1. Introduction

The 8-puzzle problem is a classic puzzle that has captivated the minds of puzzle enthusiasts and computer scientists. It is a sliding puzzle that consists of a 3x3 grid with 8 numbered tiles and an empty space. The goal is to rearrange the tiles from an initial state to a specified goal state using the minimum number of moves while obeying the rules of the game.

The purpose of this report is to investigate and compare the performance of three different search algorithms in solving the 8-puzzle problem. We aim to implement and analyze the following algorithms:

1. Breadth-First Search (BFS)
2. Depth-First Search (DFS)
3. A* Search (Using Manhattan Distance and Euclidean Distance heuristics)

We will evaluate these algorithms in terms of their ability to find optimal solutions, their efficiency, and the quality of the paths they generate. This report will provide insights into the strengths and weaknesses of each algorithm in the context of the 8-puzzle problem.

2. Problem Description

The 8-puzzle is a classic sliding puzzle that consists of a 3x3 grid with 8 numbered tiles and one empty space. The puzzle's objective is to rearrange the tiles from an initial configuration to a specified goal configuration using the minimum number of moves. The objective is to transform the initial state into this goal state by sliding the tiles into the empty space.

In the 8-puzzle, each move involves sliding a single tile (numbered 1 to 8) into the adjacent empty space, in our case we will assume that the empty tile is '0'. For easier implementation, the moves represent the movement of the '0' tile.

The following moves are allowed:

1. Move Left: Swap the '0' tile with the one on the left.
2. Move Right: Swap the '0' tile with the one on the right.
3. Move Up: Swap the '0' tile with the one above it.
4. Move Down: Swap the '0' tile with the one below it.

In our case, we have a fixed goal state which we want to reach from any random state, here is the goal state:

■	1	2
3	4	5
6	7	7

3. Algorithms Used

3.1. Breadth First Search (BFS)

```
function BREADTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Queue.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.dequeue()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier $\cup$ explored:
                frontier.enqueue(neighbor)

    return FAILURE
```

3.2. Depth First Search (DFS)

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Stack.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier $\cup$ explored:
                frontier.push(neighbor)

