CS 599: Algorithms **(Due: October 22nd, 2020)**

Submission Homework 4

Instructor: Prof. Prasad Tadepalli *Name:* Rashmi Jadhav, *Student ID:* 934-069-574

1. Implementation of N-Queens

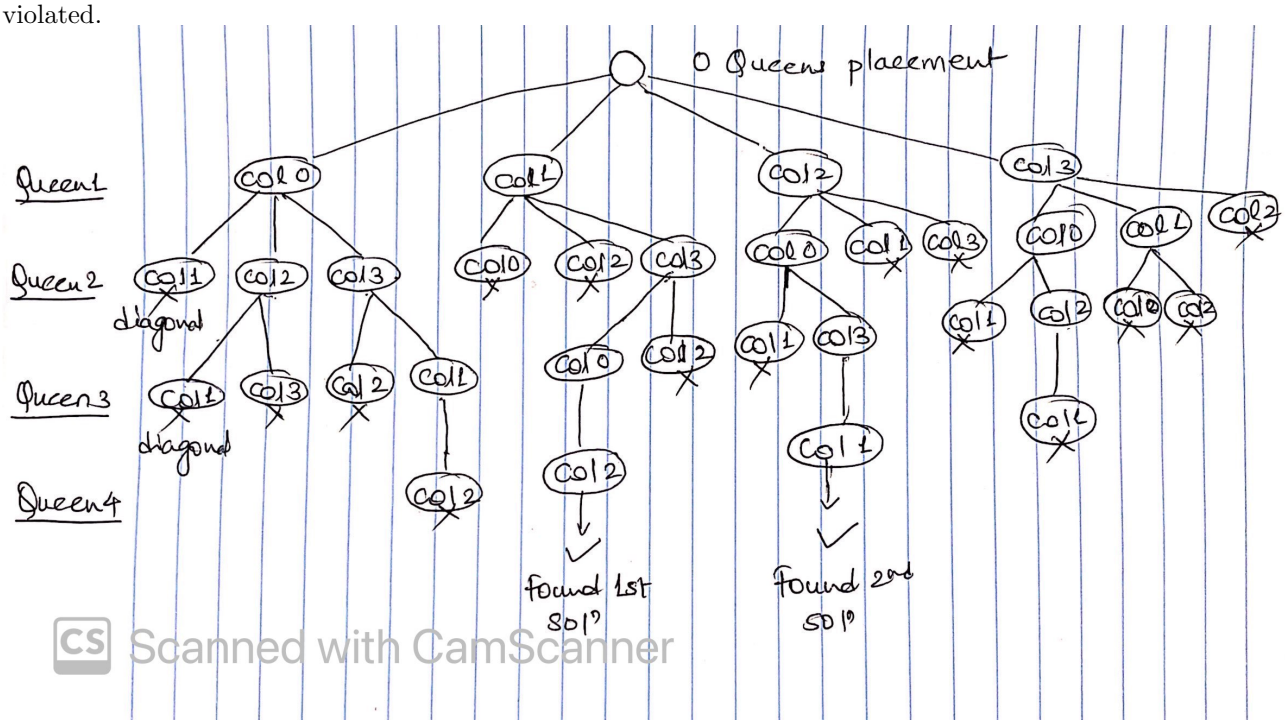
(a) For exhaustive search, we first generate all possible permutations corresponding to the board's positioning of N Queens such that the ith queen is placed at $permutation[i]^{th}$ column. Once we have those generated, we check for the condition of diagonal attack as *permutations()* takes care of not having repeated elements i.e., same column number for two different queens. If permutation follows valid check, we have a solution.

```
from itertools import permutations


def e_queens(n, solutions):  # exhaustive search
    # Generate all possible permutations representing Q1->index 0, Q2->index1, ..., Qn->index n - 1
    for perm in permutations(range(n)):
        if is_valid_perm(perm, n):  # Check if attack conditions are met
            solutions.append(perm)


def is_valid_perm(permutation, n):
    for row_id in range(n):  # for each queen
        for i in range(row_id):  # compare queen at row_id's placement with the other columns
            if abs(permutation[i] - permutation[row_id]) == row_id - i:  # diagonal clash check
                return False
    return True
```

(b) For backtracking search, we try each column for a queen and terminate the recursion if attack conditions are violated.



