Assignment No. 7

Aim

8-queens matrix

Problem Definition

8-Queens Matrix is Stored using JSON/XML having first Queen placed, use back-tracking to place remaining Queens to generate final 8-queen's Matrix using Python

Learning Objectives

- To understand the working of 8-queens problem
- To learn the concept of Backtracking
- To implement 8-queens problem in python

Learning Outcome

- Understood the concept of Backtracking
- Implemented 8-queens problem

Software And Hardware Requirements

- Latest 64-BIT Version of Linux Operating System
- Python editor and compiler
- Modelio Software

Mathematical Model

Let S be the system of solution set for given problem statement such that,
S = { s, e, X, Y, F, DD, NDD, Su, Fu }
where,
s = start state
such that, y = { y1 }

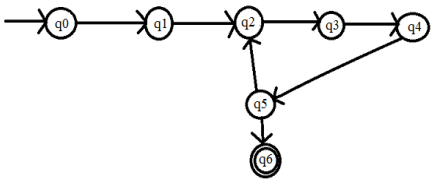
y1 = First queen is placed at 0^{th} row.
e = end state
such that, y = { y2 }
where, y2 = 8 Queens are palced on chessboard
X = set of inputs
such that, X = { x1 }
where, x1 = Row number for first Queen i.e. JSON data
Y = set of output
such that Y = { y1 }
where, y1 = [8 X 8] i.e binary matrix of order 8 X 8
Y_{ij} = 0 OR 1
If Y_{ij} == 1 means i^{th} queen is placed at i^{th} row and j^{th} column
F = set of function
such that F = { f1, f2, f3, f4 }
where,
f1 = function to check input is valid or not i.e. x and y is in between 0 to 7
f2 = function to check whether queens are attacking each other or not
f3 = function to place queen on right position
f4 = function to print board status
DD = Deterministic data
such that, DD = { x1 }
NDD = Nondeterministic data
such that, NDD = { y1 }
Su = Success case

- If the value of x and y is in between 0 to 7 then problem of 8 queen problem solved successfully.

Fu = Failure case

- If the value of x and y is not in between 0 to 7 then problem of 8 queen problem is not solved.

State Diagram



where,
q0 = Start state
q1 = Validate Input

q2 = Add queen in column
q3 = Check whether queen is attacking queen or not
q4 = Place queen
q5 = Check all queens are placed or not
q6 = End state

Theory

Backtracking

Backtracking is a refinement of the brute force approach, which systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors (v1, ..., vm) of values and by traversing, in a depth first manner, the domains of the vectors until the solutions are found.

When invoked, the algorithm starts with an empty vector. At each stage it extends the partial vector with a new value. Upon reaching a partial vector (v1, ..., vi) which cant represent a partial solution, the algorithm backtracks by removing the trailing value from the vector, and then proceeds by trying to extend the vector with alternative values.

Algorithm

```
ALGORITHM try(v1,...,vi)

IF (v1,...,vi) is a solution
THEN RETURN (v1,...,vi)

FOR each v DO

        IF (v1,...,vi,v) is acceptable vector
                THEN sol = try(v1,...,vi,v)

                IF sol != () THEN RETURN sol

        END

END

RETURN
```

If S_i is the domain of v_i, then S_1 x ... x S_m is the solution space of the problem. The validity criteria used in checking for acceptable vectors determines what portion of that space needs to be searched, and so it also determines the resources required by the algorithm.

The traversal of the solution space can be represented by a depth-first traversal of a tree. The tree itself is rarely entirely stored by the algorithm in discourse; instead just a path toward a root is stored, to enable the backtracking.

8 Queen's Problem

The eight queens puzzle is the problem of placing eight chess queens on an 8x8 chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. The eight queens puzzle is an example of the more general n-queens problem of placing n queens on an nxn chessboard.

The problem can be quite computationally expensive as there are 4,426,165,368 (i.e., ${}^{64}C_8$) possible arrangements of eight queens on an 88 board, but only 92 solutions. It is possible to use shortcuts that reduce computational requirements or rules of thumb that avoids brute-force computational techniques. For example, just by applying a simple rule that constrains each queen to a single column (or row), though still considered brute force, it is possible to reduce the number of possibilities to just 16,777,216 (that is, 8^8) possible combinations. Generating permutations further reduces the possibilities to just 40,320 (that is, 8!), which are then checked for diagonal attacks.

