Dronacharya College of Engineering, Gurgaon

Lab Manual

Lab: "ARTIFICIAL INTELLIGENCE LAB USING PYTHON"
Course Code: LC-CSE-326G

1. Write a Program to Implement Breadth First Search using Python.

```
# Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (max(self.graph) + 1)

        # Create a queue for BFS
        queue = []

        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from
            # queue and print it
```

```
        s = queue.pop(0)
        print (s, end = " ")

        # 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 in self.graph[s]:
            if visited[i] == False:
                queue.append(i)
                visited[i] = True

# Driver code

# Create a graph given in
# the above diagram
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 Traversal"
                  " (starting from vertex 2)")
g.BFS(2)
```

Output:

Following is Breadth First Traversal (starting from vertex 2)

> 3

2 0 3 1 3

>

2. Write a Program to Implement Depth First Search using Python.

```
# program to print DFS traversal
# from a given given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation


class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self, v, visited):

        # Mark the current node as visited
        # and print it
        visited.add(v)
        print(v, end=' ')

        # Recur for all the vertices
        # adjacent to this vertex
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self, v):

        # Create a set to store visited vertices
        visited = set()

        # Call the recursive helper function
        # to print DFS traversal
        self.DFSUtil(v, visited)

# Driver code


# Create a graph given
# in the above diagram
```

```
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)
```

Output:

Following is Depth First Traversal (starting from vertex 2)

2 0 1 9 3

3. Write a Program to Implement Tic-Tac-Toe game using Python.

```
# Tic-Tac-Toe Program using

# random number in Python


# importing all necessary libraries

import numpy as np

import random

from time import sleep


# Creates an empty board

def create_board():

    return(np.array([[0, 0, 0],

                     [0, 0, 0],

                     [0, 0, 0]]))


# Check for empty places on board

def possibilities(board):

    l = []


    for i in range(len(board)):

        for j in range(len(board)):
```

```
            if board[i][j] == 0:
                l.append((i, j))
    return(l)


# Select a random place for the player
def random_place(board, player):
    selection = possibilities(board)
    current_loc = random.choice(selection)
    board[current_loc] = player
    return(board)


# Checks whether the player has three
# of their marks in a horizontal row
def row_win(board, player):
    for x in range(len(board)):
        win = True

        for y in range(len(board)):
            if board[x, y] != player:
```

```
                win = False
                continue
        if win == True:
            return(win)
    return(win)
# Checks whether the player has three
# of their marks in a vertical row
def col_win(board, player):
    for x in range(len(board)):
        win = True
        for y in range(len(board)):
            if board[y][x] != player:
                win = False
                continue
        if win == True:
            return(win)
    return(win)


# Checks whether the player has three
# of their marks in a diagonal row
```

```
def diag_win(board, player):
    win = True
    y = 0
    for x in range(len(board)):
        if board[x, x] != player:
            win = False
    if win:
        return win
    win = True
    if win:
        for x in range(len(board)):
            y = len(board) - 1 - x
            if board[x, y] != player:
                win = False
    return win


# Evaluates whether there is
# a winner or a tie
def evaluate(board):
    winner = 0
```

```
    for player in [1, 2]:
        if (row_win(board, player) or
            col_win(board,player) or
            diag_win(board,player)):

            winner = player

    if np.all(board != 0) and winner == 0:
        winner = -1
    return winner

# Main function to start the game
def play_game():
    board, winner, counter = create_board(), 0, 1
    print(board)
    sleep(2)

    while winner == 0:
        for player in [1, 2]:
```

```
        board = random_place(board, player)

        print("Board after " + str(counter) + " move")

        print(board)

        sleep(2)

        counter += 1

        winner = evaluate(board)

        if winner != 0:

            break

    return(winner)

# Driver Code

print("Winner is: " + str(play_game()))
```

Output:

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after 1 move
[[0 0 0]
 [0 0 0]
 [1 0 0]]
Board after 2 move
[[0 0 0]
 [0 2 0]
 [1 0 0]]
Board after 3 move
[[0 1 0]
 [0 2 0]
```

```
 [1 0 0]]
Board after 4 move
[[0 1 0]
 [2 2 0]
 [1 0 0]]
Board after 5 move
[[1 1 0]
 [2 2 0]
 [1 0 0]]
Board after 6 move
[[1 1 0]
 [2 2 0]
 [1 2 0]]
Board after 7 move
[[1 1 0]
 [2 2 0]
 [1 2 1]]
Board after 8 move
[[1 1 0]
 [2 2 2]
 [1 2 1]]
Winner is: 2
```

4. Write a Program to Implement 8-Puzzle problem using Python

