CSE 356 – Artificial Intelligence

ASSIGMENT 1

Submitted by:

Ahmed Hani Ahmed 7387

Mohamed Hazem Hafez 7729

Mahmoud Haytham 7560

Due on:

March 7th, 2024

8 Puzzle Solution Using AI

Intro:

The 8-puzzle is a popular sliding puzzle that consists of a 3x3 grid with eight numbered square tiles and one blank tile. The tiles can be moved horizontally or vertically into the blank space, with the aim of rearranging them into a specific goal configuration. The goal configuration is typically a specified arrangement of the numbered tiles in ascending order, with the blank tile in a specific position.

The puzzle starts with the tiles randomly arranged within the grid, and the player's objective is to slide the tiles around the grid using the empty space to eventually achieve the goal configuration. The puzzle is often used as a recreational game or as a problem-solving exercise in artificial intelligence and computer science.

The 8-puzzle is a classic example of a search problem in artificial intelligence, and various algorithms such as breadth-first search, depth-first search, A* search, and others can be applied to find the optimal solution, i.e., the minimum number of moves required to solve the puzzle. Due to its simplicity and popularity, the 8-puzzle has been extensively studied in the field of computer science and serves as a foundational problem in various algorithmic domains.

$$parent = \begin{array}{|c|c|c|}\hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|}\hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

$$parent = \begin{array}{|c|c|c|}\hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|}\hline & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

Breath First Search (BFS):

Pseudocode:

```
function bfs(initial_state):
    queue = [initial_state]
    visited = set()
    num_expanded = 0
    expanded = []
    depth = 0

    while queue is not empty:
        current_state = queue.pop(0)

        if current_state not in visited:
            visited.add(current_state)
            num_expanded += 1
            expanded.append(current_state)

            if is_goal_state(current_state):
                return reconstruct_path(current_state), num_expanded, expanded, depth

            for neighbor_state in get_neighbors(current_state):
                if neighbor_state not in visited:
                    queue.append(neighbor_state)
                    depth = max(depth, len(queue))

    return None, num_expanded, expanded, depth
```

Explanation:

The breadth-first search (BFS) algorithm employed for solving the 8-puzzle problem begins by initializing a queue data structure, typically implemented using a First-In-First-Out (FIFO) approach, to manage the nodes representing different puzzle configurations. These nodes encapsulate the current state of the puzzle, along with information about the path taken to reach that state and the associated cost.

As the algorithm progresses, it maintains a set of visited states to prevent revisiting previously explored configurations, ensuring efficiency by avoiding redundant exploration. This set effectively serves as a form of memoization, allowing the algorithm to prune branches of the search tree that have already been traversed.

At each iteration, the algorithm dequeues a node from the front of the queue and expands it by generating all possible successor states, corresponding to valid moves from the current configuration. These successor states are then added to the queue for further exploration if they have not already been visited.

The process continues until either the goal state is encountered or the queue becomes empty, indicating that all reachable states have been explored. Upon reaching the goal state, the algorithm traces back the path from the initial configuration to the goal, recording the sequence of moves taken and the associated cost.

Throughout the search, the algorithm dynamically updates counters to keep track of the number of nodes expanded and the maximum depth reached, providing insights into the computational complexity of the search process.

Overall, the BFS algorithm systematically explores the search space in a breadth-first manner, gradually expanding the frontier of reachable states until a solution is found. Its ability to guarantee an optimal solution, along with its efficient memory usage and completeness, makes it a popular choice for solving the 8-puzzle problem and similar combinatorial optimization tasks.

Depth First Search (DFS):

Pseudocode:

```
function dfs(initial_state):
    stack = [PuzzleNode(initial_state)]
    visited = set()
    num_expanded = 0
    expanded = []
    depth = 0
```

```
while stack is not empty:
    current_node = stack.pop()

    if current_node.g > depth:
        depth = current_node.g

    visited.add(tuple(map(tuple, current_node.state)))
    num_expanded += 1
    expanded.append(current_node)

    if current_node.is_goal():
        path = []
        cost = 0

        while current_node is not None:
            path.append(current_node.state)
            cost += 1
            current_node = current_node.parent

        return reverse(path), cost, num_expanded, expanded, depth

    for neighbor_state in get_neighbors(current_node):
        neighbor_node = PuzzleNode(neighbor_state, current_node, g=current_node.g + 1)

        if tuple(map(tuple, neighbor_state)) not in visited:
            stack.append(neighbor_node)
            visited.add(tuple(map(tuple, neighbor_state)))
return None, None, None, None, None
```

Explanation:

The depth-first search (DFS) algorithm presented here tackles the 8-puzzle problem by systematically exploring potential solutions in a depth-first manner. It initializes a stack data structure, which follows the Last-In-First-Out (LIFO) principle, to manage puzzle nodes for exploration. The initial state of the puzzle is placed onto the stack to kick off the search process.

To ensure efficiency and prevent redundant exploration, the algorithm maintains a set to track visited states. This set helps avoid revisiting configurations that have already been explored, effectively pruning branches of the search tree.

