### Question 1

The abstraction that nrooks-2.py is using can be defined according to the five parameters we learned about in lecture.

The **set of valid states** is all the chess boards that have between 0 and N rooks.

Our **successor function** is adding one additional rook to the board compared to the previous board.

The **cost function** is the number of moves, which in this case, is irrelevant. Every move costs the same.

Our **goal state** is defined as an NxN board, which has N rooks on it, where no rook is in danger from another rook of being eaten.

The **Initial state** is an empty board that is N x N chessboard, where N is provided by the user at runtime.

### Question 2

Things started to get interesting at this point. I modified the code by changing line 49 by adding a zero to the pop() method:

for s in successors( fringe.pop(0))

This changed how the code goes through the fringe, by switching from DFS to BFS. pop() with no index, takes the last element first (Last In First Out – LIFO). By putting an index of 0, and calling pop(0), we are taking the first element. This converts us from a stack (DFS) to a queue (BFS) that is now First in First Out (FIFO).

This remarkably speed up the program. Under DFS, the code could only solve as high as N=2 in a reasonable amount of time. Now, under BFS, the code can solve N=4 in less than 0.1 seconds and N=5 in under five seconds. N=6 or greater still takes a very long amount of time to solve.

In reading the python documentation on Data Structures[[1]](#footnote-1), I noticed something interesting about the .pop() method. The documentation states that using lists as queues is highly inefficient, since it causes every other has to be shifted over. It recommends using collections.deque and the .popleft() method to implement a queue. I added the following to my code at the beginning:

from collections import deque

and then changed the assignment statement for the fringe to:

fringe = deque([initial\_board])

and then changed my successors for loop to read:

for s in successors( fringe.popleft())

this also had a noticeable change on the performance of the code. Below is the approximate order of magnitude of the results:

|  |  |  |
| --- | --- | --- |
| N | Nrooks w/ pop(0) | Nrooks w/ popleft() |
| 4 | 0.04 s | 0.04 s |
| 5 | ~0.46 s | ~0.20 s |
| 6 | > 10 min | > 10 min |

The result at N=5 indicates that there might be some improved performance by using this form since it was a magnitude of two faster. It was not enough of an improvement to make it complete N=6 in less than 10 minutes, which was the time limit I set to understand the limits of the program.

### Question 3

The new successors2 function incorporates two changes. First, I added an if statement which checks the count\_pieces function to make sure the code does not create a board with N+1 pieces. Secondly, within the loop generating the successors it checks to make sure it does not place a rook (assign it to value of 1) to a spot on the board already occupied by a rook, effectively creating a successor with no move.

Also, the python documentation on Data Structures also mentioned that using collection.deque is more efficient for pop from the left or right of the list. So, I also compared results for BFS and DFS with and without the use of that library.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | BFS | | DFS | |
| .pop(0) | .popleft() w/deque | .pop() | .pop() (deque) |
| 3 | 0.02 s | 0.02 s | 0.02 s | 0.03 s |
| 4 | 0.03 s | 0.03 s | 0.02 s | 0.03 s |
| 5 | 3 s | 1.6 s s | 0.3 s | 0.3 s |
| 6 | **>10 min** | **> 10 min** | 41 s | 41 s |

Overall, the algorithm is much faster now for both DFS and BFS. The use of collections.deque varies. For BFS it is much faster to use popleft() from collections.deque. With, DFS there was no noteworthy difference as N increased. With successors2, DFS is faster than BFS now.

### Question 4

We now have a new successor function that only allows us to place a rook in the first lefthand column without a rook. Starting with the successors2 function of:

def successors2(board):

if count\_pieces(board) < N:

return [add\_piece(board, r, c) for r in range(0, N) for c in range(0,N) if board[r][c]!=1]

else:

return[]

and I focused on the portion that iterates through creating the column:

This, currently iterates through every column. I want to restrict to only one column. To do so, I use the count\_pieces function again. I’ll insert that into the function and assign it to a new variable called total\_pieces. This will tell me the number of pieces on the board. Once I know the number of pieces, I know what column. If there are 0 pieces, then it should go in the first column. If there are 3 pieces, the next rook should go in the 4th column. I achieve this by simplifying how to add pieces. Specifically, I removed

for c in range(0,N)

from the statement, change the c to total\_pieces in the rest of the line. Which results in a successor3 of:

def successors3(board):

total\_pieces = count\_pieces(board)

if total\_pieces < N:

return [add\_piece(board, r, total\_pieces) for r in range(0, N) if board[r][total\_pieces]!=1]

else:

return[]

For this one, I went with the DFS implementation, since that was the fastest prior. With a time limit of 1 minute, the DFS solution could find a value for N=9 consistently in about 55 seconds on the soic servers. Under BFS, it could only find N-7 under a minute.