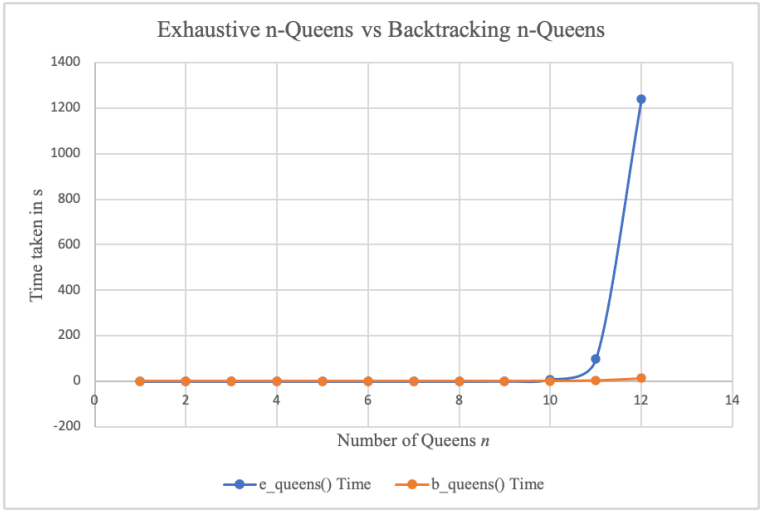
```
def b_queens(row, n, placement, solutions):  # backtrack search
    if row == n:  # when row reaches n, we have placed all the n queens without any attack conflicts
        solutions.append(placement[:])
    else:
        for column in range(n):  # explore n columns for row'th queen
            placement.append(column)
            # print(placement)
            if is_valid(placement):  # check if the row'th queen placed at column attacks already placed queens
                b_queens(row + 1, n, placement, solutions)
            placement.pop()  # done exploring column'th position


def is_valid(placement):
    row_id = len(placement) - 1
    # 0 to row_id - 1 queens are placed without attack conflicts
    for i in range(row_id):  # compare newly placed row_id'th queen with already placed queens for attacks
        diff = abs(placement[i] - placement[row_id])
        if diff == 0 or diff == row_id - i:  # same column clash or diagonal clash
            return False
    return True
```

(c) Following run experiments were performed on *orst.engr* servers:

i. exhaustive vs backtracking n-Queens in terms of time:

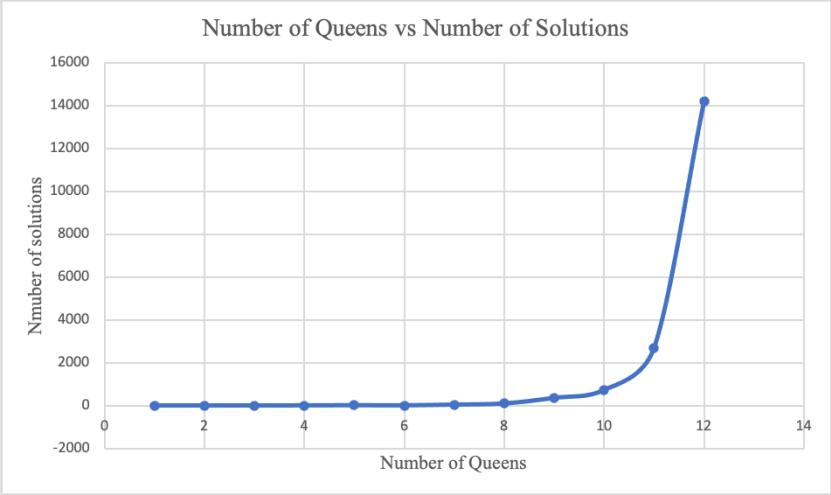
Number of Queens	*e_queens()* Time in s	*b_queens()* Time in s
1	1.09673E-05	5.00679E-06
2	7.86781E-06	1.00136E-05
3	1.09673E-05	1.90735E-05
4	4.1008E-05	6.38962E-05
5	0.000208855	0.000231028
6	0.001340866	0.000967026
7	0.009900808	0.004097939
8	0.084067106	0.018296957
9	0.802456141	0.088052988
10	8.469252825	0.440567017
11	98.41906285	2.385438919
12	1239.668924	13.86728287