The algorithm employs a loop that continues until the stack is empty, indicating that all potential states have been explored. Within each iteration, the algorithm pops a node from the top of the stack to examine it. It then generates successor states by exploring valid moves from the current configuration of the puzzle. These successor states are pushed onto the stack for further exploration if they have not been visited before.

Throughout the search process, the algorithm dynamically updates a depth variable to keep track of the maximum depth reached during exploration. If a deeper level is encountered, the depth variable is adjusted accordingly.

Upon encountering the goal state, the algorithm traces back the path from the initial configuration to the goal. It accomplishes this by recording the sequence of moves taken and the associated cost. This information provides insights into the optimal solution and the steps required to reach it.

An essential aspect of the DFS algorithm is its ability to handle search spaces with varying depths effectively. By prioritizing exploration of paths that extend deeply into the search space, DFS may discover solutions more quickly compared to other search strategies, especially when solutions are located deep within the search tree.

Overall, the DFS algorithm efficiently explores potential solutions to the 8-puzzle problem, leveraging a stack-based approach to traverse the search space in a depth-first manner. Its effectiveness lies in its ability to maintain completeness while efficiently managing memory and exploration.

A*:

pseudocode:

```
function euclidean_distance(state):
    distance = 0
    for i = 0 to 2:
        for j = 0 to 2:
            if state[i][j] ≠ 0:
                goal_row = (state[i][j]) // 3
```

```
            goal_col = (state[i][j]) % 3
            distance += sqrt((i - goal_row)^2 + (j - goal_col)^2)
    return distance

function manhattan_distance(state):
    distance = 0
    for i = 0 to 2:
        for j = 0 to 2:
            if state[i][j] ≠ 0:
                goal_row = (state[i][j]) // 3
                goal_col = (state[i][j]) % 3
                distance += |i - goal_row| + |j - goal_col|
    return distance

function astar_search(initial_state, f):
    frontier = priority_queue()
    frontier.push(initial_state)
    visited = set()
    cost = 0
    num_expanded = 0
    expanded = []
    depth = 0

    while frontier is not empty:
        current_node = frontier.pop()
        if current_node.g > depth:
            depth = current_node.g
        visited.add(current_node)
        num_expanded += 1
        expanded.append(current_node)

        if current_node.state not in visited:
            visited.add(current_node.state)
            for neighbor_state in get_neighbors(current_node):
                if f == 1:
                    neighbor = PuzzleNode(neighbor_state, current_node, g=current_node.g+1,
h=manhattan_distance(neighbor_state))
                else:
                    neighbor = PuzzleNode(neighbor_state, current_node, g=current_node.g+1,
h=euclidean_distance(neighbor_state))
                frontier.push(neighbor)

        if current_node.is_goal():
            path = []
            while current_node is not None:
                path.append(current_node.state)
                current_node = current_node.parent
```

```
            cost += 1
        return reverse(path), cost, num_expanded, expanded, depth

    return None, None, None, None, None
```

Explanation:

The astar_search function implements the A* search algorithm, a best-first search algorithm that combines the benefits of Depth First Search algorithm and greedy search. It utilizes a priority queue to explore nodes based on their combined cost (f = g + h) where g is the cost to reach the current node from the start state and h is the heuristic estimate of the cost to reach the goal from the current node.

During the search process, the algorithm maintains a frontier of nodes yet to be explored, expanding nodes with the lowest f value first. It also keeps track of visited nodes to avoid revisiting already explored states. The choice between the Manhattan and Euclidean distance heuristics is determined by the parameter f passed to the function.

The euclidean_distance function calculates the Euclidean distance heuristic for a given puzzle state by iterating over each tile and computing its straight-line distance from its current position to its goal position. This heuristic provides an estimate of the total cost required to reach the goal state, assuming optimal moves are made.

Similarly, the manhattan_distance function calculates the Manhattan distance heuristic by summing the horizontal and vertical distances of each tile from its current position to its goal position. This heuristic is widely used in grid-based pathfinding problems like the 8-puzzle due to its effectiveness and simplicity.

Once the goal state is found, the algorithm backtracks to reconstruct the optimal path, recording the sequence of moves and their associated costs. The function returns the optimal path, total cost,

number of nodes expanded during the search, list of expanded nodes, and maximum depth reached during the search.

Overall, this implementation provides a comprehensive framework for efficiently solving the 8-puzzle problem using A* search with heuristic estimates. The choice between heuristics offers flexibility in optimizing the search process based on the problem characteristics and computational resources available, ultimately leading to an effective and accurate solution.

