfsnum = [0] * G.numVertices
num = 0
low = [0] * G.numVertices
def Bridges( G, u ) :
    low[u] = num
    dfsnum[u] = num
    num = num +1
    for v in range(0,G.numVertices):
        if(G.adjMatrix[u][v] and dfsnum[v] == -1):
```

```
                cutVertices( v )
                if(low[v] > dfsnum[u]):
                        print (u,v)  #as a bridge
                low[u] = min ( low[u] , low[v] )
        else:   # (u,v) is a back edge
                low[u ] = min(low[u] , dfsnum[v])
```

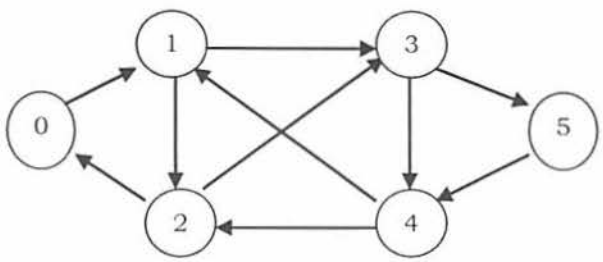
Problem-17 **DFS Application:** Discuss *Euler* Circuits

Solution: Before discussing this problem let us see the terminology:

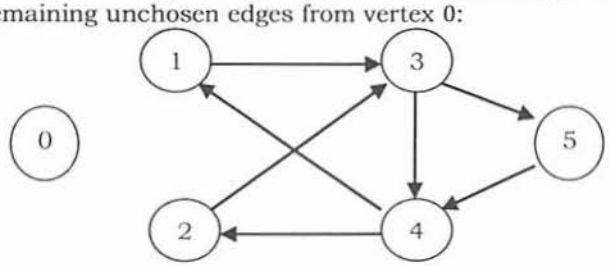
- *Eulerian tour* – a path that contains all edges without repetition.
- *Eulerian circuit* – a path that contains all edges without repetition and starts and ends in the same vertex.
- *Eulerian graph* – a graph that contains an Eulerian circuit.
- *Even vertex*: a vertex that has an even number of incident edges.
- *Odd vertex*: a vertex that has an odd number of incident edges.

Euler circuit: For a given graph we have to reconstruct the circuits using a pen, drawing each line exactly once. We should not lift the pen from the paper while drawing. That means, we must find a path in the graph that visits every edge exactly once and this problem is called an *Euler path* (also called *Euler tour*) or *Euler circuit problem*. This puzzle has a simple solution based on DFS.

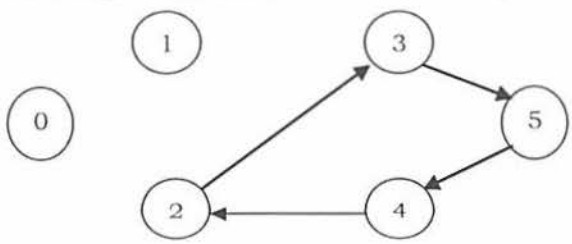
An *Euler* circuit exists if and only if the graph is connected and the number of neighbors of each vertex is even. Start with any node, select any untraversed outgoing edge, and follow it. Repeat until there are no more remaining unselected outgoing edges. For example, consider the following graph: A legal Euler Circuit of this graph is 0 1 3 4 1 2 3 5 4 2 0.



If we start at vertex 0, we can select the edge to vertex 1, then select the edge to vertex 2, then select the edge to vertex 0. There are now no remaining unchosen edges from vertex 0:



We now have a circuit 0,1,2,0 that does not traverse every edge. So, we pick some other vertex that is on that circuit, say vertex 1. We then do another depth first search of the remaining edges. Say we choose the edge to node 3, then 4, then 1. Again we are stuck. There are no more unchosen edges from node 1. We now splice this path 1,3,4,1 into the old path 0,1,2,0 to get: 0,1,3,4,1,2,0. The unchosen edges now look like this:



We can pick yet another vertex to start another DFS. If we pick vertex 2, and splice the path 2,3,5,4,2, then we get the final circuit 0,1,3,4,1,2,3,5,4,2,0.

A similar problem is to find a simple cycle in an undirected graph that visits every vertex. This is known as the *Hamiltonian cycle problem*. Although it seems almost identical to the *Euler* circuit problem, no efficient algorithm for it is known.

Notes:

- A connected undirected graph is *Eulerian* if and only if every graph vertex has an even degree, or exactly two vertices with an odd degree.
- A directed graph is *Eulerian* if it is strongly connected and every vertex has an equal *in* and *out* degree.

Application: A postman has to visit a set of streets in order to deliver mails and packages. He needs to find a path that starts and ends at the post-office, and that passes through each street (edge) exactly once. This way the postman will deliver mails and packages to all the necessary streets, and at the same time will spend minimum time/effort on the road.

Problem-18 **DFS Application:** Finding Strongly Connected Components.

Solution: This is another application of DFS. In a directed graph, two vertices u and v are strongly connected if and only if there exists a path from u to v and there exists a path from v to u. The strong connectedness is an equivalence relation.

- A vertex is strongly connected with itself
- If a vertex u is strongly connected to a vertex v, then v is strongly connected to u
- If a vertex u is strongly connected to a vertex v, and v is strongly connected to a vertex x, then u is strongly connected to x

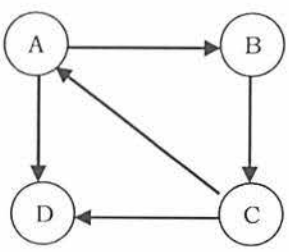
What this says is, for a given directed graph we can divide it into strongly connected components. This problem can be solved by performing two depth-first searches. With two DFS searches we can test whether a given directed graph is strongly connected or not. We can also produce the subsets of vertices that are strongly connected.

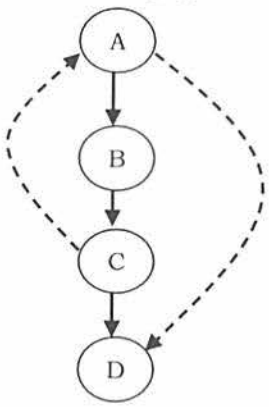
Algorithm

- Perform DFS on given graph G.
- Number vertices of given graph G according to a post-order traversal of depth-first spanning forest.
- Construct graph G_r by reversing all edges in G.
- Perform DFS on G_r: Always start a new DFS (initial call to Visit) at the highest-numbered vertex.
- Each tree in the resulting depth-first spanning forest corresponds to a strongly-connected component.

Why this algorithm works?

Let us consider two vertices, v and w. If they are in the same strongly connected component, then there are paths from v to w and from w to v in the original graph G, and hence also in G_r. If two vertices v and w are not in the same depth-first spanning tree of G_r, clearly they cannot be in the same strongly connected component. As an example, consider the graph shown below on the left. Let us assume this graph is G.



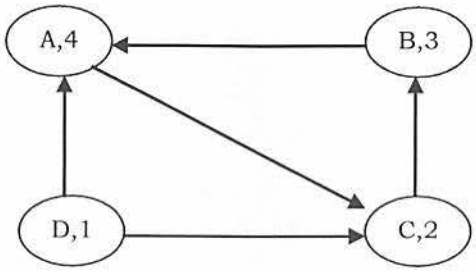


Now, as per the algorithm, performing *DFS* on this G graph gives the following diagram. The dotted line from C to A indicates a back edge.

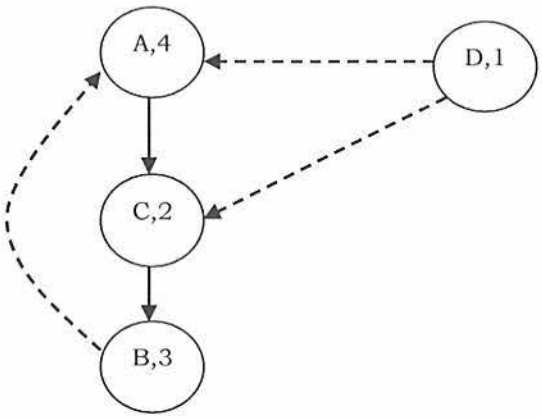
Now, performing post order traversal on this tree gives: D, C, B and A.

Vertex	Post Order Number
A	4
B	3
C	2
D	1

Now reverse the given graph G and call it G_r and at the same time assign postorder numbers to the vertices. The reversed graph G_r will look like:



The last step is performing DFS on this reversed graph G_r. While doing *DFS*, we need to consider the vertex which has the largest DFS number. So, first we start at A and with *DFS* we go to C and then B. At B, we cannot move further. This says that $\{A, B, C\}$ is a strongly connected component. Now the only remaining element is D and we end our second *DFS* at D. So the connected components are: $\{A, B, C\}$ and $\{D\}$.



The implementation based on this discussion can be shown as:

```
def stronglyConnectedComponents(G):
    indexCounter = [0]
    stack = []
    lowLinks = {}
    index = {}
    result = []
    def strongConnect(node):
        # set the depth index for this node to the smallest unused index
        index[node] = indexCounter[0]
        lowLinks[node] = indexCounter[0]
        indexCounter[0] += 1
        stack.append(node)

        # Consider successors of `node`
        try:
            successors = G[node]
        except:
            successors = []
        for successor in successors:
            if successor not in lowLinks:
                # Successor has not yet been visited; recurse on it
                strongConnect(successor)
                lowLinks[node] = min(lowLinks[node],lowLinks[successor])
            elif successor in stack:
                # the successor is in the stack and hence in the current strongly connected component (SCC)
                lowLinks[node] = min(lowLinks[node],index[successor])

        # If `node` is a root node, pop the stack and generate an SCC
        if lowLinks[node] == index[node]:
            connectedComponent = []

            while True:
                successor = stack.pop()
                connectedComponent.append(successor)
```

```
            if successor == node: break
        component = tuple(connectedComponent)
        # storing the result
        result.append(component)
    for node in G:
        if node not in lowLinks:
            strongConnect(node)
    return result
```

Problem-19 Count the number of connected components of Graph G which is represented in the adjacent matrix.

Solution: This problem can be solved with one extra counter in *DFS*.

```
def dfs(G, currentVert, visited):
    visited[currentVert]=True                          # mark the visited node
    print "traversal: " + currentVert.getVertexID()
    for nbr in currentVert.getConnections():           # take a neighbouring node
            if nbr not in visited: #condition to check whether the neighbour node is already visited
                    dfs(G, nbr, visited) #recursively traverse the neighbouring node
def countConnectedComponentsWithDFS(G):
    visited = {} # Dictionary to mark the visited nodes
    count = 0
    for currentVert in G:                  # G contains vertex objects
            if currentVert not in visited: # Start traversing from the root node only if its not visited
                    count += 1
                    dfs(G, currentVert, visited) # For a connected graph this is called only once
    return count
```

Time Complexity: Same as that of DFS and it depends on implementation. With adjacency matrix the complexity is $O(|E| + |V|)$ and with adjacency matrix the complexity is $O(|V|^2)$.

Problem-20 Can we solve the Problem-19, using BFS?

Solution: Yes. This problem can be solved with one extra counter in BFS.

