**Group-B**

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

**8 Queens Problem using Branch and Bound**

The N-Queens problem is a puzzle of placing exactly N queens on an NxN chessboard, such that no two queens can attack each other in that configuration. Thus, no two queens can lie in the same row, column or diagonal.

The branch and bound solution is somehow different, it generates a partial solution until it figures that there's no point going deeper as we would ultimately lead to a dead end.

In the backtracking approach, we maintain an 8x8 binary matrix for keeping track of safe cells (by eliminating the unsafe cells, those that are likely to be attacked) and update it each time we place a new queen. However, it required O(n^2) time to check safe cell and update the queen.

In the 8 queens problem, we ensure the following:

no two queens share a row

no two queens share a column

no two queens share the same left diagonal

no two queens share the same right diagonal

we already ensure that the queens do not share the same column by the way we fill out our auxiliary matrix (column by column). Hence, only the left out 3 conditions are left out to be satisfied.

Applying the branch and bound approach :

The branch and bound approach suggest that we create a partial solution and use it to ascertain whether we need to continue in a particular direction or not. For this problem, we create 3 arrays to check for conditions 1,3 and 4. The Boolean arrays tell which rows and diagonals are already occupied. To achieve this, we need a numbering system to specify which queen is placed.

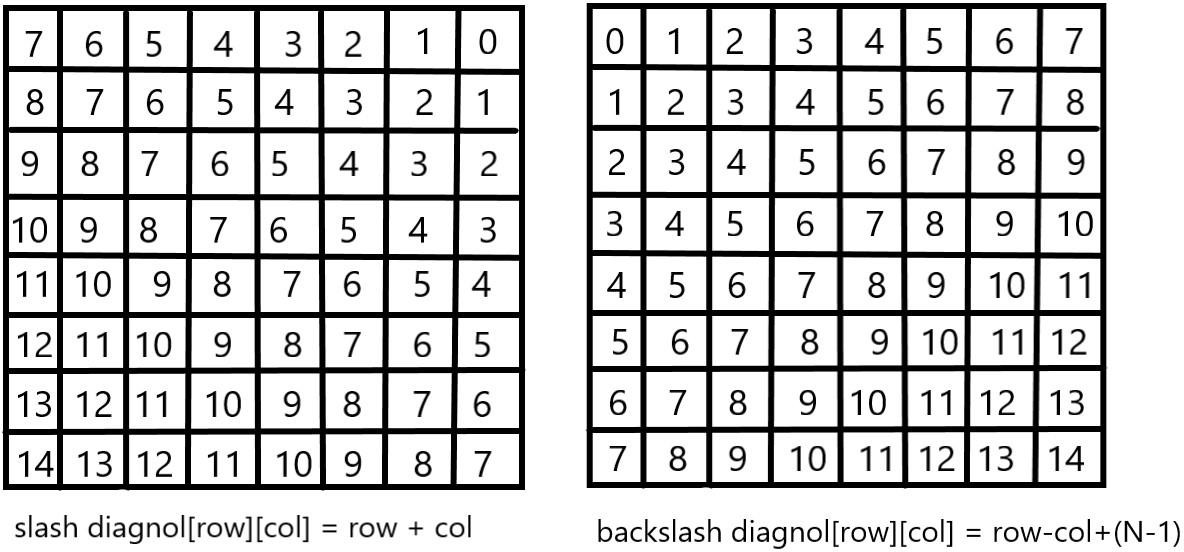
The indexes on these arrays would help us know which queen we are analyzing.

**Preprocessing** - create two NxN matrices, one for top-left to bottom- right diagonals, and other for top-right to bottom-left diagonal. We need to fill these in such a way that two queens sharing same top-left-bottom-right diagonal will have same value in slash Diagonal and two queens sharing same top-right bottom-left diagonal will have same value in backSlashDiagonal.

slashDiagnol(row)(col) = row + col

backSlashDiagnol(row)(col) = row - col + (N-1) { N = 8 }

{ we added (N-1) as we do not need negative values in backSlashDiagnol }



For placing a queen *i* on row *j*, check the following :

1. whether row 'j' is used or not
2. whether slashDiagonal 'i+j' is used or not
3. whether backSlashDiagonal 'i-j+7' is used or not

If the answer to any one of the following is true, we try another location for queen **i** on row **j**, mark the row and diagonals; and recur for queen **i+1**.

