Solving the New 8-Puzzle with Bidirectional A* and Custom Heuristic

1 Problem:

The new 8-puzzle has eight tiles as usual, but each tile may move Up, Up-Right, Right, Down-Right, Down, Down-Left, Left, and Up-Left. The cost of a regular move (Up, Down, Left, or Right) is 1, and the cost of a diagonal move (Up-Right, Down-Right, Down-Left, and Up-Left) is 1.4 (approximately equal to $\sqrt{2} \approx 1.414$).

2 Solution:

The classical 8-puzzle problem is a well-known problem in Artificial Intelligence and heuristic search, where a 3×3 board contains 8 numbered tiles and one empty space. The objective is to slide the tiles using legal moves to reach a predefined goal configuration from a given initial state. Traditionally, only the four cardinal moves (Up, Down, Left, Right) are allowed, each with a unit cost.

In this extended version of the 8-puzzle, diagonal moves are permitted as well: Up-Right, Down-Right, Down-Left, and Up-Left. While cardinal moves retain a cost of 1, diagonal moves are assigned a cost of approximately 1.4, a close approximation of $\sqrt{2}$, representing the Euclidean cost for diagonal traversal. This increases the complexity and diversity of the state space, necessitating more sophisticated algorithms to efficiently explore the search space.

This study utilizes the **Bidirectional A* Search** algorithm for solving the extended 8-puzzle problem. Bidirectional A* enhances the classical A* approach by conducting two simultaneous searches—one forward from the initial state and one backward from the goal state—with the aim of meeting in the middle. The convergence of both searches indicates a viable solution path and significantly reduces the number of explored nodes compared to unidirectional search.

Methodology

Each puzzle state is represented as a 3×3 matrix configuration. Legal actions include all eight possible directions: Up, Down, Left, Right, Up-Right, Down-Right, Down-Left, and Up-Left. The cost of each move depends on its direction; cardinal directions (Up, Down, Left, Right) each have a cost of 1, while diagonal moves have a cost of 1.4. The initial and

goal states used for this problem are presented below.

Initial State

$$\begin{bmatrix} 1 & 3 & 0 \\ 8 & 2 & 6 \\ 7 & 4 & 5 \end{bmatrix}$$

Goal State

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

The Bidirectional A* algorithm improves efficiency by simultaneously expanding nodes from both the initial and goal states using two coordinated A* searches. Each direction maintains its own priority queue and employs a heuristic to estimate the cost to reach the target state. In this implementation, the Manhattan Distance heuristic is employed, which calculates the sum of the distances of each tile from its goal position, considering only horizontal and vertical movements.

Each state in the search tracks the current tile configuration, its parent state, the accumulated cost to reach it (denoted as g), and the estimated remaining cost to the goal (h). The total cost function is $f(n) = g(n) + h(n)$, and the node with the smallest f-value is expanded next. The search begins with two frontiers: one from the initial state and one from the goal state. Nodes are alternately expanded from each frontier. For every expansion, all legal moves from the current state are generated, their costs are computed, and if the resulting state has not yet been visited or a better path is found, it is added to the priority queue. The process continues until a state is encountered that appears in both frontiers, indicating that the two searches have converged. At this point, the shortest path is reconstructed by tracing back from the meeting point to both the initial and goal states.

The provided code structure reflects this logic. The *MOVES* array defines all possible movement directions and their associated costs. The function *is_valid_move(x, y)* ensures that attempted moves remain within the bounds of the 3×3 grid. The function *get_successors(state)* generates all valid successor states from a given state by applying the legal moves and computing the resulting costs. The heuristic function *manhattan_distance(state, goal)* calculates the estimated distance from the current state to the goal configuration. The main logic resides in the *bi_directional_astar(initial, goal)* function, which performs the search by managing the forward and backward queues, expanding nodes, and checking for convergence.

The expected output of this algorithm includes the full search tree with nodes annotated by their expansion order and cumulative path costs. Upon convergence, the final solution path is reconstructed and returned as the optimal sequence of moves from the initial to the goal state.

Graduate students enrolled in CSc 6810 are required to improve upon this baseline method by introducing novel extensions. Such enhancements may include adaptive or domain-specific heuristics, dynamic weighting strategies, or cost functions that more precisely model diagonal movement complexity. These innovations aim to further reduce the search space and improve solution efficiency in highly branched or complex puzzle configurations.

Explanation of `get_successors` Function

```python
def get_successors(state):
    # Initialize an empty list to store valid successor states and their costs
    successors = []

    # Iterate through each cell in the 3x3 puzzle grid
    for i in range(3):
        for j in range(3):
            # Check if the current tile is the empty space (0)
            if state[i][j] == 0:
                # Explore all possible movements from the MOVES list
                for dx, dy in MOVES:
                    # Compute the new coordinates after moving
                    new_x, new_y = i + dx, j + dy

                    # Check if the new coordinates are within puzzle bounds
                    if is_valid_move(new_x, new_y):
                        # Determine move cost: diagonal = DIAGONAL_COST, else 1.0
                        cost = DIAGONAL_COST if (dx != 0 and dy != 0) else 1.0

                        # Make a deep copy of the state to avoid mutation
                        new_state = [list(row) for row in state]

                        # Swap the empty tile with the adjacent tile
                        new_state[i][j], new_state[new_x][new_y] = \
                            new_state[new_x][new_y], new_state[i][j]

                        # Convert the state to a hashable tuple and append with cost
                        successors.append((tuple(map(tuple, new_state)), cost))

    # Return the list of all valid successor states and their move costs
    return successors
```

Explanation of `manhattan_distance` Function

```python
def manhattan_distance(state, goal_state):
    # Calculate the Manhattan distance heuristic
    distance = 0

    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0:
                goal_x, goal_y = divmod(tile - 1, 3)
                distance += abs(i - goal_x) + abs(j - goal_y)

    return distance
```

The `manhattan_distance` function calculates the Manhattan distance heuristic between the current `state` and the `goal_state` in your sliding puzzle problem. Here's a breakdown of each line:

- `def manhattan_distance(state, goal_state):`

Defines a function named `manhattan_distance` that takes two arguments: `state` and `goal_state`.

- `distance = 0`

Initializes a variable `distance` to zero, to accumulate the total Manhattan distance between tiles in the current state and their goal positions.