    return FAILURE
```

3.3. A* Search

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE : /* Cost $f(n) = g(n) + h(n)$ */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

For the cost function, we will use the Manhattan Distance heuristic and the Euclidean Distance heuristic and also compare between both of them in terms of their efficiency and optimality.

3.3.1. Manhattan Distance Formula

$$\text{h} = \text{abs(current\_cell:x - goal:x)} + \text{abs(current\_cell:y - goal:y)}$$

3.3.2. Euclidean Distance Formula

$$\text{h} = \text{sqrt((current\_cell:x - goal:x)}^2 + \text{(current\_cell.y - goal:y)}^2\text{)}$$

4. Data Structures Used

4.1. The 8-Puzzle Game

For our implementation, we are using C++ programming language. The search tree nodes are represented by the 8-puzzle states, so we created the "State" struct. This struct contains:

- 2D-Array (3x3) representing the puzzle itself
- Index of the '0' tile in the array (i, j)
- State Number represents the number of the expanded states
- String Path which contains the path moves to the goal (l, r, u, d)
- Manhattan Distance heuristic for the state (integer) -> h(n)
- Euclidean Distance heuristic for the state (float) -> h(n)
- Actual Cost to the State (integer) -> g(n)
- Cost Function -> f(n) = h(n) + g(n) (Integer and Float)

4.2. Breadth First Search (BFS)

Here are the main data structures used for implementing the BFS algorithm:

- Frontier Queue (FIFO): this stores all the states which have been expanded, these are the states candidate to be explored.
- Explored Queue (FIFO): this stores the states which have already been explored.

4.3. Depth First Search (DFS)

Here are the main data structures used for implementing the DFS algorithm:

- Frontier Stack (LIFO): this stores all the states which have been expanded, these are the states candidate to be explored.
- Explored Queue (FIFO): this stores the states which have already been explored.

4.4. A* Search

Here are the main data structures used for implementing the A* Search algorithm:

- Frontier Priority Queue (FIFO): this stores all the states which have been expanded, they are sorted in the queue ascendingly according to their cost function; the least cost is popped out first.
- Explored Queue (FIFO): this stores the states which have already been explored.

In all 3 algorithms, a queue is used for the explored states instead of a set, this is done for easier implementation. "Contains" function is implemented to check whether the state already exists in the queue or not to avoid exploring the same node more than once.

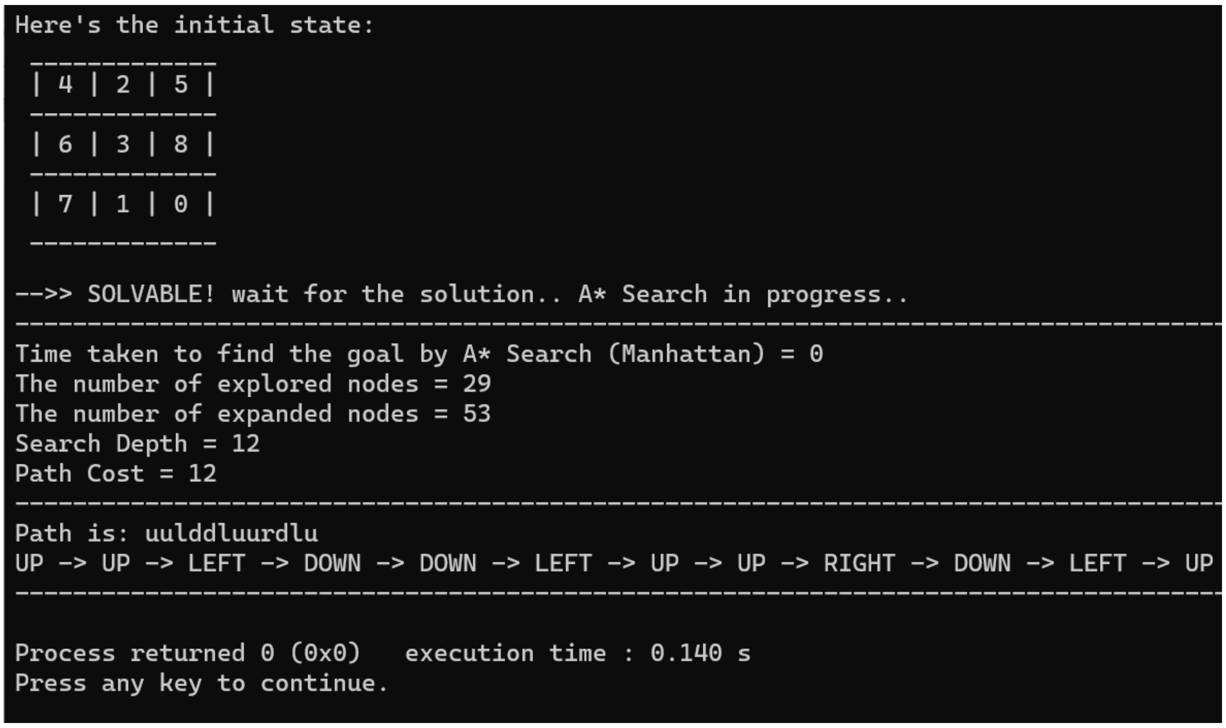
5. Assumptions

Here are the assumptions made throughout the code:

- Search Depth: It is assumed that the depth starts at Level-0 which is when the search tree only contains the root state.
- Manhattan Distance Heuristic: Assumes that each numbered tile can reach its desired position in the puzzle without having any obstacles in its way, moving only in vertical and horizontal directions.
- Euclidean Distance Heuristic: Assumes that each numbered tile can reach its desired position in the puzzle without having any obstacles in its way, moving in a straight line directly between the 2 positions.
- Path Cost: The path cost is assumed to be the number of moves made to reach the goal by any of the search algorithms.
- Complete Search: The algorithms aim for complete search, meaning they will explore all possible states until they find the goal. No pruning or optimization techniques are used to limit the search space.
- Solvable Puzzles: It is assumed that NOT all puzzles are solvable; a solvable function checks the solvability of the puzzle before running any search algorithm.

6. Sample Runs

In this part, we will investigate and compare the performance of three different search algorithms in solving the 8-puzzle problem. There are 4 main sample runs, their outputs are recorded in tables, making it easier to visualize the major differences between the algorithms. Here is a sample of how the run looks like in the command window (Running Manhattan heuristic A* Search on {4,2,5,6,3,8,7,1,0} -> Sample Run #2.



```
Here's the initial state:
 -------------
 | 4 | 2 | 5 |
 -------------
 | 6 | 3 | 8 |
 -------------
 | 7 | 1 | 0 |
 -------------

-->> SOLVABLE! wait for the solution.. A* Search in progress..
--------------------------------------------------------------------------------
Time taken to find the goal by A* Search (Manhattan) = 0
The number of explored nodes = 29
The number of expanded nodes = 53
Search Depth = 12
Path Cost = 12
--------------------------------------------------------------------------------
Path is: uulddluurdlu
UP -> UP -> LEFT -> DOWN -> DOWN -> LEFT -> UP -> UP -> RIGHT -> DOWN -> LEFT -> UP
--------------------------------------------------------------------------------