```
class Solution:
```

```
def solve(self, board):
  dict = {}
  flatten = []
  for i in range(len(board)):
    flatten += board[i]
  flatten = tuple(flatten)

  dict[flatten] = 0

  if flatten == (0, 1, 2, 3, 4, 5, 6, 7, 8):
    return 0

  return self.get_paths(dict)

def get_paths(self, dict):
  cnt = 0
  while True:
    current_nodes = [x for x in dict if dict[x] == cnt]
    if len(current_nodes) == 0:
      return -1

    for node in current_nodes:
      next_moves = self.find_next(node)
      for move in next_moves:
        if move not in dict:
          dict[move] = cnt + 1
        if move == (0, 1, 2, 3, 4, 5, 6, 7, 8):
          return cnt + 1
```

```
        cnt += 1


    def find_next(self, node):
        moves = {
            0: [1, 3],
            1: [0, 2, 4],
            2: [1, 5],
            3: [0, 4, 6],
            4: [1, 3, 5, 7],
            5: [2, 4, 8],
            6: [3, 7],
            7: [4, 6, 8],
            8: [5, 7],
        }

        results = []
        pos_0 = node.index(0)
        for move in moves[pos_0]:
            new_node = list(node)
            new_node[move], new_node[pos_0] = new_node[pos_0], new_node[move]
            results.append(tuple(new_node))

        return results
ob = Solution()
matrix = [
    [3, 1, 2],
    [4, 7, 5],
    [6, 8, 0]
```

```
]

print(ob.solve(matrix))
```

Input:

```
matrix = [
[3, 1, 2],
[4, 7, 5],
[6, 8, 0] ]
```

Output:

4

5. Write a Program to Implement Water-Jug problem using Python

```
# This function is used to initialize the
# dictionary elements with a default value.
```

```
from collections import defaultdict

# jug1 and jug2 contain the value
# for max capacity in respective jugs
# and aim is the amount of water to be measured.
jug1, jug2, aim = 4, 3, 2

# Initialize dictionary with
# default value as false.
visited = defaultdict(lambda: False)

# Recursive function which prints the
# intermediate steps to reach the final
# solution and return boolean value
# (True if solution is possible, otherwise False).
# amt1 and amt2 are the amount of water present
# in both jugs at a certain point of time.
def waterJugSolver(amt1, amt2):

    # Checks for our goal and
    # returns true if achieved.
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True

    # Checks if we have already visited the
    # combination or not. If not, then it proceeds further.
    if visited[(amt1, amt2)] == False:
        print(amt1, amt2)

        # Changes the boolean value of
        # the combination as it is visited.
        visited[(amt1, amt2)] = True

        # Check for all the 6 possibilities and
        # see if a solution is found in any one of them.
        return (waterJugSolver(0, amt2) or
                waterJugSolver(amt1, 0) or
                waterJugSolver(jug1, amt2) or
                waterJugSolver(amt1, jug2) or
                waterJugSolver(amt1 + min(amt2, (jug1-amt1)),
                amt2 - min(amt2, (jug1-amt1))) or
                waterJugSolver(amt1 - min(amt1, (jug2-amt2)),
                amt2 + min(amt1, (jug2-amt2))))

    # Return False if the combination is
    # already visited to avoid repetition otherwise
    # recursion will enter an infinite loop.
    else:
        return False
```

```
print("Steps: ")

# Call the function and pass the
# initial amount of water present in both jugs.
waterJugSolver(0, 0)
```

Output:

```
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2
```

6. Write a Program to Implement Travelling Salesman Problem using Python.

```
# program to implement traveling salesman
# problem using naive approach.
from sys import maxsize
from itertools import permutations
V = 4
# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):
    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)
    # store minimum weight
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:

        # store current Path weight(cost)
        current_pathweight = 0
```

```
        # compute current path weight
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]

        # update minimum
        min_path = min(min_path, current_pathweight)

    return min_path
# Driver Code
if __name__ == "__main__":
    # matrix representation of graph
    graph = [[0, 10, 15, 20], [10, 0, 35, 25],
            [15, 35, 0, 30], [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s))
```

Output

80

7. Write a Program to Implement Tower of Hanoi using Python.