- `for i in range(3):`
`for j in range(3):`

Nested loops iterate over each row (`i`) and column (`j`) of the 3x3 puzzle grid to examine every tile.

- `tile = state[i][j]`

Retrieves the tile value at position (i, j) in the current state.

- `if tile != 0:`

Skips the empty tile (represented by zero), considering only numbered tiles for distance calculation.

- `goal_x, goal_y = divmod(tile - 1, 3)`

Computes the goal coordinates $(goal\_x, goal\_y)$ for the tile using `divmod`. Since tiles are numbered from 1, subtract 1 to align with zero-based indexing.

- `distance += abs(i - goal_x) + abs(j - goal_y)`

Adds the Manhattan distance (sum of absolute horizontal and vertical offsets) from the current tile position to its goal position.

- `return distance`

Returns the total Manhattan distance as the heuristic estimate of how far the current state is from the goal state.

Explanation of `bi_directional_astar` Function

```python
def bi_directional_astar(initial_state, goal_state):
```

```
    forward_queue = [(manhattan_distance(initial_state, goal_state), 0, initial_state)]
    backward_queue = [(manhattan_distance(goal_state, initial_state), 0, goal_state)]

    forward_visited = {tuple(map(tuple, initial_state)): (0, None)}
    backward_visited = {tuple(map(tuple, goal_state)): (0, None)}

    found_solution = False

    while forward_queue and backward_queue and not found_solution:
        _, forward_cost, forward_state = heapq.heappop(forward_queue)
        _, backward_cost, backward_state = heapq.heappop(backward_queue)

        if forward_state == backward_state:
            found_solution = True

        print("Forward Queue:")
        for item in forward_queue:
            print(item)

        print("Backward Queue:")
        for item in backward_queue:
            print(item)

        for next_state, cost in get_successors(forward_state):
            if next_state not in forward_visited:
                forward_visited[next_state] = (forward_cost + cost, forward_state)
                forward_priority = forward_cost + cost + manhattan_distance(next_state, goal_state)
                heapq.heappush(forward_queue, (forward_priority, forward_cost + cost, next_state))

        for next_state, cost in get_successors(backward_state):
            if next_state not in backward_visited:
                backward_visited[next_state] = (backward_cost + cost, backward_state)
                backward_priority = backward_cost + cost + manhattan_distance(next_state, initial_state)
                heapq.heappush(backward_queue, (backward_priority, backward_cost + cost, next_state))

```

```
37      if found_solution:
38          path = []
39          current_state = forward_state
40
41          while current_state:
42              path.append(current_state)
43              forward_cost, current_state = forward_visited[
                    current_state]
44
45          path.reverse()
46
47          current_state = backward_state
48
49          while current_state:
50              path.append(current_state)
51              backward_cost, current_state = backward_visited[
                    current_state]
52
53          return path
```

The `bi_directional_astar` function implements a bidirectional A* search algorithm to find a path from the `initial_state` to the `goal_state` by simultaneously searching forward and backward. Here's the line-by-line explanation:

- `def bi_directional_astar(initial_state, goal_state):`

Defines the function accepting the starting and goal states of the puzzle.

- `forward_queue = [...]` and `backward_queue = [...]`

Initialize two priority queues (min-heaps) for forward and backward searches. Each entry contains: 1. Priority (cost + heuristic). 2. Cost so far. 3. Current state.

- `forward_visited` and `backward_visited` dictionaries store visited states with their cost and parent.

- `found_solution = False`

Tracks whether the searches have met.

- `while forward_queue and backward_queue and not found_solution:`

Runs the main loop until one queue is empty or a solution is found.

- `heapq.heappop(...)`

Pops the lowest priority state from each queue to explore.

- `if forward_state == backward_state:`

If both searches meet at the same state, sets `found_solution` to `True`.

- Printing the queues allows you to observe the search progress after each iteration.

- For each successor of `forward_state`, if unvisited, update `forward_visited`, calculate priority, and push onto `forward_queue`.

- For each successor of `backward_state`, similarly update `backward_visited` and `backward_queue`.

- Upon finding a solution: - Reconstruct the path by tracing parents in `forward_visited` from the meeting state backward. - Reverse the forward path. - Continue tracing parents in `backward_visited` forward. - Return the combined path.

Explanation of the Call to `bi_directional_astar`

```
path = bi_directional_astar(tuple(map(tuple, initial_state)),
    tuple(map(tuple, goal_state)))
```

- `tuple(map(tuple, initial_state))` converts the 2D list `initial_state` into a nested tuple, making it hashable and usable as a dictionary key.

- `tuple(map(tuple, goal_state))` similarly converts the `goal_state`.

- `bi_directional_astar(...)` calls the bidirectional A* function with these tuple states.

- The resulting `path` variable will hold the solution path connecting the initial and goal states, if one exists.

*Search Tree with Marked Expansion Order Numbers and Costs

Due to the high complexity and size of the 8-puzzle search tree, it is impractical to display the entire structure graphically. Instead, we provide the ordered contents of the forward and backward queues as the search progresses. Each entry contains:

1. The total priority (heuristic + cost), 2. The path cost from the parent node, 3. The 3x3 puzzle state.

Two separate search trees are maintained: - One for the forward search (initial state to goal), - One for the backward search (goal to initial state).

The progression of both search directions is presented below step-by-step.

Forward Queue1: Backward Queue1:

Forward Queue2: (9.4, 1.4, ((2, 1, 3), (8, 0, 6), (7, 4, 5)))
(10.0, 1.0, ((8, 1, 3), (0, 2, 6), (7, 4, 5)))

Backward Queue2: (8.0, 1.0, ((1, 2, 3), (0, 8, 4), (7, 6, 5)))
(8.0, 1.0, ((1, 2, 3), (8, 4, 0), (7, 6, 5)))
(10.0, 1.0, ((1, 0, 3), (8, 2, 4), (7, 6, 5)))
(11.4, 1.4, ((0, 2, 3), (8, 1, 4), (7, 6, 5)))
(8.0, 1.0, ((1, 2, 3), (8, 6, 4), (7, 0, 5)))
(11.4, 1.4, ((1, 2, 3), (8, 7, 4), (0, 6, 5)))
(11.4, 1.4, ((1, 2, 0), (8, 3, 4), (7, 6, 5)))

