ASSIGNMENT 0

PART 1

Abstraction for N-rooks Problem (As per given program)

Suppose we create an abstraction for 2- Rooks problem.

1. Set of States

   For N= 2, there are 16 set of states. These are as follows:-

   [[0,0],[0,0]] [[1,0],[0,0]] [[0,1],[0,0]] [[0,0],[1,0]]
   [[0,0],[0,1]] [[1,1],[0,0]] [[1,0],[1,0]] [[0,0],[1,1]]
   [[0,1],[0,1]] [[1,0],[0,1]] [[1,1],[1,0]] [[0,1],[1,1]]
   [[1,0],[1,1]] [[1,1],[0,1]] [[0,1],[1,0]] [[1,1],[1,1]]

   There are four places and 2 possible values for each place (0 and 1). Therefore,

   Number of possible states = 2 * 2 * 2 * 2

   = 16

   Similarly, for N= 3 problem, set of states = 2^9 = 512

2. Initial State

   S_0 = state when there are no rooks on board
   Therefore S_0 = [[0,0],[0,0]]

3. Successor function S(x)

   S (S_0) = {[[1,0],[0,0]] , [[0,1],[0,0]] , [[0,0],[1,0]] , [[0,0],[0,1]]}

   Successor function is given by adding one rook to the board. It adds up to 4 rooks to the board.

   S([[0,1],[0,0]]) = {[[1,1],[0,0]] , [[0,1],[0,0]], [[0,1],[1,0]], [[0,1],[0,1]]}
   S([[1,0],[0,0]]) = {[[1,0],[0,0]] , [[1,1],[0,0]], [[1,0],[1,0]], [[1,0],[0,1]]}
   S([[0,0],[1,0]]) = {[[1,0],[1,0]] , [[0,1],[1,0]], [[0,0],[1,0]], [[0,0],[1,1]]}
   S([[0,0],[0,1]]) = {[[1,0],[0,1]] , [[0,1],[0,1]], [[0,0],[1,1]], [[0,0],[0,1]]}

   S([[1,0],[1,0]]) = {[[1,0],[1,0]] , [[1,1],[1,0]], [[1,0],[1,0]], [[1,0],[1,1]]}

   In this way, we can obtain successor function for each state.

Every state has 2^2 i.e. 4 successor states.

4. Set Of Goal States

Goal State is reached when there is only one rook in whole row and only one rook in whole column.

For N= 2, there are two goal states possible.