Process returned 0 (0x0)   execution time : 0.140 s
Press any key to continue.
```

SAMPLE RUN #1

1	5	8
6		2
4	3	7

	1	2
3	4	5
6	7	7

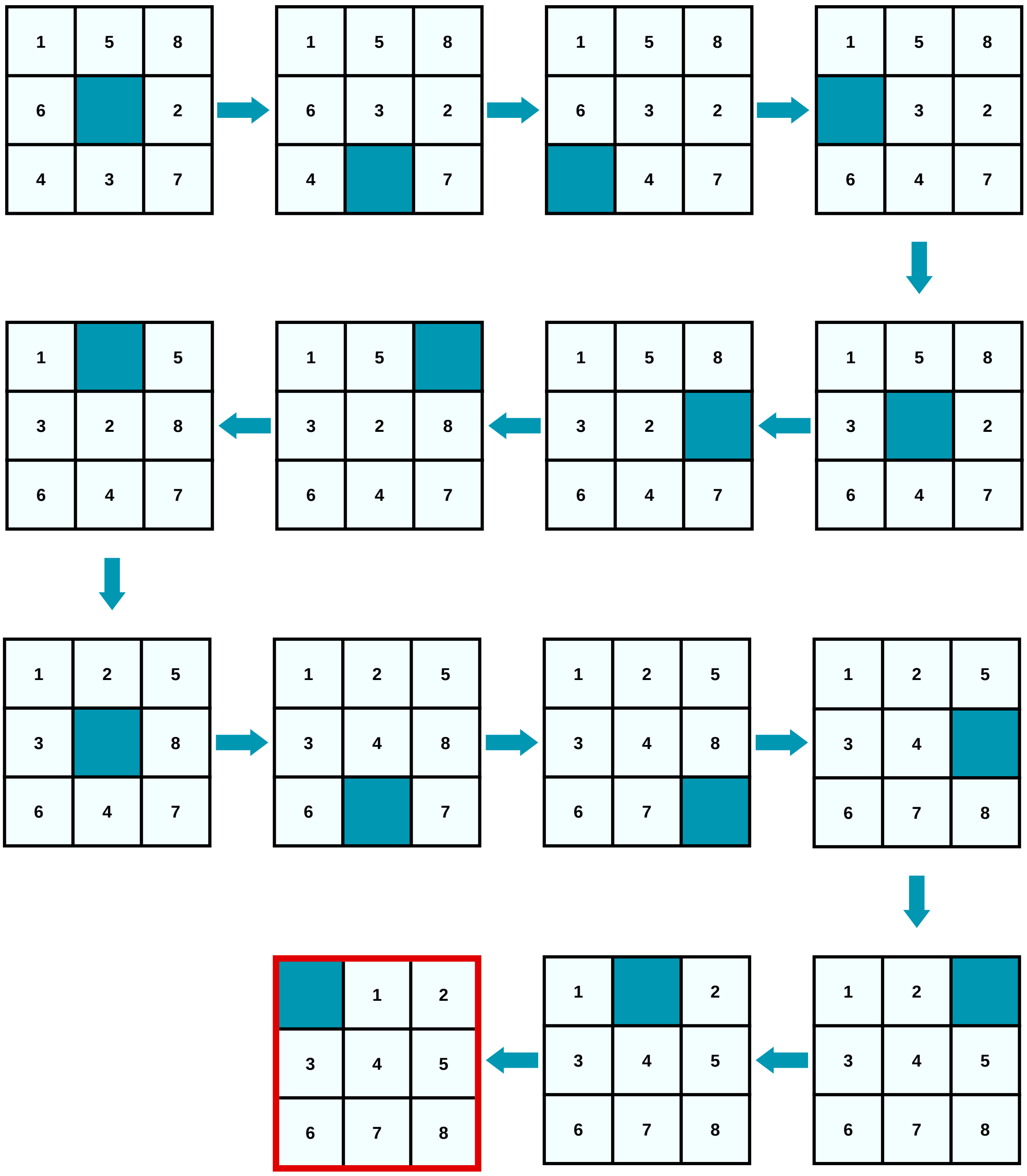
COMPARISON BETWEEEN ALGORITHMS

Algorithm	No. of Explored States	No. of Expanded States	Running Time	Search Depth	Path Cost
A* **Manhattan**	41	68	0.001 s	14 levels	14 moves dlurrulddruull
A* **Euclidean**	84	134	0.008 s	14 levels	14 moves dlurrulddruull
BFS	4561	7219	8.003 s	14 levels	14 moves rulddlurdruull
DFS					

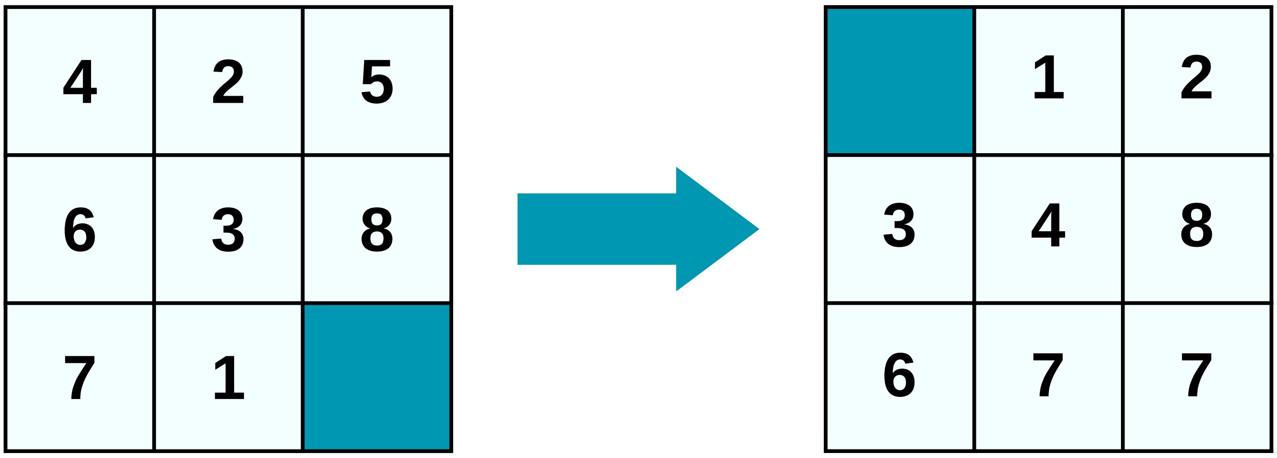
SAMPLE RUN #1

PATH OF A* SEARCH

This is the path tracking for the A* (Manhattan) search algorithm, it's noticed from the previous comparison that all algorithms have the same cost (14 moves), but the A* (Manhattan) search algorithm has the least running-time.



SAMPLE RUN #2



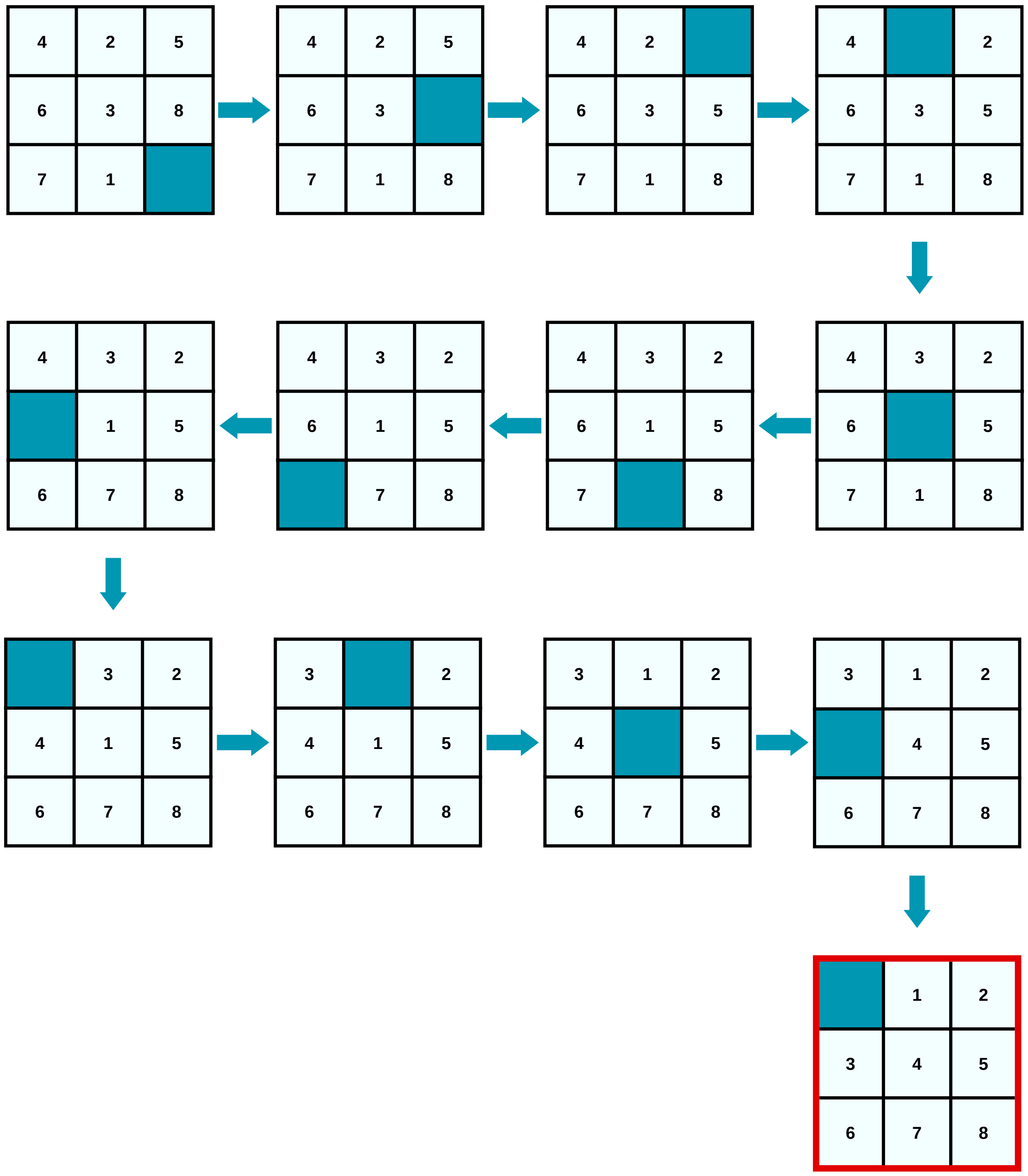
COMPARISON BETWEEEN ALGORITHMS

