Elements of AI - Assignment 0

Soln 1. The abstraction used in the program with N×N chessboard and using N pieces has been summarized as follows:-

i) Initial state :-

In the beginning the initial board is a matrix of 0's (representation of empty squares). Thus, the initial state could be said to be an empty chessboard with none of the pieces placed on it. Eg. for a board, N=2, the initial state is a list of lists : [[0,0],[0,0]] shown as:

_ _

_ _

ii) Goal state :-

The goal for the problem is the state with all the N pieces placed at such a position that none of the pieces attack one another. For the n rooks problem, it'll such that none of the n rooks are in the same row and column. There can be multiple solutions / goal states. One eg :- for N=2.

R _

_ R

iii) Successor Function: The sw

The successor function generates the list of next valid states and appends them to the fringe. In the original program, it just adds a piece to all the squares on the board that is passed to it. If the new board isn't a goal state then it is appended to the fringe. A point to note is that if the square already has a piece we still go through the process of putting a piece there and return technically the same board. The Depth first search paradigm has been used to implement the ~~fringe~~ search with the help of stacks data structure.

Eg. lets assume a board of _ R and the
_ _
successor function will give :-

R R ; _ R ; _ R ; _ R
_ _ _ _ R _ _ R

this will give

R R ; R R ; R R ; R R
_ _ _ _ R _ _ R

3 more can also exist on a 2×2 board with this function.

iv) Cost function :-

The cost function represents the cost of going from one state to another state. Since we are just placing a rook at a square (if it already has one, we still go through the entire process), the cost function for going from one state to the next is a constant factor of 1. (The cost of placing a rook anywhere on the board is the same)

v) Valid states :-

A valid state can be said to be state that is possible / attainable by the successor function. A valid state for this problem could be a state with rooks placed randomly anywhere on it. The total number of rooks can exceed N also. We also don't take into account the attacking constraint while generating valid states.

eg of valid states : _ _ ; R _ ; _ _ R _

R R . R _ . R R , etc.
R _ ' _ R ' _ _

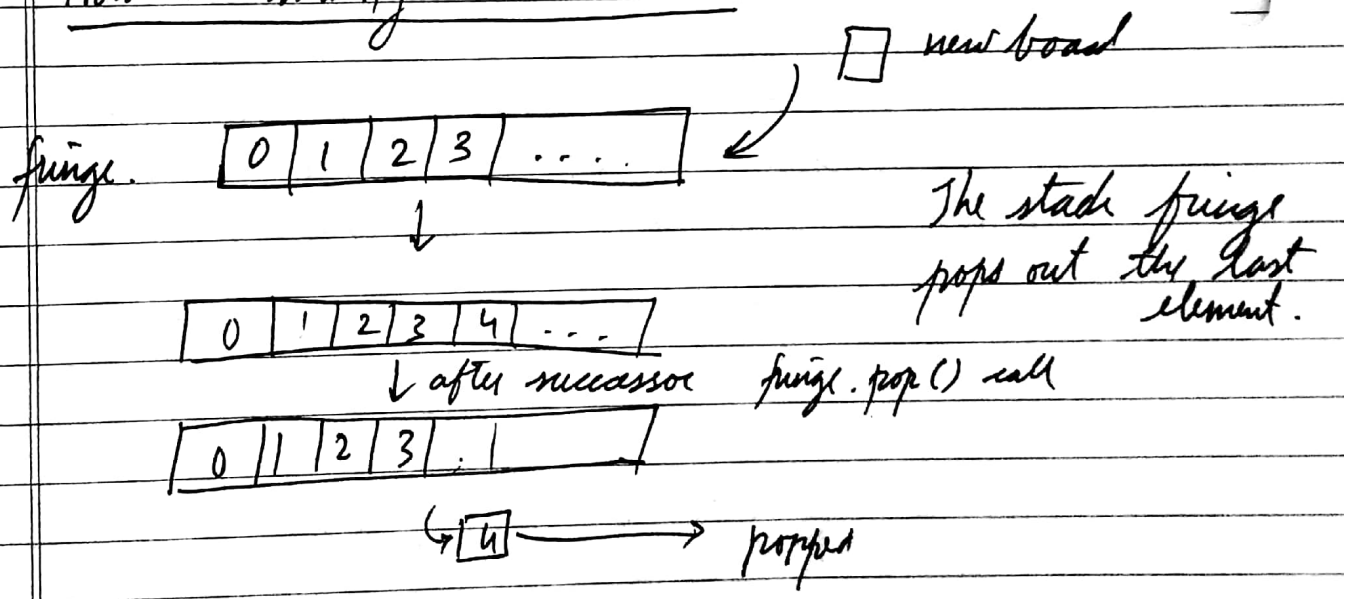
Soln 2. The code is converted to BFS by using (fringe.pop(0)) which pops the elements from the beginning of the fringe ie. first in first out (FIFO) paradigm.