Solution

We can divide this problem into a sub problem of placing single queen on the chess board To identify if this leads to a solution, we can check and see does any of the already placed queens attack this position. If yes, then go to next column. If no, then place this queen and move on to next queen.

To illustrate this further,

1. Did we place all of the queens?
   - YES, we found the successful configuration. Print it.
   - NO, Take the queen
2. For each column from 1 to 8
   - Does this column is safe YES, then iterate this process for next queens call this method recursively
   - NO continue with next column

Program Code

```
import json

def isattack(board,r,c):
    for i in range(r):
        if(board[i][c]==1):
```

```
            return True

    i=r-1
    j=c-1
    while((i>=0) and (j>=0)):
        if(board[i][j]==1):
            return True
        i=i-1
        j=j-1

    i=r-1
    j=c+1
    while((i>=0) and (j<8)):
        if(board[i][j]==1):
            return True
        i=i-1
        j=j+1
    return False

def solve(board,row):
    i=0
    while(i<8):
        if(not isattack(board, row, i)):
            board[row][i]=1
            if(row==7):
                return True
            else:
                if(solve(board, row+1)):
                    return True
                else:
                    board[row][i]=0
        i=i+1

    if(i==8):
        return False

def printboard(board):
    for i in range(8):
        for j in range(8):
            print str(board[i][j])+"  ",
        print "\n"

board = [[0 for x in range(8)] for x in range(8)]

if __name__ == '__main__':
    data=[]
```

```
with open('input.json') as f:
    data=json.load(f)

if(data["start"]<0 or data["start"]>7):
    print "Invalid JSON input"
    exit()

board[0][data["start"]]=1
if(solve(board, 1)):
    print "8 Queens problem solved!!!"
    print "Board Configuration:"
    printboard(board)
else:
    print "Queens problem not solved!!!"
```

Output:

```
root@prasad-Lenovo-G550:/home/CL1/B1# python solve.py //start=0
8 Queens problem solved!!!
Board Configuration:
1 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 1

0 0 0 0 0 1 0 0

0 0 1 0 0 0 0 0

0 0 0 0 0 0 1 0

0 1 0 0 0 0 0 0

0 0 0 1 0 0 0 0

root@prasad-Lenovo-G550:/home/CL1/B1# python solve.py //start=3
8 Queens problem solved!!!
Board Configuration:
0 0 0 1 0 0 0 0

1 0 0 0 0 0 0 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 0 0 0 1 0

0 0 1 0 0 0 0 0

0 0 0 0 0 1 0 0

root@prasad-Lenovo-G550:/home/CL1/B1# python solve.py //start=8
Invalid JSON input

root@prasad-Lenovo-G550:/home/CL1/B1#
```

Testing

```
import unittest
from assgn import Queens

class test8Queens(unittest.TestCase):

    def testLoadFile(self):
        filename = 'input.json'
        obj = Queens(filename)
        try:
            obj.loadfile()
        except Exception:
            self.fail("File cannot be loaded.")

    def testValidFile(self):
        filename = 'input1.json'
        obj = Queens(filename)
        obj.loadfile()
        if(not obj.isvalid()):
            self.fail("Invalid Input")
```

