# CS250-AI Lab

# Assignment-1: BFS, DFS

### By- Archit Sharma (2101AI05)

***Submitted files- ass1.py, ass1-test.py, explanation-docs.docx***

## Code Explanation-

### Imported functions and libraries-

1. **shuffle function from random library-** This function is imported to shuffle the board to generate a random starting state. One of the 9! states is generated.
2. **deepcopy from copy library-** Used to copy 2d arrays, as by default ‘.copy’ attribute of python copies arrays by reference.
3. **defaultdict from collections library-** defaultdict acts as the map which is used for lookup both by dfs and bfs. We use this over the default dict offered by python as this gives the option to give out a default value even when the key has not been added yet.
4. **perf\_counter from time library-** Used to measure the time taken by dfs and bfs in seconds.

### Functions implemented-

1. **hash(arr)-** Since list object in python are not hashable, they cannot be used as keys for defaultdict. So, given a 2d 3x3 array, this function converts and returns a string which is hashable and can be used for lookup by bfs and dfs. This is done in O(1).
2. **printBoard(arr)-** Given a 2d 3x3 array, a grid representation is printed. As we are representing the board by using integers where 0 represents blank space, it is printed as ‘B’. Again, it is done in O(1).
3. **moveUp(currentBoard, i, j)-** Given a state ‘currentBoard’ and row (i) and column (j) of the position of blank piece, if it is possible to move the piece upward, exchange the top and blank piece and return the board; else return an empty list. **Time complexity: O(1).**
4. **moveDown(currentBoard, i, j)**- Given a state ‘currentBoard’ and row (i) and column (j) of the position of blank piece, if it is possible to move the piece downward, exchange the bottom and blank piece and return the board; else return an empty list. **Time complexity: O(1).**
5. **moveLeft(currentBoard, i, j)**- Given a state ‘currentBoard’ and row (i) and column (j) of the position of blank piece, if it is possible to move the piece left, exchange the left and blank piece and return the board; else return an empty list. **Time complexity: O(1).**
6. **moveRight(currentBoard, i, j)**- Given a state ‘currentBoard’ and row (i) and column (j) of the position of blank piece, if it is possible to move the piece rightward, exchange the right and blank pieces and return the board; else return an empty list. **Time complexity: O(1).**
7. **getPossibleStates(currentBoard)-** Given a state ‘currentBoard’. It first finds the location of the blank piece and then using the four previous functions, it checks all possible actions that could be taken by it and then returns all the possible future states in a list.
8. **bfs(initial, final)-** For initial and final state, an iterative bfs algorithm has been implemented which uses list object as a FIFO data structure, \_bfs object stores the number of states that have been compared to the final state. statesTaken is a defaulDict which uses the hashed lists for lookup. Also, for user interface, it prints the initial state that has been passed to it, \_bfs is printed in real time and on reaching the final state, it prints “BFS has checked: {\_bfs} states”. First we insert the intitial state into the queue. Then untill the queue is empy, do the following- pop out the list at 0th index, if it has been taken (use statesTaken) , continue; else set it to be visited and increase \_bfs by 1, if this visited list is equal to the final state, break the loop, else for all the possible states that could be reached from this state, add them to the queue. On termination return the last state examined and \_bfs. **Time Complexity: O(bd+1 ), Space Complexity: O(bd) where b is the branch factor (4 here) and d is the depth of the shallowest solution.**
9. **dfs(initial, final)-** For initial and final state, an iterative dfs algorithm has been implemented which uses list object as a LIFO data structure, \_dfs object stores the number of states that have been compared to the final state. statesTaken is a defaulDict which uses the hashed lists for lookup. Also, for user interface, it prints the initial state that has been passed to it, \_bfs is printed in real time and on reaching the final state, it prints “DFS has checked: {\_bfs} states”. First we insert the intitial state into the queue. Then untill the queue is empy, do the following- pop out the list at the last position, if it has been taken (use statesTaken) , continue; else set it to be visited and increase \_dfs by 1, if this visited list is equal to the final state, break the loop, else for all the possible states that could be reached from this state, add them to the queue. On termination return the last state examined and \_dfs. **Time Complexity: O(bm+1 ), Space Complexity: O(bm) where b is the branch factor (4 here) and m is the depth of the deepest solution.**
10. **auxBFS(initial, final)-** This function is implemented so as to call the bfs function for the initial and final states and upon termination print appropriate message if the final state has been reached or not.
11. **auxDFS(initial, final)-** This function is implemented so as to call the dfs function for the initial and final states and upon termination print appropriate message if the final state has been reached or not.