Algorithm	No. of Explored States	No. of Expanded States	Running Time	Search Depth	Path Cost
A* **Manhattan**	29	53	0 s	12 levels	12 moves uulddluurdlu
A* **Euclidean**	28	52	0 s	12 levels	12 moves uulddluurdlu
BFS	1830	2847	1.343 s	14 levels	12 moves uulddluurdlu
DFS	114962	127902	1730 s (30 mins)	65982 levels	65982 moves

SAMPLE RUN #2

PATH OF A* SEARCH

This is the path tracking for the A* (Manhattan) search algorithm, it's noticed from the previous comparison that all algorithms have the same cost (12 moves), but the A* (Manhattan) search algorithm has the least running-time.



SAMPLE RUN #3

1	2	
3	4	5
6	7	8

$\longrightarrow$

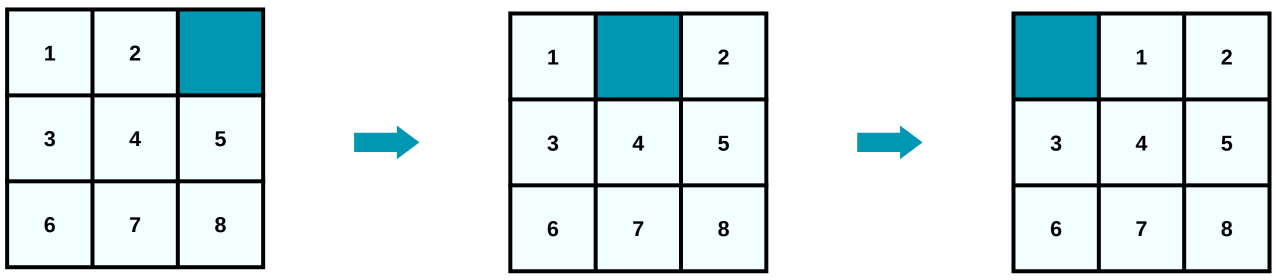
	1	2
3	4	8
6	7	7

COMPARISON BETWEEEN ALGORITHMS

Algorithm	No. of Explored States	No. of Expanded States	Running Time	Search Depth	Path Cost
A* **Manhattan**	3	5	0 s	2 levels	2 moves LL