Testing

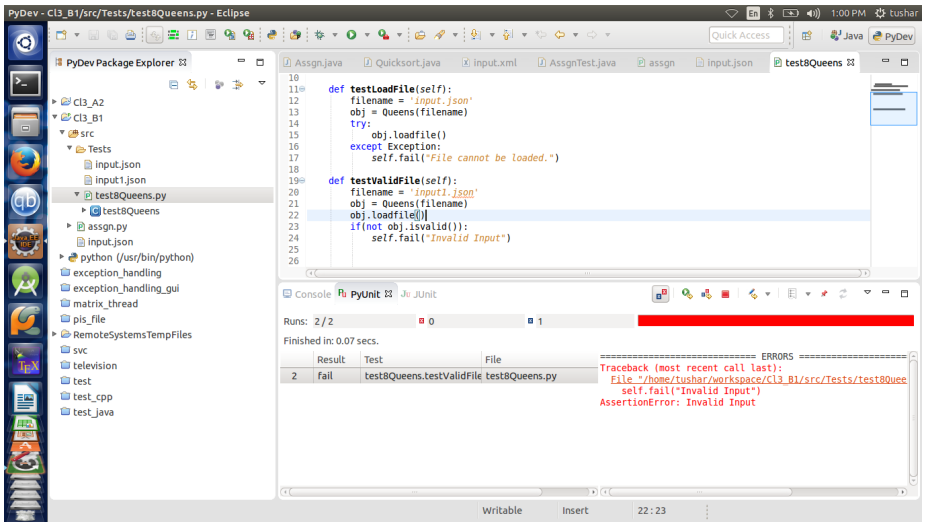


Fig1 : Testing

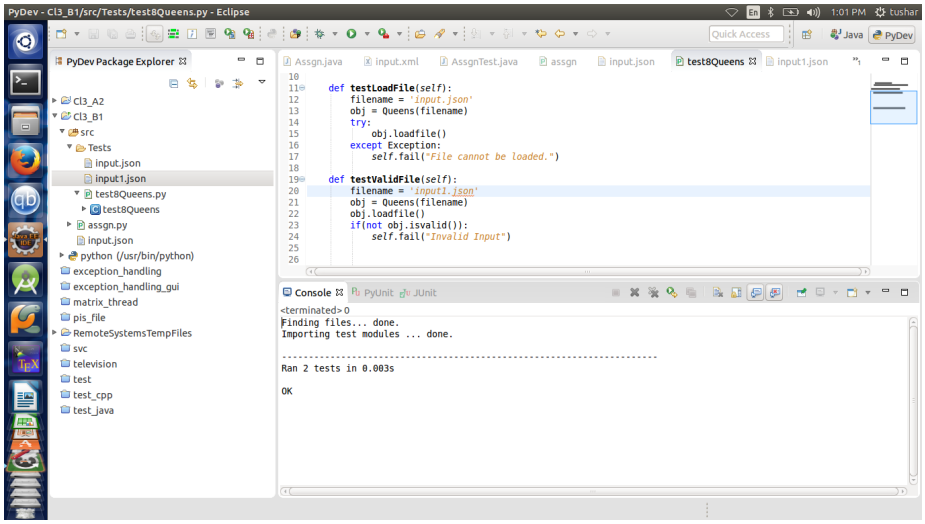


Fig2 : Testing

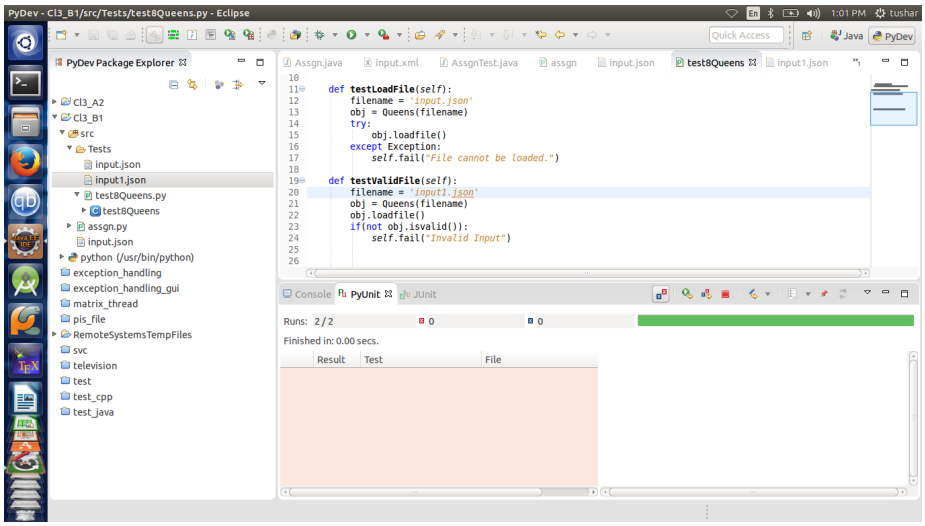
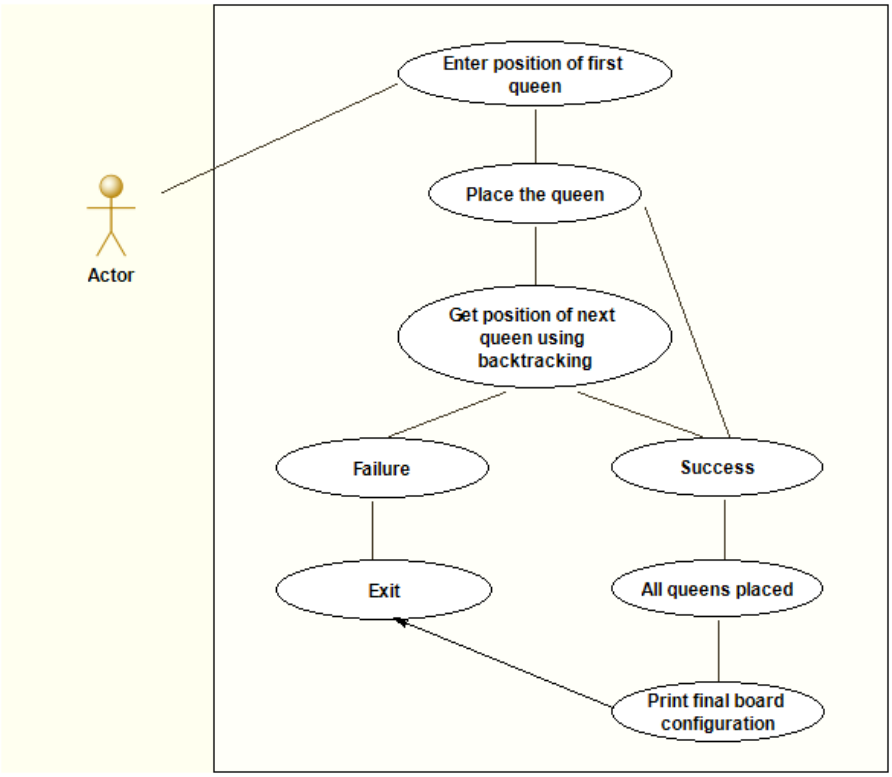