```
# Recursive Python function to solve tower of hanoi


def TowerOfHanoi(n , from_rod, to_rod, aux_rod):
    if n == 1:
        print("Move disk 1 from rod",from_rod,"to rod",to_rod)
        return
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print("Move disk",n,"from rod",from_rod,"to rod",to_rod)
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)


# Driver code
n = 4
TowerOfHanoi(n, 'A', 'C', 'B')
# A, C, B are the name of rods
```

Output

```
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
```

Move disk 1 from rod B to rod C

Move disk 2 from rod B to rod A

Move disk 1 from rod C to rod A

Move disk 3 from rod B to rod C

Move disk 1 from rod A to rod B

Move disk 2 from rod A to rod C

Move disk 1 from rod B to rod C

Output:

Tower of Hanoi Solution for 4 disks:

A: [4, 3, 2, 1] B: [] C: []

Move disk from rod A to rod B
A: [4, 3, 2] B: [1] C: []

Move disk from rod A to rod C
A: [4, 3] B: [1] C: [2]

Move disk from rod B to rod C
A: [4, 3] B: [] C: [2, 1]

Move disk from rod A to rod B
A: [4] B: [3] C: [2, 1]

Move disk from rod C to rod A
A: [4, 1] B: [3] C: [2]

Move disk from rod C to rod B
A: [4, 1] B: [3, 2] C: []

Move disk from rod A to rod B
A: [4] B: [3, 2, 1] C: []

Move disk from rod A to rod C
A: [] B: [3, 2, 1] C: [4]

Move disk from rod B to rod C
A: [] B: [3, 2] C: [4, 1]

Move disk from rod B to rod A
A: [2] B: [3] C: [4, 1]

Move disk from rod C to rod A
A: [2, 1] B: [3] C: [4]

Move disk from rod B to rod C
A: [2, 1] B: [] C: [4, 3]

Move disk from rod A to rod B
A: [2] B: [1] C: [4, 3]

Move disk from rod A to rod C
A: [] B: [1] C: [4, 3, 2]

Move disk from rod B to rod C
A: [] B: [] C: [4, 3, 2, 1]

8. Write a Program to Implement Monkey Banana Problem using Python.

```
from poodle import Object, schedule

from typing import Set

class Position(Object):

    def __str__(self):

        if not hasattr(self, "locname"): return "unknown"

        return self.locname

class HasHeight(Object):

    height: int

class HasPosition(Object):

    at: Position

class Monkey(HasHeight, HasPosition): pass

class PalmTree(HasHeight, HasPosition):

    def __init__(self, *args, **kwargs):

        super().__init__(*args, **kwargs)

        self.height = 2

class Box(HasHeight, HasPosition): pass

class Banana(HasHeight, HasPosition):

    owner: Monkey

    attached: PalmTree
```

```
class World(Object):

    locations: Set[Position]

p1 = Position()

p1.locname = "Position A"

p2 = Position()

p2.locname = "Position B"

p3 = Position()

p3.locname = "Position C"



w = World()

w.locations.add(p1)

w.locations.add(p2)

w.locations.add(p3)



m = Monkey()

m.height = 0 # ground

m.at = p1



box = Box()
```

```
box.height = 2

box.at = p2



p = PalmTree()

p.at = p3



b = Banana()

b.attached = p



def go(monkey: Monkey, where: Position):

    assert where in w.locations

    assert monkey.height < 1, "Monkey can only move while on the ground"

    monkey.at = where

    return f"Monkey moved to {where}"



def push(monkey: Monkey, box: Box, where: Position):

    assert monkey.at == box.at

    assert where in w.locations

    assert monkey.height < 1, "Monkey can only move the box while on the ground"
```

```
    monkey.at = where

    box.at = where

    return f"Monkey moved box to {where}"



def climb_up(monkey: Monkey, box: Box):

    assert monkey.at == box.at

    monkey.height += box.height

    return "Monkey climbs the box"



def grasp(monkey: Monkey, banana: Banana):

    assert monkey.height == banana.height

    assert monkey.at == banana.at

    banana.owner = monkey

    return "Monkey takes the banana"



def infer_owner_at(palmtree: PalmTree, banana: Banana):

    assert banana.attached == palmtree

    banana.at = palmtree.at

    return "Remembered that if banana is on palm tree, its location is where palm tree is"
```

```
def infer_banana_height(palmtree: PalmTree, banana: Banana):

    assert banana.attached == palmtree

    banana.height = palmtree.height

    return "Remembered that if banana is on the tree, its height equals tree's height"

print('\n'.join(x() for x in schedule(

        [go, push, climb_up, grasp, infer_banana_height, infer_owner_at],

        [w,p1,p2,p3,m,box,p,b],

        goal=lambda: b.owner == m)))
```

Result:

```
$ pip install poodle

$ python ./monkey.py

Monkey moved to Position B

Remembered that if banana is on the tree, its height equals tree's height

Remembered that if banana is on palm tree, its location is where palm tree is

Monkey moved box to Position C

Monkey climbs the box

Monkey takes the banana
```

9. Write a Program to Implement Missionaries-Cannibals Problems using Python.