Forward Queue3: (9.4, 1.4, ((2, 1, 3), (8, 0, 6), (7, 4, 5)))
(10.0, 1.0, ((8, 1, 3), (0, 2, 6), (7, 4, 5)))
(9.4, 2.4, ((1, 8, 3), (0, 2, 6), (7, 4, 5)))
(11.4, 2.4, ((1, 6, 3), (8, 2, 0), (7, 4, 5)))
(10.0, 2.0, ((1, 3, 0), (8, 2, 6), (7, 4, 5)))

Backward Queue3: (8.0, 1.0, ((1, 2, 3), (0, 8, 4), (7, 6, 5)))
(8.0, 1.0, ((1, 2, 3), (8, 4, 0), (7, 6, 5)))
(10.0, 1.0, ((1, 0, 3), (8, 2, 4), (7, 6, 5)))

(9.4, 2.4, ((1, 2, 3), (8, 5, 0), (7, 6, 4)))
(8.0, 1.0, ((1, 2, 3), (8, 6, 4), (7, 0, 5)))
(11.4, 1.4, ((1, 2, 3), (8, 7, 4), (0, 6, 5)))
(11.4, 1.4, ((1, 2, 0), (8, 3, 4), (7, 6, 5)))
(11.4, 1.4, ((0, 2, 3), (8, 1, 4), (7, 6, 5)))

Forward Queue4: (8.0, 3.0, ((1, 2, 3), (0, 8, 6), (7, 4, 5)))
(8.0, 3.0, ((1, 2, 3), (8, 4, 6), (7, 0, 5)))
(9.4, 2.4, ((1, 8, 3), (0, 2, 6), (7, 4, 5)))
(9.4, 1.4, ((2, 1, 3), (8, 0, 6), (7, 4, 5)))
(10.0, 2.0, ((1, 3, 0), (8, 2, 6), (7, 4, 5)))
(11.4, 3.4, ((0, 2, 3), (8, 1, 6), (7, 4, 5)))
(10.0, 3.0, ((1, 2, 3), (8, 6, 0), (7, 4, 5)))
(11.4, 2.4, ((1, 6, 3), (8, 2, 0), (7, 4, 5)))
(10.0, 1.0, ((8, 1, 3), (0, 2, 6), (7, 4, 5)))
(11.4, 3.4, ((1, 2, 3), (8, 7, 6), (0, 4, 5)))
(11.4, 3.4, ((1, 2, 0), (8, 3, 6), (7, 4, 5)))

Backward Queue4: (8.0, 1.0, ((1, 2, 3), (0, 8, 4), (7, 6, 5)))
(8.0, 1.0, ((1, 2, 3), (8, 4, 0), (7, 6, 5)))
(10.0, 1.0, ((1, 0, 3), (8, 2, 4), (7, 6, 5)))
(9.4, 2.4, ((1, 2, 3), (8, 5, 0), (7, 6, 4)))
(8.0, 1.0, ((1, 2, 3), (8, 6, 4), (7, 0, 5)))
(11.4, 1.4, ((1, 2, 3), (8, 7, 4), (0, 6, 5)))
(11.4, 1.4, ((1, 2, 0), (8, 3, 4), (7, 6, 5)))
(11.4, 1.4, ((0, 2, 3), (8, 1, 4), (7, 6, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 0, 4), (7, 5, 6)))
(8.8, 3.8, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(9.4, 3.4, ((1, 2, 3), (8, 5, 4), (0, 7, 6)))

Forward Queue5: (8.0, 3.0, ((1, 2, 3), (8, 4, 6), (7, 0, 5)))
(9.4, 1.4, ((2, 1, 3), (8, 0, 6), (7, 4, 5)))
(9.4, 2.4, ((1, 8, 3), (0, 2, 6), (7, 4, 5)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 6), (7, 0, 4)))
(10.0, 2.0, ((1, 3, 0), (8, 2, 6), (7, 4, 5)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(10.0, 3.0, ((1, 2, 3), (8, 6, 0), (7, 4, 5)))
(11.4, 2.4, ((1, 6, 3), (8, 2, 0), (7, 4, 5)))
(10.0, 1.0, ((8, 1, 3), (0, 2, 6), (7, 4, 5)))
(11.4, 3.4, ((1, 2, 3), (8, 7, 6), (0, 4, 5)))
(11.4, 3.4, ((1, 2, 0), (8, 3, 6), (7, 4, 5)))
(11.4, 3.4, ((0, 2, 3), (8, 1, 6), (7, 4, 5)))

Backward Queue5: (8.0, 1.0, ((1, 2, 3), (8, 4, 0), (7, 6, 5)))
(8.0, 1.0, ((1, 2, 3), (8, 6, 4), (7, 0, 5)))
(8.8, 4.8, ((0, 2, 3), (1, 5, 4), (7, 8, 6)))
(9.4, 2.4, ((1, 2, 3), (8, 5, 0), (7, 6, 4)))
(8.8, 3.8, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(10.0, 1.0, ((1, 0, 3), (8, 2, 4), (7, 6, 5)))
(8.8, 4.8, ((1, 2, 3), (5, 0, 4), (7, 8, 6)))
(11.4, 1.4, ((0, 2, 3), (8, 1, 4), (7, 6, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 0, 4), (7, 5, 6)))
(8.8, 4.8, ((1, 2, 3), (7, 5, 4), (0, 8, 6)))
(9.4, 3.4, ((1, 2, 3), (8, 5, 4), (0, 7, 6)))
(11.4, 1.4, ((1, 2, 3), (8, 7, 4), (0, 6, 5)))
(10.2, 5.2, ((1, 0, 3), (2, 5, 4), (7, 8, 6)))
(11.4, 1.4, ((1, 2, 0), (8, 3, 4), (7, 6, 5)))

