DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PRINCIPLES OF AI LAB

2K22 AFI Batch

LAB FILE

SUBMITTED TO:
Gull Kaur Ma'am

SUBMITTED BY:
Shikhar Asthana
(Roll No. 2K22/AFI/24)

EXPERIMENT 4

AIM: Consider the 8-Puzzle problem. Implement a solution for the problem using the following paradigms: Breadth First Search, Depth First Search and Iterative Deeping Depth First Search. Use tkinter to design a UI capable of showing the tiles and moves taken. Compare all three different approaches on time complexity, space complexity, completeness, and optimality.

THEORY: The 8-Puzzle problem states that given a 3x3 board of tiles, have numbers 1-8 and one empty tile (signified in red color), we need to arrange them in the goal state configuration i.e., a sorted series where tile at (0,0) is 1, (0,1) is 2, (0,2) is 3, (1,0) is 4 and so on. The only tile which can be moved or swapped is the red colored empty tile. And the movement is only considered valid if it is left, right, up or down. In Artificial Intelligence, this problem can be formulated as a search problem where the given 8-puzzle is the starting state and we need to search for the final state using different searching paradigms.

These search paradigms can be briefly described as follows:

- Breadth First Search (BFS) – This search goes in a level-by-level manner where the nodes in the upper levels are explored first and later levels are explored later. Generally, this finds a goal state nearest to the source/starting state.
- Depth First Search (DFS) – This search goes till the deepest possible node (leaf node) of the state space tree. In this paradigm, there is a possibility that successive states might actually be taking us further away from the goal state.
- Iterative Deepening Depth First Search (IDDFS) – This can be considered as a combination of BFS and DFS where we first limit the depth of the state space tree and then apply DFS on it. If the goal state is not reached, we can increase the depth and try again. This continues till we reach a maximum depth (which can vary based on the problem properties).

ALGORITHM:

Approach 1 – BFS

1. Q ← Initialize a queue
2. Push starting state of 8-puzzle into Q

3. While Q is not empty
    a. Pop a state from Q
    b. If state == goal state:
        i. Solution found – Break
    c. Else:
        i. Find neighboring 8-puzzle states
        ii. Push states into Q one by one

Approach 2 – DFS

1. S ← Initialize a stack
2. Push starting state of 8-puzzle into Q

3. While S is not empty
    a. Pop a state from S
    b. If state == goal state:
        i. Solution found – Break
    c. Else:
        i. Find neighboring 8-puzzle states
        ii. Push states into Q one by one