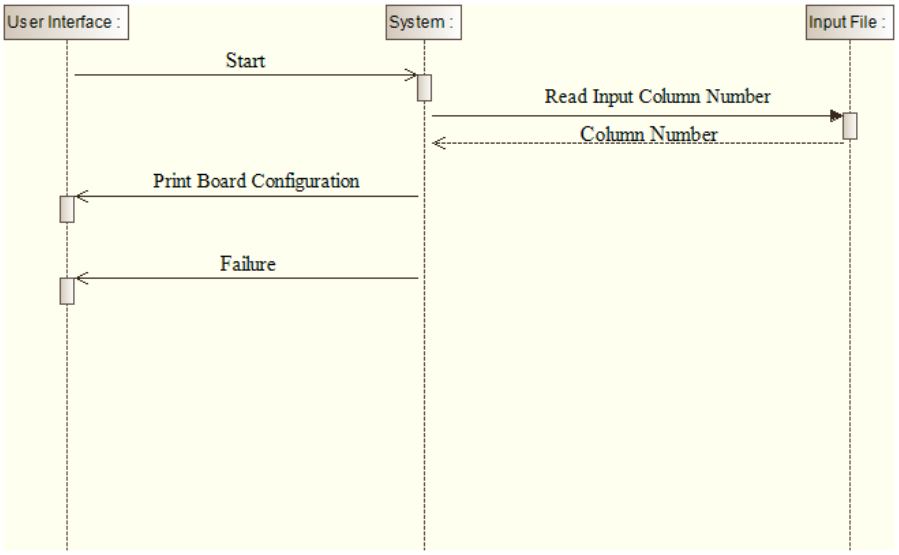
Fig3 : Testing

UML Diagrams

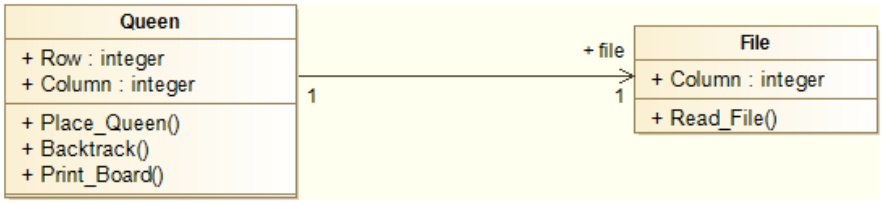
USE-CASE Diagram



Sequence Diagram



Class Diagram



Conclusion

Thus, using back-tracking strategy we successfully solved the 8-Queens Problem.