**1)**

#include <bits/stdc++.h>

#define N 8

using namespace std;

void printSolution(int board[N][N])

{

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++)

cout << " " << board[i][j] << " ";

printf("\n");

}

}

bool isSafe(int board[N][N], int row, int col)

{

int i, j;

/\* Check this row on left side \*/

for (i = 0; i < col; i++)

if (board[row][i])

return false;

/\* Check upper diagonal on left side \*/

for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

if (board[i][j])

return false;

/\* Check lower diagonal on left side \*/

for (i = row, j = col; j >= 0 && i < N; i++, j--)

if (board[i][j])

return false;

return true;

}

/\* A recursive utility function to solve N

Queen problem \*/

bool solveNQUtil(int board[N][N], int col)

{

if (col >= N)

return true;

for (int i = 0; i < N; i++) {

if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveNQUtil(board, col + 1))

return true;

board[i][col] = 0; // BACKTRACK

}

}

return false;

}

bool solveNQ()

{

int board[N][N] = { { 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0 }

};

if (solveNQUtil(board, 0) == false) {

cout << "Solution does not exist";

return false;

}

printSolution(board);

return true;

}

int main()

{

solveNQ();

return 0;

}

**2)bfs**

visited = [] # List for visited nodes.

queue = [] #Initialize a queue

def bfs(visited, graph, node): #function for BFS

visited.append(node)

queue.append(node)

while queue: # Creating loop to visit each node

m = queue.pop(0)

print (m, end = " ")

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

graph = {

'0' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : ['4'],

'4' : ['8'],

'8' : []

}

# Driver Code

print("Following is the Breadth-First Search")

bfs(visited, graph, '0') # function calling

**DFS**

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node): #function for dfs

if node not in visited:

print (node)

visited.add(node)

for neighbour in graph[node]:

dfs(visited, graph, neighbour)

# Driver Code

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

print("Following is the Depth-First Search")

dfs(visited, graph, '5')

**A\***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | class Graph:      def \_\_init\_\_(self, adjacency\_list):          self.adjacency\_list = adjacency\_list      def get\_neighbors(self, v):          return self.adjacency\_list[v]      # heuristic function with equal values for all nodes      def h(self, n):          H = {              'A': 1,              'B': 1,              'C': 1,              'D': 1          }          return H[n]      def a\_star\_algorithm(self, start\_node, stop\_node):            open\_list = set([start\_node])          closed\_list = set([])          g = {}          g[start\_node] = 0          parents = {}          parents[start\_node] = start\_node          while len(open\_list) > 0:              n = None              # find a node with the lowest value of f() - evaluation function              for v in open\_list:                  if n == None or g[v] + self.h(v) < g[n] + self.h(n):                      n = v              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:                  reconst\_path = []                  while parents[n] != n:                      reconst\_path.append(n)                      n = parents[n]                  reconst\_path.append(start\_node)                  reconst\_path.reverse()                  print('Path found: {}'.format(reconst\_path))                  return reconst\_path              # for all neighbors of the current node do              for (m, weight) in self.get\_neighbors(n):                  # if the current node isn't in both open\_list and closed\_list                  # add it to open\_list and note n as it's parent                  if m not in open\_list and m not in closed\_list:                      open\_list.add(m)                      parents[m] = n                      g[m] = g[n] + weight                  # otherwise, check if it's quicker to first visit n, then m                  # and if it is, update parent data and g data                  # and if the node was in the closed\_list, move it to open\_list                  else:                      if g[m] > g[n] + weight:                          g[m] = g[n] + weight                          parents[m] = n                          if m in closed\_list:                              closed\_list.remove(m)                              open\_list.add(m)              # remove n from the open\_list, and add it to closed\_list              # because all of his neighbors were inspected              open\_list.remove(n)              closed\_list.add(n)          print('Path does not exist!')          return None  adjacency\_list = {      'A': [('B', 1), ('C', 3), ('D', 7)],      'B': [('D', 5)],      'C': [('D', 12)]  }  graph1 = Graph(adjacency\_list)  graph1.a\_star\_algorithm('A', 'D')   |  |  | | --- | --- | |  |  | |  |