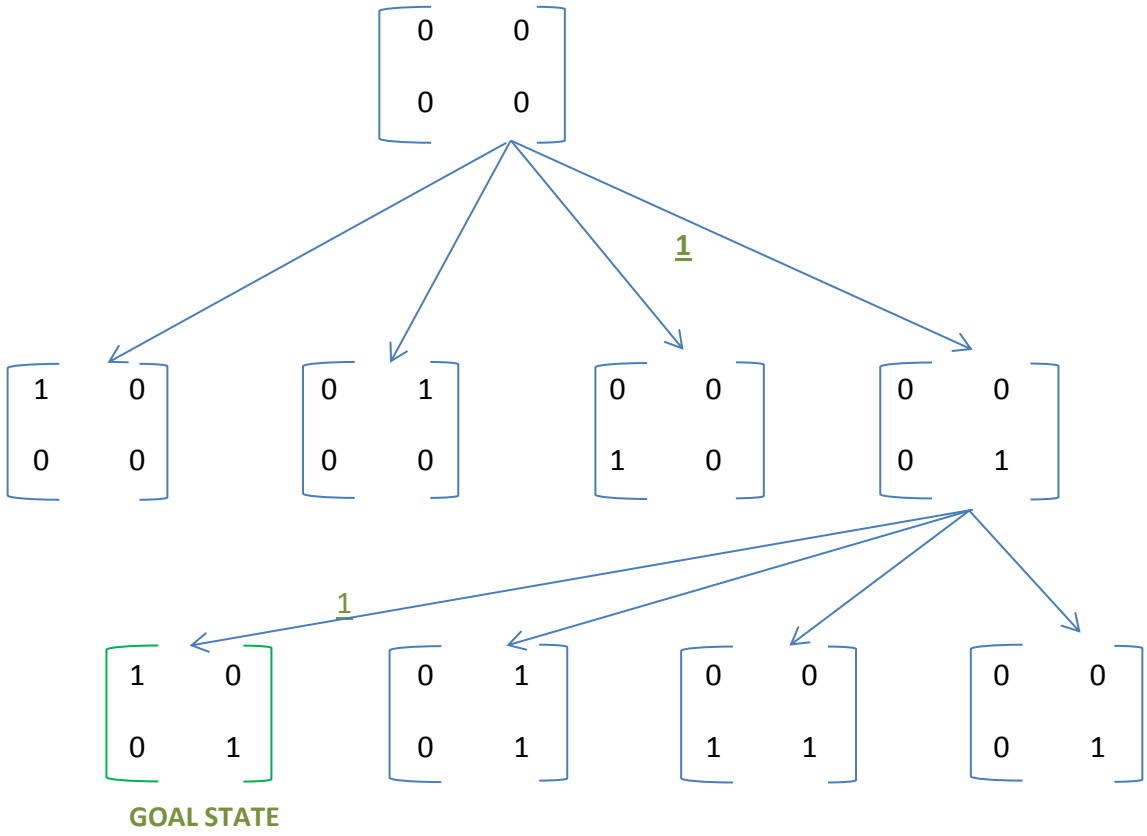
G = [[1,0],[0,1]] , [[0,1],[1,0]]

5. Cost function

Suppose cost to travel from one state to another is 1.

Therefore cost (S_0, S_1) = 1

Cost function is the number of moves required to travel from initial to goal state.



Therefore, In this case,

Cost to reach goal state is 2.

PART 2

The given program is not at all efficient. It runs for N=2 but returns a memory error for N=3.

Please look at screenshot below:-

```
[guptaapu@silo ~]$ emacs nrooks-2.py
[guptaapu@silo ~]$ time ./nrooks-2.py 2
Starting from initial board:
_ _
_ _

Looking for solution...

R _
_ R

real    0m0.017s
user    0m0.013s
sys     0m0.002s
[guptaapu@silo ~]$ time ./nrooks-2.py 3
Starting from initial board:
_ _ _
_ _ _
_ _ _

Looking for solution...

Traceback (most recent call last):
  File "./nrooks-2.py", line 61, in <module>
    solution = solve(initial_board)
  File "./nrooks-2.py", line 48, in solve
    for s in successors2( fringe.pop() ):
  File "./nrooks-2.py", line 37, in successors2
    return [ add_piece(board, r, c) for r in range(0, N) for c in range(0,N) ]
  File "./nrooks-2.py", line 29, in add_piece
    return board[0:row] + [board[row][0:col] + [1,] + board[row][col+1:]] + board[row+1:]
MemoryError
```

The successor function in code returns a lot of states which are not going to lead us to the goal state.

Example – [[1, 1], [1, 0]]

S([[1,0],[1,0]]) = {[[1,0],[1,0]] , [[1,1],[1,0]], [[1,0],[1,0]], [[1,0],[1,1]]}

All the states returned by successor function of [[1, 0], [1, 0]] are invalid states.

Eliminating these states can help us to reduce the size of state space graph.

