1 Program to demonstrate Breadth First Search.

```
# to print BFS traversal from a given source vertex.

from collections import defaultdict

class Graph:
    def __int__(self):
        self.graph = defaultdict(list)
    def addEdge(self, u, v):
        self.graph[u].append(v)
    def BFS(self, s):
        visited = [False] * (max(self.graph) + 1)
        queue = []
        queue.append(s)
        visited[s] = True
        while queue:
            s = queue.pop(0)
            print(s, end = " ")
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True


g = Graph()
g.addEdge(0,1)
g.addEdge(0,2)
g.addEdge(1,2)
g.addEdge(2,0)
```

```
g.addEdge (2,3)
g.addEdge (3,3)

print ("Following is Breadth First Search Traversal "
                                "(starting from vertex 2)")
g.BFS(2)
```

OUTPUT :

Following is Breadth First Search Traversal (starting from vertex 2)
2 0 3 1

Date

Expt. No.

2. Program to demonstrate Depth First Search.

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self, u, v):
        self.graph[u].append(v)
    def DFSUtil(self, v, visited):
        visited.add(v)
        print(v, end='')
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)
    def DFS(self, v):
        visited = set()
        self.DFSUtil(v, visited)

g = Graph()
g.addEdge(0,1)
g.addEdge(0,2)
g.addEdge(1,2)
g.addEdge(2,0)
g.addEdge(2,3)
g.addEdge(3,3)

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

Teacher's Signature

OUTPUT:-

Following is DFS from (starting from vertex 2)
2 0 1 3

3 Program to demonstrate A* Algorithm.

```
class Node ():
    def __init__(self, parent = None, position = None):
        self. parent = parent
        self. position = position
        self. g = 0
        self. h = 0
        self. f = 0
    def __eq__ (self, other):
        return self. position == other. position
def astar (maze, start, end):
    start_node = Node (None, start)
    start_node. g = start_node. h = start_node. f = 0
    end_node = Node (None, end)
    end_node. g = end_node. h = end_node. f = 0
    open_list = []
    closed_list = []
    open_list. append (start_node)
    while len(open_list) >0:
        current_node = open_list [0]
        current_index = 0
        for index, item in enumerate (open_list):
            if item. f < current_node. f:
                current_node = item
                current_idex = index
        open_list. pop (current_index)
        closed_list. append (current_node)
        if current_node == end_node:
```

```
path = []
current = current_node
while current is not None:
    path.append (current.position)
    current = current.parent
return path [::-1]
children = []
for new_position in [(0,-1), (0,1), (-1,0), (1,0), (-1,-1),
                     (-1,1), (1,-1), (1,1)]:
    node_position = (current_node.position [0] + new_position[0],
                     current_node.position [1] + new_position[1])
    if node_position [0] > (len (maze) -1) or node_position [0]
            < 0 or node_position [1] > ( len (maze [
            len(maze-1) - 1) or node_position [1] < 0 :
        continue
    if maze [node_position [0]] [node_position [1]] != 0 :
        continue
    new_node = Node (current_node, node_position)
    children.append (new_node)
for child in children :
    for closed_child in closed_list:
        if child == closed_child :
            continue
    child.g = current_node.g + 1
    child.h = ((child.position [0] - end_node.position [0]) ** 2)
              + ((child.position [1] - end_node.
              position [1]) ** 2)
    child.f = child.g + child.h
    for open_node in open_list:
```

```
            if child == open_node and child.g > open_node.g:
                continue
        open_list.append(child)
def main():
    maze = [[0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 0]]
    graph = [[0, 1, 0, 0, 0, 0],
             [1, 0, 1, 0, 1, 0],
             [0, 1, 0, 0, 0, 1],
             [0, 0, 0, 0, 1, 0],
             [0, 1, 0, 1, 0, 0],
             [0, 0, 1, 0, 0, 0]]
    start = (0,0)
    end = (5,5)
    end1 = (5,5)
    path = astar(maze, start, end)
    print(path)
    path1 = astar(graph, start, end1)
    print(path1)
if __name__ == '__main__':
    main()
```

OUTPUT:

[(0,0), (1,1), (2,2), (3,3), (4,3), (5,4), (5,5)]
[(0,0), (1,1), (2,2), (3,3), (4,4), (5,5)]

4. Program to demonstrate Hill Climbing.

```
import random
def randomSolution(tsp):
    cities = list(range(len(tsp)))
    solution = []
    for i in range(len(tsp)):
        randomCity = cities[random.randint(0, len(cities)-1)]
        solution.append(randomCity)
        cities.remove(randomCity)
    return solution

def routeLength(tsp, solution):
    routeLength = 0
    for i in range(len(solution)):
        routeLength += tsp[solution[i-1]][solution[i]]
    return routeLength

def getNeighbours(solution):
    neighbours = []
    for i in range(len(solution)):
        for j in range(i+1, len(solution)):
            neighbour = solution.copy()
            neighbour[i] = solution[j]
            neighbour[j] = solution[i]
            neighbours.append(neighbour)
    return neighbours

def getBestNeighbour(tsp, neighbours):
```

```
    bestRouteLength = routeLength (tsp, neighbours [0])
    bestNeighbour = neighbours [0]
    for neighbour in neighbours:
        currentRouteLength = routeLength (tsp, neighbour)
        if currentRouteLength < bestRoute Length:
            bestRouteLength = currentRouteLength
            bestNeighbour = neighbour
    return bestNeighbour, bestRouteLength

def hillClimbing (tsp):
    currentSolution = randomSolution (tsp)
    currentRouteLength = routeLength (tsp, currentSolution)
    neighbours = getNeighbours (currentSolution)
    bestNeighbour, bestNeighbourRouteLength = getBestNeighbour
                                              (tsp, neighbours)
    while bestNeighbourRouteLength < currentRouteLength:
        currentSolution = bestNeighbour
        currentRouteLength = bestNeighbourRouteLength =
                                 getBestNeighbour (tsp, neighbours)
    return currentSolution, currentRouteLength

def main ():
    tsp = [[0, 400, 500, 300], [400, 0, 300, 500], [500, 300, 0, 400],
           [300, 500, 400, 0]]

    print (hillClimbing (tsp))

if __name__ == "__main__":
    main()
```

OUTPUT

([0,1, 2,3], 1400)