```
def bfs(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 currentVert.getVertexID()
            for nbr in currentVert.getConnections():
                    if (nbr.getColor() == 'white'):
                            nbr.setColor('gray')
                            nbr.setDistance(currentVert.getDistance() + 1)
                            nbr.setPrevious(currentVert)
                            vertQueue.enQueue(nbr)
                    currentVert.setColor('black')
def countConnectedComponentsWithBFS(G):
    edges = []
    count = 0
    for v in G:
            if (v.getColor() == 'white'):
                    count += 1
                    bfs(G, v.getVertexID())
    print count
```

Time Complexity: Same as that of *BFS* and it depends on implementation. With adjacency matrix the complexity is $O(|E| + |V|)$ and with adjacency matrix the complexity is $O(|V|^2)$.

Problem-21 Let us assume that $G(V, E)$ is an undirected graph. Give an algorithm for finding a spanning tree which takes $O(|E|)$ time complexity (not necessarily a minimum spanning tree).

Solution: The test for a cycle can be done in constant time, by marking vertices that have been added to the set S. An edge will introduce a cycle, if both its vertices have already been marked.

Algorithm:

```
S = {} # Assume S is a set
for each edge e in E:
    if(adding e to S doesn't form a cycle):
        add e to S
        mark e
```

Problem-22 Is there any other way of solving 0?

Solution: Yes. We can run *BFS* and find the *BFS* tree for the graph (level order tree of the graph). Then start at the root element and keep moving to the next levels and at the same time we have to consider the nodes in the next level only once. That means, if we have a node with multiple input edges then we should consider only one of them; otherwise they will form a cycle.

Problem-23 Detecting a cycle in an undirected graph

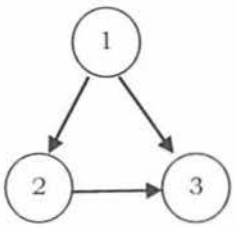
Solution: An undirected graph is acyclic if and only if a *DFS* yields no back edges, edges (u, v) where v has already been discovered and is an ancestor of u.

- Execute *DFS* on the graph.
- If there is a back edge - the graph has a cycle.

If the graph does not contain a cycle, then $|E| < |V|$ and *DFS* cost $O(|V|)$. If the graph contains a cycle, then a back edge is discovered after $2|V|$ steps at most.

Problem-24 Detecting a cycle in DAG

Solution:



Cycle detection on a graph is different than on a tree. This is because in a graph, a node can have multiple parents. In a tree, the algorithm for detecting a cycle is to do a depth first search, marking nodes as they are encountered. If a previously marked node is seen again, then a cycle exists. This won't work on a graph. Let us consider the graph shown in the figure below. If we use a tree cycle detection algorithm, then it will report the wrong result. That means that this graph has a cycle in it. But the given graph does not have a cycle in it. This is because node 3 will be seen twice in a *DFS* starting at node 1.

The cycle detection algorithm for trees can easily be modified to work for graphs. The key is that in a *DFS* of an acyclic graph, a node whose descendants have all been visited can be seen again without implying a cycle. But, if a node is seen for the second time before all its descendants have been visited, then there must be a cycle.

Can you see why this is? Suppose there is a cycle containing node A. This means that A must be reachable from one of its descendants. So when the *DFS* is visiting that descendant, it will see *A* again, before it has finished visiting all of *A's* descendants. So there is a cycle.

In order to detect cycles, we can modify the depth first search.

```
def DetectCycle(G) :
    for i in range(0, G.numVertices):
            Visited[s]=0
            Predecessor[i] = 0
    for i in range(0, G.numVertices):
            if(not Visited[i] and HasCycle(G, i)):
                    return 1
    return False
def HasCycle(G, u) :
    Visited[u]=1
    for i in range(0, G.numVertices):
            if(G.adjMatrix[s][i]) :
                    if(Predecessor[i] != u and Visited[i]):
                            return 1
                    else:
                            Predecessor[i] = u
```

```
                    return HasCycle(G, i)
    return 0
```

Time Complexity: $O(V + E)$.

Problem-25 For Problem-24, is there any other way of solving the problem?

Solution: We can topological sort to check whether a given graph is directed acyclic or not. As seen in topological sort section, it will return None if there is a cycle in given directed graph.

```
def isDirectedAcyclicGraph(G):
  """Return True if the graph G is a directed acyclic graph (DAG). Otherwise return False. """
  if topologicalSort(G) :          # Refer Topological sort section for topologicalSort()
    return True
  else:
    return False
```

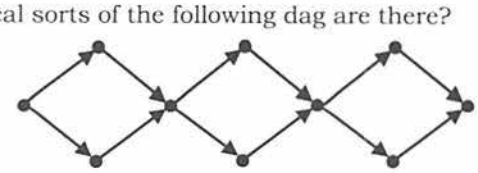
Problem-26 Given a directed acyclic graph, give an algorithm for finding its depth.

Solution: If it is an undirected graph, we can use the simple unweighted shortest path algorithm (check *Shortest Path Algorithms* section). We just need to return the highest number among all distances. For directed acyclic graph, we can solve by following the similar approach which we used for finding the depth in trees. In trees, we have solved this problem using level order traversal (with one extra special symbol to indicate the end of the level).