A* **Euclidean**	3	5	0 s	2 levels	2 moves LL
BFS	4561	7219	8.003 s	2 levels	2 moves LL
DFS	84897	127093	1373 s (23 mins)	64328 levels	64328 moves

SAMPLE RUN #3

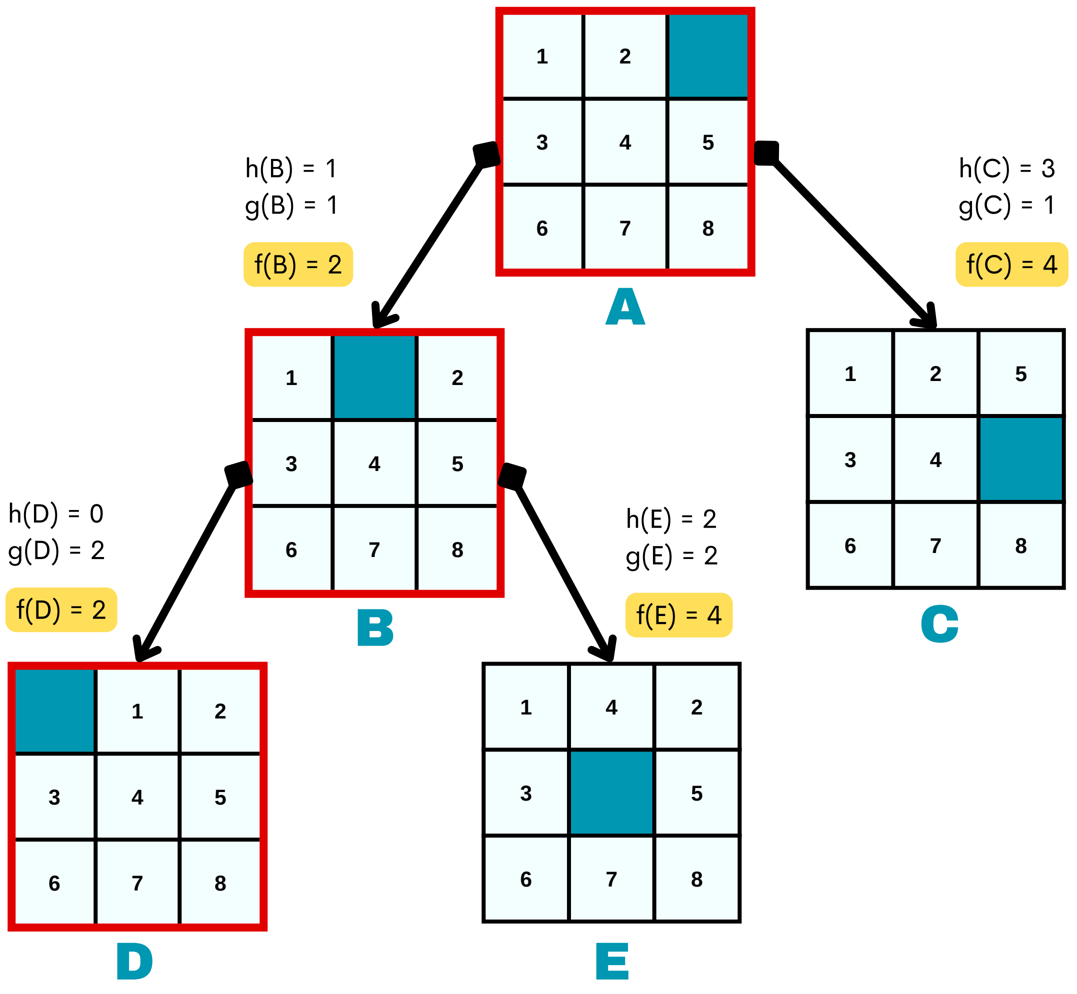
PATH OF A* SEARCH



In the next part, the search tree for each algorithm will be traced in detail, showing all expanded states to illustrate how the algorithm works.

Explored states have a bold RED border.

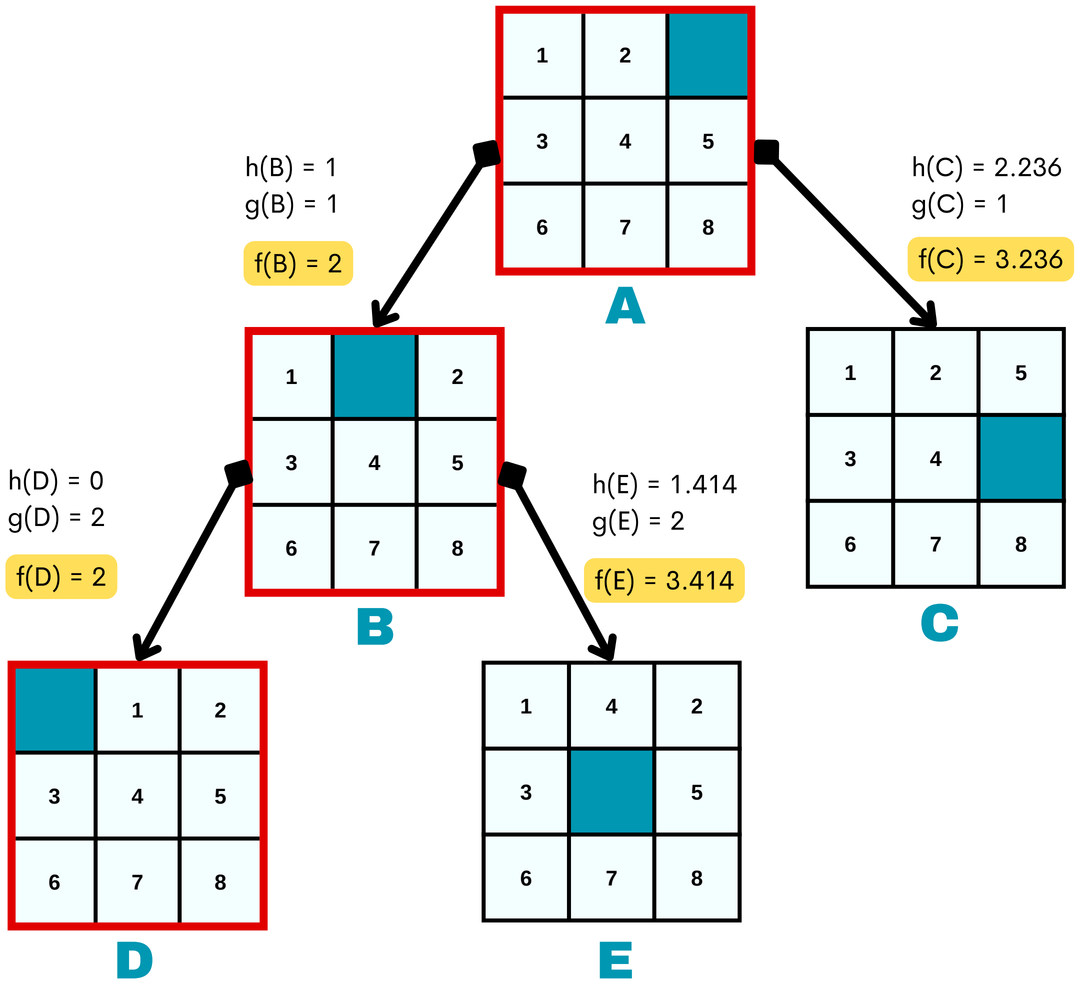
SEARCH TREE FOR A* (MANHATTAN)



SAMPLE RUN #3

SEARCH TREE FOR A* (EUCLIDEAN)

Explored states have a bold RED border.

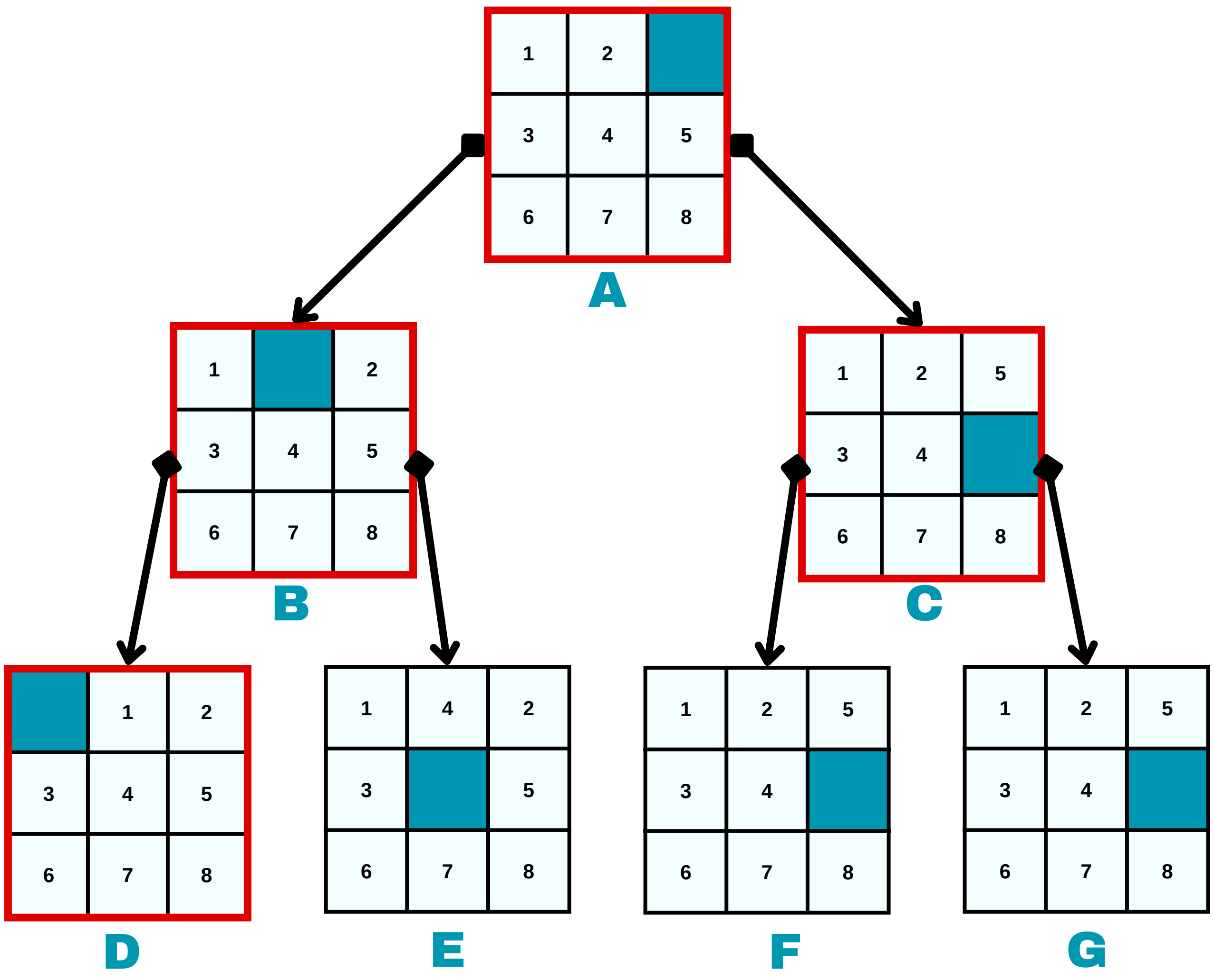


It's noticed that in this exact case, the same states are explored when using the Manhattan and the Euclidean distances as heuristics for A* search, this is due to the simplicity of the path, which does not happen in all cases. If further states are expanded, they will differ in both.