```
''' mclib.py '''
class MCState:
    ### MC is missionaries and cannibals
    def __init__(self, state_vars, num_moves=0, parent=None):
        self.state_vars = state_vars
        self.num_moves = num_moves
        self.parent = parent

    ### decorator
    @classmethod
    def root(cls):
        return cls((3,3,1))

    def get_possible_moves(self):
        ''' return all possible moves in the game as tuples
        possible moves:
            1 or 2 mis
            1 or 2 cannibals
            1 mis, 1 can
        '''

        moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]
        return moves

    def is_legal(self):
        missionaries = self.state_vars[0]
        cannibals = self.state_vars[1]
        ## could have done tuple unpacking too:
        ## missionaries, cannibals, boat = self.state_vars

        if missionaries < 0 or missionaries > 3:
            return False
        elif cannibals < 0 or cannibals > 3:
            return False
        return True

        ## alternate
        # if 0 <= missionaries <= 3 and 0 <= cannibals <= 3
        #     return True
        ###

    def is_solution(self):
        if self.state_vars == (0,0,0):
            return True
        return False

    def is_failure(self):
        missionaries = self.state_vars[0]
        cannibals = self.state_vars[1]
        boat = self.state_vars[2]
```

```
        ## could have done tuple unpacking too:
        ## missionaries, cannibals, boat = self.state_vars

        ### missionaries on right side AND more cannibals than missionaries
        if missionaries > 0 and missionaries < cannibals:
            return True

        ## to make this easier to understand, I will create temporary variables
        ## but we could just substitute the math and skip the variables
        missionaries_on_left = 3 - missionaries
        cannibals_on_left = 3 - cannibals
        if missionaries_on_left > 0 and missionaries_on_left < cannibals_on_left:
            return True
        ## if you replace the math in, you get:
        #if 3 - missionaries > 0 and 3 - missionaires < 3 - cannaibals
        # which leads to:
        #if missionaries < 3 and cannibals < missionaries:

        ### if we make it here, we aren't in a failed state!
        return False


    def get_next_states(self):
        ## using possible move, get next states

        moves = self.get_possible_moves()
        all_states = list()
        mis_right, can_right, raft_right = self.state_vars
        ## if raft is on right, subtract move from these numbers
        ## if raft is on left, add these move numbers to these numbers
        for move in moves:
            change_mis, change_can = move
            if raft_right == 1:  ## mis_right = 3; can_right = 3, raft_right = 1
                new_state_vars = (mis_right-change_mis, can_right-change_can, 0)
            else:
                new_state_vars = (mis_right+change_mis, can_right+change_can, 1)

            ## notice the number of moves is increasing by 1
            ## also notice we are passing self to our child.
            new_state = MCState(new_state_vars, self.num_moves+1, self)
            if new_state.is_legal():
                all_states.append(new_state)

        return all_states

    def __str__(self):
        return "MCState[{}]".format(self.state_vars)

    def __repr__(self):
        return str(self)
```

```python
def search(dfs=True):


    ### this is the stack/queue that we used before
    from collections import deque

    ### create the root state
    root = MCState.root()


    ### we use the stack/queue for keeping track of where to search next
    to_search = deque()

    ### use a set to keep track fo where we've been
    seen_states = set()

    ### use a list to keep track of the solutions that have been seen
    solutions = list()

    ### start the search with the root
    to_search.append(root)

    ### safety variable for infinite loops!
    loop_count = 0
    max_loop = 10000

    ### while the stack/queue still has items
    while len(to_search) > 0:
        loop_count += 1
        if loop_count > max_loop:
            print(len(to_search))
            print("Escaping this super long loop!")
            break

        ### get the next item
        current_state = to_search.pop()

        ## look at the current state's children
        ## this uses the rule for actions and moves to create next states
        ## it is also removing all illegal states
        next_states = current_state.get_next_states()

        ## next_states is a list, so iterate through it
        for possible_next_state in next_states[::-1]:

            ## to see if we've been here before, we look at the state variables
            possible_state_vars = possible_next_state.state_vars
```