### Main Body-

In the main body od the code, we initialize the final\_state that we have to reach and also generate the random initial state by calling the shuffle function. The initial\_state and final\_state is initally a 9x1 1d vector but it is converted to a 3x3 vector using two nested-for loops. We then call auxBFS() and auxDFS() functions to initialize the search and while doing so we use the perf\_counter() to store the time taken for the search.

Finally we print the time taken by each search, upto 3 places of decimal. Emperically, it was found that DFS was significantly faster whenever the solution was further from the intial state.

## Answers to the Questions

#### Compare Breadth First Search(BFS) and Depth First Search(DFS) with respect to the number of steps required to reach the solution, if they are reachable.

Ans: The number of steps taken to reach the solution depends on the initial state generated. For this three cases can emerge-

1. The solution is shallow- In this case BFS is going to find the answer earlier and in less number of steps as it expands and searches for breadth-first while DFS algorithm may start exploring deep into branches which do not have solution. eg.- [[1,2,3],[4,5,6],[7,0,8]] was solved by BFS in 3 steps, while DFS took 42 steps to reach the answer.
2. The solution is deep- In cases where the nearest solution is deep in the generated graph, DFS shines over BFS as it searches depth first and hence converges faster, while BFS expands and searches breadth-wises and so consumes both time and memory to reach the solution. eg.- For [[4,8,2],[0,1,5],[3,7,6]] was solved by BFS in 59816 steps, while DFS did so in 5256 states..

#### Comment on which algorithm will be faster and when, by mentioning proper intuition and examples.

Ans: Since the time complexity of BFS is O(bd+1) and DFS is O(bm+1) where d and m are depth of the shallowest solution and m is the depth of the deepest solution, we can intuitively say that in cases where a solution is available at a shallower level, BFS does a better job. While in places where the solution is available at deeper level, DFS converges faster. Our hypothesis is verified emperically-

eg.- [[1,2,3],[4,5,6],[7,0,8]] was solved by BFS in 3 steps, while DFS took 42 steps to reach the answer. In terms of time BFS reached solution in 0.0 seconds and DFS in 0.004 seconds, the time are approximated to 3 places of decimal.

eg.- For [[4,8,2],[0,1,5],[3,7,6]] was solved by BFS in 59816 steps, while DFS did so in 5256 states., BFS took 5.471s while DFS did so in 0.481s.

The speed of algorithm can be described in two ways-

1. **Number of states checked per unit time**- For this test to be fair, the number of states checked should be equal so, I checked the time for the state that is impossible to reach, i.e. - [[1,2,3],[4,5,6],[8,7,0]]. Each function did so by scouring through 181440 states. BFS did it in 17.72s and DFS in 14.975. This translates to BFS scanning 10239.278 states per seconds and DFS 12116.194 states per second. I hypothesize that this difference is due to the fact that queue in BFS grows exponentially with each depth, while for DFS it is linear with depth hence pop attribute for lists in python works faster for BFS.
2. **Emperically Faster-** Since we are not aware of the distribution of the times the solution is shallow or deep or impossible to reach. I ran the algorithms 50 times to know how many times which algorithm is faster and the average time per algorithm for a general case. For this I modified the main ass1.py file as ass1-test.py file. The following was observed-

* Average time for BFS: 5.149s
* Average time for DFS: 4.971s
* Times when BFS converged faster: 15/50 = 30%
* Times when DFS converged faster: 35/40: 70%

This also in the line with the intuitive explanation that since the number of nodes grow exponentially with depth, if the solution is assumed to be equally likely placed, it has higher odds of beeing deeper than being shallower.

Both of our metric suggest that DFS is faster both asymptoticaly and behavorially for graph search.