Forward Queue6: (8.0, 3.0, ((1, 2, 3), (8, 4, 6), (7, 0, 5)))
(9.4, 1.4, ((2, 1, 3), (8, 0, 6), (7, 4, 5)))
(9.4, 2.4, ((1, 8, 3), (0, 2, 6), (7, 4, 5)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 6), (7, 0, 4)))
(10.0, 2.0, ((1, 3, 0), (8, 2, 6), (7, 4, 5)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(10.0, 3.0, ((1, 2, 3), (8, 6, 0), (7, 4, 5)))
(10.0, 4.0, ((1, 2, 3), (7, 8, 6), (0, 4, 5)))
(10.0, 1.0, ((8, 1, 3), (0, 2, 6), (7, 4, 5)))
(11.4, 3.4, ((1, 2, 3), (8, 7, 6), (0, 4, 5)))
(11.4, 3.4, ((1, 2, 0), (8, 3, 6), (7, 4, 5)))
(11.4, 3.4, ((0, 2, 3), (8, 1, 6), (7, 4, 5)))
(10.0, 4.0, ((0, 2, 3), (1, 8, 6), (7, 4, 5)))
(11.4, 4.4, ((1, 0, 3), (2, 8, 6), (7, 4, 5)))
(11.4, 2.4, ((1, 6, 3), (8, 2, 0), (7, 4, 5)))

Backward Queue6: (8.0, 1.0, ((1, 2, 3), (8, 6, 4), (7, 0, 5)))
(8.8, 3.8, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(8.8, 4.8, ((0, 2, 3), (1, 5, 4), (7, 8, 6)))
(9.4, 2.4, ((1, 2, 3), (6, 8, 4), (7, 0, 5)))
(8.8, 4.8, ((1, 2, 3), (7, 5, 4), (0, 8, 6)))
(10.0, 1.0, ((1, 0, 3), (8, 2, 4), (7, 6, 5)))
(8.8, 4.8, ((1, 2, 3), (5, 0, 4), (7, 8, 6)))
(9.4, 2.4, ((1, 2, 3), (8, 5, 0), (7, 6, 4)))
(9.4, 3.4, ((1, 2, 3), (8, 0, 4), (7, 5, 6)))
(10.0, 2.0, ((1, 2, 3), (7, 8, 4), (0, 6, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 5, 4), (0, 7, 6)))
(11.4, 1.4, ((1, 2, 3), (8, 7, 4), (0, 6, 5)))
(10.2, 5.2, ((1, 0, 3), (2, 5, 4), (7, 8, 6)))

(11.4, 1.4, ((1, 2, 0), (8, 3, 4), (7, 6, 5)))
(10.0, 2.0, ((0, 2, 3), (1, 8, 4), (7, 6, 5)))
(11.4, 2.4, ((1, 0, 3), (2, 8, 4), (7, 6, 5)))
(11.4, 1.4, ((0, 2, 3), (8, 1, 4), (7, 6, 5)))

Forward Queue7 : (7.4, 5.4, ((1, 2, 3), (4, 8, 6), (7, 5, 0)))
(8.0, 3.0, ((1, 2, 3), (8, 4, 6), (7, 0, 5)))
(9.4, 2.4, ((1, 8, 3), (0, 2, 6), (7, 4, 5)))
(9.4, 1.4, ((2, 1, 3), (8, 0, 6), (7, 4, 5)))
(10.0, 2.0, ((1, 3, 0), (8, 2, 6), (7, 4, 5)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(10.0, 3.0, ((1, 2, 3), (8, 6, 0), (7, 4, 5)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 6), (7, 0, 4)))
(9.4, 5.4, ((1, 2, 3), (4, 8, 6), (0, 7, 5)))
(11.4, 3.4, ((1, 2, 3), (8, 7, 6), (0, 4, 5)))
(11.4, 3.4, ((1, 2, 0), (8, 3, 6), (7, 4, 5)))
(11.4, 3.4, ((0, 2, 3), (8, 1, 6), (7, 4, 5)))
(10.0, 4.0, ((0, 2, 3), (1, 8, 6), (7, 4, 5)))
(11.4, 4.4, ((1, 0, 3), (2, 8, 6), (7, 4, 5)))
(11.4, 2.4, ((1, 6, 3), (8, 2, 0), (7, 4, 5)))
(10.0, 4.0, ((1, 2, 3), (7, 8, 6), (0, 4, 5)))
(10.8, 5.800000000000001, ((1, 2, 3), (4, 8, 0), (7, 6, 5)))
(10.0, 1.0, ((8, 1, 3), (0, 2, 6), (7, 4, 5)))

Backward Queue7 : (8.0, 1.0, ((1, 2, 3), (8, 6, 4), (7, 0, 5)))
(8.0, 2.0, ((1, 2, 3), (8, 4, 5), (7, 6, 0)))
(8.8, 4.8, ((0, 2, 3), (1, 5, 4), (7, 8, 6)))
(8.8, 3.8, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(8.8, 4.8, ((1, 2, 3), (7, 5, 4), (0, 8, 6)))
(10.0, 1.0, ((1, 0, 3), (8, 2, 4), (7, 6, 5)))
(8.8, 4.8, ((1, 2, 3), (5, 0, 4), (7, 8, 6)))
(9.4, 2.4, ((1, 2, 3), (8, 5, 0), (7, 6, 4)))
(9.4, 2.4, ((1, 2, 3), (6, 8, 4), (7, 0, 5)))
(10.0, 2.0, ((1, 2, 3), (7, 8, 4), (0, 6, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 5, 4), (0, 7, 6)))
(11.4, 1.4, ((1, 2, 3), (8, 7, 4), (0, 6, 5)))
(10.2, 5.199999999999999, ((1, 0, 3), (2, 5, 4), (7, 8, 6)))
(11.4, 1.4, ((1, 2, 0), (8, 3, 4), (7, 6, 5)))
(10.0, 2.0, ((0, 2, 3), (1, 8, 4), (7, 6, 5)))
(11.4, 2.4, ((1, 0, 3), (2, 8, 4), (7, 6, 5)))
(11.4, 1.4, ((0, 2, 3), (8, 1, 4), (7, 6, 5)))
(10.0, 2.0, ((1, 2, 0), (8, 4, 3), (7, 6, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 0, 4), (7, 5, 6)))
(11.4, 2.4, ((1, 0, 3), (8, 4, 2), (7, 6, 5)))