A better successor function can be created by doing four things:-

1. Eliminate all the states which have N>2. The number of hooks can never be greater than 2. Thus, it is useless exploring those states further. We can reduce the sample space to a great extent.
   Example:
   S**([[1,0],[1,0]])** = **{[[1,0],[1,0]]** , [[1,1],[1,0]], **[[1,0],[1,0]],** [[1,0],[1,1]]**}**

2. The successor function returns the same state again. I.e. Successor function of [[1, 0], [1, 0]] returns the same state again. Further finding successor function of this state will repeat this in loop.
   Example:
   S**([[1,0],[1,0]])** = **{[[1,0],[1,0]]** , [[1,1],[1,0]], **[[1,0],[1,0]]**, [[1,0],[1,1]]**}**

   These states should be removed from successor function.

3. We need to check if a single row contains more than 1 queen. If yes, we can eliminate those states in successor function. This helps us by not expanding the states which are not going to lead us to goal state.

   Example:
   S(**[[1,0],[0,0]])** = {[[1,0],[0,0]] , **[[1,1],[0,0]],** [[1,0],[1,0]], [[1,0],[0,1]]}

   Expanding state [[1, 1] ,[0, 0]] is useless. Therefore, we will eliminate this state in successor function.

4. We need to check if a single column contains more than 1 queen. If they contain more than 1 queen, such states have to be eliminated and should not be expanded.

   Example:
   S(**[[1,0],[0,0]])** = {[[1,0],[0,0]] , [[1,1],[0,0]], **[[1,0],[1,0]],** [[1,0],[0,1]]}

   [[1, 0], [1, 0]] this state has 2 queens in same column. We should avoid expanding such a state.

   Therefore, we will put these four conditions in our code.

MODIFICATIONS IN EXISTING CODE

Existing successor function

```
# Get list of successors of given board state
def successors(board):
    return [ add_piece(board, r, c) for r in range(0, N) for c in range(0,N) ]
```

