DAA assignment

(N-queens problem)

Submitted to:Satyam sir

Submitted by:

1. Mohit
2. Tarun
3. Dheeraj
4. Mukul

N Queen Problem

We have discussed Knight's tour and Rat in a Maze problems in Set 1 and Set 2 respectively. Let us discuss N Queen as another example problem that can be solved using Backtracking.

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.

The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

{ 0, 1, 0, 0}

{ 0, 0, 0, 1}

{ 1, 0, 0, 0}

{ 0, 0, 1, 0}

Naive Algorithm

Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

```
while there are untried configurations
{
   generate the next configuration
   if queens don't attack in this configuration then
   {
      print this configuration;
   }
}
```

Backtracking Algorithm

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

1) Start in the leftmost column

2) If all queens are placed

return true

3) Try all rows in the current column. Do following for every tried row.

a) If the queen can be placed safely in this row then mark this [row,

column] as part of the solution and recursively check if placing queen here leads to a solution.

b) If placing the queen in [row, column] leads to a solution then return

true.

c) If placing queen doesn't lead to a solution then umark this [row,

column] (Backtrack) and go to step (a) to try other rows.

3) If all rows have been tried and nothing worked, return false to trigger

backtracking.

Implementation of Backtracking solution:

```
# 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:

The 1 values indicate placements of queens

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```