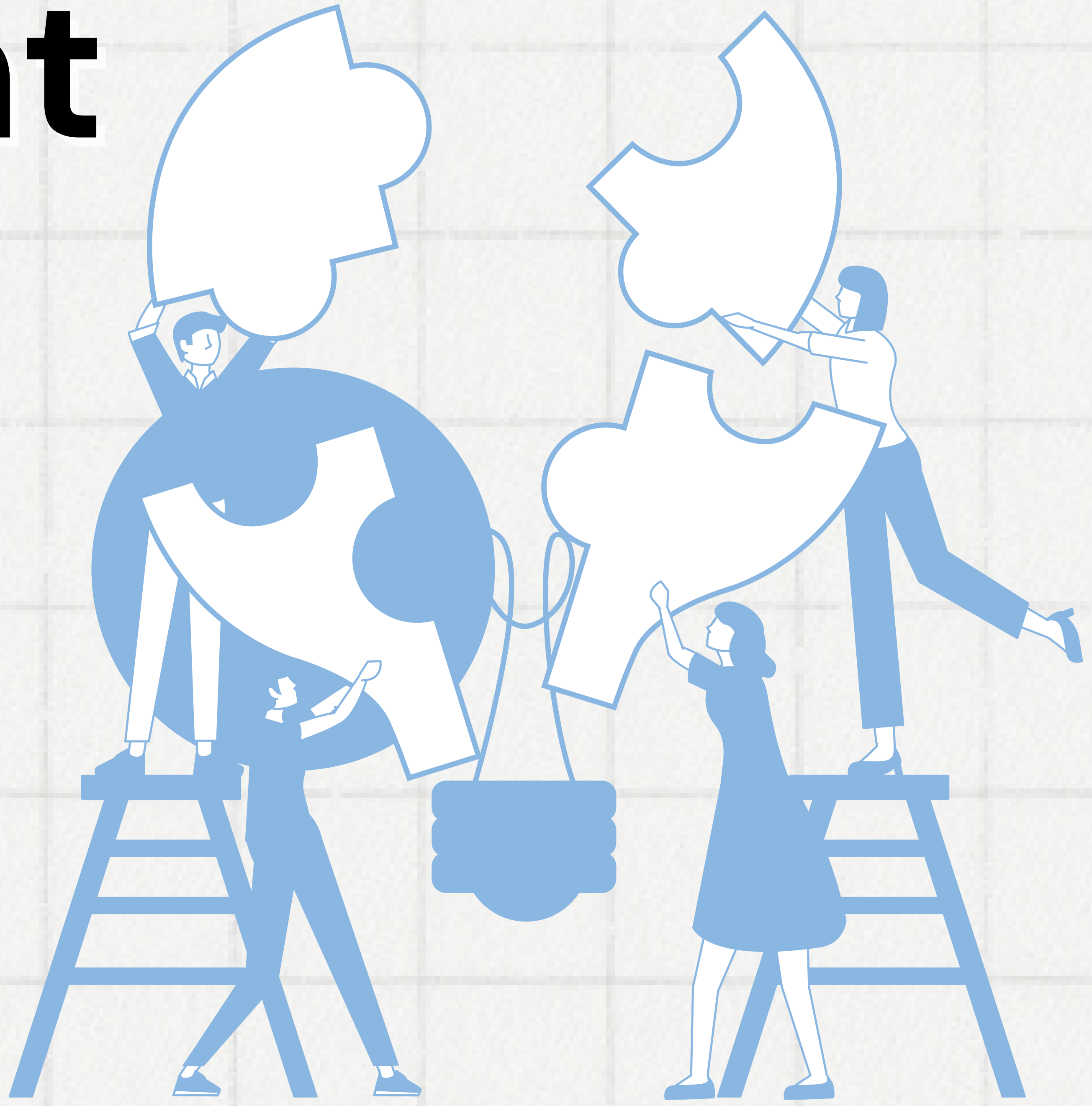
8-Puzzle Problem using A* search

Problem Statement

- Given an 8-puzzle with a starting configuration of numbered tiles (1 to 8) and one empty space on a 3x3 grid, develop an algorithm that finds a sequence of legal moves (solution path) to reach the goal state from the starting configuration.
- The 8-puzzle problem asks for an algorithm to solve a sliding tile puzzle: rearrange numbered tiles on a 3x3 grid to reach a specific goal state using legal moves (sliding tiles into the empty space).
- Many possibilities: There are numerous ways to move the tiles, leading to a vast number of potential states to explore.
- Finding the best path: Not all moves lead directly to the goal. The challenge is to identify the sequence of moves that gets you there in the fewest steps or using some other optimality criteria.



