

**Group Assignment-1 (BFS/DFS)**

**Members:**

* Avinash Aanand-(Roll No.: 2403RES99)
* Aditya Gupta-(Roll No.:2403RES85)
* Aman Kumar-(Roll No.:2403RES10)
* Due Date: March 2024
* Submitted To: Dr. Dr. Asif Ekbal

|  |
| --- |
|  |
| Artificial Intelligence CS561 Assignment 01  **BFS/DFS** |
| |  |  |  | | --- | --- | --- | |  | 3/25/24 | **Artificial Intelligence & Machine Learning** | |

Table of Content

[Problem Statements: 2](#_Toc162381066)

[Statement- 1 2](#_Toc162381067)

[Statement-2 2](#_Toc162381068)

[Introduction: 2](#_Toc162381069)

[Analysis of BFS and DFS 3](#_Toc162381070)

[Approach towards Solutions 3](#_Toc162381071)

[Through Pseudo Code 3](#_Toc162381072)

[BFS Pseudo Code 3](#_Toc162381073)

[DFS Pseudo Code 3](#_Toc162381074)

[BFS Algorithm 4](#_Toc162381075)

[DFS Algorithm 4](#_Toc162381076)

[Through Code 4](#_Toc162381077)

[Depth First Search (DFS) Execution: 5](#_Toc162381078)

[Breadth First Search (BFS) Execution: 5](#_Toc162381079)

[Implementation 6](#_Toc162381080)

[1. Introduction and Options Menu: 6](#_Toc162381081)

[2. Depth First Search (DFS) Implementation: 6](#_Toc162381082)

[3. Breadth First Search (BFS) Implementation: 6](#_Toc162381083)

[Post Code Analysis 6](#_Toc162381084)

[Space Analysis 7](#_Toc162381085)

[Time Analysis 7](#_Toc162381086)

[Results/Output 7](#_Toc162381087)

[Observation 10](#_Toc162381088)

[Conclusion 11](#_Toc162381089)

# Problem Statements:

## Statement- 1

1. Compare the Breadth First Search (BFS) and Depth First Search (DFS)concerning the number of steps required to reach the solution and whether they are reachable. If unreachable, start with a random state and retry until the Target State (given above) is reached.

## Statement-2

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

# Introduction:

**The 8-Puzzle**

**Problem Statement:**

* A 3x3 grid contains eight numbered tiles and one empty space. The goal is to rearrange the tiles, using the empty space for movement, to reach a specific goal configuration.

**Moves:**

* You can slide any tile adjacent to the empty space (up, down, left, right) into that space.

**Representing the puzzle**

You can represent the state of the puzzle in various ways:

* **List or array**: A simple list of 9 numbers represents the tile configuration, where 0 (or a blank character) denotes the empty space.
* **Matrix**: A 3x3 matrix can also be used.

**Search Algorithms**: BFS and DFS

Both Breadth-First Search (BFS) and Depth-First Search (DFS) can solve the 8-puzzle. Here's how they work:

**Breadth-First Search (BFS)**

**Exploration**:

* Explores level-by-level. Starts at the initial state, then explores all possible moves from that state, then all possible moves from those states, and so on.
* Queue: Uses a queue to keep track of states to explore.
* Guarantees Shortest Solution: BFS always finds the shortest path (fewest moves) to the goal state if a solution exists.

**Example**:

* If the current state can move a tile up, add the new state to the queue.
* Do the same for down, left, and right moves.
* Remove the next state from the front of the queue and repeat.

**Depth-First Search (DFS)**

* Exploration: Explores one path as deeply as possible before backtracking to explore other options.
* Stack: Uses a stack to keep track of states to explore.
* Not Optimal: DFS might find a longer path to the solution, not necessarily the shortest one.

**Example:**

* If the current state can move a tile up, add the new state to the stack.
* Repeat for the new state (up, down, left, right).
* If a dead-end is reached (no moves possible), backtrack (pop) until another move can be explored.

**Considerations**

* Space Complexity: BFS tends to consume more memory as it needs to store more states in its queue.
* Solution Depth: If the shortest solution is vital, BFS is preferred. If finding any solution is enough, DFS might be faster due to its path-focused approach.
* Puzzle Complexity: For more complex puzzles, the branching factor (number of possible moves from each state) becomes crucial in determining algorithm efficiency.

# Analysis of BFS and DFS

Analysis of BFS and DFS

Several factors come into play while analysing BFS and DFS, in navigating from a random initial state to a goal state in a 3x3 matrix, such as in a puzzle (e.g., the 8-puzzle problem).