SAMPLE RUN #3

SEARCH TREE FOR BFS

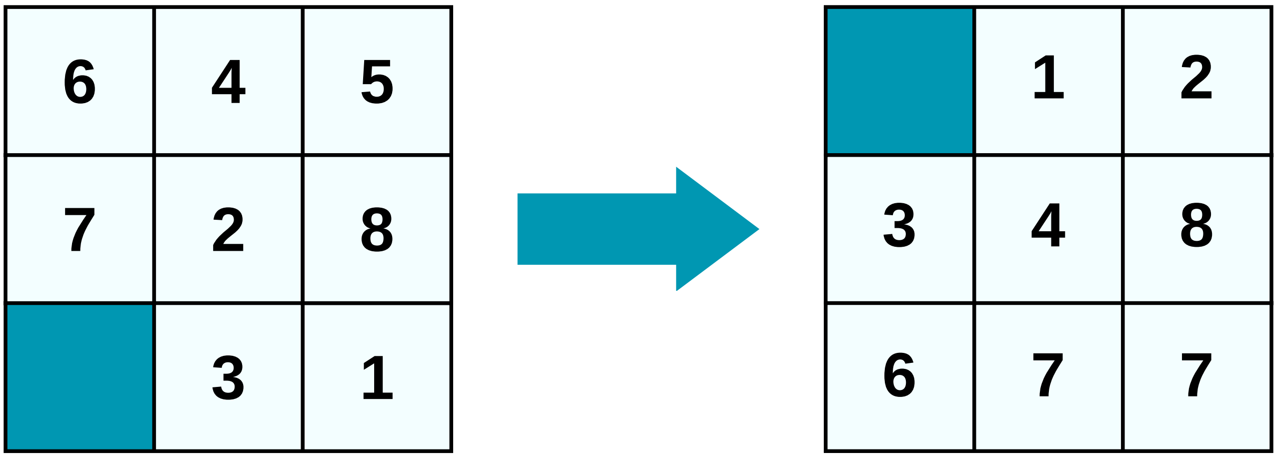
Explored states have a bold RED border.



In this specific sample run, we were able to dive into the search trees for the A* and the BFS search algorithms due to the simplicity of the path (Left -> Left) only, but for more complex paths like in sample run #1, it would be so hard to keep track of all the nodes.

DFS algorithm may explore the nodes infinitely, even for simple paths which only require 2 moves like this one. It might keep exploring infinitely as seen above, it gave a path cost of 64328 moves and ran for 23 minutes which is huge compared to the other search algorithms.

SAMPLE RUN #4



COMPARISON BETWEEEN ALGORITHMS

Algorithm	No. of Explored States	No. of Expanded States	Running Time	Search Depth	Path Cost
A* **Manhattan**	167	277	0.007 s	18 levels	18 moves uurddruulddl uurdlu