Approaches

01.

Brute Force Method

This approach involves systematically exploring all possible states of the puzzle until the goal state is reached. While guaranteed to find a solution if one exists, it is generally impractical due to the vast number of possible states, resulting in exponential time complexity.

02.

Breadth-First Search

BFS explores all nodes at the current depth level before moving to the next level. It guarantees finding the shortest solution path, but it may require a large amount of memory, especially for deeper search trees.

03.

Depth-First Search

DFS explores as far as possible along each branch before backtracking. It can be memory efficient but may not always find the shortest solution, as it can get stuck in deep branches.

04.

A* Search Algorithm

A* search combines the advantages of informed and uninformed search by using a heuristic function to guide the search towards the goal state efficiently. It evaluates nodes based on a combination of the cost to reach them from the start and an estimate of the cost to reach the goal from them.

A* Algorithm

1. **Start:** Add the initial state to the Open List. Set its f(n) score to 0 (initial cost is 0).
2. **Iteration:** While the Open List is not empty:
    - a. Select the state with the lowest f(n) score from the Open List (**prioritizes states closer to the goal**).
    - b. Move this state to the Closed List.
    - c. If the selected state is the goal state, the solution path is found (**stop**).
    - d. Generate all possible successor states (reachable by legal moves).
    - e. For each successor:
    - f. Calculate its g(n) score (actual cost to reach that state from the start).
    - g. Estimate its h(n) score using a heuristic (e.g., number of misplaced tiles or Manhattan distance).
    - h. Calculate its total cost f(n) = g(n) + h(n).
    - i. If the successor is not in the Closed List:
    - j. Add the successor to the Open List with its f(n) score.
    - k. If the successor is already in the Open List with a higher f(n) score (less promising path):
    - l. Update the Open List with the lower f(n) score for this successor (**explores more efficient paths first**).
3. **No Solution:** If the Open List becomes empty, there's no solution reachable from the starting configuration.

A* Algorithm example

f(n) = g(n) + h(n)

g(n): actual cost from start node to n

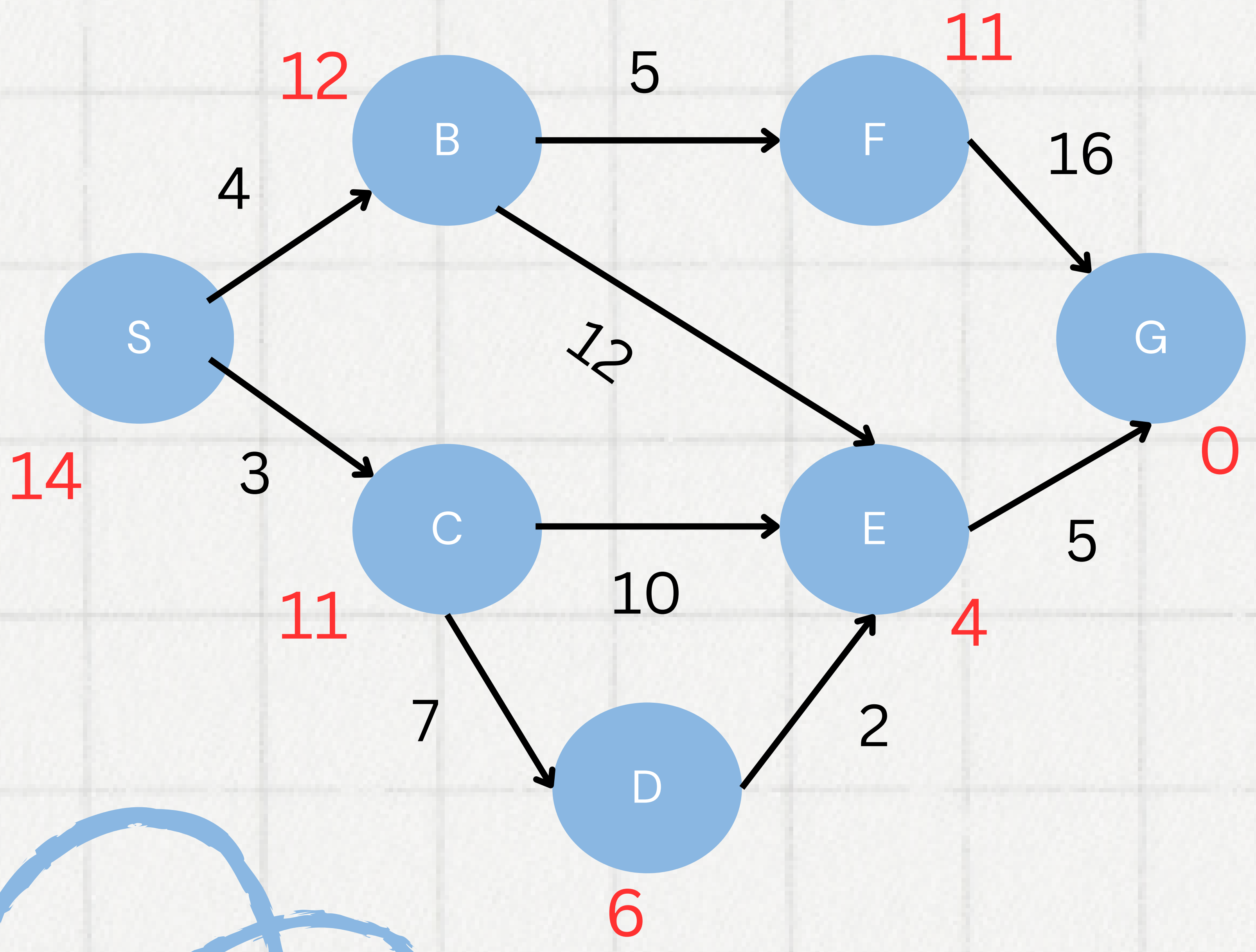
h(n): estimate cost from n to Goal node using Manhattan distance