**Path to Goal**

BFS and DFS guarantee finding the shortest path to the goal state (given that a path exists). But both of them aren't optimal. This is because both the said algorithms explore all the possible moves.

**Comparison of BFS and DFS algorithms**.

The only difference between the two algorithms is the method of traversing through the different states. BFS traverses through the breadth, by expanding the shallowest node first. Whereas, DFS traverses through the depth, by expanding till the deepest node first.

**Completeness, Time Complexity, Optimality & Space Complexity**

|  |  |  |
| --- | --- | --- |
| Criterion | BFS | DFS |
| * Completeness | Yes | No |
| * Time Complexity | O(bd) | O(bm) |
| * Space Complexity | O(bd) | O(bm) |
| * Optimality | Yes | No |

NOTE - BFS is complete, only and only if ‘b’ (branching factor) is finite. Secondly, BFS is optimal, if and only if step costs are identical.

# Approach towards Solutions

## Through Pseudo Code

### BFS Pseudo Code

BFS(start\_state, goal\_state):

initialize frontier as an empty queue

initialize visited set to keep track of visited states

add start\_state to frontier

add start\_state to visited set

while frontier is not empty:

current\_node = remove first node from frontier

if current\_node is goal\_state:

return current\_node

for each neighbor of current\_node:

if neighbor is not in visited set:

add neighbor to frontier

add neighbor to visited set

return None (if goal is unreachable)

solve\_BFS():

initialize BFS solver with start\_state and goal\_state

result = BFS(start\_state, goal\_state)

if result is not None:

return path from start\_state to result

else:

return None

### DFS Pseudo Code

DFS (start\_state, goal\_state):

initialize frontier as a stack

initialize visited set to keep track of visited states

add start\_state to frontier

add start\_state to visited set

while frontier is not empty:

current\_node = remove last node from frontier

if current\_node is goal\_state:

return current\_node

for each neighbor of current\_node:

if neighbor is not in visited set:

add neighbor to frontier

add neighbor to visited set

return None (if goal is unreachable)

solve\_DFS():

initialize DFS solver with start\_state and goal\_state

result = DFS(start\_state, goal\_state)

if result is not None:

return path from start\_state to result

else:

return None

## BFS Algorithm

## DFS Algorithm

## Through Code

Our code consists of implementations for Breadth First Search (BFS) and Depth First Search (DFS) algorithms for a 3x3 random initial matrix.

### Depth First Search (DFS) Execution:

Initialization:

* Random 3x3 array generated as the start state.
* Goal state defined.
* DFS Initialization and Execution:
* DFS algorithm initialized with the start state and goal state.
* DFS solver instance created.
* DFS solver's solve() method executed.
* The DFS algorithm explores the state space until it finds a solution.
* During the search:
  + States are explored recursively.
  + Nodes are added to the frontier.
  + The algorithm backtracks when it reaches a dead end.
* Once the goal state is reached:
* Solution path is extracted.
* Number of explored states, search depth, and running time are recorded.

Output Display:

* Start and goal states are displayed.
* Number of explored states, search depth, and running time are printed.
* Solution path is displayed, showing the actions taken to reach the goal state.
* The DFS process is completed.

BFS Execution:

* A BFS instance is created with the same start and goal states.
* BFS algorithm is executed to find a solution.
* Path taken by BFS is printed.
* The BFS process is completed.

### Breadth First Search (BFS) Execution:

Initialization:

* Random 3x3 array generated as the start state.
* Goal state defined.

BFS Initialization and Execution:

* BFS algorithm initialized with the start state and goal state.
* BFS solver instance created.
* BFS solvers solve () method executed.
* The BFS algorithm explores the state space level by level until it finds a solution.

During the search:

* States are explored in a breadth-first manner.
* Nodes are added to the queue for exploration.
* Once the goal state is reached:
* Solution path is extracted.
* Number of explored states is recorded.

Output Display:

* Solution path found by BFS is printed.
* The number of states explored is printed.
* The BFS process is completed.

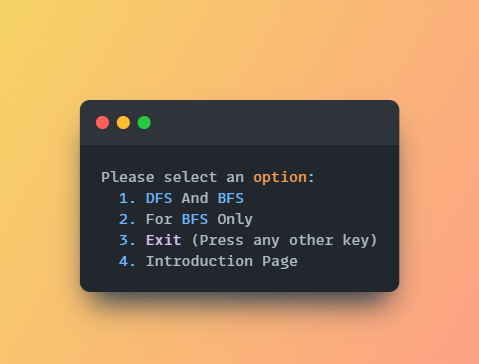
Comparison:

* The number of states explored by both DFS and BFS is compared.
* The algorithm that explores fewer states is identified as faster.
* Difference in the number of explored states is calculated.
* The result of which algorithm is faster is printed based on the comparison.