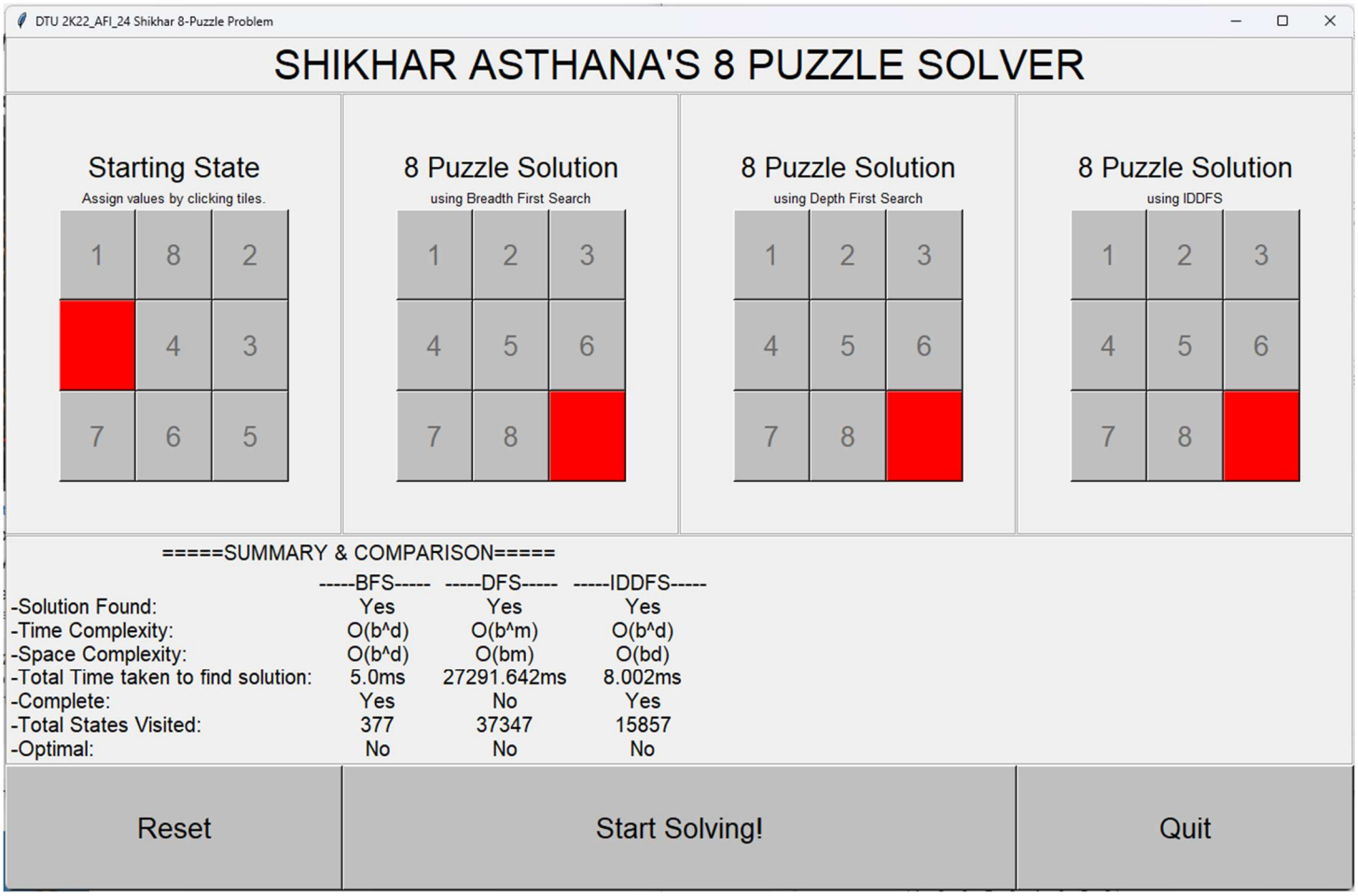
Approach 3 – IDDFS

1. Run a for loop where i from 0 to max depth:
        i. IF DFS (start state, goal state, i):
            1. Return True – Search Successful
        ii. Else:
            1. Return False - Search failed
2. DFS(state, goal state, max depth)
    a. IF state == goal state:
        i. Return True
    b. IF max depth <= 0
        i. Return False
    c. For nstate in neighbours of state:
        i. IF DFS(nstate, goal state, max depth -1)
            1. Return True

ii. Else:
    1. Return False

CODE: Implemented code is submitted over the google classroom assignment.

OUTPUT:



As evident from the output parameters, for the above shown input state, the performance of the approaches can be arranged as follows:

BFS > IDDFS > DFS

As shown in the above screenshot, I have taken into consideration the following parameters:

- Solution Found – Whether the approach was able to find the solution or not
- Time Complexity – The time complexity in terms of max branching factor (b), depth of least cost solution (d), maximum depth of state space tree.
- Space Complexity – The space complexity in terms of max branching factor (b), depth of least cost solution (d), maximum depth of state space tree.
- Total Time Taken to find solution – The time difference based on my system clock in ending the algorithm and starting of the algorithm – in ms.
- Complete – If a solution exists, can the paradigm find the solution.
- Total States visited – The total number of states visited by the algorithm in attempting to find the goal state.
- Optimal – Is this approach optimal.

Summary & Comparison			
Parameter	BFS	DFS	IDDFS
Solution Found	Yes	Yes	Yes
Time Complexity	$O(b^d)$	$O(b^m)$	$O(b^d)$
Space Complexity	$O(b^d)$	O(bm)	O(bd)
Total Time Taken to Find the Solution	5.0ms	27291.642ms	8.002ms
Complete	Yes	No	Yes
Total States Visited	377	37347	15857
Optimal	No	No	No

EXPERIMENT 5

AIM: Consider the 8-Puzzle problem. Implement a solution for the problem using the following paradigms: A*, Iterative Deeping A* and Branch & Bound. Use tkinter to design a UI capable of showing the tiles and moves taken. Compare all three different approaches on time complexity, space complexity, completeness, and optimality. Compare the results with the uninformed search techniques of Experiment 4.

THEORY: The 8-Puzzle problem states that given a 3x3 board of tiles, have numbers 1-8 and one empty tile (signified in red color), we need to arrange them in the goal state configuration i.e., a sorted series where tile at (0,0) is 1, (0,1) is 2, (0,2) is 3, (1,0) is 4 and so on. The only tile which can be moved or swapped is the red colored empty tile. And the movement is only considered valid if it is left, right, up or down. In Artificial Intelligence, this problem can be formulated as a search problem where the given 8-puzzle is the starting state and we need to search for the final state using different searching paradigms.

