PRACTICAL 01

BFS algorithm

SOURCE CODE :

```
import collections
def bfs(graph, root):
    visited, queue = set(), collections.deque([root])
    visited.add(root)
    while queue:
        # Dequeue a vertex from queue
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")
        # If not visited, mark it as visited, and
        # enqueue it
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)
if __name__ == '__main__':
    graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}
    print("Following is Breadth First Traversal: ")
    bfs(graph, 0)
```

OUTPUT

```
Following is Breadth First Traversal:
> 3
0 1 2 3 3
```

DFS ALGORITHM

SOURCE CODE

```
def dfs_recursive(graph, source,path = []):
    if source not in path:
        path.append(source)
        if source not in graph:
            # leaf node, backtrack
            return path

        for neighbour in graph[source]:

            path = dfs_recursive(graph, neighbour, path)
    return path
graph = {"A":["B","C","D"],
  "B":["E"],
  "C":["G","F"],
  "D":["H"],
  "E":["I"],
  "F":["J"],
  "G":["K"]}
dfs_element = dfs_recursive(graph, "A")
print(dfs_element)
```

OUTPUT

```
['A', 'B', 'E', 'I', 'C', 'G', 'K', 'F', 'J', 'D', 'H']
```

PRACTICAL 02

A* SEARCH ALGORITHM

SOURCE CODE

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}              #store distance from starting node
    parents = {}        # parents contains an adjacency map of all nodes
    #distance of starting node from itself is zero
    g[start_node] = 0
    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
        #node with lowest f() is found
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                #nodes 'm' not in first and last set are added to first
                #n is set its parent
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
```

```
                g[m] = g[n] + weight
            #for each node m,compare its distance from start i.e g(m) to the
            #from start through n node
            else:
                if g[m] > g[n] + weight:
                    #update g(m)
                    g[m] = g[n] + weight
                    #change parent of m to n
                    parents[m] = n
                    #if m in closed set,remove and add to open
                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)
    if n == None:
        print('Path does not exist!')
        return None

    # if the current node is the stop_node
    # then we begin reconstructin the path from it to the start_node
    if n == stop_node:
        path = []
        while parents[n] != n:
            path.append(n)
            n = parents[n]
        path.append(start_node)
        path.reverse()
        print('Path found: {}'.format(path))
        return path
```

```
        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
```

Graph and Heuristic Values

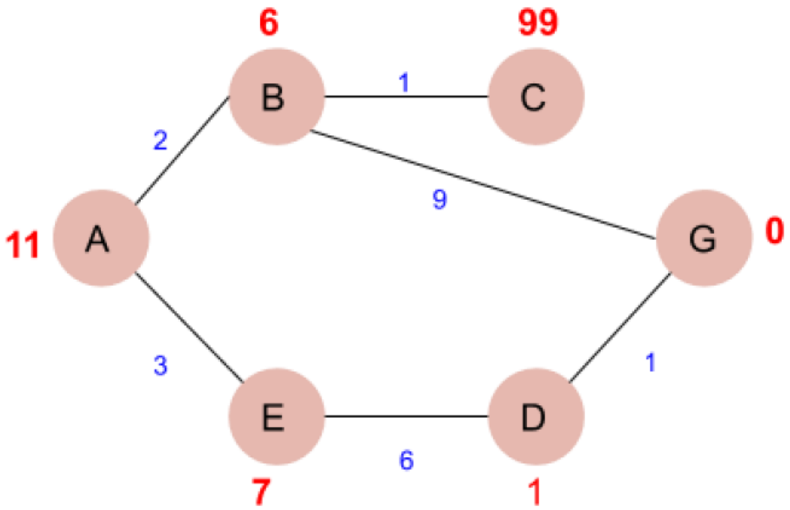
```
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0
```

```
    }
    return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}
aStarAlgo('A', 'J')
```

OUTPUT

```
Path found: ['A', 'F', 'G', 'I', 'J']
```



PRACTICAL 03

PRIMS ALGORITHM

SOURCE CODE

```
from typing import List, Dict # For annotations

class Node :

    def __init__(self, arg_id) :
        self._id = arg_id

class Graph :

    def __init__(self, source : int, adj_list : Dict[int, List[int]]) :
        self.source = source
        self.adjlist = adj_list

    def PrimsMST (self) -> int :

first node
        # in the priority queue
        priority_queue = { Node(self.source) : 0 }
        added = [False] * len(self.adjlist)
        min_span_tree_cost = 0

        while priority_queue :
            # Choose the adjacent node with the least edge cost
            node = min (priority_queue, key=priority_queue.get)
            cost = priority_queue[node]