```
def BFSTraversal(G,s):
  global maxPathLength
  pathLength = 0
  start = G.getVertex(s)
  start.setDistance(0)
  start.setPrevious(None)
  vertQueue = Queue()
  vertQueue.enQueue(start)
  vertQueue.enQueue(None)
  while (vertQueue.size > 0):
        currentVert = vertQueue.deQueue()
        if(currentVert == None):
              pathLength += 1
              if vertQueue.size > 0:
                    vertQueue.enQueue(None)
              continue
        print currentVert.getVertexID()
        for nbr in currentVert.getConnections():
              if (nbr.getColor() == 'white'):
                    nbr.setColor('gray')
                    nbr.setDistance(currentVert.getDistance() + 1)
                    nbr.setPrevious(currentVert)
                    vertQueue.enQueue(nbr)
              currentVert.setColor('black')
  if pathLength > maxPathLength:
        maxPathLength = pathLength
maxPathLength = 0
def LongestPathInDAG(G):
  for v in G:
        if (v.getColor() == 'white'):
              BFSTraversal(G, v.getVertexID())
  return maxPathLength
```

Total running time is $O(V + E)$.

Problem-27 How many topological sorts of the following dag are there?



Solution: If we observe the above graph there are three stages with 2 vertices. In the early discussion of this chapter, we saw that topological sort picks the elements with zero indegree at any point of time. At each of the

two vertices stages, we can first process either the top vertex or the bottom vertex. As a result, at each of these stages we have two possibilities. So the total number of possibilities is the multiplication of possibilities at each stage and that is, $2 \times 2 \times 2 = 8$.

Problem-28 **Unique topological ordering:** Design an algorithm to determine whether a directed graph has a unique topological ordering.

Solution: A directed graph has a unique topological ordering if and only if there is a directed edge between each pair of consecutive vertices in the topological order. This can also be defined as: a directed graph has a unique topological ordering if and only if it has a Hamiltonian path. If the digraph has multiple topological orderings, then a second topological order can be obtained by swapping a pair of consecutive vertices.

Problem-29 Let us consider the prerequisites for courses at *IIT Bombay*. Suppose that all prerequisites are mandatory, every course is offered every semester, and there is no limit to the number of courses we can take in one semester. We would like to know the minimum number of semesters required to complete the major. Describe the data structure we would use to represent this problem, and outline a linear time algorithm for solving it.

Solution: Use a directed acyclic graph (DAG). The vertices represent courses and the edges represent the prerequisite relation between courses at *IIT Bombay*. It is a DAG, because the prerequisite relation has no cycles. The number of semesters required to complete the major is one more than the longest path in the dag. This can be calculated on the DFS tree recursively in linear time. The longest path out of a vertex x is 0 if x has outdegree 0, otherwise it is $1 + max\ \{longest\ path\ out\ of\ y \mid (x, y)\ is\ an\ edge\ of\ G\}$.

Problem-30 At a university let's say *IIT Bombay*), there is a list of courses along with their prerequisites. That means, two lists are given:

A - Courses list

B – Prerequisites: B contains couples (x, y) where $x, y \in A$ indicating that course x can't be taken before course y.

Let us consider a student who wants to take only one course in a semester. Design a schedule for this student.

Example: A = {C-Lang, Data Structures, OS, CO, Algorithms, Design Patterns, Programming }. B = { (C-Lang, CO), (OS, CO), (Data Structures, Algorithms), (Design Patterns, Programming) }. *One possible schedule could be*:

Semester 1: Data Structures
Semester 2: Algorithms
Semester 3: C-Lang
Semester 4: OS
Semester 5: CO
Semester 6: Design Patterns
Semester 7: Programming

Solution: The solution to this problem is exactly the same as that of topological sort. Assume that the courses names are integers in the range $[1..n]$, n is known (n is not constant). The relations between the courses will be represented by a directed graph $G = (V, E)$, where V are the set of courses and if course i is prerequisite of course j, E will contain the edge (i, j). Let us assume that the graph will be represented as an Adjacency list.

First, let's observe another algorithm to topologically sort a DAG in $O(|V| + |E|)$.

- Find in-degree of all the vertices - $O(|V| + |E|)$
- Repeat:
  Find a vertex v with in-degree=0 - $O(|V|)$
  Output v and remove it from G, along with its edges - $O(|V|)$
  Reduce the in-degree of each node u such as (v, u) was an edge in G and keep a list of vertices with in-degree=0 - $O(degree(v))$
  Repeat the process until all the vertices are removed

The time complexity of this algorithm is also the same as that of the topological sort and it is $O(|V| + |E|)$.

Problem-31 In Problem-30, a student wants to take all the courses in A, in the minimal number of semesters. That means the student is ready to take any number of courses in a semester. Design a schedule for this scenario. *One possible schedule is*:
Semester 1: C-Lang, OS, Design Patterns
Semester 2: Data Structures, CO, Programming
Semester 3: Algorithms

Solution: A variation of the above topological sort algorithm with a slight change: In each semester, instead of taking one subject, take all the subjects with zero indegree. That means, execute the algorithm on all the nodes with degree 0 (instead of dealing with one source in each stage, all the sources will be dealt and printed).

Time Complexity: $O(|V| + |E|)$.

Problem-32 **LCA of a DAG:** Given a DAG and two vertices v and w, find the *lowest common ancestor* (LCA) of v and w. The LCA of v and w is an ancestor of v and w that has no descendants that are also ancestors of v and w.

Hint: Define the height of a vertex v in a DAG to be the length of the longest path from *root* to v. Among the vertices that are ancestors of both v and w, the one with the greatest height is an LCA of v and w.

Problem-33 **Shortest ancestral path:** Given a DAG and two vertices v and w, find the *shortest ancestral path* between v and w. An ancestral path between v and w is a common ancestor x along with a shortest path from v to x and a shortest path from w to x. The shortest ancestral path is the ancestral path whose total length is minimized.

Hint: Run BFS two times. First run from v and second time from w. Find a DAG where the shortest ancestral path goes to a common ancestor x that is not an LCA.

Problem-34 Let us assume that we have two graphs G_1 and G_2. How do we check whether they are isomorphic or not?

Solution: There are many ways of representing the same graph. As an example, consider the following simple graph. It can be seen that all the representations below have the same number of vertices and the same number of edges.

Definition: Graphs $G_1 = \{V_1, E_1\}$ and $G_2 = \{V_2, E_2\}$ are isomorphic if

1) There is a one-to-one correspondence from V_1 to V_2 and
2) There is a one-to-one correspondence from E_1 to E_2 that map each edge of G_1 to G_2.

Now, for the given graphs how do we check whether they are isomorphic or not?

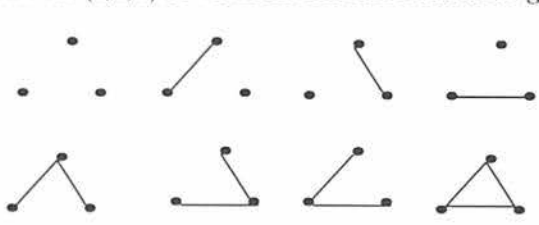
In general, it is not a simple task to prove that two graphs are isomorphic. For that reason we must consider some properties of isomorphic graphs. That means those properties must be satisfied if the graphs are isomorphic. If the given graph does not satisfy these properties then we say they are not isomorphic graphs.

Property: Two graphs are isomorphic if and only if for some ordering of their vertices their adjacency matrices are equal.

Based on the above property we decide whether the given graphs are isomorphic or not. I order to check the property, we need to do some matrix transformation operations.

Problem-35 How many simple undirected non-isomorphic graphs are there with n vertices?

Solution: We will try to answer this question in two steps. First, we count all labeled graphs. Assume all the representations below are labeled with $\{1, 2, 3\}$ as vertices. The set of all such graphs for $n = 3$ are:



There are only two choices for each edge: it either exists or it does not. Therefore, since the maximum number of edges is $\binom{n}{2}$ (and since the maximum number of edges in an undirected graph with n vertices is $\frac{n(n-1)}{2} = n_{C_2} = \binom{n}{2}$), the total number of undirected labeled graphs is $2^{\binom{n}{2}}$.

Problem-36 **Hamiltonian path in DAGs:** Given a DAG, design a linear time algorithm to determine whether there is a path that visits each vertex exactly once.

Solution: The *Hamiltonian* path problem is an NP-Complete problem (for more details ref *Complexity Classes* chapter). To solve this problem, we will try to give the approximation algorithm (which solves the problem, but it may not always produce the optimal solution).

Let us consider the topological sort algorithm for solving this problem. Topological sort has an interesting property: that if all pairs of consecutive vertices in the sorted order are connected by edges, then these edges form a directed *Hamiltonian* path in the DAG. If a *Hamiltonian* path exists, the topological sort order is unique. Also, if a topological sort does not form a *Hamiltonian* path, the DAG will have two or more topological orderings.

Approximation Algorithm: Compute a topological sort and check if there is an edge between each consecutive pair of vertices in the topological order.

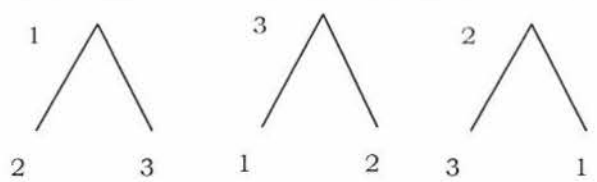
In an unweighted graph, find a path from s to t that visits each vertex exactly once. The basic solution based on backtracking is, we start at *s* and try all of its neighbors recursively, making sure we never visit the same vertex twice. The algorithm based on this implementation can be given as:

```
def HamiltonianPath( G, u ):
    if( u == t )
            # Check that we have seen all vertices.
    else:
            for v in range(0,G.numVertices)
                    if( !seenTable[v] and G.adjMatrix[u][v]):
                            seenTable[v] = True
                            HamiltonianPath( v )
                            seenTable[v] = False
```

Note that if we have a partial path from s to u using vertices $s = v_1, v_2, \ldots, v_k = u$, then we don't care about the order in which we visited these vertices so as to figure out which vertex to visit next. All that we need to know is the set of vertices we have seen (the seenTable[] array) and which vertex we are at right now (u). There are 2^n possible sets of vertices and n choices for u. In other words, there are 2^n possible *seenTable*[ ] arrays and n different parameters to HamiltonianPath(). What HamiltonianPath() does during any particular recursive call is completely determined by the *seenTable*[ ] array and the parameter u.

Problem-37 For a given graph *G* with *n* vertices how many trees we can construct?

Solution: There is a simple formula for this problem and it is named after Arthur Cayley. For a given graph with *n* labeled vertices the formula for finding number of trees on is n^{n-2}. Below, the number of trees with different *n* values is shown.

n value	Formula value: n^{n-2}	Number of Trees
2	1	1 ——— 2
3	3	

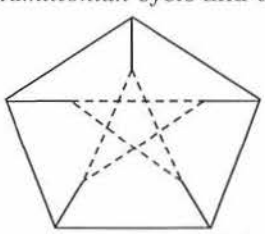
Problem-38 For a given graph G with *n* vertices how many spanning trees can we construct?

Solution: The solution to this problem is the same as that of Problem-37. It is just another way of asking the same question. Because the number of edges in both regular tree and spanning tree are the same.

Problem-39 The *Hamiltonian cycle* problem: Is it possible to traverse each of the vertices of a graph exactly once, starting and ending at the same vertex?

Solution: Since the *Hamiltonian* path problem is an NP-Complete problem, the *Hamiltonian* cycle problem is an NP-Complete problem. A *Hamiltonian* cycle is a cycle that traverses every vertex of a graph exactly once. There are no known conditions in which are both necessary and sufficient, but there are a few sufficient conditions.

- For a graph to have a *Hamiltonian* cycle the degree of each vertex must be two or more.
- The Petersen graph does not have a *Hamiltonian* cycle and the graph is given below.



- In general, the more edges a graph has, the more likely it is to have a *Hamiltonian* cycle.
- Let G be a simple graph with $n \geq 3$ vertices. If every vertex has a degree of at least $\frac{n}{2}$, then G has a *Hamiltonian* cycle.
- The best known algorithm for finding a *Hamiltonian* cycle has an exponential worst-case complexity.

Note: For the approximation algorithm of *Hamiltonian* path, refer to the *Dynamic Programming* chapter.

Problem-40 What is the difference between *Dijkstra's* and *Prim's* algorithm?

Solution: *Dijkstra's* algorithm is almost identical to that of *Prim's*. The algorithm begins at a specific vertex and extends outward within the graph until all vertices have been reached. The only distinction is that *Prim's* algorithm stores a minimum cost edge whereas *Dijkstra's* algorithm stores the total cost from a source vertex to the current vertex. More simply, *Dijkstra's* algorithm stores a summation of minimum cost edges whereas *Prim's* algorithm stores at most one minimum cost edge.

Problem-41 **Reversing Graph: :** Give an algorithm that returns the reverse of the directed graph (each edge from v to w is replaced by an edge from w to v).

Solution: In graph theory, the reverse (also called *transpose*) of a directed graph G is another directed graph on the same set of vertices with all the edges reversed. That means, if G contains an edge (u, v) then the reverse of G contains an edge (v, u) and vice versa.

Algorithm:

```
def ReverseTheDirectedGraph(G):
    Create new graph with name ReversedGraph and
            let us assume that this will contain the reversed graph.
    #The reversed graph also will contain same number of vertices and edges.
    for each vertex of given graph G:
            for each vertex w adjacent to v:
                    Add the w to v edge in ReversedGraph;
                    # That means we just need to reverse the bits in adjacency matrix.
    return ReversedGraph
```

Problem-42 **Travelling Sales Person Problem:** Find the shortest path in a graph that visits each vertex at least once, starting and ending at the same vertex?

Solution: The Traveling Salesman Problem (*TSP*) is related to finding a Hamiltonian cycle. Given a weighted graph G, we want to find the shortest cycle (may be non-simple) that visits all the vertices.

Approximation algorithm: This algorithm does not solve the problem but gives a solution which is within a factor of 2 of optimal (in the worst-case).

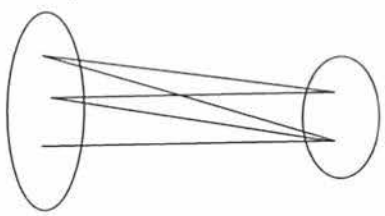
1) Find a Minimal Spanning Tree (MST).
2) Do a DFS of the MST.

For details, refer to the chapter on *Complexity Classes*.

Problem-43 Discuss Bipartite matchings?

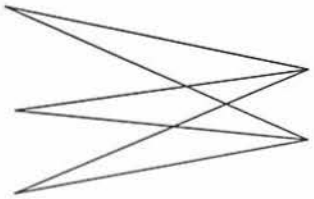
Solution: In Bipartite graphs, we divide the graphs in to two disjoint sets, and each edge connects a vertex from one set to a vertex in another subset (as shown in figure).

Definition: A simple graph $G = (V, E)$ is called a *bipartite graph* if its vertices can be divided into two disjoint sets $V = V_1 \cup V_2$, such that every edge has the form $e = (a, b)$ where $a \in V_1$ and $b \in V_2$. One important condition is that no vertices both in V_1 or both in V_2 are connected.

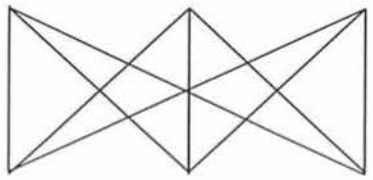


Properties of Bipartite Graphs

- A graph is called bipartite if and only if the given graph does not have an odd length cycle.
- A *complete bipartite graph* $K_{m,n}$ is a bipartite graph that has each vertex from one set adjacent to each vertex from another set.

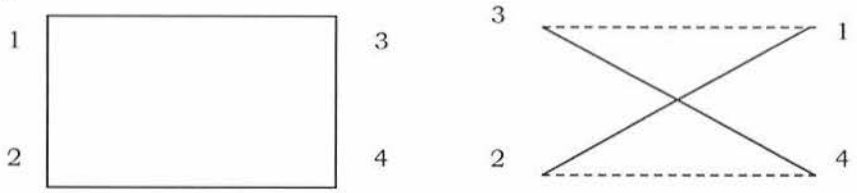


$K_{2,3}$



$K_{3,3}$

- A subset of edges $M \subset E$ is a *matching* if no two edges have a common vertex. As an example, matching sets of edges are represented with dotted lines. A matching M is called *maximum* if it has the largest number of possible edges. In the graphs, the dotted edges represent the alternative matching for the given graph.



- A matching M is *perfect* if it matches all vertices. We must have $V_1 = V_2$ in order to have perfect matching.
- An *alternating path* is a path whose edges alternate between matched and unmatched edges. If we find an alternating path, then we can improve the matching. This is because an alternating path consists of matched and unmatched edges. The number of unmatched edges exceeds the number of matched edges by one. Therefore, an alternating path always increases the matching by one.

The next question is, how do we find a perfect matching? Based on the above theory and definition, we can find the perfect matching with the following approximation algorithm.

Matching Algorithm (Hungarian algorithm)

1) Start at unmatched vertex.
2) Find an alternating path.
3) If it exists, change matching edges to no matching edges and conversely. If it does not exist, choose another unmatched vertex.
4) If the number of edges equals $V/2$, stop. Otherwise proceed to step 1 and repeat, as long as all vertices have been examined without finding any alternating paths.

Time Complexity of the Matching Algorithm: The number of iterations is in $O(V)$. The complexity of finding an alternating path using BFS is $O(E)$. Therefore, the total time complexity is $O(V \times E)$.

Problem-44 Marriage and Personnel Problem?

Marriage Problem: There are X men and Y women who desire to get married. Participants indicate who among the opposite sex could be a potential spouse for them. Every woman can be married to at most one man, and every man to at most one woman. How can we marry everybody to someone they like?

Personnel Problem: You are the boss of a company. The company has M workers and N jobs. Each worker is qualified to do some jobs, but not others. How will you assign jobs to each worker?

Solution: These two cases are just another way of asking about bipartite graphs, and the solution is the same as that of Problem-43.

Problem-45 How many edges will be there in complete bipartite graph $K_{m,n}$?

Solution: $m \times n$. This is because each vertex in the first set can connect all vertices in the second set.

Problem-46 A graph is called a regular graph if it has no loops and multiple edges where each vertex has the same number of neighbors; i.e., every vertex has the same degree. Now, if $K_{m,n}$ is a regular graph, what is the relation between m and n?

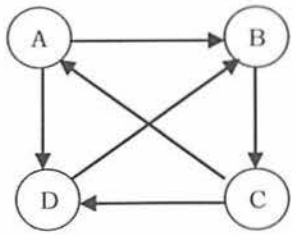
Solution: Since each vertex should have the same degree, the relation should be $m = n$.

Problem-47 What is the maximum number of edges in the maximum matching of a bipartite graph with n vertices?

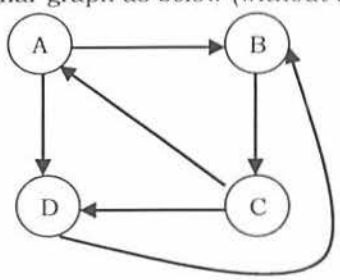
Solution: From the definition of *matching*, we should not have edges with common vertices. So in a bipartite graph, each vertex can connect to only one vertex. Since we divide the total vertices into two sets, we can get the maximum number of edges if we divide them in half. Finally the answer is $\frac{n}{2}$.

Problem-48 Discuss Planar Graphs. *Planar graph*: Is it possible to draw the edges of a graph in such a way that the edges do not cross?

Solution: A graph G is said to be planar if it can be drawn in the plane in such a way that no two edges meet each other except at a vertex to which they are incident. Any such drawing is called a plane drawing of G. As an example consider the below graph:



This graph we can easily convert to a planar graph as below (without any crossed edges).



How do we decide whether a given graph is planar or not?

The solution to this problem is not simple, but researchers have found some interesting properties that we can use to decide whether the given graph is a planar graph or not.

Properties of Planar Graphs

- If a graph G is a connected planar simple graph with V vertices, where $V = 3$ and E edges, then $E = 3V - 6$.
- K_5 is non-planar. [K_5 stands for complete graph with 5 vertices].
- If a graph G is a connected planar simple graph with V vertices and E edges, and no triangles, then $E = 2V - 4$.
- $K_{3,3}$ is non-planar. [$K_{3,3}$ stands for bipartite graph with 3 vertices on one side and the other 3 vertices on the other side. $K_{3,3}$ contains 6 vertices].
- If a graph G is a connected planar simple graph, then G contains at least one vertex of 5 degrees or less.
- A graph is planar if and only if it does not contain a subgraph that has K_5 and $K_{3,3}$ as a contraction.
- If a graph G contains a nonplanar graph as a subgraph, then G is non-planar.
- If a graph G is a planar graph, then every subgraph of G is planar.
- For any connected planar graph $G = (V, E)$, the following formula should hold: $V + F - E = 2$, where F stands for the number of faces.
- For any planar graph $G = (V, E)$ with K components, the following formula holds: $V + F - E = 1 + K$.

In order to test the planarity of a given graph, we use these properties and decide whether it is a planar graph or not. Note that all the above properties are only the necessary conditions but not sufficient.

Problem-49 How many faces does $K_{2,3}$ have?

Solution: From the above discussion, we know that $V + F - E = 2$, and from an earlier problem we know that $E = m \times n = 2 \times 3 = 6$ and $V = m + n = 5$. $\therefore 5 + F - 6 = 2 \Rightarrow F = 3$.

Problem-50 Discuss Graph Coloring

Solution: A k –coloring of a graph G is an assignment of one color to each vertex of G such that no more than k colors are used and no two adjacent vertices receive the same color. A graph is called k –colorable if and only if it has a k –coloring.

Applications of Graph Coloring: The graph coloring problem has many applications such as scheduling, register allocation in compilers, frequency assignment in mobile radios, etc.

Clique: A *clique* in a graph G is the maximum complete subgraph and is denoted by $\omega(G)$.

Chromatic number: The chromatic number of a graph G is the smallest number k such that G is k –colorable, and it is denoted by $X(G)$.

The lower bound for $X(G)$ is $\omega(G)$, and that means $\omega(G) \leq X(G)$.

Properties of Chromatic number: Let G be a graph with n vertices and G′ is its complement. Then,

- $X(G) \leq \Delta(G) + 1$, where Δ (G) is the maximum degree of G.
- $X(G)\,\omega(G') \geq n$
- $X(G) + \omega(G') \leq n + 1$
- $X(G) + (G') \leq n + 1$

K-colorability problem: Given a graph $G = (V, E)$ and a positive integer $k \leq V$. Check whether G is k –colorable?

This problem is NP-complete and will be discussed in detail in the chapter on *Complexity Classes*.

Graph coloring algorithm: As discussed earlier, this problem is *NP*-Complete. So we do not have a polynomial time algorithm to determine $X(G)$. Let us consider the following approximation (no efficient) algorithm.

- Consider a graph G with two non-adjacent vertices a and b. The connection G_1 is obtained by joining the two non-adjacent vertices a and b with an edge. The contraction G_2 is obtained by shrinking $\{a, b\}$ into a single vertex $c(a, b)$ and by joining it to each neighbor in G of vertex a and of vertex b (and eliminating multiple edges).
- A coloring of G in which a and b have the same color yields a coloring of G_1. A coloring of G in which a and b have different colors yields a coloring of G_2.
- Repeat the operations of connection and contraction in each graph generated, until the resulting graphs are all cliques. If the smallest resulting clique is a K –clique, then $(G) = K$.

Important notes on Graph Coloring

- Any simple planar graph G can be colored with 6 colors.
- Every simple planar graph can be colored with less than or equal to 5 colors.

Problem-51 What is the four coloring problem?

Solution: A graph can be constructed from any map. The regions of the map are represented by the vertices of the graph, and two vertices are joined by an edge if the regions corresponding to the vertices are adjacent. The resulting graph is planar. That means it can be drawn in the plane without any edges crossing.

The *Four Color Problem* is whether the vertices of a planar graph can be colored with at most four colors so that no two adjacent vertices use the same color.

History: The *Four Color Problem* was first given by *Francis Guthrie*. He was a student at *University College London* where he studied under *Augusts De Morgan*. After graduating from London he studied law, but some years later his brother Frederick Guthrie had become a student of *De Morgan*. One day Francis asked his brother to discuss this problem with *De Morgan*.

Problem-52 When an adjacency-matrix representation is used, most graph algorithms require time $O(V^2)$. Show that determining whether a directed graph, represented in an adjacency-matrix that contains a sink can be done in time $O(V)$. A sink is a vertex with in-degree $|V| - 1$ and out-degree 0 (Only one can exist in a graph).

Solution: A vertex i is a sink if and only if $M[i, j] = 0$ for all j and $M[j, i] = 1$ for all $j \neq i$. For any pair of vertices i and j:

$M[i, j] = 1 \rightarrow$ vertex i can't be a sink
$M[i, j] = 0 \rightarrow$ vertex j can't be a sink

Algorithm:

- Start at $i = 1, j = 1$
- If $M[i, j] = 0 \rightarrow i$ wins, $j++$
- If $M[i, j] = 1 \rightarrow j$ wins, $i++$
- Proceed with this process until $j = n$ or $i = n + 1$
- If $i == n + 1$, the graph does not contain a sink
- Otherwise, check row i – it should be all zeros; and check column i – it should be all but $M[i, i]$ ones; – if so, i is a sink.

Time Complexity: $O(V)$, because at most $2|V|$ cells in the matrix are examined.

Problem-53 What is the worst – case memory usage of DFS?

Solution: It occurs when the $O(|V|)$, which happens if the graph is actually a list. So the algorithm is memory efficient on graphs with small diameter.

Problem-53 Does DFS find the shortest path from start node to some node w ?

Solution: No. In DFS it is not compulsory to select the smallest weight edge.

Problem-54 Give an algorithm that takes as input a directed graph G. The algorithm should check if there is a vertex v so that there is a path from v to at most 10 vertices in the graph. Assume that the graph is represented via an array of adjacency lists (an array of linked lists).

Solution: For every v, the algorithm starts to do a BFS search. We initiate a counter to 0. Each time a new vertex is encountered (a new vertex is labeled, so it has finite distance from v, namely, is reachable from v), we augment this counter by 1. If the counter gets to 11 then there are more than 10 vertices reachable from v and we go to the next vertex. Otherwise, the BFS checks a constant number of vertices for every v. The total running time is O(n).

CHAPTER

10

SORTING

10.1 What is Sorting?

Sorting is an algorithm that arranges the elements of a list in a certain order [either *ascending* or *descending*]. The output is a permutation or reordering of the input.

10.2 Why is Sorting Necessary?

Sorting is one of the important categories of algorithms in computer science and a lot of research has gone into this category. Sorting can significantly reduce the complexity of a problem, and is often used for database algorithms and searches.

10.3 Classification of Sorting Algorithms

Sorting algorithms are generally categorized based on the following parameters.

By Number of Comparisons

In this method, sorting algorithms are classified based on the number of comparisons. For comparison based sorting algorithms, best case behavior is O($nlogn$) and worst case behavior is O(n^2). Comparison-based sorting algorithms evaluate the elements of the list by key comparison operation and need at least O($nlogn$) comparisons for most inputs.

Later in this chapter we will discuss a few *non – comparison* (*linear*) sorting algorithms like Counting sort, Bucket sort, Radix sort, etc. Linear Sorting algorithms impose few restrictions on the inputs to improve the complexity.

By Number of Swaps

In this method, sorting algorithms are categorized by the number of *swaps* (also called *inversions*).

By Memory Usage

Some sorting algorithms are "*in place*" and they need O(1) or O($logn$) memory to create auxiliary locations for sorting the data temporarily.

By Recursion

Sorting algorithms are either recursive [quick sort] or non-recursive [selection sort, and insertion sort], and there are some algorithms which use both (merge sort).

By Stability

Sorting algorithm is *stable* if for all indices i and j such that the key $A[i]$ equals key $A[j]$, if record $R[i]$ precedes record $R[j]$ in the original file, record $R[i]$ precedes record $R[j]$ in the sorted list. Few sorting algorithms maintain the relative order of elements with equal keys (equivalent elements retain their relative positions even after sorting).

By Adaptability

With a few sorting algorithms, the complexity changes based on pre-sortedness [quick sort]: pre-sortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

10.4 Other Classifications

Another method of classifying sorting algorithms is:

- Internal Sort
- External Sort

Internal Sort

Sort algorithms that use main memory exclusively during the sort are called *internal* sorting algorithms. This kind of algorithm assumes high-speed random access to all memory.

External Sort

Sorting algorithms that use external memory, such as tape or disk, during the sort come under this category.

10.5 Bubble Sort

Bubble sort is the simplest sorting algorithm. It works by iterating the input array from the first element to the last, comparing each pair of elements and swapping them if needed. Bubble sort continues its iterations until no more swaps are needed. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Generally, insertion sort has better performance than bubble sort. Some researchers suggest that we should not teach bubble sort because of its simplicity and high time complexity.

The only significant advantage that bubble sort has over other implementations is that it can detect whether the input list is already sorted or not.

Implementation

```
def BubbleSort( A ):
 for i in range( len( A ) ):
        for k in range( len( A ) - 1, i, -1 ):
            if ( A[k] < A[k - 1] ):
                swap( A, k, k - 1 )

def swap( A, x, y ):
 temp = A[x]
 A[x] = A[y]
 A[y] = temp

A = [534,246,933,127,277,321,454,565,220]
BubbleSort(A)
print(A)
```

Algorithm takes $O(n^2)$ (even in best case). We can improve it by using one extra flag. No more swaps indicate the completion of sorting. If the list is already sorted, we can use this flag to skip the remaining passes.

```
def BubbleSort( A ):
 swapped = 1
 for i in range( len( A ) ):
        if ( swapped == 0 ):
            return
        for k in range( len( A ) - 1, i, -1 ):
```