A tabular description of N versus run-time.

Type \ N	1	2	3	4
DFS	runs	runs	no output	no output
BFS	runs	runs	runs	runs.

For N > 2 DFS doesn't give output whereas BFS gives an output quickly. DFS goes on an infinite loop checking the same board again & again.

How to modify to BFS?

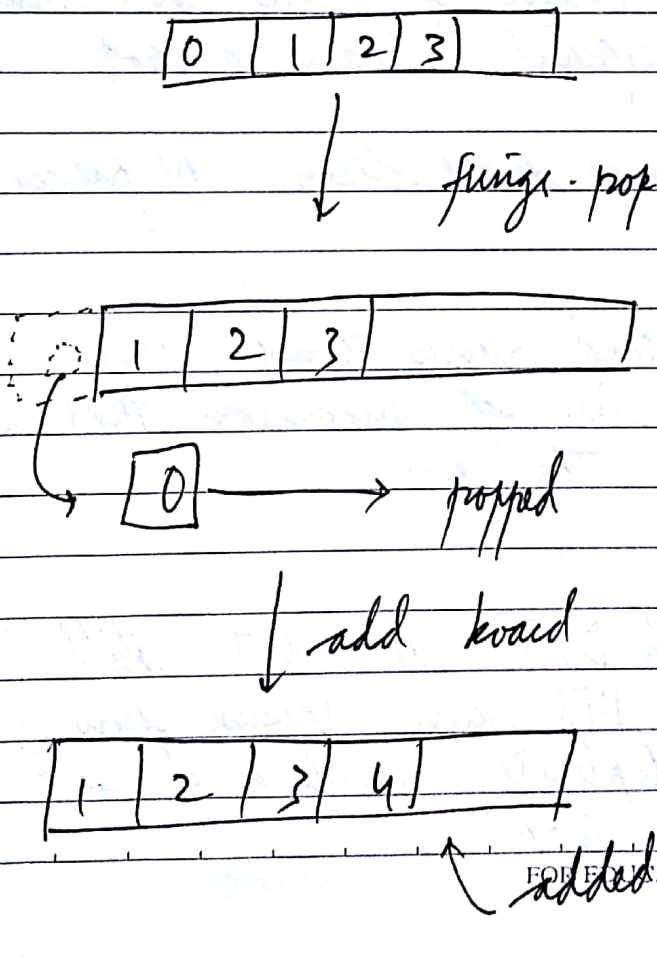


So correct to a BFS we must remove the board that came first into the list. We can do this by using:

fringe.pop(0)

↑ specify the location to pop from.

Now all the boards will be taken from the 0th position and we get a BFS implementation instead of a DFS implementation.



Soln 3. Successors 2 is implemented as follows (given in code file). The following logic is added:

successors2

successors list

→ Only append to fringe, if new_board generated from add_piece is not same as original board.

append

→ Only return the board to successors if the total number of pieces on board are less than equal to N.

⇒ The add_piece function returns a board with a piece at (r, c) passed to it.

- We must now check if this new board is equal to original board or not and
- check if there are more than N pieces on the new board.

If the new_board passes these tests we append it to a list of successors that is returned to solve at the end.

⇒ Now both BFS & DFS run but still they both are very slow. DFS runs because now fringe doesn't have duplicate boards in it and don't go on infinite loop.

Soln 4. Successors 3 has been implemented in code file

The logic used has been :-

- We specify a list empty_s [ ] with [1, 2, 3 ... , N]
- Now iterate over the all the squares in the board passed to successors. If the square has a piece, remove the row from empty_s [ ] ie. if board [3][4] has a piece (=1) then remove "3" from empty_s.
- Now specify a variable "col" equal to the number of pieces on the board. ie. it'll be "4" if board already has 4 rooks placed & so on.
- Now simply iterate over all the elements in empty_s [ ] place a piece at the values of empty_s [ ] & col. (fixed).
- Use the limiting condition defined in successors2 as is and append to successors function if it is valid.
- Return successors list.
- Fringe uses DFS (BFS is very slow)

Tabular representation of my successor 3 with time:-

N	Time taken in seconds
5	0.00014
20	0.00454
30	0.0162
40	0.0437
50	0.095
100	1.214
150	5.214
200	15.683
250	37.542
300	75.367

The program runs for N = 280 in 58 seconds. So, I guess N=280 will be the limit for running under 60 seconds.