```
        ## we use the set and the "not in" boolean comparison
        if possible_state_vars not in seen_states:

            if possible_next_state.is_failure():
                #print("Failure!")
                continue
            elif possible_next_state.is_solution():
                ## Save it into our solutions list
                solutions.append(possible_next_state)
                #print("Solution!")
                continue



            # the state variables haven't been seen yet
            # so we add the state itself into the searching stack/queue

            #### IMPORTANT
            ## which side we append on changes how the search works
            ## why is this?
            if dfs:
                to_search.append(possible_next_state)
            else:
                to_search.appendleft(possible_next_state)

            # now that we have "seen" the state, we add the state vars to the set.
            # this means next time when we do the "not in", that will return False
            # because it IS in
            #seen_states.add(possible_state_vars)
            seen_states.add(possible_state_vars)

    ## finally, we reach this line when the stack/queue is empty (len(to_searching==))
    print("Found {} solutions".format(len(solutions)))
    return solutions

sol_dfs = search(True)
sol_bfs = search(False)


current_state = sol_dfs[0]
while current_state:
    print(current_state)
    current_state = current_state.parent

print("--")
current_state = sol_dfs[1]
while current_state:
    print(current_state)
    current_state = current_state.parent
```

```python
print("--")
current_state = sol_bfs[0]
while current_state:
    print(current_state)
    current_state = current_state.parent


print("--")
current_state = sol_bfs[1]
while current_state:
    print(current_state)
    current_state = current_state.parent
```

```
Found 2 solutions
Found 2 solutions
MCState[(0, 0, 0)]
MCState[(1, 1, 1)]
MCState[(0, 1, 0)]
MCState[(0, 3, 1)]
MCState[(0, 2, 0)]
MCState[(2, 2, 1)]
MCState[(1, 1, 0)]
MCState[(3, 1, 1)]
MCState[(3, 0, 0)]
MCState[(3, 2, 1)]
MCState[(3, 1, 0)]
MCState[(3, 3, 1)]
--
MCState[(0, 0, 0)]
MCState[(0, 2, 1)]
MCState[(0, 1, 0)]
MCState[(0, 3, 1)]
MCState[(0, 2, 0)]
MCState[(2, 2, 1)]
MCState[(1, 1, 0)]
MCState[(3, 1, 1)]
MCState[(3, 0, 0)]
MCState[(3, 2, 1)]
MCState[(3, 1, 0)]
MCState[(3, 3, 1)]
--
MCState[(0, 0, 0)]
MCState[(0, 2, 1)]
MCState[(0, 1, 0)]
MCState[(0, 3, 1)]
MCState[(0, 2, 0)]
MCState[(2, 2, 1)]
MCState[(1, 1, 0)]
MCState[(3, 1, 1)]
MCState[(3, 0, 0)]
```

```
MCState[(3, 2, 1)]
MCState[(2, 2, 0)]
MCState[(3, 3, 1)]
--
MCState[(0, 0, 0)]
MCState[(1, 1, 1)]
MCState[(0, 1, 0)]
MCState[(0, 3, 1)]
MCState[(0, 2, 0)]
MCState[(2, 2, 1)]
MCState[(1, 1, 0)]
MCState[(3, 1, 1)]
MCState[(3, 0, 0)]
MCState[(3, 2, 1)]
MCState[(2, 2, 0)]
MCState[(3, 3, 1)]
```

10. Write a Program to Implement N-Queens Problem using Python.

```
# Python program to solve N Queen
```

```
# Problem using backtracking

global N
N = 4

def printSolution(board):
    for i in range(N):
        for j in range(N):
            print board[i][j],
        print


# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens
def isSafe(board, row, col):

    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    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):
    # base case: If all queens are placed
    # then return true
    if col >= N:
        return True

    # Consider this column and try placing
    # this queen in all rows one by one
    for i in range(N):

        if isSafe(board, i, col):
            # Place this queen in board[i][col]
            board[i][col] = 1
```

```
            # recur to place rest of the queens
            if solveNQUtil(board, col + 1) == True:
                return True

            # If placing queen in board[i][col
            # doesn't lead to a solution, then
            # queen from board[i][col]
            board[i][col] = 0

    # if the queen can not be placed in any row in
    # this colum col  then return false
    return False

# This function solves the N Queen problem using
# Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one  of the
# feasible solutions.
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 program to test above function
solveNQ()
```

Output:

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0