A* **Euclidean**	218	360	0.011 s	18 levels	18 moves uurddruulddl uurdlu
BFS	26915	37791	304.3 s (5 mins)	18 levels	18 moves uurddruulddl uurdlu
DFS					

7. Results Breakdown

Breadth-First Search (BFS)

- Efficiency: BFS explores a large number of states in a breadth-first manner, making it less efficient in terms of memory usage and time complexity.
- Optimality: BFS is optimal as it guarantees the shortest path to the goal.

Depth-First Search (DFS)

- Efficiency: DFS is memory-efficient as it explores deeper branches first, but it may take a long time to find the goal. This was clear in all previous sample runs, where DFS actually ran infinitely for most of them and couldn't generate the solution.
- Optimality: DFS is not guaranteed to be optimal. It can return the first solution it encounters, which may not be the shortest path.

A* Search

- Efficiency: A* combines the advantages of both BFS and DFS by using heuristics to guide the search efficiently. It was noticed that A* took almost no time in most sample runs compared to both BFS and DFS.
- Optimality: A* with an admissible heuristic guarantees optimality, meaning it finds the shortest path to the goal.
- For our specific problem (8-Puzzle) it was noticed that the Manhattan distance heuristic gave a more efficient search for more complex puzzles.

8. Conclusion

The choice of algorithm depends on the specific requirements and constraints of the problem at hand. Deepening our understanding of these search algorithms, including their strengths and limitations, is valuable for making informed decisions about which algorithm to employ.

For the 8-Puzzle problem, the best algorithm was the A* Manhattan distance heuristic search algorithm, guaranteeing optimal and efficient solutions for all cases.