### Question 5

I began by creating a successor4 function that would be used for the nrooks case. We know that rooks can be in danger if they are in the same column or row of another rook. Since the column has already been limited, the first improvement is to limit the row. This is done by using the count\_on\_row function that was provided in the skeleton code. This function is added to a second condition to the end of the if statement that was created in successor3. The successor4 function for nrooks now looks like:

def successors4(board):

total\_pieces = count\_pieces(board)

if total\_pieces < N:

return [add\_piece(board, r, total\_pieces) for r in range(0, N) if board[r][total\_pieces]!=1 and count\_on\_row(board,r)==0]

else:

return[]

Now, it will only place a rook on that row if it knows there are no other rows on that row. This results in about nrooks running to over 300 in under 30 seconds.

Looking towards nqueens, before creating an nqueens successor function, I created a function that would count the pieces in a diagonal from a particular position. Since our pieces are going from left to right, I only have to worry about diagonal pieces to the left of the position, with one diagonal go up to the left and one going down to the left. If on the top or bottom edge, there will only be diagonal. There will be no diagonal is placing in the first column on the left. This resulted in the following function:

# Count # of pieces in the diagonals from a particular spot on the board

def count\_on\_diagonals(board,row,col):

diags = 0

# Opportunity to streamline code in future below

for colpos in range(col-1,-1,-1):

for rowpos in range(row+colpos-col,row+colpos-col-1,-1):

if rowpos >= 0 and board[rowpos][colpos]==1:

diags += board[rowpos][colpos]

for rowpos in range(row-colpos+col,row-colpos+col+1):

if rowpos <= N-1 and board[rowpos][colpos]==1:

diags += board[rowpos][colpos]

return diags

Admittingly, this code could likely be refactored to something more streamlined. The important thing to note is to show how it handles the diagonals and uses logic to cut off when the diagonals would continue off the board.

Once, that was completed, I added I built on successors4() and created the following:

def successors5(board):

total\_pieces = count\_pieces(board)

if total\_pieces < N:

return [add\_piece(board, r, total\_pieces) for r in range(0, N) if board[r][total\_pieces] != 1 and count\_on\_row(board,r)==0 and count\_on\_diagonals(board,r,total\_pieces)==0]

else:

return[]

Similarly to how I added the count\_on\_row function and made that a condition of adding a piece, I then added an additional condition of count\_on\_diagonals==0. With this new function completed, I was now ready to fully incorporate the nqueens.

Next, I set-up the program to handle the nqueens details. First, added a line to read in the nrook or nqueen

# This will tell us whether to run nrook or nqueen. It is passed through command line arguments

ntype = str(sys.argv[1])

and then renumbered the input for N to:

# This is N, the size of the board. It is passed through command line arguments.

N = int(sys.argv[2])

Next, need to accommodate the number of spots where a piece could not be placed. It takes the command input, then cycles through and places them on a second mirror board called blocked\_board. Since, much of the other logic depends on counting the number of pieces in rows, columns, and the board, I decided it would be better to save that information on a second board and reference it later.