Conclusion :

**Implementation :**

""" Python3 program to solve N Queen Problem

using Branch or Bound """

N = 8

""" A utility function to print solution """

def printSolution(board):

for i in range(N):

for j in range(N):

print(board[i][j], end = " ")

print()

""" A Optimized function to check if

a queen can be placed on board[row][col] """

def isSafe(row, col, slashCode, backslashCode,

rowLookup, slashCodeLookup,

backslashCodeLookup):

if (slashCodeLookup[slashCode[row][col]] or

backslashCodeLookup[backslashCode[row][col]] or

rowLookup[row]):

return False

return True

""" A recursive utility function

to solve N Queen problem """

def solveNQueensUtil(board, col, slashCode, backslashCode,

rowLookup, slashCodeLookup,

backslashCodeLookup):

""" base case: If all queens are

placed then return True """

if(col >= N):

return True

for i in range(N):

if(isSafe(i, col, slashCode, backslashCode,

rowLookup, slashCodeLookup,

backslashCodeLookup)):

""" Place this queen in board[i][col] """

board[i][col] = 1

rowLookup[i] = True

slashCodeLookup[slashCode[i][col]] = True

backslashCodeLookup[backslashCode[i][col]] = True

""" recur to place rest of the queens """

if(solveNQueensUtil(board, col + 1,

slashCode, backslashCode,

rowLookup, slashCodeLookup,

backslashCodeLookup)):

return True

""" If placing queen in board[i][col]

doesn't lead to a solution,then backtrack """

""" Remove queen from board[i][col] """

board[i][col] = 0

rowLookup[i] = False

slashCodeLookup[slashCode[i][col]] = False

backslashCodeLookup[backslashCode[i][col]] = False

""" If queen can not be place in any row in

this column col then return False """

return False

""" This function solves the N Queen problem using

Branch or Bound. It mainly uses solveNQueensUtil()to

solve the problem. It returns False if queens

cannot be placed,otherwise return True or

prints placement of queens in the form of 1s.

Please note that there may be more than one

solutions,this function prints one of the

feasible solutions."""

def solveNQueens():

board = [[0 for i in range(N)]

for j in range(N)]

# helper matrices

slashCode = [[0 for i in range(N)]

for j in range(N)]

backslashCode = [[0 for i in range(N)]

for j in range(N)]

# arrays to tell us which rows are occupied

rowLookup = [False] \* N

# keep two arrays to tell us

# which diagonals are occupied

x = 2 \* N - 1

slashCodeLookup = [False] \* x

backslashCodeLookup = [False] \* x

# initialize helper matrices

for rr in range(N):

for cc in range(N):

slashCode[rr][cc] = rr + cc

backslashCode[rr][cc] = rr - cc + 7

if(solveNQueensUtil(board, 0, slashCode, backslashCode,

rowLookup, slashCodeLookup,

backslashCodeLookup) == False):

print("Solution does not exist")

return False

# solution found

printSolution(board)

return True

# Driver Code

solveNQueens()

Output :

1 0 0 0 0 0 0 0

0 0 0 0 0 0 1 0

0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 1

0 1 0 0 0 0 0 0

0 0 0 1 0 0 0 0

0 0 0 0 0 1 0 0

0 0 1 0 0 0 0 0

True

The **graph coloring problem** is to discover whether the nodes of the graph G can be covered in such a way, that no two adjacent nodes have the same color yet only m colors are used. This graph coloring problem is also known as M- colorability decision problem.

The M – colorability optimization problem deals with the smallest integer m for which the graph G can be colored. The integer is known as a chromatic number of the graph.

Here, it can also be noticed that if d is the degree of the given graph, then it can be colored with d+ 1 color.

A graph is also known to be planar if and only if it can be drawn in a planar in such a way that no two edges cross each other. A special case is the 4 - colors problem for planar graphs. The problem is to color the region in a map in such a way that no two adjacent regions have the same color. Yet only four colors are needed. This is a problem for which graphs are very useful because a map can be easily transformed into a graph. Each region of the map becomes the node, and if two regions are adjacent, they are joined by an edge.

**Graph coloring problem** can also be solved using a state space tree, whereby applying a backtracking method required results are obtained.

For solving the **graph coloring problem**, we suppose that the graph is represented by its adjacency matrix G[ 1:n, 1:n], where, G[ i, j]= 1 if (i, j) is an edge of G, and G[i, j] = 0 otherwise.

The colors are represented by the integers 1, 2, ..., m and the solutions are given by the n-tuple (x1, x2, x3, ..., xn), where x1 is the color of node i.

#### Algorithm for finding the m - colorings of a graph

1.

2.

3.

4.

5.

6.

7.

Algorithm mcoloring ( k )

// this algorithm is formed using the recursive backtracking

// schema. The graph is represented by its Boolean adjacency

// matrix G [1: n, 1: n]. All assignments of 1, 2, …, m to the

// vertices of the graph such that adjacent vertices are

// assigned distinct are printed. K is the index

// of the next vertex to color.

8.

9.

10.

11.

12.

13.

14.

{

Repeat

{

// generate all legal assignments for x[k], Next value (k); // assign to x[k] a legal color.

If ( x[k] = 0 ) then return; // no new color possible

If (k = n) then // at most m colors have been used to color the n

vertices.

1. Write (x[1 : n ]);
2. Else mcoloring (k + 1);

17. }

18. Until (false);

19. }

This algorithm uses the recursive backtracking schema. In this algorithm colors to be assigned are to determine from the range (0, m), i.e., m colors are available.

The total time required by the above algorithm is **O (nm^n)**.

Conclusion :

**Implementation :**

# 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],end=' ')

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 column 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

True