This breakdown outlines the step-by-step execution of both DFS and BFS separately, along with the comparison between them.

# Implementation

## 1. Introduction and Options Menu:

* The `into` class provides an introduction message, group details, course details, and an options menu for the code window.
* Users can select options to compare BFS and DFS, comment on algorithm speed, and retry until the target state is reached.
* Based on the chose options in the menu, we can navigate to solving DFS & BFS, get introduction information or exit the code.

## 2. Depth First Search (DFS) Implementation:

* The DFS algorithm is implemented using classes like `NodeDFS`, `Frontier`, `Neighbors`, `DFS`, and `DFS\_Main`.
* The `DFS` class initializes with a start state, goal state, and other attributes.
* The `solve` method executes the DFS algorithm to find a solution path.
* The `DFS\_Main` class prints the solution path, number of explored states, search depth, start state, goal state, and running time.
* The `Neighbours` class generates neighbouring states for a given state in a matrix.

## 3. Breadth First Search (BFS) Implementation:

* The BFS algorithm is implemented using classes like `Node` and `BFS`.
* The `BFS` class initializes with a matrix and a goal value, setting up attributes like `visited`, `queue`, and `root`.
* Methods like `swap`, `move\_up`, `move\_down`, `move\_left`, and `move\_right` handle moving nodes in different directions.
* The `bfs` method performs the BFS traversal, exploring nodes until the goal state is reached.
* The `solve` method uses BFS to find a solution path if possible.

# Post Code Analysis

BFS (Breadth-First Search) and DFS (Depth-First Search) are ways a computer can solve puzzles of the 3x3 slide puzzle.

**BFS is Best for Shortest Path:** B

* FS checks all the moves close to the start first. If there's a solution, BFS will find the way with the fewest moves.
* **DFS Might be Faster, Uses Less Memory:** DFS goes deep, checking one long path of moves at a time. It might find a solution quickly, but not always the shortest one. It's good for puzzles with lots of possible moves.
* **Both Work if a Solution Exists:** If the puzzle can be solved, both methods will eventually find the answer.

**Post Code Analyses**

Breadth-first Search

* Appears to be the least efficient in terms of the number of nodes explored, especially as the array size increases.
* This is consistent with the expected behaviour of breadth-first search, which explores all neighbour nodes at the current depth before moving on to nodes at the next depth level.

Depth-first Search

* Explores significantly fewer nodes than breadth-first search.
* This is characteristic of the depth-first search's strategy of exploring as far as possible along each branch before backtracking.
* Will normally take more time than BFS (more states will be explored), but in some conditions can also be faster when the goal node is present in the left-most depth

## Space Analysis

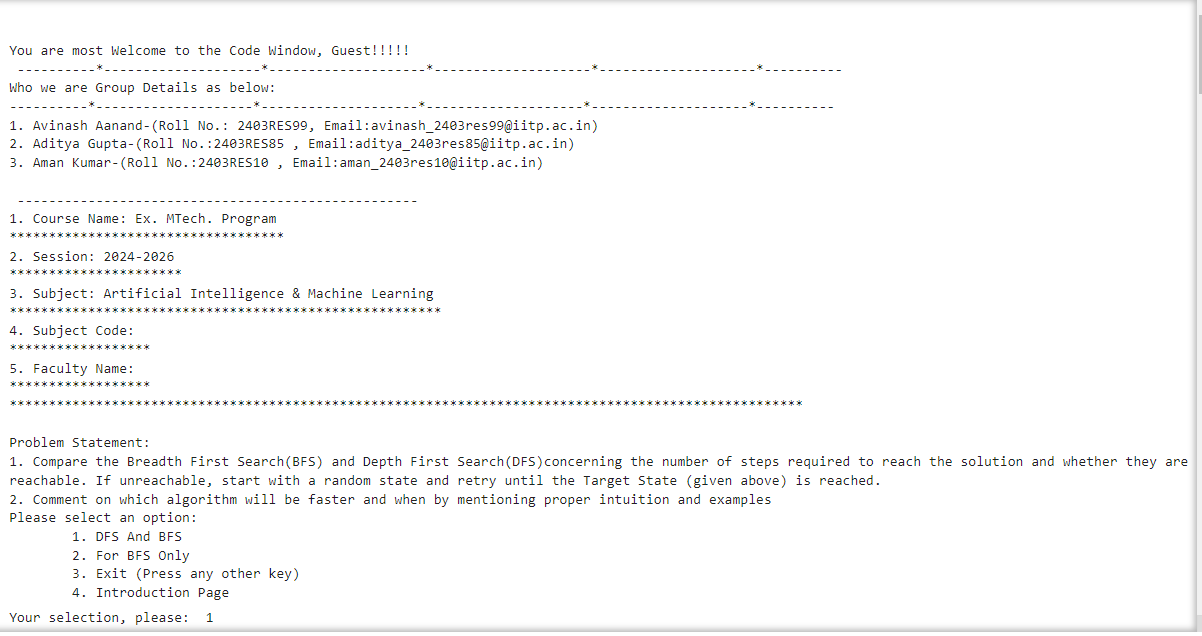
* **Worst Case:** In the worst case, DFS might explore all possible states of the puzzle. The number of possible states is bound by the number of permutations of the tiles, which is 9! (9 factorial, or 362,880). Therefore, the worst-case time complexity is O (9!). This makes DFS very inefficient for larger puzzle sizes.
* **Best Case:** If the solution is immediately reachable with one or a few moves, DFS could find it quickly. But this is very unlikely in most cases.