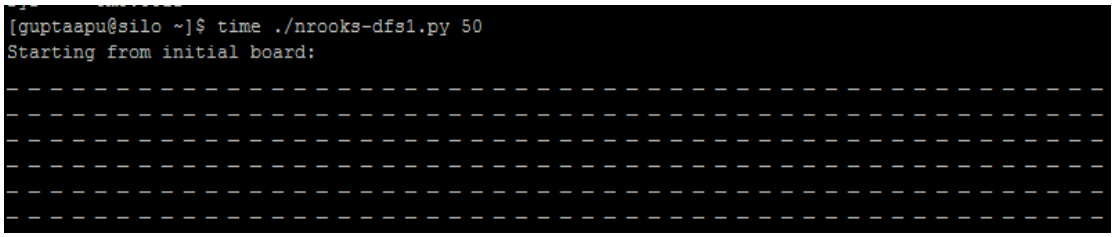
New successor function

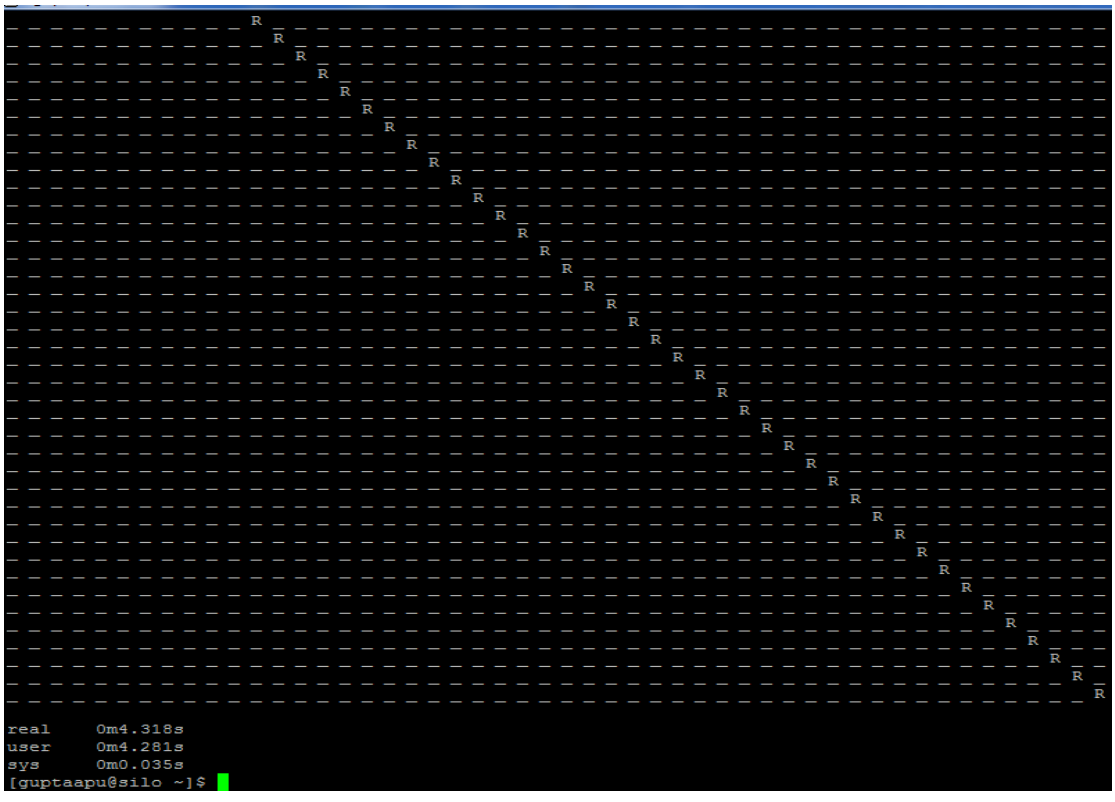
```
def successors2(board):
    new_board = []
    for r in range(0,N):
        for c in range(0,N):
            state = add_piece(board,r,c)
            if state !=board and count_pieces(state) <= N and count_on_row(state,r)<=1 and count_on_col(state,c)<=1:
                new_board.append(state)
    return new_board
```

This new successor function improves the performance drastically.

PERFORMANCE MEASURE OF NEW SUCCESSSOR FUNCTION

- The initial code gave a memory error for N=3.
- The new code runs in 4.318 seconds for N=50 which is far better from its initial performance.



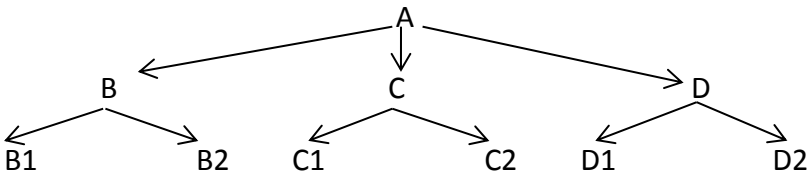


Therefore, the new successor function is very efficient.

PART 3

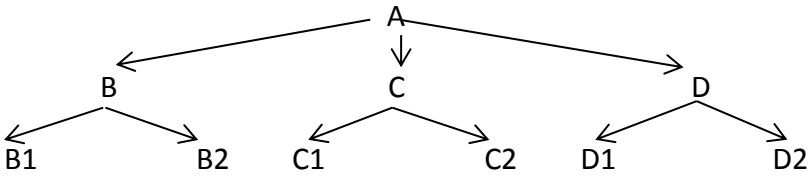
Converting DFS code to BFS

Depth First Search



In the above state space, depth first search will first search A and will give B, C and D successor nodes. After that, it pops out the last element D and adds successor nodes at the end of list i.e. D1 and D2.It then checks D1 and D2 and adds its successor elements.

Breadth First Search



In the above state space, Breadth first search will first search A and will give B, C and D successor nodes. Unlike depth first search (which pops out last element), it now pops out the first element i.e. B and checks for B1 and B2. If B1 and B2 are not goal nodes, it further adds its successors and check for goal.

Thus, to convert depth first search to Breadth first search, I popped out the first element instead of last. Rest code remains same.

