1. **Valid States:** The Valid states would be the set of all the squares on the chess board where the squares would be either one or zero based on whether the rook is present in the square or not respectively and the total number of the valid states would be 2^(N*N).

   Successor Function: The successor function would be a function that adds 1 rook to each of the element in the fringe i.e. a state of the chess one by one to each square, resulting in successive boards.

   Cost Function: The cost function would be uniform because all the squares would be processed one by one and will take equal time to get processed.

   Goal State: The goal state would be a state where N rooks will be placed on the board such that no rook can attack each other i.e. once a rook is placed on the board, there will be no other rook which will lie on its row or column.

   Initial State: The initial state would be a state where the board will be empty i.e. all the squares would be set to a 0.

2. The algorithm of N-rooks uses DFS as the fringe is set to be a stack i.e. the fringe gets popped and pushed from the same side. However, if we change the code a little bit as follows, the fringe becomes a queue and the whole algorithm changes to become a BFS.

   I Method

```
def solve(initial_board):
    fringe = [initial_board]
    while len(fringe) > 0:
        for s in successors( fringe.pop() ):
            if is_goal(s):
                return(s)
            #fringe.append(s)#part of original code
            fringe.insert(0,s)#this is the change
    return False
```

   II Method

```
def solve(initial_board):
    fringe = [initial_board]
    while len(fringe) > 0:
        for s in successors( fringe.pop(0) ):#this is the change
            if is_goal(s):
                return(s)
            fringe.append(s)
            return False
```

   Either of the methods can be used to implement a BFS, however, DFS is more efficient in terms of speed than BFS but the problem in DFS is that using the current algorithm, it enters an infinite loop for the values N>=3. BFS on the contrary gives the desired result till N=5.

What Happens now for N=4 or N=8??

The program if implemented using DFS, does not work for N>2, however, if we change the implementation to BFS, it gives us results for N<=5. The higher values of N are still slow (including N=8).

3. Currently, the nrooks.py runs only till N=2 with DFS(keeping the timeout after 1 min in consideration).

   Following is the Successors2 function:

```
def successors(board):
  piece_count = count_pieces(board)
  successor_boards=[]
  for c in range(0,N):
    for r in range(0,N):
      if piece_count < N :
        temp=add_piece(board, r, c)
        if count_pieces(temp)==piece_count+1:
          successor_boards.append(temp)
  return successor_boards
```

   After this change, the program runs till N=5 for both BFS and DFS within 1 min. Hence solving the infinite loop problem in DFS.
   0.013693 min for DFS
   0.567422 minutes for BFS
   Thus, now DFS outperforms BFS.

   Does the choice of BFS or DFS matter? Why and Why not?
   Yes, the choice of implementing BFS or DFS will matter. Even though BFS is complete, efficiency of the program suffers while we use BFS. This is because when we use BFS, the algorithm searches the solution one node after another in a sequential manner. The algorithm basically searches all the possible board settings with zero rooks, one rook, two rooks and so on whereas the solution will always be found on the maximum depth of the tree, unless it is a worst case scenario or there is no solution for that particular N. DFS is likely to find the solution sooner in such cases as the algorithm penetrates to the depths first and then backtracks to the sibling nodes. However, if the solution does not exist, both the algorithm will perform equally in terms of running time as their time complexity is same. We will still save the memory using DFS.

   BFS
   Time Complexity: b^d
   Space Complexity: b^d

   DFS
   Time Complexity: b^d
   Space Complexity: b*d

   Where b is the branching factor of the search tree and d is the depth of the search tree.

4. The current code for N=4 is still slow because of the large fringe. If we design our fringe in such a way so as to filter the unwanted boards, our program will run more efficiently. In the current fringe we append the possible boards just by checking whether the number of pieces is less than or equal to N. However, if we add an additional check before adding a board to the fringe to check if the board does not have a two or more rooks in a row or column, we would get a goal closer to our goal state and will thus, considerably reduce the size of our fringe.

New Abstraction

Valid States: (same) The Valid states would be the set of all the squares on the chess board where the squares would be either one or zero based on whether the rook is present in the square or not respectively and the total number of the valid states would be 2^(N*N).

Successor Function: The successor function would be a function that :

- adds 1 rook to each of the square in the board (passed as a parameter)
- checks whether the total rooks on the board is less than N
- checks whether there is more than one rook in a row/column.

After checking the above conditions, the successor function would add the resultant boards to the fringe resulting in successive boards with number of rooks increased by one from the parent state or the parent board.

Cost Function: (same)The cost function would be uniform because all the squares would be processed one by one and will take equal time to get processed.

Goal State: (same)The goal state would be a state where N rooks will be placed on the board such that no rook can attack each other i.e. once a rook is placed on the board, there will be no other rook which will lie on its row or column.

Initial State: (same)The initial state would be a state where the board will be empty i.e. all the squares would be set to a 0.

The new modified successor function would be:

```
def successors3(board):
    piece_count = count_pieces(board)
    successor_boards = []
    for c in range(0, N):
        for r in range(0, N):
            if piece_count < N and count_on_row(board, r) < 1 and count_on_col(board, c) < 1:
                temp = add_piece(board, r, c)
                if count_pieces(temp) == piece_count + 1:
                    successor_boards.append(temp)
    return successor_boards
```

With this new successor function, the program of N rooks for N=100 runs in 1.032528 minutes. Hence, N<100 will run within a minute.

5. The largest value of N is less than 60 that my new version can solve for queens as N=60 takes 1.040213 minutes. Please find the code attached for the rest of the question.