T.C = O(V + E) = O (b^d) , S.C = O(b^d)

V = vertices , E = edges

b = branching factor / no. of children of a particular node

d = depth



S->S : f(S) = 0 + 14

S->B: f(B) = 4+12 = 16 , S->C : f(B) = 3 + 11 = **14**

SC->E : f(E)= 3+ 10 + 4 = 17 ,

SC->D : f(D)=3 +7+ 6 = **16**

SCD->E : f(E)= 3 +7 + 2+ 4 = **16**

SCDE->G : f(G)= 3 +7 + 2+ 5 = **17**

Solving 8-puzzle using A*

1	2	3
0	4	6
7	5	8

Initial State

1	2	3
4	5	6
7	8	0

Final State

g=0 ,

h=3
(no. of miss-placed tiles)

f = 0 + 3 = **3**

1	2	3
4	0	6
7	5	8

g=1 , h=2

f = 1 + 2 = **3**

1	2	3
4	5	6
7	0	8

g=2 , h=1

f = 2 + 1 = **3**

1	2	3
4	5	6
7	8	0

g=3 , h=0

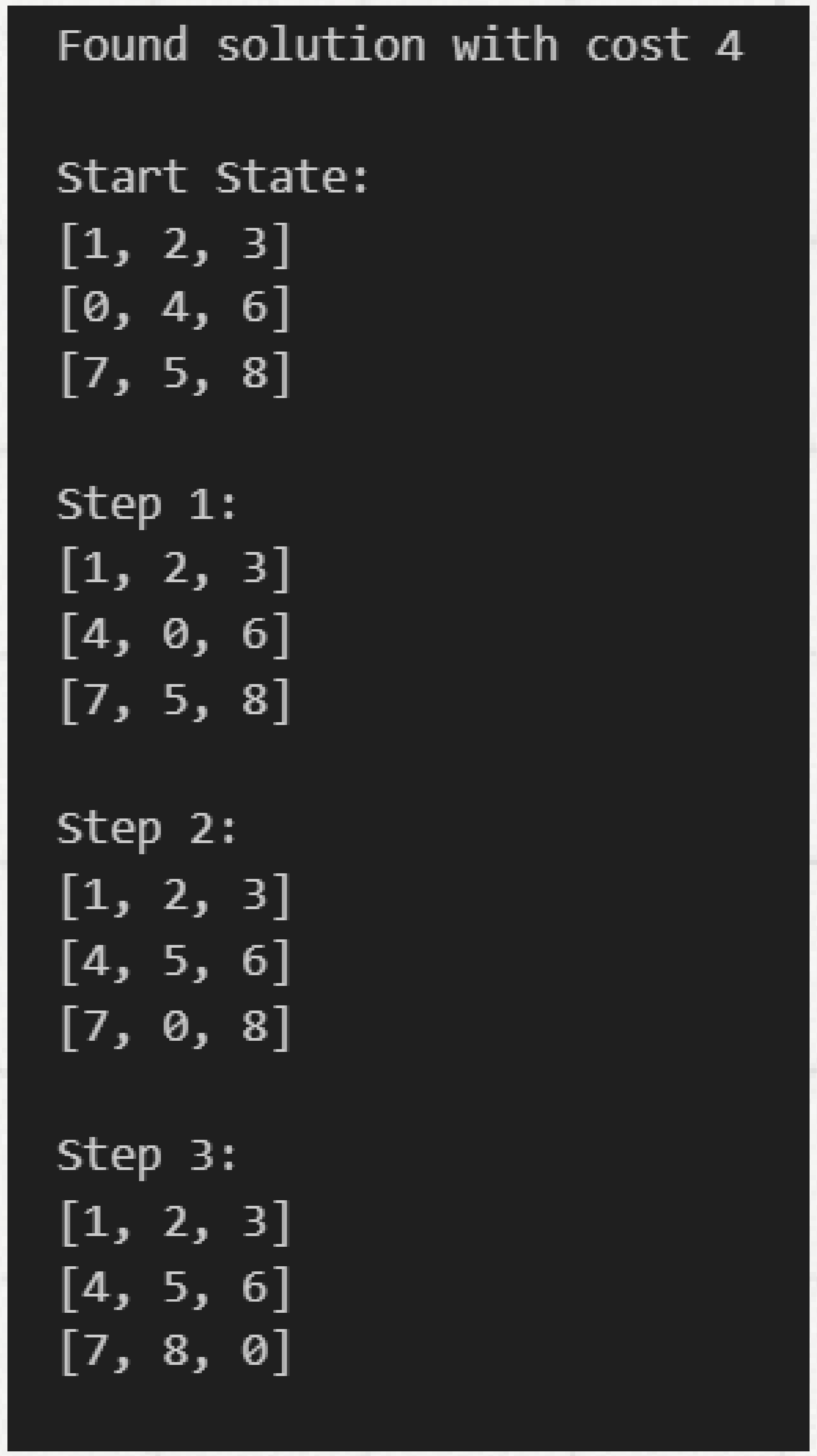
f = 3 + 0 = **3**

Data Structures

A priority queue implemented using the **heapq** module is utilized to efficiently explore states while solving the 8-puzzle problem with the A* search algorithm. Let's delve into how data structures like priority queues are utilized in the code:

1. **Priority Queue Initialization**:
    a. A priority queue is initialized using the **heapq** module's **heap** list. This list serves as the underlying data structure for the priority queue.
    b. The priority queue is used to store tuples **(cost, puzzle, path)**, where **cost** represents the cumulative cost of reaching the current state, **puzzle** represents the current state of the puzzle, and **path** represents the sequence of moves taken to reach the current state.
2. **Pushing States into the Priority Queue**:
    a. Initially, the priority queue is populated with the initial state of the puzzle, along with a cost of 0 and an empty path. This is done using **heapq.heappush(heap, (0, puzzle, []))**.
    b. States are pushed into the priority queue with their associated costs, ensuring that states with lower costs are explored first.
3. **Popping States from the Priority Queue**:
    a. States are popped from the priority queue using **heapq.heappop(heap)**. This operation retrieves the state with the lowest cost from the priority queue.
    b. The popped state represents the current state of the puzzle that needs to be explored further.
4. **Visited States**:
    a. A set named **visited** is used to keep track of visited states to avoid revisiting the same state multiple times. This helps in preventing infinite loops and redundant exploration.
    b. Before pushing a state into the priority queue, it is checked if the state has been visited before. If not, the state is added to the **visited** set, and then it is pushed into the priority queue.