MODIFICATIONS IN EXISTING CODE

DFS code

```
# Solve n-rooks!
def solve(initial_board):
    fringe = [initial_board]
    while len(fringe) > 0:
        for s in successors2( fringe.pop() ):
                if is_goal(s):
                    return(s)
                fringe.append(s)
    return False
```

BFS Code

```
# Solve n-rooks!
def solve(initial_board):
    fringe = [initial_board]
    while len(fringe) > 0:
        for s in successors2(fringe.pop(0) ):
            if is_goal(s):
                return(s)
            fringe.append(s)

    return False
```

- **Fringe.pop ()** pops the last element. Instead of that, we use **fringe.pop (0)** to pop the first element.
- Rest code remains same.

PART 4

Time required for Depth First Search

I ran the algorithm 1000 times for more accurate results.

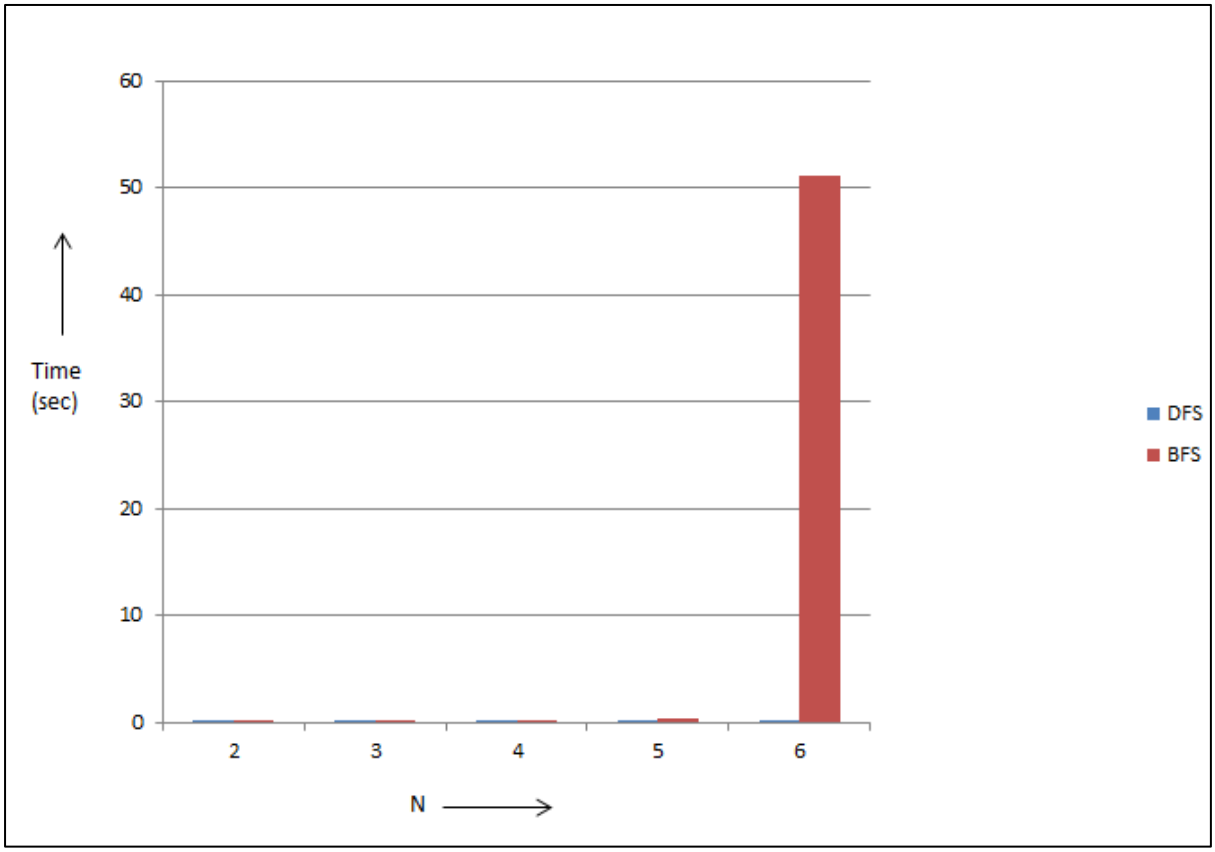
N	User Time (1000 times)	User Time (1 time)
2	8.773	8.773/1000 = 0.008773
3	8.687	8.687/1000 = 0.008687
4	8.651	8.651/1000 = 0.008651
5	9.139	9.139/1000 = 0.009139
6	9.605	9.605/1000 = 0.009605

