DAA assignment

(N-queens problem)

Submitted to: Satyam sir

Submitted by: 1.Mohit

2.Tarun

3.Dheeraj

4.Mukul

N queens problem

The N queens problem is the problem of placing N non-attacking queens on an NxN chessboard, for which solutions exist for all natural numbers N with the exception of N=2 and N=3.

When N=1, the solution is trivial so the program will ask for a value of N such that `N >= 4`.
I will solve this problem using backtracking. There are more efficient ways to solve this problem, but I will use backtracking since it's the most intuitive way to arrive at the solution without getting into the mathematics of arriving at efficient solutions. Through solving these problems, I aim to better understand Python.

What is backtracking?

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons each partial candidate *c* ("backtracks") as soon as it determines that *c*cannot possibly be completed to a valid solution.
A high level overview of how to use backtracking to solve the N queens problem:

- place a queen in the first column and first row
- place a queen in the second column such that it does not attack the queen in the first column
- continue placing non-attacking queens in the remaining columns
- if all N queens have been placed, a solution has been found. Remove the queen in the Nth column, and try incrementing the row of the queen in the (N-1)th column
- if it's a dead end, remove the queen, increment the row of the queen in the previous column
- continue doing this until the queen in the 1st column exhausts all options and is in the row N

The above explanation starts counting at 1, not 0 based counting.

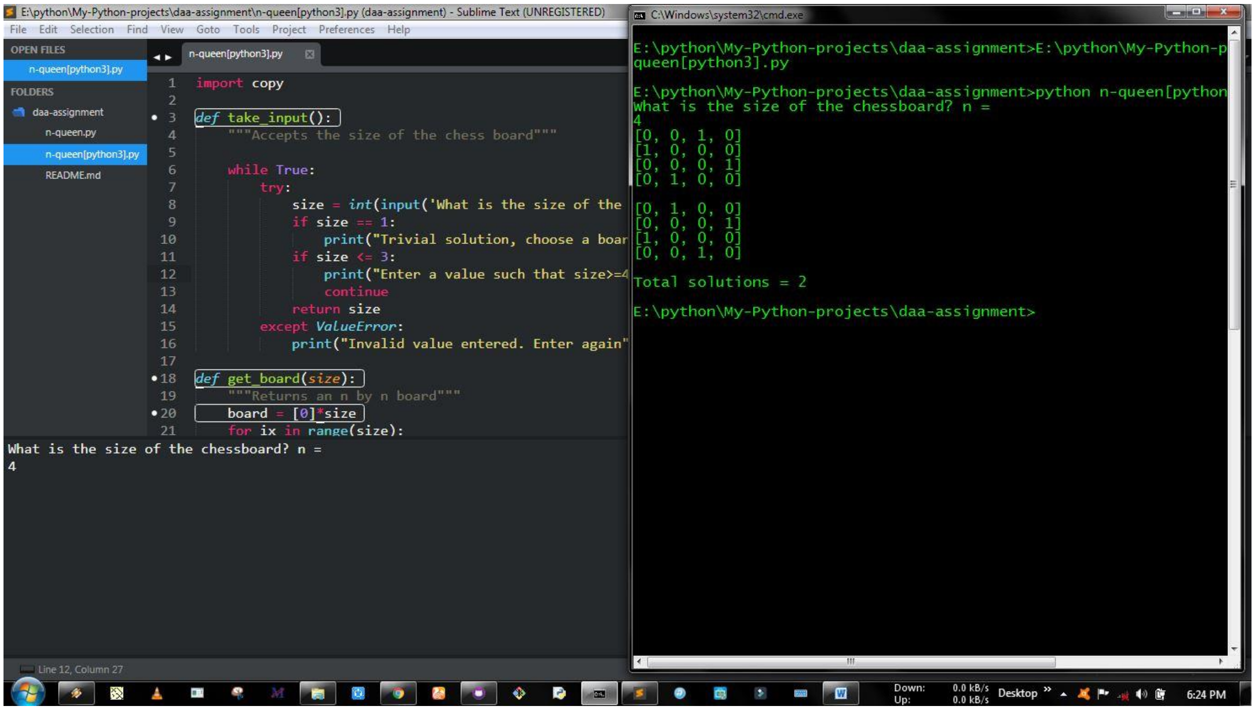
The solution:

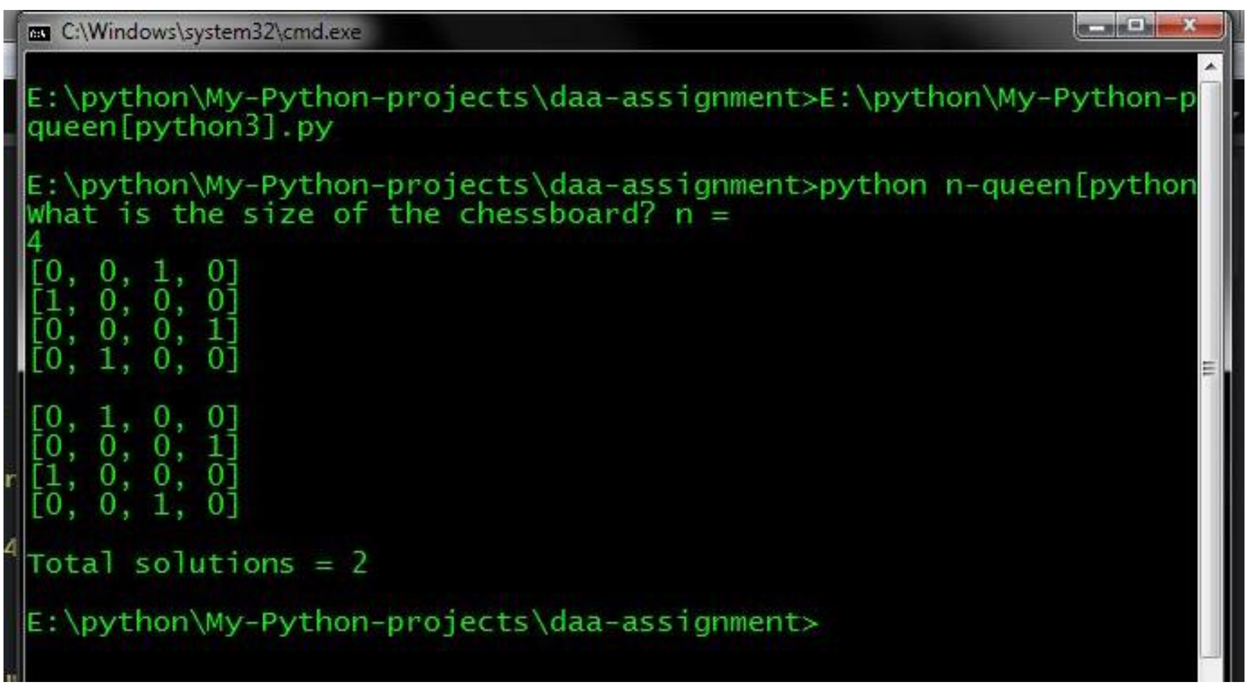
```
import copy

def take_input():
    """Accepts the size of the chess board"""

    while True:
        try:
            size = int(input('What is the size of the chessboard? n = 
\n'))
            if size == 1:
                print("Trivial solution, choose a board size of 
atleast 4")
            if size <= 3:
                print("Enter a value such that size>=4")
                continue
            return size
        except ValueError:
            print("Invalid value entered. Enter again")

def get_board(size):
    """Returns an n by n board"""
    board = [0]*size
    for ix in range(size):
        board[ix] = [0]*size
    return board

def print_solutions(solutions, size):
    """Prints all the solutions in user friendly way"""
    for sol in solutions:
        for row in sol:
            print(row)
        print()

def is_safe(board, row, col, size):
    """Check if it's safe to place a queen at board[x][y]"""

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

    ix, iy = row, col
    while ix >= 0 and iy >= 0:
        if board[ix][iy] == 1:
            return False
        ix-=1
        iy-=1

    jx, jy = row,col
```

```python
    while jx < size and jy >= 0:
        if board[jx][jy] == 1:
            return False
        jx+=1
        jy-=1

    return True

def solve(board, col, size):
    """Use backtracking to find all solutions"""
    #base case
    if col >= size:
        return

    for i in range(size):
        if is_safe(board, i, col, size):
            board[i][col] = 1
            if col == size-1:
                add_solution(board)
                board[i][col] = 0
                return
            solve(board, col+1, size)
            #backtrack
            board[i][col] = 0

def add_solution(board):
    """Saves the board state to the global variable 'solutions'"""
    global solutions
    saved_board = copy.deepcopy(board)
    solutions.append(saved_board)

size = take_input()

board = get_board(size)

solutions = []

solve(board, 0, size)

print_solutions(solutions, size)

print("Total solutions = {}".format(len(solutions)))
```

link: https://github.com/mmaithani/daa-assignment/blob/master/n-queen%5Bpython3%5D.py

Output of the program when N=4:





Here 1 represent queen

Floyd Warshall Algorithm

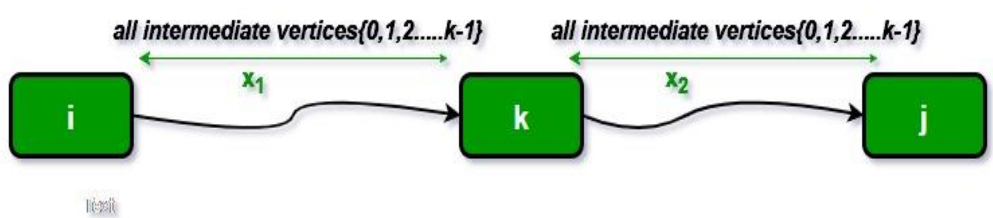
The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

In Floyd Warshall Algorithm We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