Forward Queue8 : (7.4, 5.4, ((1, 2, 3), (4, 8, 6), (7, 5, 0)))
(8.0, 3.0, ((1, 2, 3), (8, 4, 6), (7, 0, 5)))
(9.4, 2.4, ((1, 8, 3), (0, 2, 6), (7, 4, 5)))
(9.4, 1.4, ((2, 1, 3), (8, 0, 6), (7, 4, 5)))
(10.0, 2.0, ((1, 3, 0), (8, 2, 6), (7, 4, 5)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(10.0, 3.0, ((1, 2, 3), (8, 6, 0), (7, 4, 5)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 6), (7, 0, 4)))
(9.4, 5.4, ((1, 2, 3), (4, 8, 6), (0, 7, 5)))
(11.4, 3.4, ((1, 2, 3), (8, 7, 6), (0, 4, 5)))
(11.4, 3.4, ((1, 2, 0), (8, 3, 6), (7, 4, 5)))
(11.4, 3.4, ((0, 2, 3), (8, 1, 6), (7, 4, 5)))
(10.0, 4.0, ((0, 2, 3), (1, 8, 6), (7, 4, 5)))
(11.4, 4.4, ((1, 0, 3), (2, 8, 6), (7, 4, 5)))
(11.4, 2.4, ((1, 6, 3), (8, 2, 0), (7, 4, 5)))
(10.0, 4.0, ((1, 2, 3), (7, 8, 6), (0, 4, 5)))
(10.8, 5.800000000000001, ((1, 2, 3), (4, 8, 0), (7, 6, 5)))
(10.0, 1.0, ((8, 1, 3), (0, 2, 6), (7, 4, 5)))
(10.0, 4.0, ((1, 2, 3), (4, 8, 6), (7, 0, 5)))

Backward Queue8 : (7.4, 3.4, ((1, 2, 3), (8, 4, 6), (7, 5, 0)))
(8.0, 1.0, ((1, 2, 3), (8, 6, 4), (7, 0, 5)))
(8.8, 4.8, ((0, 2, 3), (1, 5, 4), (7, 8, 6)))
(8.8, 3.8, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(8.0, 2.0, ((1, 2, 3), (8, 4, 5), (7, 6, 0)))
(10.0, 1.0, ((1, 0, 3), (8, 2, 4), (7, 6, 5)))
(8.8, 4.8, ((1, 2, 3), (5, 0, 4), (7, 8, 6)))
(9.4, 2.4, ((1, 2, 3), (8, 5, 0), (7, 6, 4)))
(9.4, 2.4, ((1, 2, 3), (6, 8, 4), (7, 0, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 0, 6), (7, 4, 5)))
(8.8, 4.8, ((1, 2, 3), (7, 5, 4), (0, 8, 6)))
(11.4, 1.4, ((1, 2, 3), (8, 7, 4), (0, 6, 5)))
(10.2, 5.199999999999999, ((1, 0, 3), (2, 5, 4), (7, 8, 6)))
(11.4, 1.4, ((1, 2, 0), (8, 3, 4), (7, 6, 5)))
(10.0, 2.0, ((0, 2, 3), (1, 8, 4), (7, 6, 5)))
(11.4, 2.4, ((1, 0, 3), (2, 8, 4), (7, 6, 5)))
(11.4, 1.4, ((0, 2, 3), (8, 1, 4), (7, 6, 5)))
(10.0, 2.0, ((1, 2, 0), (8, 4, 3), (7, 6, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 0, 4), (7, 5, 6)))
(11.4, 2.4, ((1, 0, 3), (8, 4, 2), (7, 6, 5)))
(10.0, 2.0, ((1, 2, 3), (7, 8, 4), (0, 6, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 5, 4), (0, 7, 6)))

(9.4, 3.4, ((1, 2, 3), (8, 4, 6), (0, 7, 5)))

Forward Queue9 : (8.0, 3.0, ((1, 2, 3), (8, 4, 6), (7, 0, 5)))
(9.4, 1.4, ((2, 1, 3), (8, 0, 6), (7, 4, 5)))
(8.8, 7.800000000000001, ((1, 2, 3), (4, 5, 0), (7, 8, 6)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 6), (7, 0, 4)))
(9.4, 6.4, ((1, 2, 3), (4, 6, 0), (7, 8, 5)))
(8.8, 7.800000000000001, ((1, 2, 3), (4, 5, 6), (7, 0, 8)))
(10.0, 3.0, ((1, 2, 3), (8, 6, 0), (7, 4, 5)))
(10.0, 4.0, ((0, 2, 3), (1, 8, 6), (7, 4, 5)))
(9.4, 5.4, ((1, 2, 3), (4, 8, 6), (0, 7, 5)))
(10.0, 2.0, ((1, 3, 0), (8, 2, 6), (7, 4, 5)))
(10.8, 6.800000000000001, ((0, 2, 3), (4, 1, 6), (7, 8, 5)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(9.4, 2.4, ((1, 8, 3), (0, 2, 6), (7, 4, 5)))
(11.4, 4.4, ((1, 0, 3), (2, 8, 6), (7, 4, 5)))
(11.4, 2.4, ((1, 6, 3), (8, 2, 0), (7, 4, 5)))
(10.0, 4.0, ((1, 2, 3), (7, 8, 6), (0, 4, 5)))
(10.8, 5.800000000000001, ((1, 2, 3), (4, 8, 0), (7, 6, 5)))
(10.0, 1.0, ((8, 1, 3), (0, 2, 6), (7, 4, 5)))
(9.4, 6.4, ((1, 0, 3), (4, 2, 6), (7, 8, 5)))
(11.4, 3.4, ((1, 2, 3), (8, 7, 6), (0, 4, 5)))
(10.8, 6.800000000000001, ((1, 2, 0), (4, 3, 6), (7, 8, 5)))
(11.4, 3.4, ((1, 2, 0), (8, 3, 6), (7, 4, 5)))
(10.8, 6.800000000000001, ((1, 2, 3), (4, 7, 6), (0, 8, 5)))
(11.4, 3.4, ((0, 2, 3), (8, 1, 6), (7, 4, 5)))
(9.4, 6.4, ((1, 2, 3), (0, 4, 6), (7, 8, 5)))