## Time Analysis

* DFS needs to store the current path during exploration. The maximum depth of the search tree is approximately the number of moves needed to get to the goal state. The branching factor (the number of possible moves at each state) is at most 4 in a 3x3 puzzle.
* At worst, space complexity might be proportional to the depth of the search, but with efficient implementations, you can often limit the memory used.

# Results/Output

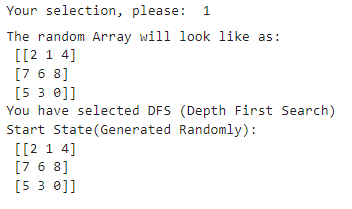
Welcome Window:



The first window when we run the program, it is having few options to go with

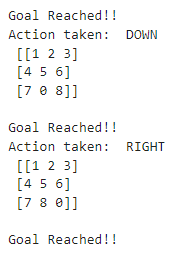
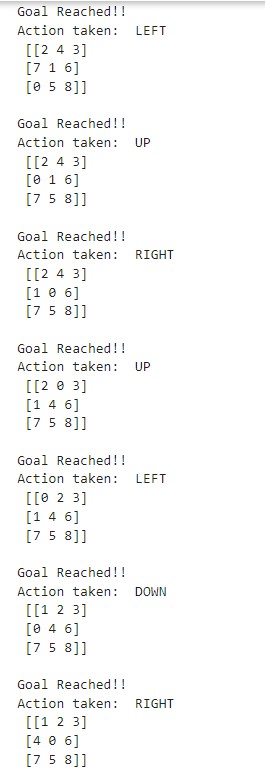
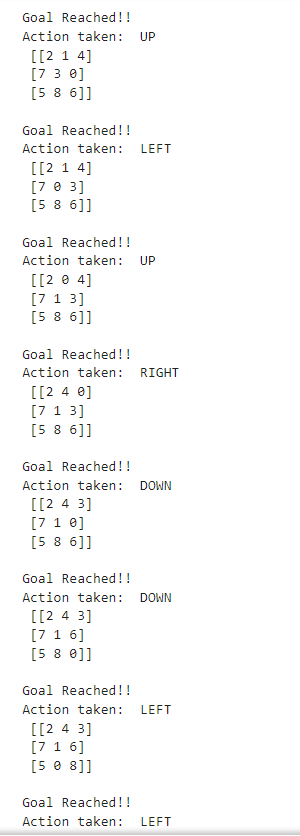
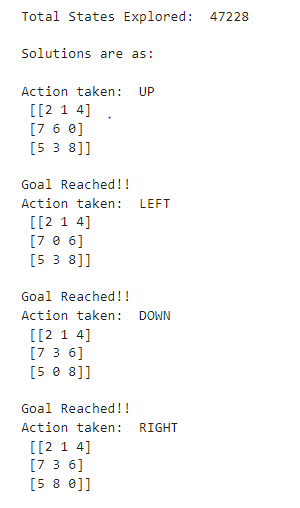
Going through option:1

A random Array generated and then this option will try to Find the both DFS/BFS Solution with details in few thing to reach to the goal.