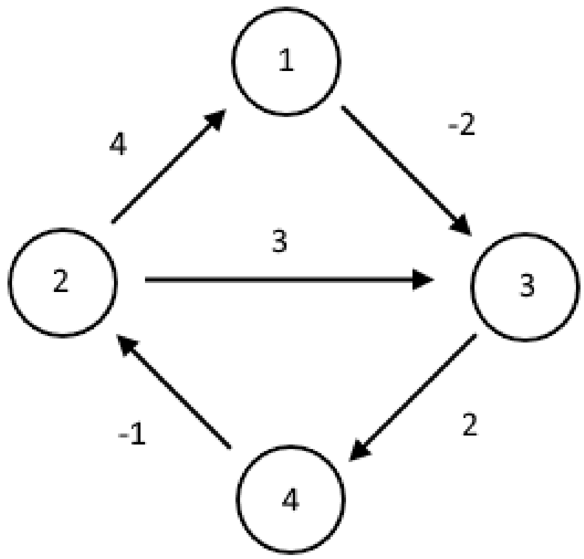
2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.



Task

Find the lengths of the shortest paths between all pairs of vertices of the given directed graph. Your code may assume that the input has already been checked for loops, parallel edges and negative cycles.

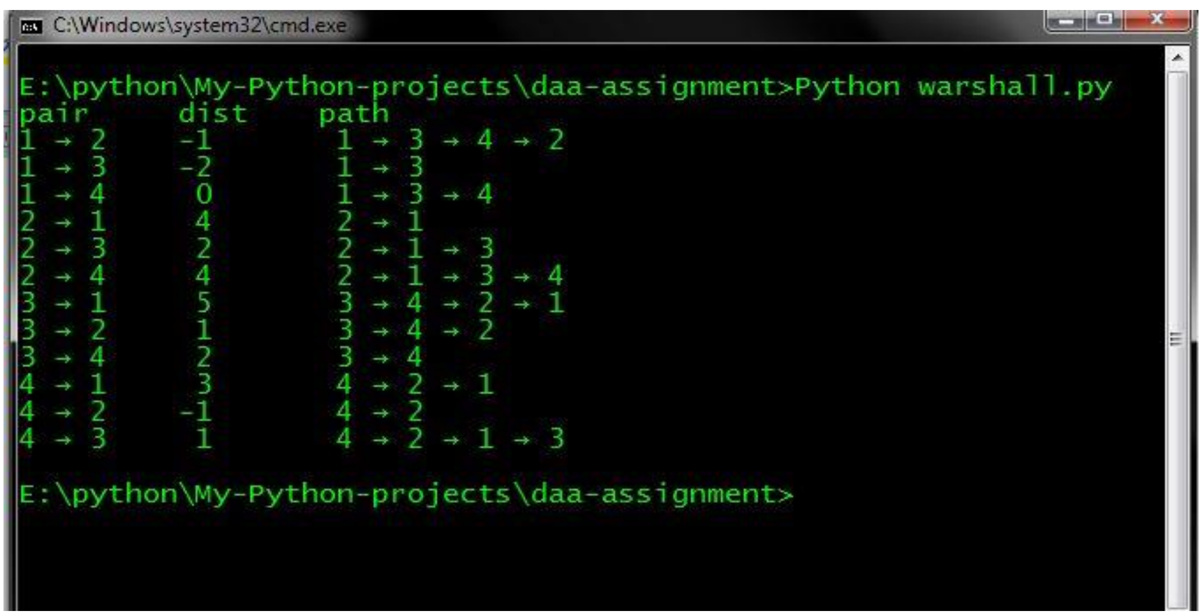


Print the pair, the distance and (optionally) the path.

Link: https://github.com/mmaithani/daa-assignment/blob/master/warshall.py

 # Python Program for Floyd Warshall Algorithm :

```python
from math import inf
from itertools import product

def floyd_warshall(n, edge):
    rn = range(n)
    dist = [[inf] * n for i in rn]
    nxt  = [[0]   * n for i in rn]
    for i in rn:
        dist[i][i] = 0
    for u, v, w in edge:
        dist[u-1][v-1] = w
        nxt[u-1][v-1] = v-1
    for k, i, j in product(rn, repeat=3):
        sum_ik_kj = dist[i][k] + dist[k][j]
        if dist[i][j] > sum_ik_kj:
            dist[i][j] = sum_ik_kj
            nxt[i][j]  = nxt[i][k]
    print("pair     dist    path")
    for i, j in product(rn, repeat=2):
        if i != j:
            path = [i]
            while path[-1] != j:
                path.append(nxt[path[-1]][j])
            print("%d → %d  %4d       %s"
                  % (i + 1, j + 1, dist[i][j],
                     ' → '.join(str(p + 1) for p in path)))

if __name__ == '__main__':
    floyd_warshall(4, [[1, 3, -2], [2, 1, 4], [2, 3, 3], [3, 4, 2], [4, 2, -1
]])
```



output