Backward Queue9 : (7.4, 3.4, ((1, 2, 3), (8, 4, 6), (7, 5, 0)))
(8.0, 1.0, ((1, 2, 3), (8, 6, 4), (7, 0, 5)))
(8.8, 4.8, ((0, 2, 3), (1, 4, 6), (7, 8, 5)))
(8.8, 3.8, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(8.0, 2.0, ((1, 2, 3), (8, 4, 5), (7, 6, 0)))
(8.8, 4.8, ((0, 2, 3), (1, 5, 4), (7, 8, 6)))
(8.8, 4.8, ((1, 2, 3), (5, 0, 4), (7, 8, 6)))
(9.4, 2.4, ((1, 2, 3), (8, 5, 0), (7, 6, 4)))
(9.4, 2.4, ((1, 2, 3), (6, 8, 4), (7, 0, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 0, 6), (7, 4, 5)))
(8.8, 4.8, ((1, 2, 3), (7, 5, 4), (0, 8, 6)))
(10.0, 1.0, ((1, 0, 3), (8, 2, 4), (7, 6, 5)))
(8.8, 4.8, ((1, 2, 3), (7, 4, 6), (0, 8, 5)))
(11.4, 1.4, ((1, 2, 0), (8, 3, 4), (7, 6, 5)))
(10.0, 2.0, ((0, 2, 3), (1, 8, 4), (7, 6, 5)))
(11.4, 2.4, ((1, 0, 3), (2, 8, 4), (7, 6, 5)))

(11.4, 1.4, ((0, 2, 3), (8, 1, 4), (7, 6, 5)))
(10.0, 2.0, ((1, 2, 0), (8, 4, 3), (7, 6, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 0, 4), (7, 5, 6)))
(11.4, 2.4, ((1, 0, 3), (8, 4, 2), (7, 6, 5)))
(10.0, 2.0, ((1, 2, 3), (7, 8, 4), (0, 6, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 5, 4), (0, 7, 6)))
(9.4, 3.4, ((1, 2, 3), (8, 4, 6), (0, 7, 5)))
(11.4, 1.4, ((1, 2, 3), (8, 7, 4), (0, 6, 5)))
(10.2, 5.199999999999999, ((1, 0, 3), (2, 4, 6), (7, 8, 5)))
(10.2, 5.199999999999999, ((1, 0, 3), (2, 5, 4), (7, 8, 6)))

Forward Queue10: (8.8, 6.800000000000001, ((1, 2, 3), (4, 0, 6), (7, 5, 8)))
(9.4, 1.4, ((2, 1, 3), (8, 0, 6), (7, 4, 5)))
(8.8, 7.800000000000001, ((1, 2, 3), (4, 5, 0), (7, 8, 6)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 6), (7, 0, 4)))
(9.4, 6.4, ((1, 2, 3), (4, 6, 0), (7, 8, 5)))
(8.8, 7.800000000000001, ((1, 2, 3), (4, 5, 6), (7, 0, 8)))
(10.0, 3.0, ((1, 2, 3), (8, 6, 0), (7, 4, 5)))
(10.0, 4.0, ((0, 2, 3), (1, 8, 6), (7, 4, 5)))
(9.4, 5.4, ((1, 2, 3), (4, 8, 6), (0, 7, 5)))
(10.0, 2.0, ((1, 3, 0), (8, 2, 6), (7, 4, 5)))
(10.8, 6.800000000000001, ((0, 2, 3), (4, 1, 6), (7, 8, 5)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(9.4, 2.4, ((1, 8, 3), (0, 2, 6), (7, 4, 5)))
(11.4, 4.4, ((1, 0, 3), (2, 8, 6), (7, 4, 5)))
(11.4, 2.4, ((1, 6, 3), (8, 2, 0), (7, 4, 5)))
(10.0, 4.0, ((1, 2, 3), (7, 8, 6), (0, 4, 5)))
(10.8, 5.800000000000001, ((1, 2, 3), (4, 8, 0), (7, 6, 5)))
(10.0, 1.0, ((8, 1, 3), (0, 2, 6), (7, 4, 5)))
(9.4, 6.4, ((1, 0, 3), (4, 2, 6), (7, 8, 5)))
(11.4, 3.4, ((1, 2, 3), (8, 7, 6), (0, 4, 5)))
(10.8, 6.800000000000001, ((1, 2, 0), (4, 3, 6), (7, 8, 5)))
(11.4, 3.4, ((1, 2, 0), (8, 3, 6), (7, 4, 5)))
(10.8, 6.800000000000001, ((1, 2, 3), (4, 7, 6), (0, 8, 5)))
(11.4, 3.4, ((0, 2, 3), (8, 1, 6), (7, 4, 5)))
(9.4, 6.4, ((1, 2, 3), (0, 4, 6), (7, 8, 5)))
(9.4, 6.4, ((1, 2, 3), (4, 8, 0), (7, 5, 6)))

Backward Queue10: (7.4, 3.4, ((1, 2, 3), (8, 4, 6), (7, 5, 0)))
(8.0, 1.0, ((1, 2, 3), (8, 6, 4), (7, 0, 5)))
(8.8, 4.8, ((0, 2, 3), (1, 4, 6), (7, 8, 5)))
(8.8, 3.8, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(8.0, 2.0, ((1, 2, 3), (8, 4, 5), (7, 6, 0)))
(8.8, 4.8, ((0, 2, 3), (1, 5, 4), (7, 8, 6)))

(8.8, 4.8, ((1, 2, 3), (5, 0, 4), (7, 8, 6)))
(9.4, 2.4, ((1, 2, 3), (8, 5, 0), (7, 6, 4)))
(9.4, 2.4, ((1, 2, 3), (6, 8, 4), (7, 0, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 0, 6), (7, 4, 5)))
(8.8, 4.8, ((1, 2, 3), (7, 5, 4), (0, 8, 6)))
(10.0, 1.0, ((1, 0, 3), (8, 2, 4), (7, 6, 5)))
(8.8, 4.8, ((1, 2, 3), (7, 4, 6), (0, 8, 5)))
(8.8, 5.8, ((1, 2, 3), (4, 6, 0), (7, 8, 5)))
(8.8, 5.8, ((1, 2, 3), (4, 8, 6), (7, 0, 5)))
(10.2, 6.199999999999999, ((0, 2, 3), (4, 1, 6), (7, 8, 5)))
(11.4, 1.4, ((0, 2, 3), (8, 1, 4), (7, 6, 5)))
(10.0, 2.0, ((1, 2, 0), (8, 4, 3), (7, 6, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 0, 4), (7, 5, 6)))
(11.4, 2.4, ((1, 0, 3), (8, 4, 2), (7, 6, 5)))
(10.0, 2.0, ((1, 2, 3), (7, 8, 4), (0, 6, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 5, 4), (0, 7, 6)))
(9.4, 3.4, ((1, 2, 3), (8, 4, 6), (0, 7, 5)))
(11.4, 1.4, ((1, 2, 3), (8, 7, 4), (0, 6, 5)))
(10.2, 5.199999999999999, ((1, 0, 3), (2, 4, 6), (7, 8, 5)))
(10.2, 5.199999999999999, ((1, 0, 3), (2, 5, 4), (7, 8, 6)))
(8.8, 5.8, ((1, 0, 3), (4, 2, 6), (7, 8, 5)))
(11.4, 1.4, ((1, 2, 0), (8, 3, 4), (7, 6, 5)))
(10.2, 6.199999999999999, ((1, 2, 0), (4, 3, 6), (7, 8, 5)))
(10.0, 2.0, ((0, 2, 3), (1, 8, 4), (7, 6, 5)))
(10.2, 6.199999999999999, ((1, 2, 3), (4, 7, 6), (0, 8, 5)))
(11.4, 2.4, ((1, 0, 3), (2, 8, 4), (7, 6, 5)))