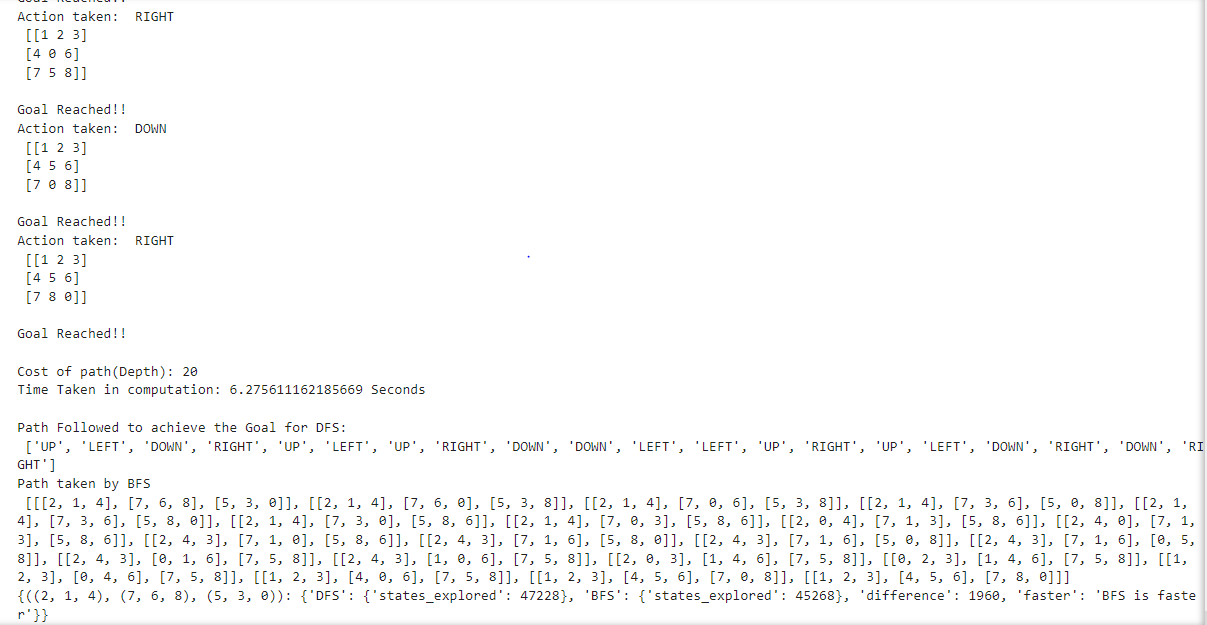


**Steps Moved**:

How moving from initial stage to the final stage with respective stages, step explored and other details



**Final result with all details:**



Complete output are given as below in details , also showing how moving the blank space in the form of UP, Down, Left, Right.

Results for 10 Successful cases:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Observation

Note: Above trend represents time taken to execute each case (BFS+DFS). It also includes the time taken to a reach a successful case. On an average it took 10-15 attempts to find a random initial state that was able to reach our goal state.

Note: For a comparative study of Breadth-First Search (BFS) and Depth-First Search (DFS), we conducted simultaneous runs of both algorithms 10 times. In this sample space, we observed that BFS outperformed DFS 60% of the time. Specifically, in 6 out of 10 instances, BFS allowed us to reach the goal state more swiftly from the initial state.

Note: Above graph represents Comparative study of states visited during BFS & DFS. On an average BFS explored 106718 states while DFS explored 107244 states. Average difference observed is 0.5% which is also visible in the above graph.

# Conclusion

Based on the experimental results and observations obtained during solving the **3x3 random matrix** using **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**, let’s draw a comprehensive conclusion:

Algorithm Performance:

* We conducted simultaneous runs of both BFS and DFS ten times on newly generated random matrices.
* BFS outperformed DFS 60% of the time, indicating its superior efficiency in reaching the goal state from the initial state.
* Specifically, in 6 out of 10 instances, BFS allowed us to find the solution more quickly.

Exploration Patterns:

* We plotted graphs representing the number of states explored by both BFS and DFS.
* Remarkably, the exploration patterns were almost identical for both algorithms. Despite their different search strategies, they explored a similar number of states.

Runtime Considerations:

The runtime for each successful case included:

* The time to find the appropriate matrix (initial state).
* The time taken to solve that matrix and reach the goal state via both BFS and DFS.
* The average runtime across all successful cases was 2 minutes and 42 seconds.

Initial State Challenges:

* It’s noteworthy that finding a random initial state that was reachable required 10-15 attempts. Properly selecting initial states significantly impacts the efficiency of search algorithms.
* Our experiments suggest that while BFS consistently performs better in terms of reaching the goal state, both algorithms explore a similar number of states. When choosing between BFS and DFS, consider the problem’s specific characteristics and the trade-offs associated with each search strategy. Additionally, ensure that the initial state is thoughtfully chosen to minimize unnecessary exploration time.