These search paradigms can be briefly described as follows:

- A* – A* is one of the most popular informed search algorithms. It uses the information available from a heuristic function along with the current path cost to compute the best estimated path to the destination/goal.
- Iterative Deepening A* – This is very similar to A* and the IDDFS algorithm we saw in Experiment 4, with a major difference that instead of height of the tree (or size of the frontier), we apply a limit on the maximum function value, i.e f(n). In case a solution is not found, we increment the f(n) value and check again.
- Branch & Bound – The speed of search for an answer node can often be increased by using an "intelligent" ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an answer node. It is similar to the backtracking technique but uses a BFS-like search. We calculate the cost function of each node – the path till the node + any heuristic function value. Out of all possible cases, we choose the least total cost node (cost lower bound).

ALGORITHM:

Approach 1 – A*

1. Create Fringe
2. h_start ← Heuristic(start_state)
3. f_val ← h_start + 0
4. Fringe ← Add(start_state, 0, h_start, f_val)
5. While Fringe not empty:
    a. Find Minimum element from Fringe
    b. If Minimum element == Goal_State
        i. Add state to visited_states
        ii. Return True
    c. If Minimum element != Goal_State
        i. Add state to visited_states
        ii. Find neighboring states of current state
        iii. For each neighboring state not in in visited set:
            1. Find h_val
            2. g_val ← g_val of current state + 1
            3. Fringe ← Add(current_state, 0, h_start, f_val)

Approach 2 – IDA*

1. idastar_algo(start_state, Goal_state, Threshold_val)
    a. Create Fringe
    b. h_start ← Heuristic(start_state)
    c. f_val ← h_start + 0
    d. Fringe ← Add(start_state, 0, h_start, f_val)
    e. While Fringe not empty:
        i. Find Minimum element from Fringe
        ii. If Minimum element == Goal_State
            1. Add state to visited_states
            2. Return 111111
        iii. If Minimum element != Goal_State
            1. Add state to visited_states
            2. Find neighboring states of current state
            3. For each neighboring state not in in visited set:
                a. Find h_val
                b. g_val ← g_val of current state + 1
                c. Fringe ← Add(current_state, 0, h_start, f_val)

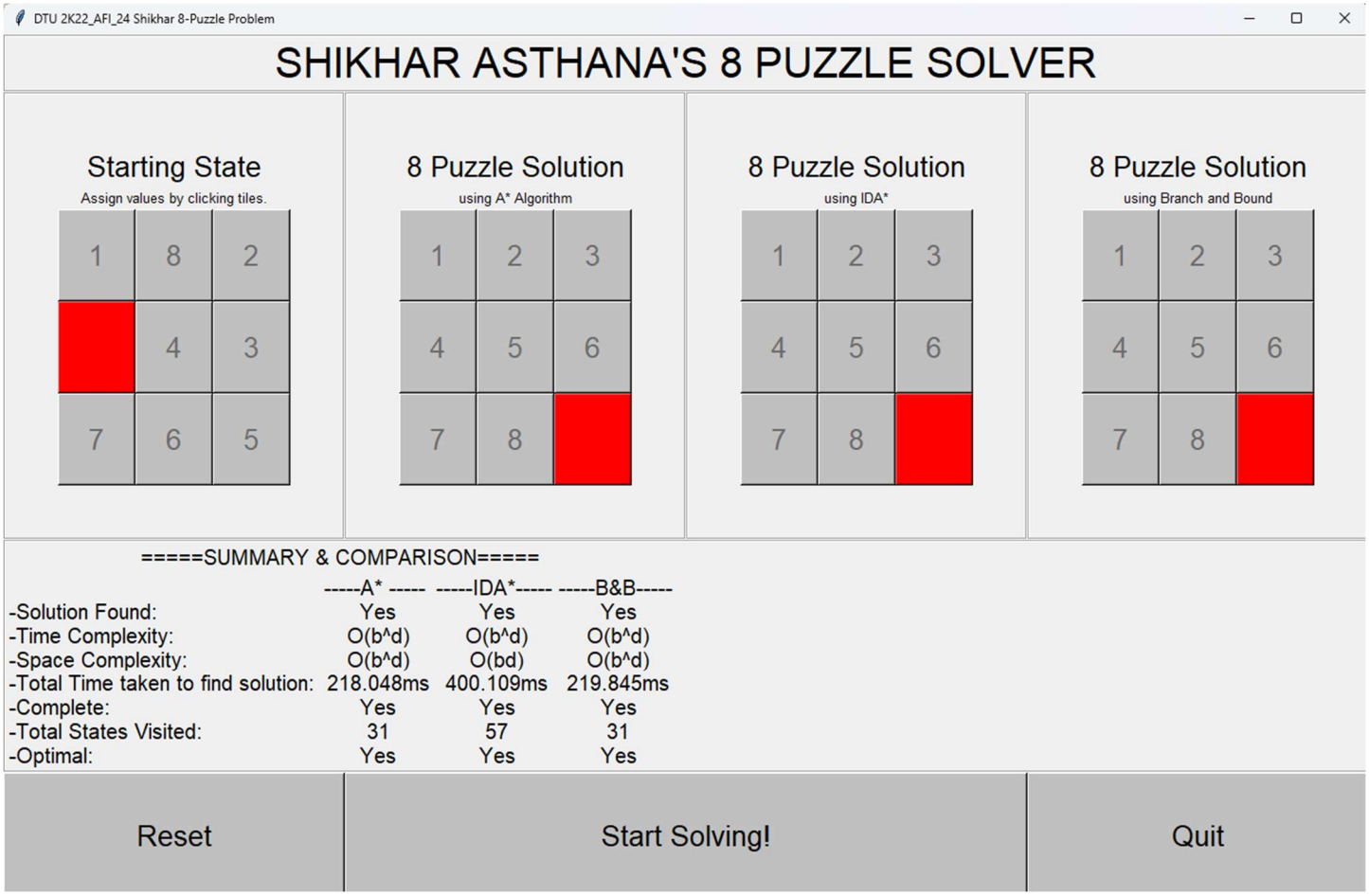
2. Solve_using_idastar(start_state, Goal_state)
    a. Threshold_val ← Heuristic(start_state)
    b. Max_Threshold_val ← Max allowed value based on problem
    c. While (Threshold_val <= Max_Threshold_val)
        i. Updated Threshold = idastar_algo(start_state, end_state, Threshold_val)
        ii. If Updated Threshold == 111111
            1. Return True
        iii. Threshold_val = Updated Threshold
    d. Return False