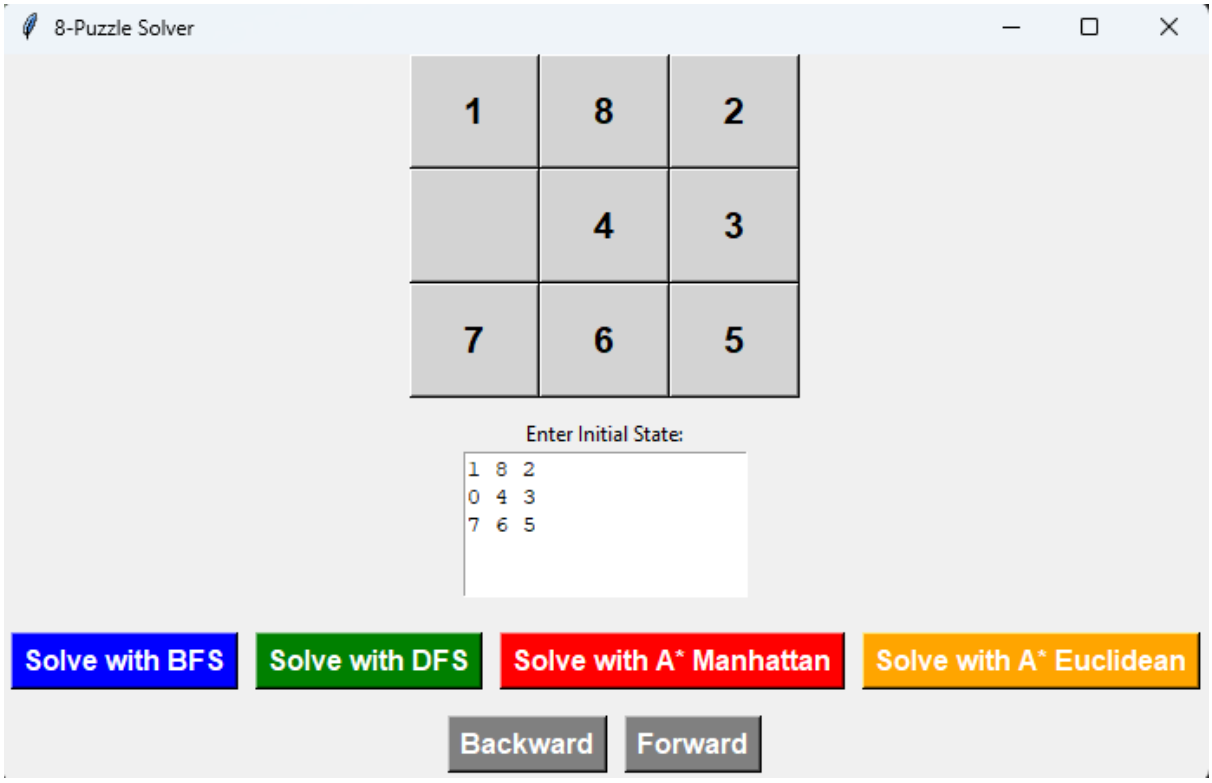
How it works:

First: enter the initial state that you want to solve.

Second: choose the algorithm you want to solve the game with by clicking on its button.

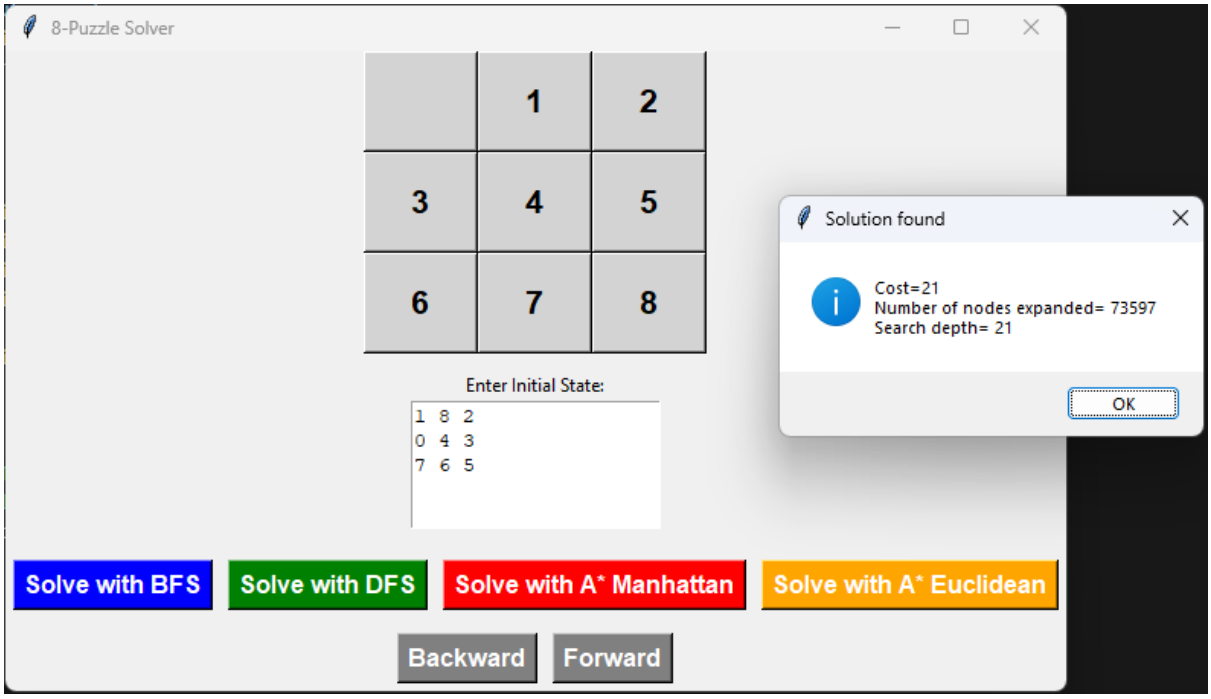
Third: after the algorithm gives the output it will show the solving path on the puzzle tiles.