Forward Queue11: (8.8, 6.800000000000001, ((1, 2, 3), (4, 0, 6), (7, 5, 8)))
(9.4, 1.4, ((2, 1, 3), (8, 0, 6), (7, 4, 5)))
(8.8, 7.800000000000001, ((1, 2, 3), (4, 5, 0), (7, 8, 6)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 6), (7, 0, 4)))
(9.4, 6.4, ((1, 2, 3), (4, 6, 0), (7, 8, 5)))
(8.8, 7.800000000000001, ((1, 2, 3), (4, 5, 6), (7, 0, 8)))
(10.0, 3.0, ((1, 2, 3), (8, 6, 0), (7, 4, 5)))
(10.0, 4.0, ((0, 2, 3), (1, 8, 6), (7, 4, 5)))
(9.4, 5.4, ((1, 2, 3), (4, 8, 6), (0, 7, 5)))
(10.0, 2.0, ((1, 3, 0), (8, 2, 6), (7, 4, 5)))
(10.8, 6.800000000000001, ((0, 2, 3), (4, 1, 6), (7, 8, 5)))
(9.4, 4.4, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(9.4, 2.4, ((1, 8, 3), (0, 2, 6), (7, 4, 5)))
(10.0, 4.0, ((1, 2, 3), (8, 4, 6), (0, 7, 5)))
(11.4, 2.4, ((1, 6, 3), (8, 2, 0), (7, 4, 5)))
(10.0, 4.0, ((1, 2, 3), (7, 8, 6), (0, 4, 5)))

(10.8, 5.800000000000001, ((1, 2, 3), (4, 8, 0), (7, 6, 5)))
(10.0, 1.0, ((8, 1, 3), (0, 2, 6), (7, 4, 5)))
(9.4, 6.4, ((1, 0, 3), (4, 2, 6), (7, 8, 5)))
(11.4, 3.4, ((1, 2, 3), (8, 7, 6), (0, 4, 5)))
(10.8, 6.800000000000001, ((1, 2, 0), (4, 3, 6), (7, 8, 5)))
(11.4, 3.4, ((1, 2, 0), (8, 3, 6), (7, 4, 5)))
(10.8, 6.800000000000001, ((1, 2, 3), (4, 7, 6), (0, 8, 5)))
(11.4, 3.4, ((0, 2, 3), (8, 1, 6), (7, 4, 5)))
(9.4, 6.4, ((1, 2, 3), (0, 4, 6), (7, 8, 5)))
(9.4, 6.4, ((1, 2, 3), (4, 8, 0), (7, 5, 6)))
(11.4, 4.4, ((1, 2, 3), (8, 4, 0), (7, 6, 5)))
(11.4, 4.4, ((1, 0, 3), (2, 8, 6), (7, 4, 5)))

Backward Queue11: (8.0, 1.0, ((1, 2, 3), (8, 6, 4), (7, 0, 5)))
(8.0, 2.0, ((1, 2, 3), (8, 4, 5), (7, 6, 0)))
(8.8, 4.8, ((0, 2, 3), (1, 4, 6), (7, 8, 5)))
(8.2, 7.199999999999999, ((1, 2, 3), (4, 5, 0), (7, 8, 6)))
(8.8, 4.8, ((1, 2, 3), (7, 5, 4), (0, 8, 6)))
(8.8, 4.8, ((0, 2, 3), (1, 5, 4), (7, 8, 6)))
(8.8, 4.8, ((1, 2, 3), (5, 0, 4), (7, 8, 6)))
(8.2, 7.199999999999999, ((1, 2, 3), (4, 5, 6), (7, 0, 8)))
(9.4, 2.4, ((1, 2, 3), (6, 8, 4), (7, 0, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 0, 6), (7, 4, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 4, 6), (0, 7, 5)))
(10.0, 1.0, ((1, 0, 3), (8, 2, 4), (7, 6, 5)))
(8.8, 4.8, ((1, 2, 3), (7, 4, 6), (0, 8, 5)))
(8.8, 5.8, ((1, 2, 3), (4, 6, 0), (7, 8, 5)))
(8.8, 5.8, ((1, 2, 3), (4, 8, 6), (7, 0, 5)))
(9.4, 2.4, ((1, 2, 3), (8, 5, 0), (7, 6, 4)))
(8.8, 3.8, ((1, 2, 3), (8, 5, 0), (7, 4, 6)))
(10.0, 2.0, ((1, 2, 0), (8, 4, 3), (7, 6, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 0, 4), (7, 5, 6)))
(11.4, 2.4, ((1, 0, 3), (8, 4, 2), (7, 6, 5)))
(10.0, 2.0, ((1, 2, 3), (7, 8, 4), (0, 6, 5)))
(9.4, 3.4, ((1, 2, 3), (8, 5, 4), (0, 7, 6)))
(11.4, 1.4, ((0, 2, 3), (8, 1, 4), (7, 6, 5)))
(11.4, 1.4, ((1, 2, 3), (8, 7, 4), (0, 6, 5)))
(10.2, 5.199999999999999, ((1, 0, 3), (2, 4, 6), (7, 8, 5)))
(10.2, 5.199999999999999, ((1, 0, 3), (2, 5, 4), (7, 8, 6)))
(8.8, 5.8, ((1, 0, 3), (4, 2, 6), (7, 8, 5)))
(11.4, 1.4, ((1, 2, 0), (8, 3, 4), (7, 6, 5)))
(10.2, 6.199999999999999, ((1, 2, 0), (4, 3, 6), (7, 8, 5)))
(10.0, 2.0, ((0, 2, 3), (1, 8, 4), (7, 6, 5)))