Results



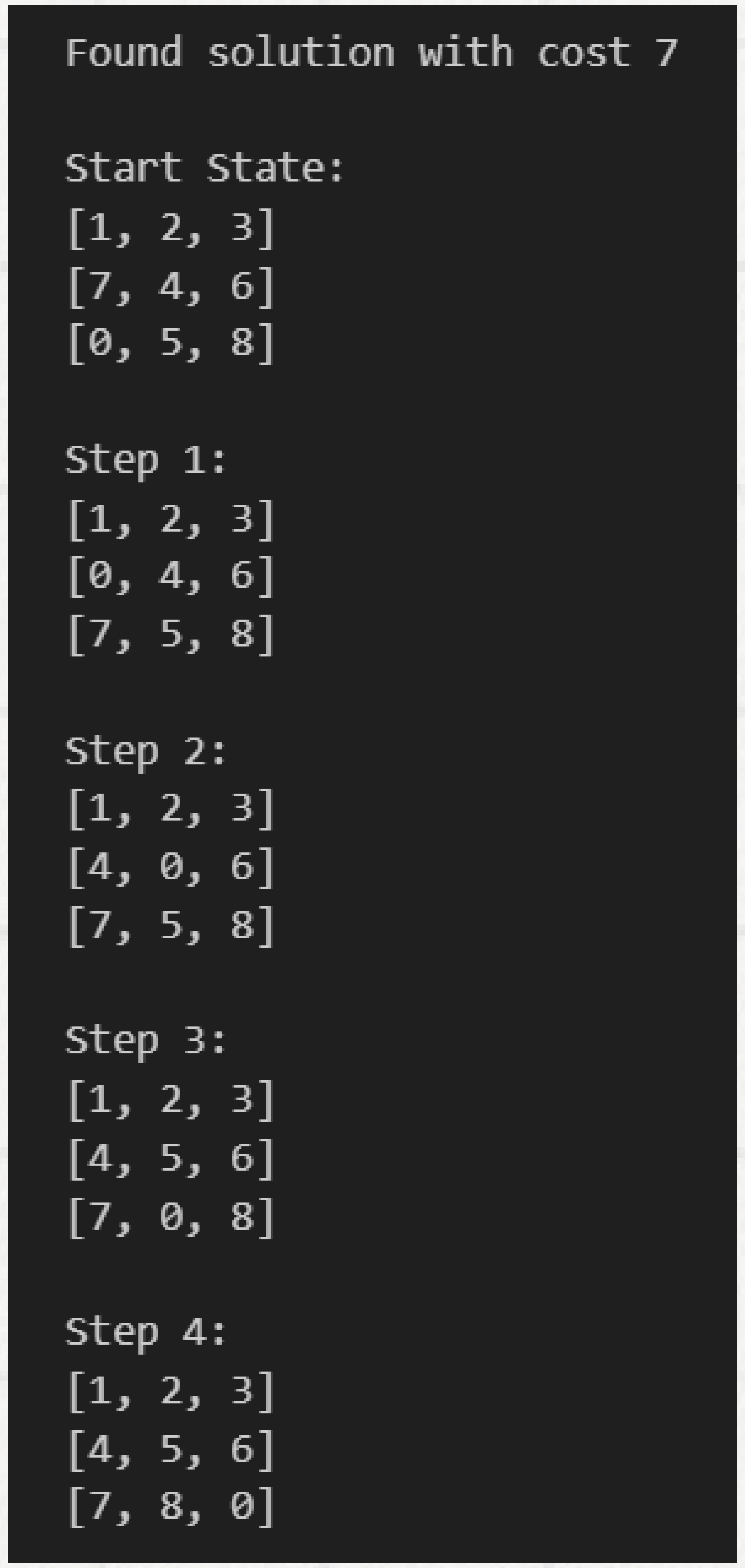
```
Found solution with cost 4

Start State:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

Step 1:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Step 2:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Step 3:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```



```
Found solution with cost 7

Start State:
[1, 2, 3]
[7, 4, 6]
[0, 5, 8]

Step 1:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

Step 2:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Step 3:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Step 4:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

Implementation Link

Advantages & Disadvantages

Pros	Cons
1. Finds the best path (if the guesstimate is good). 2. Saves time by focusing on promising paths first. 3. Adaptable to different problems with different guesstimates (heuristics).	1. Can be computationally expensive for very complex problems. 2. Relies on a good guesstimate (heuristic) - a bad one can lead it astray. 3. Might miss the best path if the guesstimate is overly optimistic.