Fourth: after the animation ends you can navigate between the solving states using the forward and backward buttons.



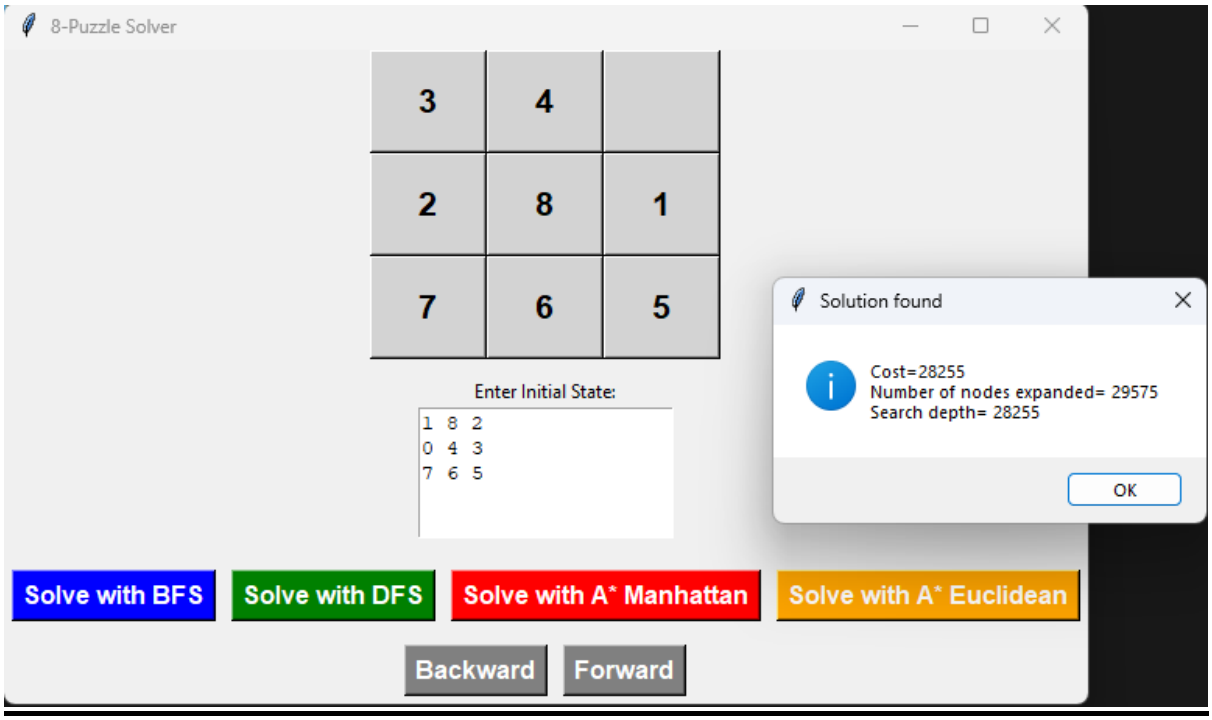
Test Cases:

BFS Solution:



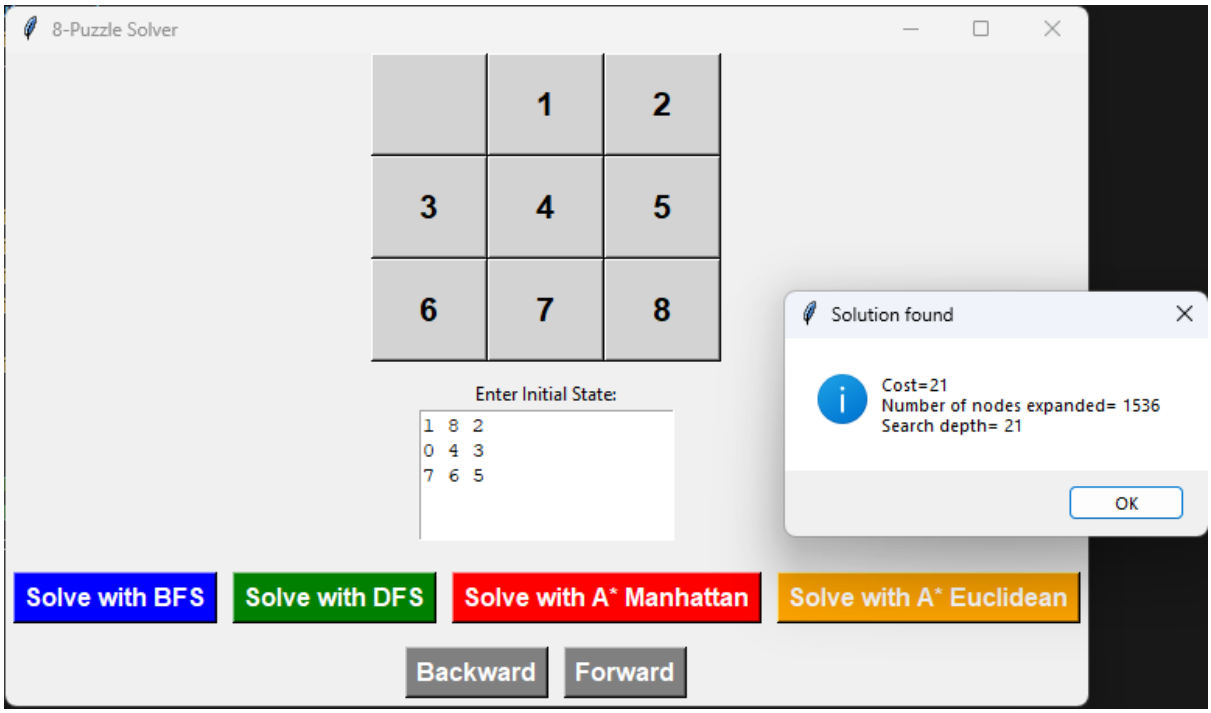

```
[1, 0, 2]
[3, 4, 5]
[6, 7, 8]
-----------
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
-----------
BFS Cost=21
Number of nodes expanded= 73597
Search depth= 21
Time taken: 2.0802841186523438 seconds
```

