Question 1:

1. Initial state = Empty board of N*N size
2. Set of valid state = {s| s is a state with all rooks placed on the board are at valid position (i.e. position should be available on the board)}
   Size of set of valid states will be 2^(n^2)
3. Goal state = {s| s is a state where no more than one rook is placed in same column or row and number of rooks placed = N}
4. Successor(s) = {s1| s1 is the state with one additional rook placed at valid position in state s}
5. Cost function = No. of conflicts occurred after moving from source state to destination state + 1 (+1 for just understanding that cost of path from initial state to goal state is >0)

Note: There can be multiple possible ways to define cost function. I have defined a one from the all possible ways.

Question 2:

The given code is implemented by DFS. In given implementation, new successors are inserted and popped from the same side in fringe. i.e. fringe uses stack implementation of list.

To convert this to BFS, we must convert the logic of fringe implementation from stack to queue. For this, we should insert and pop the successors from different end. There are 2 possible ways to convert the given starter code from DFS to BFS. either use fringe.insert(0, s) instead of using fringe.append(s) or use fringe.pop(0) instead of using fringe.pop().

With the given code, for N=2, DFS and BFS both are working fine. But in case of n>=3, DFS is looping into an infinite loop while BFS is working fine and able to find a solution. Because, given code puts piece at every position even if piece already exists at any point. Means there are some cases where it is not placing any piece at all i.e., it creates same state again and pushes into fringe. Hence, DFS will try to find successors of same state again and again, hence goes to infinite loop. But in case of BFS, it will try to find successors of each state which are in fringe and push. Hence, it can find solution instead of going into the loop.

For larger values of N, DFS will work faster than BFS. Because, DFS hunts for goal node by digging a complete branch at a time, so depth of the goal node will not impact as much in case of DFS. But in case of DFS, it visits every node on a same level before moving to the next level. Hence, depth of the goal node will impact much more which makes BFS slower than DFS.

Question 3:

Yes, even after fixing the mentioned two problems in question 3, choice of DFS or BFS implementation will matter.

DFS will be faster than BFS. Because in case of DFS, it will dig in a branch till end and if it found any solution there then it will stop execution. Even if it does not find any solution in the branch will back-track one level at a time and investigate other branches. But, in case of BFS, it will find all successors of current state. Then, it will all successors of first successors and add to the fringe and will do the same for 2nd, 3rd and all 1st level states and then it will go for next level. So, even if the goal state in 3rd level which can be

searched by DFS in no time, BFS will visit all states of level 2 first and then will go to level 3. That means BFS will take more than DFS.

There is a special case like when goal state in bottom right most node of the tree, then DFS and BFS both will take same time.

Question 4:

To optimize the code to the next level, I have created a new abstraction. As per the new abstraction, the states which have more than one rook in a same column or a row are not considered as successor of previous state, because there will not be any profit in expanding these branches further. To implement this, I have written a new successor function successor3(). This is trying to place rooks row by row. Means before placing any rook in a row, it will not create new states which tries to place rook in next row. This reduces the no. of successors of each state. In addition to this, I am calling add_piece() only if current row don't has any rook and excluding those columns of current row which already has rook. Finally, the rate of creation of new state has been reduced by a big margin, which saves the time of iterating and searching those avoided branches of the tree. Again, I am using DFS for this question for faster execution of search operation.

Now with the optimized code (with successor3()), it can handle N = 370 within 56 secs.

Question 5:

As asked in question 5, written a code which works for both nqeen and nrook as the input provided from command line. To implement both, made some changes in successor() which checks conditions depends on type (i.e. nrook, nqueen) and calls add_piece() accordingly. Written a new function count_on_diag() to check no of queens placed on diagonal of the cell. If there is no queen placed on diagonal and row or column, then only it will call add_piece() to place Queen in that position.

To implement unavailable positions, all unavailable positions are stored in a list and checked for not availability of a cell in successor() before calling add_piece(). Hence, this will restrict program to create a state with queen/rook on unavailable position.

While printing the board, used Q indicate in case of nqueen and R to indicate nrook.

To implement nknight, a new count_on_knight_pos() is written to check conflicts in placing knight at any position. Made some changes in successor() to implement nrook/nqueen/nknight. In all this, I have implemented DFS. This works good for N = 35 for nrook within 1 min, N = 25 for nqueen within 1 min and N = 25 for nknight within 1 min.

Note: I have temporarily used character 'R' to display in case of nknights, because the code-test python script allows only R, Q, X, _ characters in display format.