Approach 3 – B&B*

1. Create Fringe
2. h_start ← Heuristic(start_state)
3. c_val ← h_start + 0
4. Fringe ← Add(start_state, 0, h_start, c_val)
5. While Fringe not empty:
    a. Find Minimum cost element from Fringe
    b. If Minimum element == Goal_State
        i. Add state to visited_states
        ii. Return True
    c. If Minimum element != Goal_State
        i. Add state to visited_states
        ii. Find neighboring states of current state
        iii. For each neighboring state not in in visited set:
            1. Find h_val
            2. c_val ← g_val of current state + 1
            3. Fringe ← Add(current_state, 0, h_start, f_val)

CODE: Implemented code is submitted over the google classroom assignment.

OUTPUT:



As evident from the output parameters, for the above shown input state, the performance of the approaches can be arranged as follows:

$$A^* > B\&B > IDA^*$$

But is the above arrangement always the situation? No. IDA* has an additional advantage of taking lesser amount space – polynomial space complexity. Hence, in situations where space might be constrained, **IDA* would be a better choice**.

As shown in the above screenshot, I have taken into consideration the following parameters:

- Solution Found – Whether the approach was able to find the solution or not
- Time Complexity – The time complexity in terms of max branching factor (b), depth of least cost solution (d), maximum depth of state space tree.
- Space Complexity – The space complexity in terms of max branching factor (b), depth of least cost solution (d), maximum depth of state space tree.
- Total Time Taken to find solution – The time difference based on my system clock in ending the algorithm and starting of the algorithm – in ms.
- Complete – If a solution exists, can the paradigm find the solution.
- Total States visited – The total number of states visited by the algorithm in attempting to find the goal state.
- Optimal – Is this approach optimal.

Table 5.1 - Summary & Comparison			
Parameter	A*	IDA*	B&B
Solution Found	Yes	Yes	Yes
Time Complexity	$O(b^d)$	$O(b^d)$	$O(b^d)$
Space Complexity	$O(b^d)$	O(bd)	$O(b^d)$
Total Time Taken to Find the Solution	218.048ms	400.109ms	219.845ms
Complete	Yes	Yes	Yes
Total States Visited	31	57	31
Optimal	Yes	Yes	Yes

Comparison between uninformed search of Experiment 4 and informed searches of Experiment 5 can be made based on our observations/results and parameters. These can be highlighted as follows:

- The most major difference is the optimality of the algorithms. We saw that in the case of uninformed searches, all of them were non-optimal. Moreover, DFS was not even complete.
- Total States visited in informed searches are far less as compared to the total states visited in uninformed searches. The maximum difference can be seen

between DFS and A*. A* algorithm was able to reach solution after visiting just 31 states but DFS had to visit a total of 37,347 states.

- One major point to be noted is that, in our observations, the time taken to find the solution in uninformed search (BFS and IDDFS) seem to be lower than informed search time – This can be attributed to the fact that there is quite minimal calculation required in deciding the next step in terms of uninformed searches. However, when we deal with much larger problems, where the state-space tree is having very high branching factors, the effect due to lesser number of states in informed searches dominates the lesser next step computation time of uninformed searches.

Therefore, we can see that both uninformed and informed searches are mere tools and strategies which have their own unique advantages and disadvantages. Which algorithm will be best will depend upon the nature of the problem and the computational limitations we are presented with.