DFS Solution:



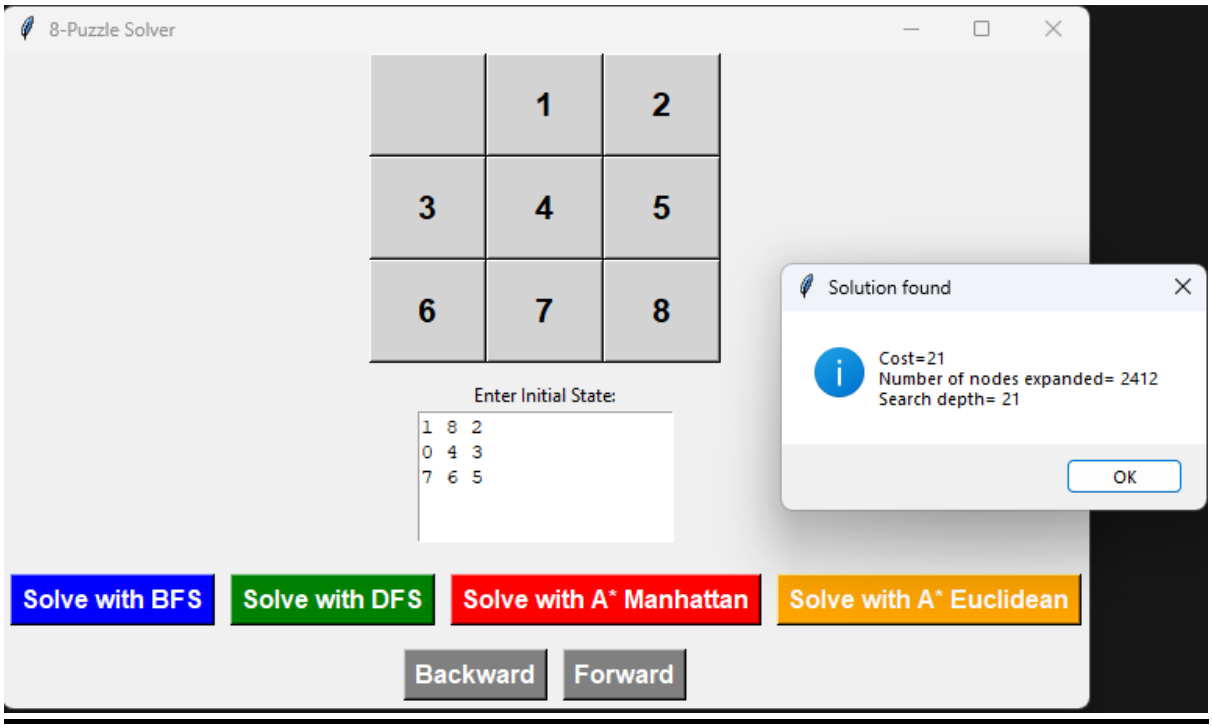

```
[3, 1, 2]
[0, 4, 5]
[6, 7, 8]
-----------
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
-----------
DFS Cost=28255
Number of nodes expanded= 29575
Search depth= 28255
Time taken: 0.7739701271057129 seconds
```

A* Manhattan Solution:




```
[1, 0, 2]
[3, 4, 5]
[6, 7, 8]
-----------
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
-----------
A star manhattan Cost=21
Number of nodes expanded= 1536
Search depth= 21
Time taken: 0.02406930923461914 seconds
```

A* Euclidean Solution:



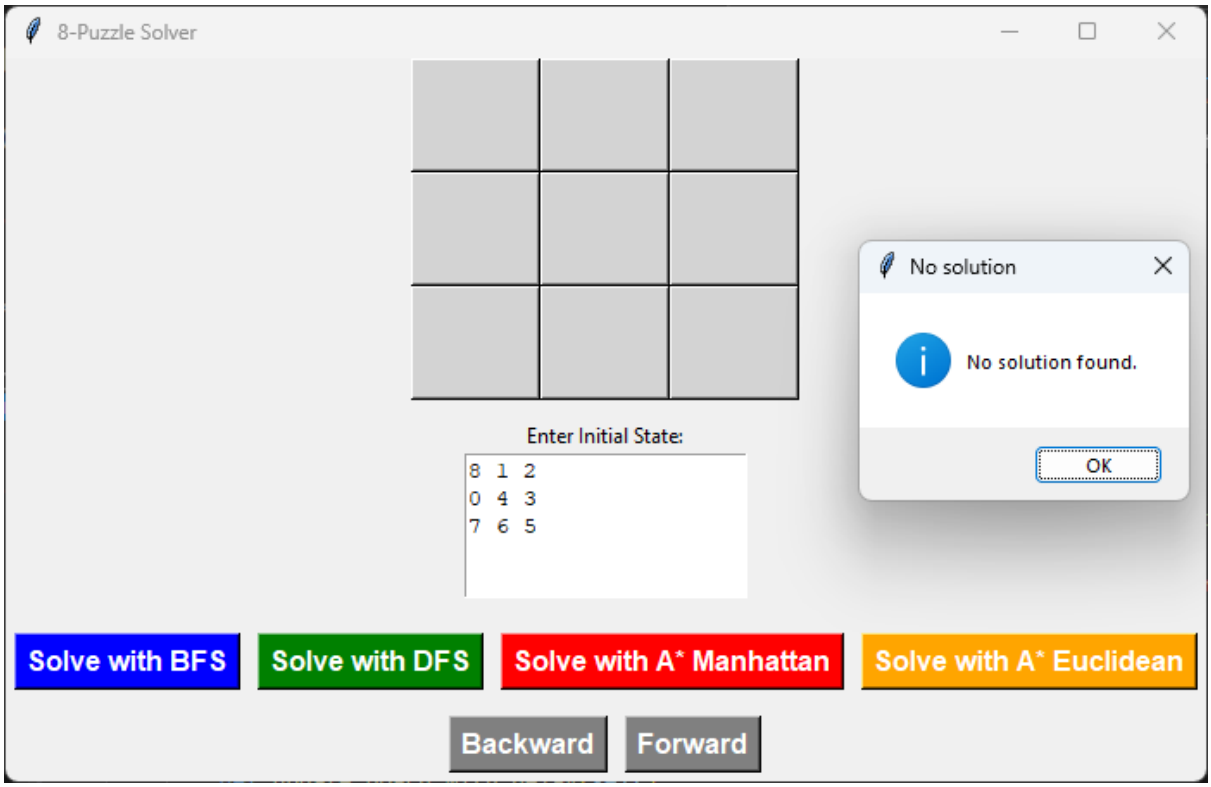

```
[3, 1, 2]
[0, 4, 5]
[6, 7, 8]
-----------
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
-----------
A star euclidean Cost=21
Number of nodes expanded= 2412
Search depth= 21
Time taken: 0.07199764251708984 seconds
```

Observations:

According to the results given results the best algorithm is the A star according to time taken and number of nodes expanded and using Manhattan distance is slightly better than using the Euclidean distance. The BFS takes most time while the DFS expands most nodes.

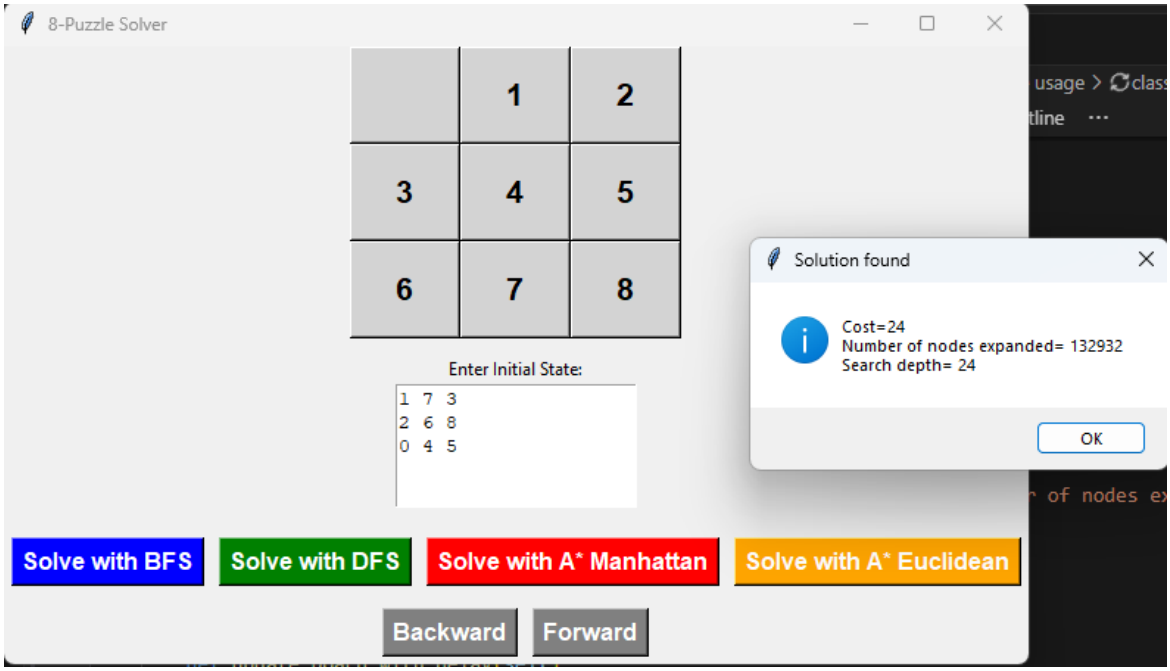
More test cases:

Testing a case that has no solution:



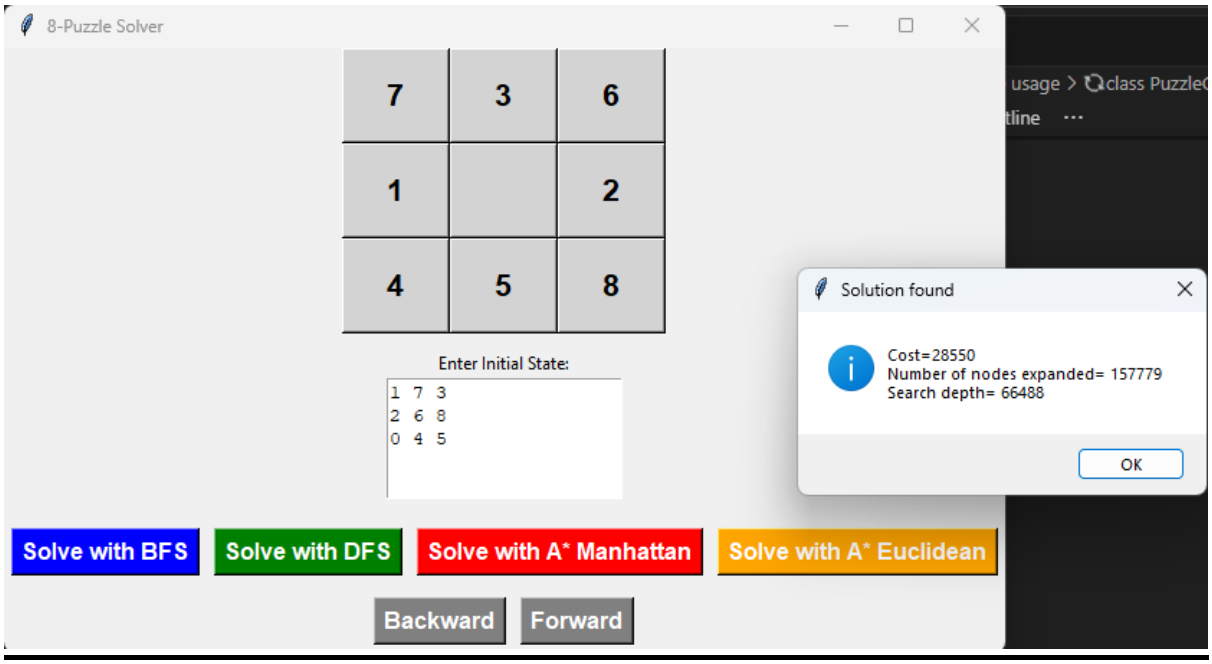
Another solvable case for demonstration

BFS Solution:



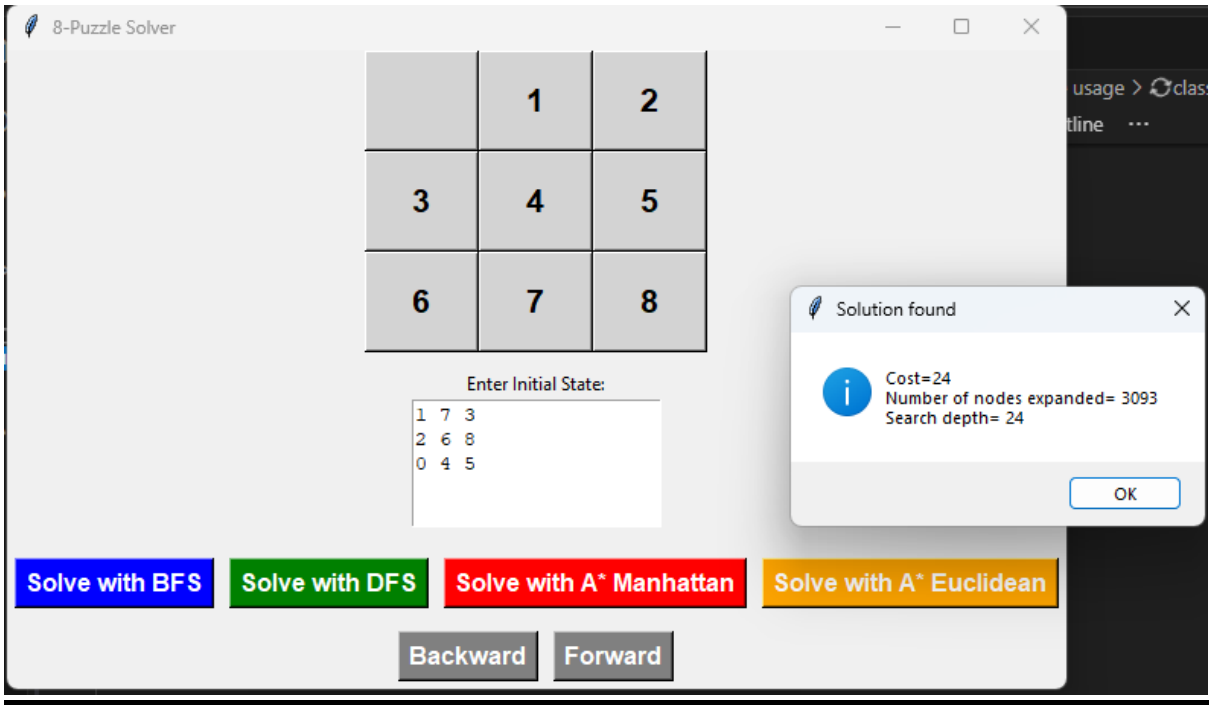
```
-----------
[3, 1, 2]
[0, 4, 5]
[6, 7, 8]
-----------
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
-----------
BFS Cost=24
Number of nodes expanded= 132932
Search depth= 24
Time taken: 2.8912429809570312 seconds
```

