# CSCI B-551 Elements of Artificial Intelligence

*Prof. David J Crandall*

Assignment - 0

Submission Date: 9/10/2017

Name: Sagar Suresh Panchal

IU Email: **spanchal@iu.edu**

1. Spend some time familiarizing yourself with the code. Write down the precise abstraction that the program is using and include it in your report. In other words, what is the set of valid states, the successor function, the cost function, the goal state definition, and the initial state?

**A. Set of valid states:**

- The set of valid states consists of any configuration of 0 to N rooks on an N*N board.
- Also, there should be only one rook present in a single block.

B. **Initial State**:

- This consists of an empty board of size N*N which does not include any rooks on it.

**C. Successor function:**

- Successor function will generate successors for the current state by adding a new rook on the board for the specified block.

**D. Goal State:**

- A state can be called as a goal state if and only if the below conditions are fulfilled:
    - There should be no more than 'N' rooks placed on the board.
    - No rook should be able to attack another rook, which means that there should be only one rook on a single row and column.

2. The current code is far from efficient. Recall from class that there are usually many ways to define a state space and successor function, and some are more practical than others (e.g. some have much larger search spaces). Create a more efficient successor function called successors2(). Hint: avoid generating states that have N+1 rooks on them or allowing "moves" that involve not adding a rook at all.

Ans: Below is the snippet of the successors2() function:

```
def successors2(board):
    add_piece_list = []

    for row in range(N):
        for col in range(N):
            if sum(board[row]) >= 1:
                continue
            elif count_on_col(board, col) >= 1:
                continue
            elif board[row][col] == 1:
                continue
            else:
                add_piece_list.append(board[0:row] +
[board[row][0:col] + [1,] + board[row][col+1:]] + board[row+1:])

    return (add_piece_list)
```

a. An add piece list is created which will store the successors generated for the current state through the add piece function.

b. The conditions placed inside the successor function are as follows:
    i. Check if there is any rook already present at the current row or column; if yes, it won't allow to add a new rook in the same row. And will continue the loop to check for the next positions.
    ii. If there is a rook already present at the current block, it won't allow the function to add one more in the same block.

3. You've probably noticed that the code given is implemented by depth first search (DFS). Modify the code to switch to BFS instead.

Ans: To switch the code to Breadth First Strategy (BFS), we must implement a queue which will add an element from one side and remove from the other end. Below is the code change performed for the same (highlighted):

```
def solve_using_bfs(initial_board):
    fringe = [initial_board]

    while len(fringe) > 0:
        for s in successors2(fringe.pop(0)):[1]
            if is_goal(s):
                return(s)
            fringe.append(s)
    return False
```

Here, an index value is given to the pop() function, which is 0 (as per the above code); hence, it will remove the 1st element from the list from the front and will add append a new element in the list at the end.

4. Run your program and measure the time required for both BFS and DFS for various values of N (e.g. from 2 to 6). (On a Linux machine, you can use the time command.) Draw a figure or create a table showing how running time varies with N. Explain the behavior you notice – why are the two similar or different?

Ans: Running time analysis for BFS and DFS for n-rooks is shown below in a tabular format as follows:

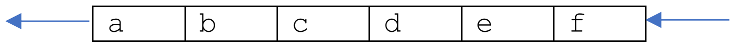
| Value of N | BFS (secs) | DFS (secs) |
|---|---|---|
| 2 | 0.0082 | 0.0082 |
| 3 | 0.0083 | 0.0081 |
| 4 | 0.013 | 0.0092 |
| 5 | 0.1342 | 0.0085 |
| 6 | 44.736 | 0.0083 |

Table 1: BFS vs DFS Running time analysis

Both the algorithms have different running time, the reason for which is as follows:

**For BFS:**

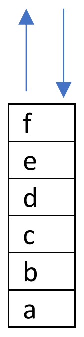
- BFS is implemented using a 'Queue' data structure, it checks for every element from start till end as it is implemented using First in First Out (FIFO) approach[2].
- Below is the example of a queue:



- Hence, with respect to the N-rooks problem, 1st element of the queue will be the "Initial Board", which is removed from the queue through the front end and then passed to the successor function for generating its successors.
- Those successors are checked to be the goal state, if they are not the goal state, then they are appended to the queue from the rear end.
- Now the queue contains the list of all the possible successors of the initial board and hence will loop through each of those successors to find out if they are the goal state or not.
- This process will continue until the goal state is found.
- As it checks each element of the queue, it takes a substantial amount of time to iterate through the entire queue for higher values of N.
- Hence, as per the Table 1, for N = 2 to 5 the running time was fast but when the value of N = 6, the execution time increased exponentially from 0.1342 secs to 44.736 secs.

**For DFS:**

- DFS is implemented using a 'Stack' data structure, in here, the elements in the list are added and removed from the same side. Hence, it is implement using a Last in First Out approach[2].
- Below is the example of Stack:



- With respect to the N-rooks problem, the 1st element of the stack will be the initial board, which is added initially and then removed from the same and passed to the successor function to generate its successors.
- Those successors are checked to be the goal state, if they are not the goal state, they are appended to the stack.
- Now the stack contains the list of all the possible successors of the initial board and hence will pop the topmost element of the stack to find out if it is a goal state or not.
- After checking, if it is not the goal state, it will find the successors for that state and append them in the stack.
- This process will continue until the goal state is found.
- As DFS checks only the topmost element of the stack, it takes a very less amount of time to execute and find the solution.
- Hence, as per the Table 1, DFS executes extremely faster even for N = 6 when compared with BFS.

**References:**

1. #https://stackoverflow.com/questions/4426663/how-do-i-remove-the-first-item-from-a-python-list

2. Russel & Norvig - Artificial Intelligence A Modern Approach (3rd Edition)