(10.2, 6.199999999999999, ((1, 2, 3), (4, 7, 6), (0, 8, 5)))
(11.4, 2.4, ((1, 0, 3), (2, 8, 4), (7, 6, 5)))
(10.2, 6.199999999999999, ((0, 2, 3), (4, 1, 6), (7, 8, 5)))

Forward and Backward Expansions

Forward Expansion 1 (Cost 0): $(0, 1, 3), (8, 2, 6), (7, 4, 5)$
Backward Expansion 1 (Cost 0): $(1, 2, 3), (8, 0, 4), (7, 6, 5)$

Forward Expansion 2 (Cost 1.0): $(1, 0, 3), (8, 2, 6), (7, 4, 5)$
Backward Expansion 2 (Cost 1.4): $(1, 2, 3), (8, 5, 4), (7, 6, 0)$

Forward Expansion 3 (Cost 2.0): $(1, 2, 3), (8, 0, 6), (7, 4, 5)$
Backward Expansion 3 (Cost 2.4): $(1, 2, 3), (8, 5, 4), (7, 0, 6)$

Forward Expansion 4 (Cost 3.4): $(1, 2, 3), (8, 5, 6), (7, 4, 0)$
Backward Expansion 4 (Cost 3.8): $(1, 2, 3), (0, 5, 4), (7, 8, 6)$

Forward Expansion 5 (Cost 3.0): $(1, 2, 3), (0, 8, 6), (7, 4, 5)$
Backward Expansion 5 (Cost 1.0): $(1, 2, 3), (0, 8, 4), (7, 6, 5)$

Forward Expansion 6 (Cost 4.4): $(1, 2, 3), (4, 8, 6), (7, 0, 5)$
Backward Expansion 6 (Cost 1.0): $(1, 2, 3), (8, 4, 0), (7, 6, 5)$

Forward Expansion 7 (Cost 5.4): $(1, 2, 3), (4, 0, 6), (7, 8, 5)$
Backward Expansion 7 (Cost 2.4): $(1, 2, 3), (8, 4, 6), (7, 0, 5)$

Forward Expansion 8 (Cost 6.8): $(1, 2, 3), (4, 5, 6), (7, 8, 0)$
Backward Expansion 8 (Cost 3.8): $(1, 2, 3), (0, 4, 6), (7, 8, 5)$

Forward Expansion 9 (Cost 5.4): $(1, 2, 3), (4, 8, 6), (7, 5, 0)$
Backward Expansion 9 (Cost 4.8): $(1, 2, 3), (4, 0, 6), (7, 8, 5)$

Forward Expansion 10 (Cost 3.0): $(1, 2, 3), (8, 4, 6), (7, 0, 5)$
Backward Expansion 10 (Cost 6.2): $(1, 2, 3), (4, 5, 6), (7, 8, 0)$

Forward Expansion 11 (Cost 4.0): $(1, 2, 3), (8, 4, 6), (7, 5, 0)$
Backward Expansion 11 (Cost 3.4): $(1, 2, 3), (8, 4, 6), (7, 5, 0)$

Bi-Directional A* Search Output Explanation

1. `initial_state` represents the starting configuration of a puzzle, which is a 3x3 grid with numbers from 0 to 8. In this grid, 0 represents an empty cell, and the goal is to rearrange the numbers to match the `goal_state`.

2. `goal_state` represents the desired configuration of the puzzle, where the numbers are arranged in ascending order from left to right and top to bottom, with 0 in the middle.

3. `path` is the result of running the `bi_directional_astar` function, which is designed to find the shortest path from the `initial_state` to the `goal_state` using the A*

search algorithm. It returns a sequence of states representing the steps to reach the goal.

4. The output provided is the `path` variable, which is a list of states. Each state is a 3x3 grid represented as a tuple of tuples. The numbers in the grids indicate the current arrangement of tiles in the puzzle at each step of the path.

Initial and Goal States

Initial State

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

Goal State

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

Path from Initial State to Goal State

1. Step 1:

$$\begin{bmatrix} 0 & 1 & 3 \\ 8 & 2 & 6 \\ 7 & 4 & 5 \end{bmatrix}$$

   Forward Cost: 0 Backward Cost: 8 Manhattan Distance: 8

2. Step 2:

$$\begin{bmatrix} 1 & 0 & 3 \\ 8 & 2 & 6 \\ 7 & 4 & 5 \end{bmatrix}$$

   Forward Cost: 1 Backward Cost: 7 Manhattan Distance: 7

3. Step 3:

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 6 \\ 7 & 4 & 5 \end{bmatrix}$$

   Forward Cost: 2 Backward Cost: 6 Manhattan Distance: 6

4. Step 4:

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 4 & 6 \\ 7 & 0 & 5 \end{bmatrix}$$

   Forward Cost: 3 Backward Cost: 5 Manhattan Distance: 5

5. Step 5:

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 4 & 6 \\ 7 & 5 & 0 \end{bmatrix}$$

   Forward Cost: 4 Backward Cost: 4 Manhattan Distance: 4

6. Step 6:

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 4 & 6 \\ 7 & 5 & 0 \end{bmatrix}$$

   Forward Cost: 5 Backward Cost: 3 Manhattan Distance: 4

7. Step 7:

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 4 & 6 \\ 7 & 0 & 5 \end{bmatrix}$$

   Forward Cost: 6 Backward Cost: 2 Manhattan Distance: 5

8. Step 8:

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 4 & 0 \\ 7 & 6 & 5 \end{bmatrix}$$

   Forward Cost: 7 Backward Cost: 1 Manhattan Distance: 7

9. Step 9:

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

   Forward Cost: 8 Backward Cost: 0 Manhattan Distance: 8

Note

1. The "Forward Cost" of 4 at Step 5 indicates it took 4 moves from the initial state to reach this state.
2. The "Backward Cost" of 4 indicates it took 4 moves from the goal state to reach this state.
3. This balance point, where forward and backward costs are equal, is where the two searches meet—highlighting the efficiency of bi-directional search algorithms.