            # Remove the node from the priority queue
            del priority_queue[node]

            if added[node._id] == False :
                min_span_tree_cost += cost
                added[node._id] = True
                print("Added Node : " + str(node._id) + ", cost now :
"+str(min_span_tree_cost))

                for item in self.adjlist[node._id] :
                    adjnode = item[0]
                    adjcost = item[1]
                    if added[adjnode] == False :
```

```
                    priority_queue[Node(adjnode)] = adjcost

        return min_span_tree_cost

def main() :

    g1_edges_from_node = {}

    # Outgoing edges from the node: (adjacent_node, cost) in graph 1.
    g1_edges_from_node[0] = [ (1,1), (2,2), (3,1), (4,1), (5,2), (6,1) ]
    g1_edges_from_node[1] = [ (0,1), (2,2), (6,2) ]
    g1_edges_from_node[2] = [ (0,2), (1,2), (3,1) ]
    g1_edges_from_node[3] = [ (0,1), (2,1), (4,2) ]
    g1_edges_from_node[4] = [ (0,1), (3,2), (5,2) ]
    g1_edges_from_node[5] = [ (0,2), (4,2), (6,1) ]
    g1_edges_from_node[6] = [ (0,1), (2,2), (5,1) ]

    g1 = Graph(0, g1_edges_from_node)
    cost = g1.PrimsMST()
    print("Cost of the minimum spanning tree in graph 1 : " + str(cost) +"\n")

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

OUTPUT

```
Added Node : 0, cost now : 0

Added Node : 1, cost now : 1

Added Node : 3, cost now : 2

Added Node : 4, cost now : 3

Added Node : 6, cost now : 4

Added Node : 2, cost now : 5

Added Node : 5, cost now : 6

Cost of the minimum spanning tree in graph 1 : 6
```

PRACTICAL 04

N QUEENS PROBLEM

SOURCE CODE

```
global N
N = 4
def printSolution(board):
    for i in range(N):
        for j in range(N):
            if board[i][j] == 1:
                print("Q",end=" ")
            else:
                print(".",end=" ")
        print()

def isSafe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, N, 1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False
```

```
        return True
def solveNQUtil(board, col):
        if col >= N:
                return True
        for i in range(N):
                if isSafe(board, i, col):
                        board[i][col] = 1
                        if solveNQUtil(board, col + 1) == True:
                                return True
                        board[i][col] = 0
        return False
def solveNQ():
        board = [[0, 0, 0, 0],
                        [0, 0, 0, 0],
                        [0, 0, 0, 0],
                        [0, 0, 0, 0]]

        if solveNQUtil(board, 0) == False:
                print("Solution does not exist")
                return False

        printSolution(board)
        return True



# Driver Code
if __name__ == '__main__':
        solveNQ()
```

OUTPUT

```
.   .   Q   .
Q   .   .   .
.   .   .   Q
.   Q   .   .
```

PRACTICAL 05

CHAT BOT USING PYTHON

SOURCE CODE

```
# importing the required modules
from chatterbot import ChatBot
from chatterbot.trainers import ListTrainer

# creating a chatbot
myBot = ChatBot(
    name = 'GS',
    read_only = True,
    logic_adapters = [
        'chatterbot.logic.MathematicalEvaluation',
        'chatterbot.logic.BestMatch'
        ]
        )

# training the chatbot
small_convo = [
    'Hi there!',
    'Hi',
    'How do you do?',
    'How are you?',
    'I\'m fine',
    'I feel awesome',
    'Excellent, glad to hear that.',
    'Not so good',
```

```
    'Sorry to hear that.',
    'What\'s your name?',
    ' I\'m GS. Ask me a math question, please.'
    ]
math_convo_1 = [
    'Pythagorean theorem',
    'a squared plus b squared equals c squared.'
    ]
math_convo_2 = [
    'Law of Cosines',
    'c**2 = a**2 + b**2 - 2*a*b*cos(gamma)'
    ]
# using the ListTrainer class
list_trainee = ListTrainer(myBot)
for i in (small_convo, math_convo_1, math_convo_2):
    list_trainee.train(i)
```

OUTPUT

```
# starting a conversation
>>> print(myBot.get_response("Hi, there!"))
Hi
>>> print(myBot.get_response("What's your name?"))
I'm GS. Ask me a math question, please.
>>> print(myBot.get_response("Do you know Pythagorean theorem"))
a squared plus b squared equals c squared.
>>> print(myBot.get_response("Tell me the formula of law of cosines"))
c**2 = a**2 + b**2 - 2*a*b*cos(gamma)
```