DFS Solution:



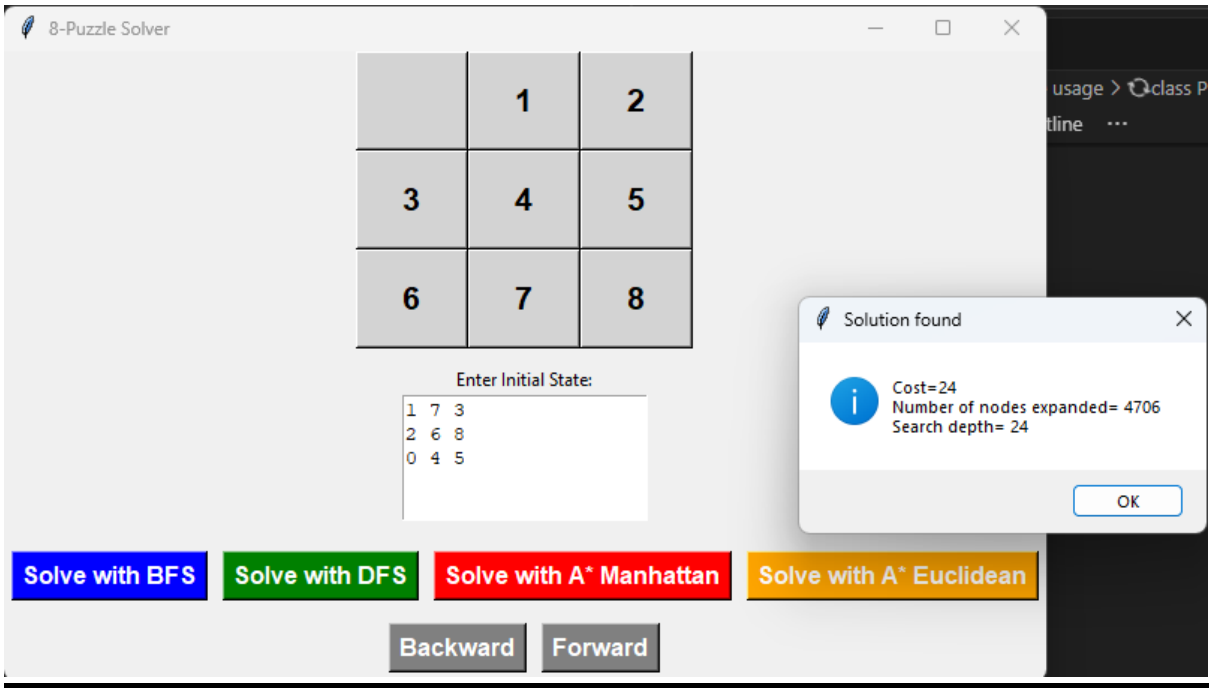

```
-----------
[3, 1, 2]
[0, 4, 5]
[6, 7, 8]
-----------
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
-----------
DFS Cost=28550
Number of nodes expanded= 157779
Search depth= 66488
Time taken: 2.8671720027923584 seconds
```

A* Manhattan Solution:




```
-----------
[3, 1, 2]
[0, 4, 5]
[6, 7, 8]
-----------
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
-----------
A star manhattan Cost=24
Number of nodes expanded= 3093
Search depth= 24
Time taken: 0.06467294692993164 seconds
```

A* Euclidean Solution:




```
-----------
[3, 1, 2]
[0, 4, 5]
[6, 7, 8]
-----------
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
-----------
A star euclidean Cost=24
Number of nodes expanded= 4706
Search depth= 24
Time taken: 0.12220954895019531 seconds
```