# Initialize a mirror blocked board. This way it doesn't mess with counting functions used elsewhere.

blocked\_board = [[0]\*N]\*N

# This will find the slots that cannot be utilized, and place a 2 in them.

total\_blocked = int(sys.argv[3])

if total\_blocked > 0:

for blocked in range(4,4+2\*total\_blocked,2):

blockedrow = int(sys.argv[blocked]) - 1

blockedcol = int(sys.argv[blocked+1]) -1

blocked\_board = block\_spot(blocked\_board,blockedrow,blockedcol)

Where block\_spot is”

# Blocks the placement on the board at a given position, and returns a new board (doesn't change original)

def block\_spot(board, row, col):

return board[0:row] + [board[row][0:col] + [2,] + board[row][col+1:]] + board[row+1:]

Then, I updated successor4 and successor5 to include another condition to add\_piece to see if there was a blocked spot where add\_piece wants to add a piece. This results in successors4 and successors5 looking like:

# NRooks Successor

def successors4(board):

total\_pieces = count\_pieces(board)

if total\_pieces < N:

return [add\_piece(board, r, total\_pieces) for r in range(0, N) if board[r][total\_pieces] != 1 and count\_on\_row(board,r)==0 and blocked\_board[r][total\_pieces] != 2]

else:

return[]

# NQueens Successor

def successors5(board):

total\_pieces = count\_pieces(board)

if total\_pieces < N:

return [add\_piece(board, r, total\_pieces) for r in range(0, N) if board[r][total\_pieces] != 1 and count\_on\_row(board,r)==0 and count\_on\_diagonals(board,r,total\_pieces)==0 and blocked\_board[r][total\_pieces] != 2]

else:

return[]

Next, I made solve\_rooks and solve\_queens functions, with the only real difference being which successor function is called.

# Solve n-rooks!

def solve\_rooks(initial\_board):

fringe = [initial\_board]

while len(fringe) > 0:

for s in successors4( fringe.pop() ):

if is\_goal(s):

return(s)

fringe.append(s)

return False

# Solve n-queens!

def solve\_queens(initial\_board):

fringe = [initial\_board]

while len(fringe) > 0:

for s in successors5( fringe.pop() ):

if is\_goal(s):

return(s)

fringe.append(s)

return False

Next, once a final solution has been created, I combined the solution board with the blocked\_board for a final board for display.

from operator import add

printable\_solution = list(map(add, solution[r], blocked\_board[r]) for r in range(0,N))

I learned how to add elements together from two lists via a helpful online article[[2]](#footnote-2), which I built since the board was a list of list.

Last, I created a function to output for nrooks and nqueens in a printable format.

# Return a string with the board rendered in a human-friendly format

def printable\_rooks\_board(board):

return "\n".join([ " ".join([ "R" if col==1 else "X" if col==2 else "\_" for col in row ]) for row in board])

def printable\_queens\_board(board):

return "\n".join([ " ".join([ "Q" if col==1 else "X" if col==2 else "\_" for col in row ]) for row in board])

And, then finally, hooray, it worked!!

In under 30 seconds, nrook will work for N = 300 and for nqueen will work for N = 25.

### Extra Credit

To do the nkight, I took similar steps in creating it that I did when I made the nqueens options.

First, I created a function to check for Ls, since knights can only be attacked in Ls that are two squares up and then one square to the left of the right. I kept the assumption, that I would only put them in the left-hand most column.

# Count number of piece in L-shapes that could attack another knight

def count\_on\_els(board,row,col):

els = 0

# Opportunity to streamline code in future below

rowpos = [row-2,row-1,row+1,row+2]

colpos = [col-1,col-2,col-2,col-1]

els = sum([board[rowpos[x]][colpos[x]] for x in range(0,4) if rowpos[x] >=0 and rowpos[x] <= N-1 and colpos[x] >=0 and colpos[x] <= N-1])

return els

Then, I created a successor function:

# NKnights Successor

def successors6(board):

total\_pieces = count\_pieces(board)

if total\_pieces < N:

return [add\_piece(board, r, total\_pieces) for r in range(0, N) if board[r][total\_pieces] != 1 and count\_on\_els(board,r,total\_pieces) == 0 and blocked\_board[r][total\_pieces] != 2]

else:

return[]

Then, I created a function to output it and for the program to handle the nrooks. Option. Right now, Nrooks can run N=9 in under 30 seconds.

#### Thanks, that was fun!!

1. <https://docs.python.org/2/tutorial/datastructures.html> [↑](#footnote-ref-1)
2. <https://stackoverflow.com/questions/18713321/element-wise-addition-of-2-lists/18713494> [↑](#footnote-ref-2)