B551 Assignment 0: N-queens and Python

-Anup Bharadwaj (anupbhar@indiana.edu)

1. In the given code that solves the n-rooks problem, it takes a value N, constructs a board of N x N size and proceeds to place N rooks in the board such that there are no other pieces in the same row or column.
    - The initial state is an empty board of size N x N
    - The set of valid states would be rooks on the board where no two pieces occupy the same row or column.
    - The successor function could be defined as the next state of the board after making a transaction. That is, add new pieces and form new states. There could be states with no new pieces added too.
    - A goal state could be defined as N rooks placed in a board of size N x N where no two pieces occupy the same row or column.
2. The algorithm is using DFS. By modifying the line of code where the states are inserted into the fringe to insert at the beginning of the list, we could switch the algorithm to follow BFS. Since, BFS is complete we now start getting solutions for up to N = 4 and times out for higher values.
3. While BFS is complete, the computational time rises exponentially with the increase in the value of N. Therefore, it is computationally efficient to use DFS instead, and with the two optimizations included in successors2(), the code will be faster.
4. To optimize the code such that the program can solve nrooks for board sizes with larger values of n, I followed two approaches. While the second approach greatly optimized the solution for nrooks, I found that the first approach was more efficient with the nqueens problem.

    Approach 1: Reduce the rows and columns as it is filled by the pieces and select the state that is the closest to the goal state.

    - Instead of iterating over N every time, we cut off a particular row and column from our iterating list every time we have a piece at a particular position. From there, we select the next best valid state which adds a piece on the board and gets one step closer to our goal state. This is implemented using 'get_available_slots' function which returns the available row and column indexes up for placing a rook on the board. The returned row and column lists are iterated over and the first state that places a new rook on a valid position will be added as the successor and the function is returned.
    - This approach always pushes for the local optimum successor and looks to reach a goal state. While there are several goal states for a given board, this approach pushes to reach the fastest goal state with the given initial board.
    - The initial state is an empty board.
    - The set of valid states and the successors are the next best state where the new piece is placed in such a way that no other rooks are in the same row and column of the new piece.
    - A goal state could be defined as N rooks placed in a board of size N x N where no two pieces occupy the same row or column.

- When run on the burrow machine, the largest value that this program could run was 956.

Approach 2: Using Bitwise operations and local optimum solutions.

- While I was looking into the code, I noticed that every time we had to iterate over the rows and the columns represented as a list of list data structure. The structure of each contents of the inner list looked very similar to a binary number. I started working on this approach with an idea where the data structure can be reduced to a list of integers, take the binary representation of each integer and the position of '1' in the binary representation can be the position of the rook in each row. This could greatly reduce the computational overhead and traverse closer to the goal state. The advantage with this approach would be that we can always ensure there is only one piece in a row by using only powers of 2 integers. A simple count function would tell us the number of rooks in a column. The obvious drawback with this approach would be that the integers could overflow.
- This approach also looks for the locally optimum solution and tries to place the rook in the next valid position such that we are closer to the goal state.
- Implementation: The data structure of the board is now a list of integers. The helper functions such as 'add_piece', 'count_row', 'count_col' etc were rewritten to match the new data structure (Usually named as *_bitwise). The get_next_value function looks for the available integers that can be placed in the given state and the row index and returns it. Thus, the successor state would now be the number added to the row index of the previous state. Using the bit manipulation operators, the list would be loaded with integers (powers of 2) ensuring that the position of the bit of value '1' will not overlap in the row or column. Ensuring the integers are always powers of two will not put more than one rook on the board and ensuring that the list is unique and the power of 2 does not exceed N will ensure the rooks do not overlap in columns.
- The initial state, valid states, successors and the goal state remains the same as approach 1.
- When this program ran on the burrow machine, the largest value where it could successfully compute the solution and print on the screen was 3200. However, it could compute the solution for N = 3300 but timed out while printing the entire solution.

5. I continued to work on both of my approaches I used to solve the nrooks problem and found out that the first approach was able to give the solution to higher N values than approach two. I was expecting the contrary and could not find out the reason behind this behavior.

   Approach 1: Find the local optimum solution for each row as successors and try to traverse towards the goal.

   - The initial state is the empty board of size N x N.
   - Set of valid states are the set of successors with one queen piece on the next row whose position does not intersect with any other pieces on the row, column or diagonal.

- For the nqueens problem, just like the nrooks problem, there are multiple goal states. But unlike nrooks, every starting point will not lead to a solution. Therefore, to accommodate that, when the board is empty, the successor states are one queen piece placed at every column position of row 0. This is to give enough starting points so that eventually one would achieve the goal state if it can be reached. After that, the consecutive successor states would be every valid position in a row where a queen can be placed. This approach tries to find the local optimum which would eventually reach a goal state but would backtrack all the way to find the next successor if the goal can't be reached.
- There could be multiple goal states for a given N value and it can be defined as N queens arranged on a board of size N x N where there are no other queens on the same row, column or diagonal.
- **Implementation:** function no_queen_in_diagonal returns a Boolean indicating if there are any queens along its diagonals for the given row and column. Using this, the function get_next_queen returns a list of possible positions the next queen can be placed on the next row. Each of these possibilities will be a successor state. A couple of minor hacks help in reducing some function calls and aid in increased efficiency. Function is_valid returns a Boolean value indicating if the given row and column does not conflict with the row, column or diagonal of any other piece on the current state of the board. It is ensured that all the successor states contain queens placed at a position that is always valid. This eliminates the need to check for all the diagonal sides when checking if the current state is the goal state. If the successor function was able to place N queens on N x N board, the goal state is achieved.
- When run on the burrow machine, the largest value of N the program could solve within one minute was 31. Sometimes, it was able to solve for 32.

Approach 2: Using bitwise operations and find the locally optimum solutions traversing towards the goal state.

- Maintaining the same logic as approach one, I changed the data structure of the board to use a single list instead of a list of list and perform bitwise operations on them. While this had many advantages which increases the efficiency of computation, it did not fare better than approach 1. Converting the resultant list having the goal state into a print friendly format also became costly.
- When run on the burrow machine, the largest value of N the program could solve within one minute was 31. Sometimes, it was able to solve for 32.

Why is there no changes to 'is_goal' function for solving nqueens?

Since the logic involved checking for diagonal elements and ensuring the successor state does not have more than one queen in a given row, column or diagonal, it can be concluded that the goal state has been reached if the program was able to successfully place N queens on a N x N board.

Notes about the code file:

The submitted nrooks.py takes a command line argument – the N value. If no or invalid value is supplied, it takes a default value of 2. The submitted code works on default implementation with Approach 1 for questions 4 and 5. To run the bitwise implementation. Uncomment the lines which call the respective bitwise functions