Question 1

Spend some time familiarizing yourself with the code. Write down the precise abstraction that the program is using and include it in your report. In other words, what is the set of valid states, the successor function, the cost function, the goal state definition, and the initial state?

Answer

Valid States : The state of the board where the pieces are on the valid locations(all values of tuple (r,c) for 0<=r<N and 0<=c<N) in the board is the valid state.
For example,
The valid states for 2x2 board are as follows:
0 0 , 1 0 , 0 1 , 0 0 , 0 0 , 1 0 , 0 1 , 0 0 , 0 1 , 1 1 , 1 0 , 1 0 , 0 1 , 1 1 , 1 1 , 1 1
0 0 0 0 0 0 1 0 0 1 0 1 0 1 1 1 1 0 0 0 1 0 1 1 1 1 0 1 1 0 1 1

So if the board is of NxN size the total number of valid states will be 2^(NxN)

Successor function: The successor function adds one piece extra that the previous board state to one of the possible tuple value for (r,c) where r and c are the values in the range of 0 to N.

Cost Function: Number of pieces on the board in the current state. So if the solution is found, the board will have N pieces on the board. So the cost of the goal state will be N.

Goal State definition: The goal state here is defined as the one board where

- total number of pieces on the board is N
- each row contains only one or zero rooks
- each column contains only one or zero rooks

Initial state definition: The initial state is defined here as the board with all the zero values indicating that there are no pieces placed initially on the board.

Question 2

You've probably noticed that the code given is implemented by depth first search (DFS). Modify the code to switch to BFS instead. What happens when you run the code for different values of N now, and why?

Answer

Initially the states were added to the end of the fringe (fringe.append(s)).
These states were popped from the end of the fringe (fringe.pop()).
Since the boards were added and removed from the same end it was the stack implementation of the lists. - Last In First Out (LIFO)

The code has been modified so that the fringe now is implemented as a queue.
passing argument 0 to fringe.pop function fringe.pop(0) - gives the queue implementation

Now the states has been added to the end and the states has been removed from the start of the list by fringe.pop(0). This is the queue implementation of the lists. - First in First out (FIFO)

For larger values of N, the DFS(depth first search) is significantly faster than the BFS(breadth first search).
The BFS algorithm takes much longer time than DFS because given the way the fringe is populated. Fringe is populated in such a way that the board with the larger number of pieces are added to the end of the fringe. Now in queue the elements are extracted from the start. As the starting states of the board included lesser number of pieces compared to the ones which were added to the end of the fringe, BFS algorithm processes many more state boards before it reaches the goal state.

Hence it is taking longer time.

Alternative explanation:
Each level contains same number of pieces on the board.
Each next level contains an additional piece compared to the previous one.
In DFS, the search algorithm goes on traversing levels. Hence each next successor board has one more piece on the board and hence will reach the goal state faster.
In BFS, each board is traversed breadth wise. So all the states are traversed.
First level: All the boards with one piece on them.
Second Level: All the boards with 2 pieces on them and so on.
Hence BFS will take longer time.

Question 3
Recall from class that there are usually many ways to define a state space and successor function, and some are more practical than others (e.g. some have much larger search spaces). The successor function in the code is defined in a very simplistic way, including generating states that have N+1 rooks on them, and allowing \moves" that involve not adding a rook at all. Create a new successors() function called successors2() that fixes these two problems. Now does the choice of BFS or DFS matter? Why or why not?

Answer
The two problems have been resolved with these conditions:

- Add successor only when count_pieces(board) < N - (ensuring no N+1 rooks on any successor)
- add piece on board only if board[r][c] == 0 - (ensuring the piece is not added where it already is added)

The choice of BFS/DFS still matters because still the successors2() function is adding the successors in the fringe the same way the successor() function was doing.
Here also BFS will be significantly slower than DFS for larger values of N.
Hence DFS will be a prefered choice for successors2() function as well.

Question 4

Even with the modifications so far, N=8 is still very slow. Instead of allowing the successor function to place a piece anywhere on the board, let's define a successor that is much more orderly: it's only allowed to add a piece to the leftmost column of the board that is currently empty. (For example, if the board is empty, the rook has to be placed in the first column; if there are 4 rooks on the board, the next one has to go in the fifth column, etc.) Modify the code to implement this alternative abstraction with a successors function called successors3(). Feel free to make other code improvements as well. What is the largest value of N your new version can run on within about 1 minute? Tip: In Linux, you can use the timeout command to kill a program after a specified time period: timeout 1m ./nrooks.py N

Answer

I have written a successor function in which I have added a piece to the topmost empty row. I am also making sure that while adding the piece to the board, I am not adding it to the column where the piece already exists in some other row. I am using my pre knowledge to avoid adding successor states that won't lead to goal state. My successors3() function is running for N=400 in 58 sec.(DFS)

Extra:

I have also written successors4() function which will only add the diagonal elements to the board.(which is one of the solutions)

```
def successors4(board):
  if count_pieces(board) < N:
    return [ add_piece(board, r, r) for r in range(0, count_pieces(board) + 1) if board[r][r] == 0 ]
```

The is_goal function for the same:

```
  def is_goal_faster(board):
    return count_pieces(board) == N and \
        ( sum(board[r][r] for r in range(0,N)) == N or
            sum(board[r][r] for r in range(0,N)) == N )
```

The successors4() function will run much faster as only the states which lead to only one solution are added to the fringe.

(As we were asked to find a solution and not all the solutions.)

By doing so, I am able to find solution for N=1000 under 12 seconds.

Question 5

Now, create a new program, a0.py, with several additional features, including the ability to solve both the N-rooks and N-queens problems, and the ability to solve these problems when some of the squares of the board are not available (meaning that you cannot place a piece on them). Your program must accept at least 3 arguments. The first one is either nrook or nqueen, which signifies the problem type. The second is the number of rooks or queens (i.e., N). The third argument represents the number of unavailable positions. The remaining arguments encode which positions are unavailable, using row-column coordinates and assuming a coordinate system where (1,1) is at the top-left of the board.

Answer

First argument - nrook / nqueen / nknight
Second argument - N
Third argument - Number of restricted positions
Fourth and beyond arguments - list of restricted positions

I have kept the default implementation of the algorithm as DFS.

e.g
a0.py nrook 7 1 1 1