In exhaustive n-queens, we compute all possible permutations over n which makes it $O(n!)$ time complexity. For backtracking solution, if first queen has n choices to be placed, second has $(n-2)$ since same column and diagonal are not allowed, third has $n-4$ choices and so on. Although this is a little less than exhaustive, we still say backtracking takes $n*(n-2)*(n-4)*(n-6)*...1 = O(n!)$ as a little loose upper bound. When we notice time differences, backtracking solution is indeed better than exhaustive search.

ii. number of queens vs number of possible solutions without considering chessboard reflections and rotations:

Number of Queens	Number of Solutions (same for both e_queens() and b_queens()
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200



2: Formulation of pints problem

We have three containers of sizes $a > b > c$. The action allowed is to pour water from one container to another until either the first one is empty or the second one is full. Our goal is to find if there is a sequence of actions that results in exactly 2 pints in one of the container. To formulate this problem:
1. we may consider $a, b,$ *and* c to be the nodes in a graph and the pouring actions to be the edges with weights of how much water we are pouring from one container to the other. However, at any instance, if a container is already full, it can not have an incoming edge but only outgoing. If we were to do multiple pouring actions to reach our goal state, the edges in the graph would change directions as well as weights weirdly.
2. How if we maintain a state of containers (representing amount of water in that container) after every pouring action? We may have such states as nodes in the graph, pretty much like finite state machines. Of course, at any point we can undo a pouring action we performed and go back to an old state. This implies that our graph will have undirected edges?
3. Let's assume that $a = 5; b = 4; c = 3$ and that initially, the container of size a is empty and the other two b & c are full to their capacities. We can represent this initial state as tuple (0, 4, 3). Now, we have a set of valid actions we can take wherein we make sure that a pouring action doesn't overflow any of the container capacities. From (0, 4, 3), we can have next possible valid action (3, 4, 0), or (4, 0, 3) and so on. To devise an algorithm, we may apply a depth first recursion exploring possible valid states until we finally reach (2, x, x) or (x, 2, x) or (x, x, 2).

3: Design for finding vertex from which all other vertices can be visited

To determine whether or not a graph has a vertex s from which all other vertices are reachable, we would have to apply DFS on all the vertices to check if we can reach all other vertices from that vertex. DFS for one vertex would cost $O(V + E)$, for total V vertices, it would cost $O(V * (V + E))$. To do better,
1. we can start DFS from any vertex and assign pre-visit and post-visit numbers to every visited node wherein, pre-visit number is assigned to a node when it is first visited and post-visit number is assigned after all the node's neighbors (and their descendants) have been explored.
2. According to the algorithm of finding strongly connected components, a node with highest post-visit number is guaranteed to be in source of strongly connected component. If the graph has more than one such vertex from which all other vertices are reachable, the highest post-visit number can be one of the sources.
3. Thus, we first do a DFS traversal of the graph which will take $O(V + E)$. Now if the graph has a vertex from which all other vertices can be reached, it would be the highest post-visit number. We then simply start DFS from this highest post-visited vertex. If by the end of this DFS, all vertices were marked as visited, we have found the vertex of interest. Otherwise, such a vertex doesn't exist in the graph.
4. Thus, we use two DFS runs which makes it $O(2 * (V + E)) = O(V + E)$.

4: Design for finding cycle in a graph

1. Given an undirected graph G and a particular edge $e(u, v)$, edge $e(u, v)$ would be part of a cycle if even after removing the edge from the graph, there still exists a path from vertex u to v.
2. We can thus form a graph $G' = G - e(u, v)$ and apply DFS on it with u as the start vertex. If we can still visit v during this DFS traversal, we know that there's a cycle.
3. Since we are removing an edge from the graph, it would take $O(1)$ whether the graph is in the form of adjacency matrix or adjacency list. DFS takes $O(V + E)$ and thus this would be a linear time algorithm.