Time required for Breadth First Search

I ran the algorithm 10 times for more accurate results.

N	User Time(10 times)	User Time(1 time)
2	0.083 sec	0.083/10 = 0.0083
3	0.094 sec	0.094/10 = 0.0094
4	0.168 sec	0.168/10 = 0.0168
5	3.360 sec	3.360/10 = 0.3360
6	8min 32.126 sec = 512.126 sec	512.126/10 = 51.2126

For comparison, I have plotted a column chart.



OBSERVATION

The running time for DFS and BFS is very small for values less than 6. At N=6, running time of BFS increases drastically as compared to DFS:-

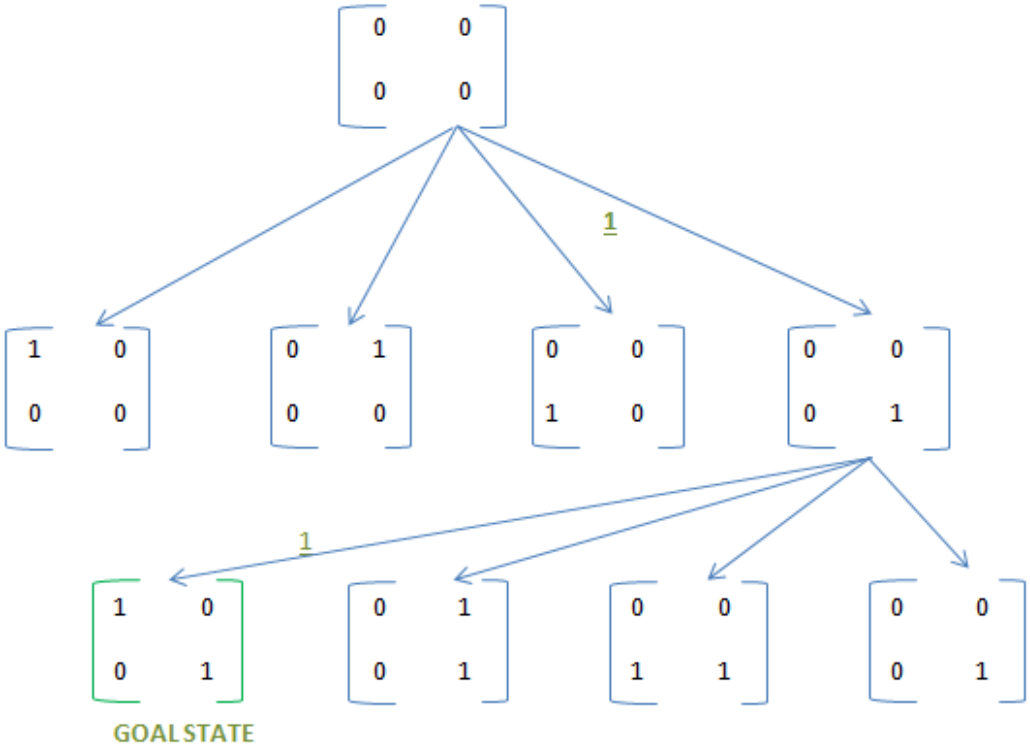
To find out the reason behind this, we need to understand the basic working of DFS and BFS.

