Group B: Assignment No. 4

Title :- Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem.

N-Queens Code:-

```
#N-Queens

class QueenChessBoard:

  def __init__(self, size):

    # board has dimensions size x size

    self.size = size

    # columns[r] is a number c if a queen is placed at row r and column c.

    # columns[r] is out of range if no queen is place in row r.

    # Thus after all queens are placed, they will be at positions

    # (columns[0], 0), (columns[1], 1), ... (columns[size - 1], size - 1)

    self.columns = []

  def place_in_next_row(self, column):

    self.columns.append(column)

  def remove_in_current_row(self):

    return self.columns.pop()

  def is_this_column_safe_in_next_row(self, column):

    # index of next row

    row = len(self.columns)
```

```
        # check column
        for queen_column in self.columns:
            if column == queen_column:
                return False
        # check diagonal
        for queen_row, queen_column in enumerate(self.columns):
            if queen_column - queen_row == column - row:
                return False
        # check other diagonal
        for queen_row, queen_column in enumerate(self.columns):
            if ((self.size - queen_column) - queen_row
                == (self.size - column) - row):
                return False
        return True
    def display(self):
        for row in range(self.size):
            for column in range(self.size):
                if column == self.columns[row]:
                    print('Q', end=' ')
                else:
                    print('.', end=' ')
            print()
def solve_queen(size):
    """Display a chessboard for each possible configuration of placing n queens
    on an n x n chessboard and print the number of such configurations."""
    board = QueenChessBoard(size)
    number_of_solutions = 0
```

```
row = 0
column = 0
# iterate over rows of board
while True:
    # place queen in next row
    while column < size:
        if board.is_this_column_safe_in_next_row(column):
            board.place_in_next_row(column)
            row += 1
            column = 0
            break
        else:
            column += 1
    # if could not find column to place in or if board is full
    if (column == size or row == size):
        # if board is full, we have a solution
        if row == size:
            board.display()
            print()
            number_of_solutions += 1
            # small optimization:
            # In a board that already has queens placed in all rows except
            # the last, we know there can only be at most one position in
            # the last row where a queen can be placed. In this case, there
            # is a valid position in the last row. Thus we can backtrack two
            # times to reach the second last row.
```

```
            board.remove_in_current_row()
            row -= 1
        # now backtrack
        try:
            prev_column = board.remove_in_current_row()
        except IndexError:
            # all queens removed
            # thus no more possible configurations
            break
        # try previous row again
        row -= 1
        # start checking at column = (1 + value of column in previous row)
        column = 1 + prev_column
    print('Number of solutions:', number_of_solutions)
n = int(input('Enter n: '))
solve_queen(n)
```

Output:-

```
Enter n: 4
. Q . .
. . . Q
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .

Number of solutions: 2
```

Graph Coloring Code:-

```
# Adjacent Matrix
G = [[ 0, 1, 1, 0, 1, 0],
     [ 1, 0, 1, 1, 0, 1],
     [ 1, 1, 0, 1, 1, 0],
     [ 0, 1, 1, 0, 0, 1],
     [ 1, 0, 1, 0, 0, 1],
     [ 0, 1, 0, 1, 1, 0]]
# inisiate the name of node.
node = "abcdef"
t_={}
for i in range(len(G)):
  t_[node[i]] = i
# count degree of all node.
degree =[]
for i in range(len(G)):
  degree.append(sum(G[i]))
# inisiate the posible color
colorDict = {}
for i in range(len(G)):
  colorDict[node[i]]=["Blue","Red","Yellow","Green"]
# sort the node depends on the degree
sortedNode=[]
indeks = []
# use selection sort
for i in range(len(degree)):
  _max = 0
```

```
 j = 0
 for j in range(len(degree)):
  if j not in indeks:
   if degree[j] > _max:
    _max = degree[j]
    idx = j
 indeks.append(idx)
 sortedNode.append(node[idx])
# The main process
theSolution={}
for n in sortedNode:
 setTheColor = colorDict[n]
 theSolution[n] = setTheColor[0]
 adjacentNode = G[t_[n]]
 for j in range(len(adjacentNode)):
  if adjacentNode[j]==1 and (setTheColor[0] in colorDict[node[j]]):
   colorDict[node[j]].remove(setTheColor[0])
# Print the solution
for t,w in sorted(theSolution.items()):
 print("Node",t," = ",w)
```

Output:-

```
Node a  =  Yellow
Node b  =  Blue
Node c  =  Red
Node d  =  Yellow
Node e  =  Blue
Node f  =  Red
```