BFS: BREADTH FIRST SEARCH ALGORITHM

Source Code: -

```
#include<iostream>
#include <list>
using namespace std;
// This class represents a directed graph using
// adjacency list representation
class Graph
{
 int V; // No. of vertices
 // Pointer to an array containing adjacency
 // lists
 list<int> *adj;
public:
 Graph(int V); // Constructor
 // function to add an edge to graph
 void addEdge(int v, int w);
 // prints BFS traversal from a given source s
 void BFS(int s);
};
Graph::Graph(int V)
{
this->V = V;
 adj = new list<int>[V];
}
void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w); // Add w to v's list.
}
void Graph::BFS(int s)
{
 // Mark all the vertices as not visited
 bool *visited = new bool[V];
```

```
for(int i = 0; i < V; i++)
visited[i] = false;
// Create a queue for BFS
list<int> queue;
// Mark the current node as visited and enqueue it
visited[s] = true;
queue.push_back(s);
// 'i' will be used to get all adjacent
// vertices of a vertex
list<int>::iterator i;
while(!queue.empty())
{
// Dequeue a vertex from queue and print it
s = queue.front();
cout << s << " ";
queue.pop_front();
// Get all adjacent vertices of the dequeued
// vertex s. If a adjacent has not been visited,
// then mark it visited and enqueue it
for (i = adj[s].begin(); i != adj[s].end(); ++i)
{
if (!visited[*i])
{
visited[*i] = true;
queue.push_back(*i);
}
}
}
}
// Driver program to test methods of graph class
int main()
{
// Create a graph given in the above diagram
```

```
 Graph g(4);
 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 0);
 g.addEdge(2, 3);
 g.addEdge(3, 3);
 cout << "Following is Breadth First Traversal "
 << "(starting from vertex 2) \n";
 g.BFS(2);
 return 0;
}
```

DFS: DEPTH FIRST SEARCH ALGORITM

```
#include <bits/stdc++.h>
using namespace std;
// Graph class represents a directed graph
// using adjacency list representation
class Graph {
public:
 map<int, bool> visited;
 map<int, list<int> > adj;
 // function to add an edge to graph
 void addEdge(int v, int w);
 // DFS traversal of the vertices
 // reachable from v
 void DFS(int v);
};
void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w); // Add w to v's list.
}
void Graph::DFS(int v)
{
 // Mark the current node as visited and
 // print it
 visited[v] = true;
 cout << v << " ";
 // Recur for all the vertices adjacent
 // to this vertex
 list<int>::iterator i;
 for (i = adj[v].begin(); i != adj[v].end(); ++i)
 if (!visited[*i])
 DFS(*i);
}
// Driver code
```

```
int main()
{
 // Create a graph given in the above diagram
 Graph g;
 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 0);
 g.addEdge(2, 3);
 g.addEdge(3, 3);
 cout << "Following is Depth First Traversal"
 " (starting from vertex 2) \n";
 g.DFS(2);
 return 0;
}
```

A star (A*) 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

    #ditance 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
#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
    H_dist = {
        'A': 11,
```

```
            'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,

        }

        return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],

}
aStarAlgo('A', 'G')
```

Output:

Kruskals Algorithm

Source Code :-

```
#include<stdio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
int main() {
printf("\n Implementation of Kruskal's algorithm\n\n");
printf("\nEnter the no. of vertices\n");
scanf("%d",&n);
printf("\nEnter the cost adjacency matrix\n");
for(i=1;i<=n;i++){
for(j=1;j<=n;j++) {
scanf("%d",&cost[i][j]);
if(cost[i][j]==0)
cost[i][j]=999;
}}
printf("\nThe edges of Minimum Cost Spanning Tree are\n\n");
while(ne<n){
for(i=1,min=999;i<=n;i++) {
for(j=1;j<=n;j++){
if(cost[i][j]<min){
min=cost[i][j];
a=u=i;
b=v=j;
}}}
u=find(u);
v=find(v);
if(uni(u,v)){
printf("\n%d edge (%d,%d) =%d\n",ne++,a,b,min);
```

```
mincost +=min;
}
cost[a][b]=cost[b][a]=999;}
printf("\n\tMinimum cost = %d\n",mincost);}
int find(int i){
while(parent[i])
i=parent[i];
return i;}
int uni(int i,int j){
if(i!=j) {
parent[j]=i;
return 1;}
return 0;
}
```

OutPut :-

```
Enter the no. of vertices : 6
Enter the cost adjacency matrix
0 4 0 0 0 2
4 0 6 0 0 3
0 6 0 3 0 1
0 0 3 0 2 0
0 0 0 2 0 4
2 3 1 0 4 0
The edges of Minimum Cost Spanning Tree are
1 edge (3,6) =1
2 edge (1,6) =2
3 edge (4,5) =2
4 edge (2,6) =3
5 edge (3,4) =3
        Minimum cost = 11
```

N Queens

Source Code :-

```
#include<iostream>
using namespace std;
#define N 4
void printBoard(int board[N][N]) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++)
      cout << board[i][j] << " ";
      cout << endl;
  }
}
bool isValid(int board[N][N], int row, int col) {
  for (int i = 0; i < col; i++) //check whether there is queen in the left or not
    if (board[row][i])
      return false;
  for (int i=row, j=col; i>=0 && j>=0; i--, j--)
    if (board[i][j]) //check whether there is queen in the left upper diagonal or not
      return false;
  for (int i=row, j=col; j>=0 && i<N; i++, j--)
    if (board[i][j]) //check whether there is queen in the left lower diagonal or not
      return false;
  return true;
}
bool solveNQueen(int board[N][N], int col) {
  if (col >= N) //when N queens are placed successfully
    return true;
  for (int i = 0; i < N; i++) { //for each row, check placing of queen is possible or not
    if (isValid(board, i, col) ) {
      board[i][col] = 1; //if validate, place the queen at place (i, col)
      if ( solveNQueen(board, col + 1)) //Go for the other columns recursively
        return true;
```

```
      board[i][col] = 0; //When no place is vacant remove that queen
    }
  }
  return false; //when no possible order is found
}
bool checkSolution() {
  int board[N][N];
  for(int i = 0; i<N; i++)
  for(int j = 0; j<N; j++)
  board[i][j] = 0; //set all elements to 0
  if ( solveNQueen(board, 0) == false ) { //starting from 0th column
    cout << "Solution does not exist";
    return false;
  }
  printBoard(board);
  return true;
}
int main() {
  checkSolution();
}
```

OutPut :-

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

ChatBot(Source Code)

```
# importing the required modules
from chatterbot import ChatBot
from chatterbot.trainers import ListTrainer
[08:54, 28/04/2022] Wandhekar Shubham Bhagwan: # creating a chatbot
myBot = ChatBot(
    name = 'Sakura',
    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 cool.',
    'Always cool.',
    'I\'m Okay',
    'Glad to hear that.',
    'I\'m fine',
    'I feel awesome',
    'Excellent, glad to hear that.',
    'Not so good',
    'Sorry to hear that.',
    'What\'s your name?',
    ' I\'m Sakura. 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 = a2 + 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 Sakura. 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 = a2 + b*2 - 2*a*b*cos(gamma)
```