1. Breadth first search is efficient when we get the values at upper levels i.e. at level 1 or level 2 in state space graph. But as the value of N increases, the solution will be obtained much deeper inside the state space rather than at upper levels.
2. Depth first search goes deep inside the state space graph and finds the solution. In case of N-Rooks problem, solution will be found at the leaf of the tree.

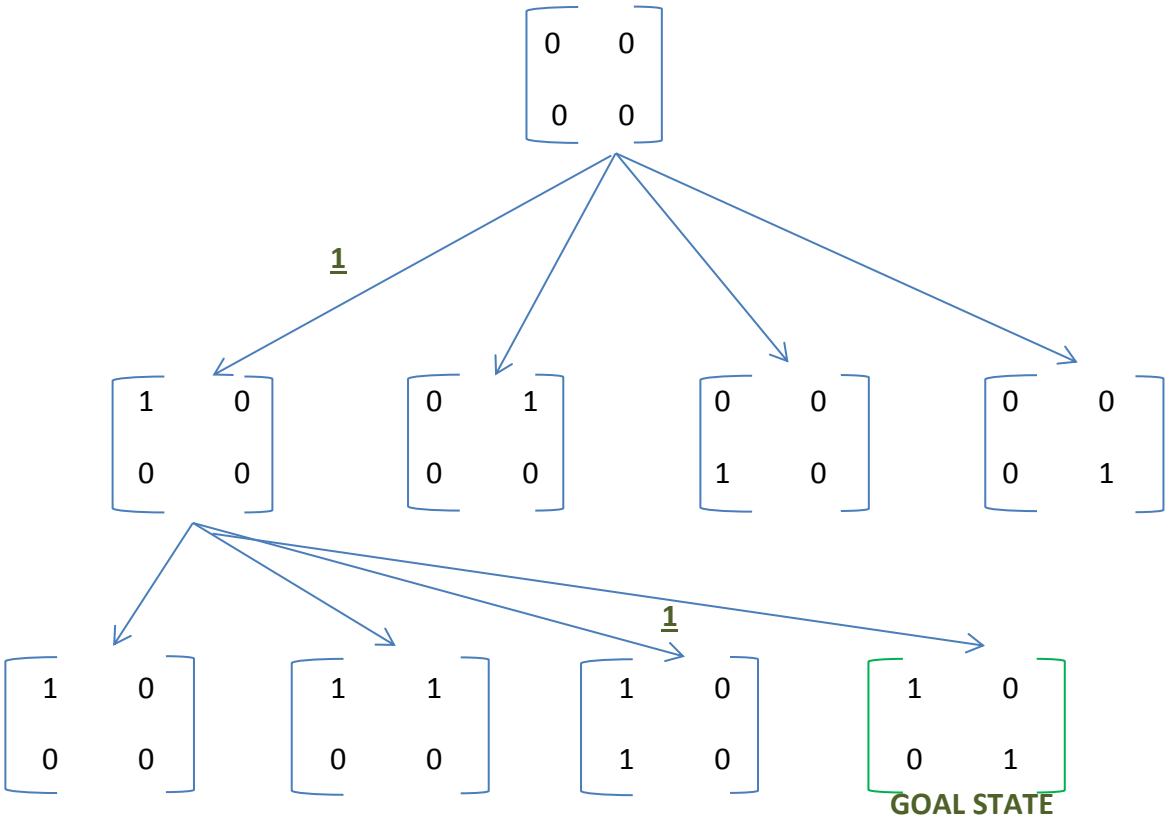
   Therefore, DFS performs better in case of N-Rooks and N-Queen problem.

   Here is an example for N=2.

Traversing by depth first search, solution will be obtained in two moves.



Traversing by breadth first search also finds the solution in two moves.



This proves that for N=2 and some small values, there won't be much difference between DFS and BFS. But, as the values increase, DFS performs much faster than BFS.