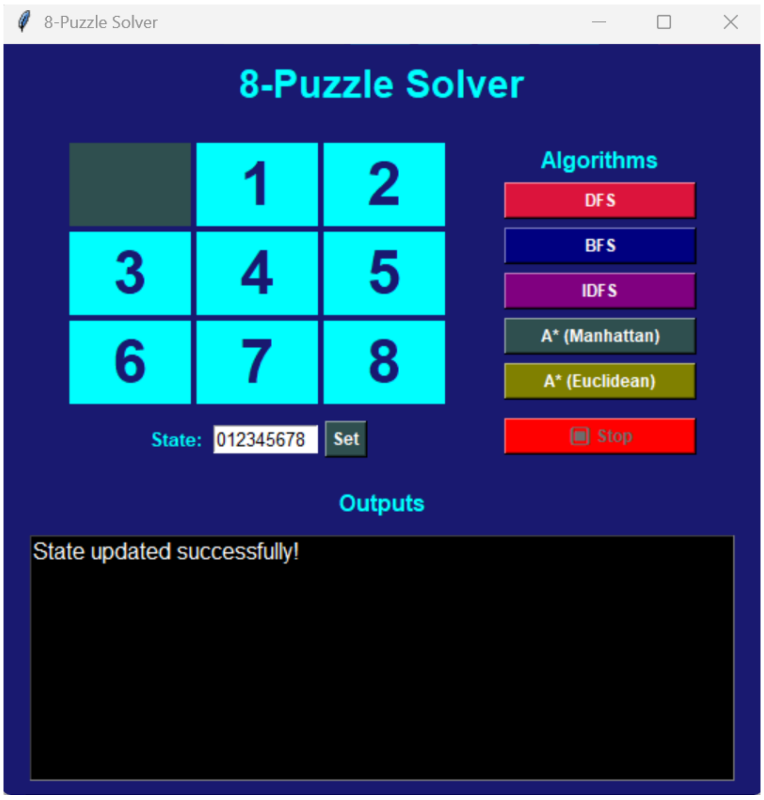
8 Puzzle Solver

Mohamed Khaled
Alaa Mohamed
Mostafa Mohamed

October 25, 2025



State updated successfully!

Figure 0.0.1: 8 puzzle solver

Abstract

This report presents the design and implementation of an intelligent 8-puzzle solver using four classical search algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), Iterative Deepening Depth-First Search (IDFS), and A* Search. The project aims to analyze, compare, and evaluate the performance of these algorithms in solving the same problem under a unified framework. The objective is to evaluate their effectiveness in finding the optimal solution by analyzing key performance metrics such as path cost, number of nodes expanded, search depth, and execution time.

1 Introduction

The implementation of the 8-puzzle solver was designed using an object-oriented programming (OOP) structure to ensure modularity, reusability, and clarity. The system is divided into three main components: the Puzzle class, the Search base class, and a set of search algorithm classes that inherit from it. The Puzzle class is responsible for representing the game state and managing all puzzle-related operations, such as generating valid successor states and checking whether a state matches the goal configuration. The Search class serves as a base (parent) class that defines common functionality used by all search algorithms. It handles performance tracking (such as execution time and search depth), path reconstruction, and the standardized reporting of results.

During development, several assumptions were made to simplify the implementation and focus on algorithmic behavior:

- States are hashable, allowing them to be efficiently stored in Python sets and dictionaries for visited-node tracking.
- The goal state is reachable from the given initial state, taking into consideration that the puzzle is only solvable if the number of inversions (pairs of tiles that are in the wrong order) is even.
- The state space is finite, meaning the number of possible configurations is limited.

Additionally, a simple Graphical User Interface (GUI) was implemented using Python's Tkinter library to visualize the puzzle and display the sequence of moves leading from the initial state to the goal state.

2 Search

In this code, several fundamental data structures are used to manage the puzzle-solving process efficiently. In the Search class, **a set called visited is used to store all previously explored puzzle states**, ensuring that no state is processed more than once — this helps prevent infinite loops and redundant exploration. **A dictionary (dict) named parent is used to record each state's predecessor**, effectively storing the "parent-child" relationships between puzzle states, which later allows for reconstructing the full solution path from the goal back to the start. Another dictionary, move_dir, maps each state to the specific move (Up, Down, Left, or Right) that led to it, making it possible to retrieve the sequence of actions that solve the puzzle.

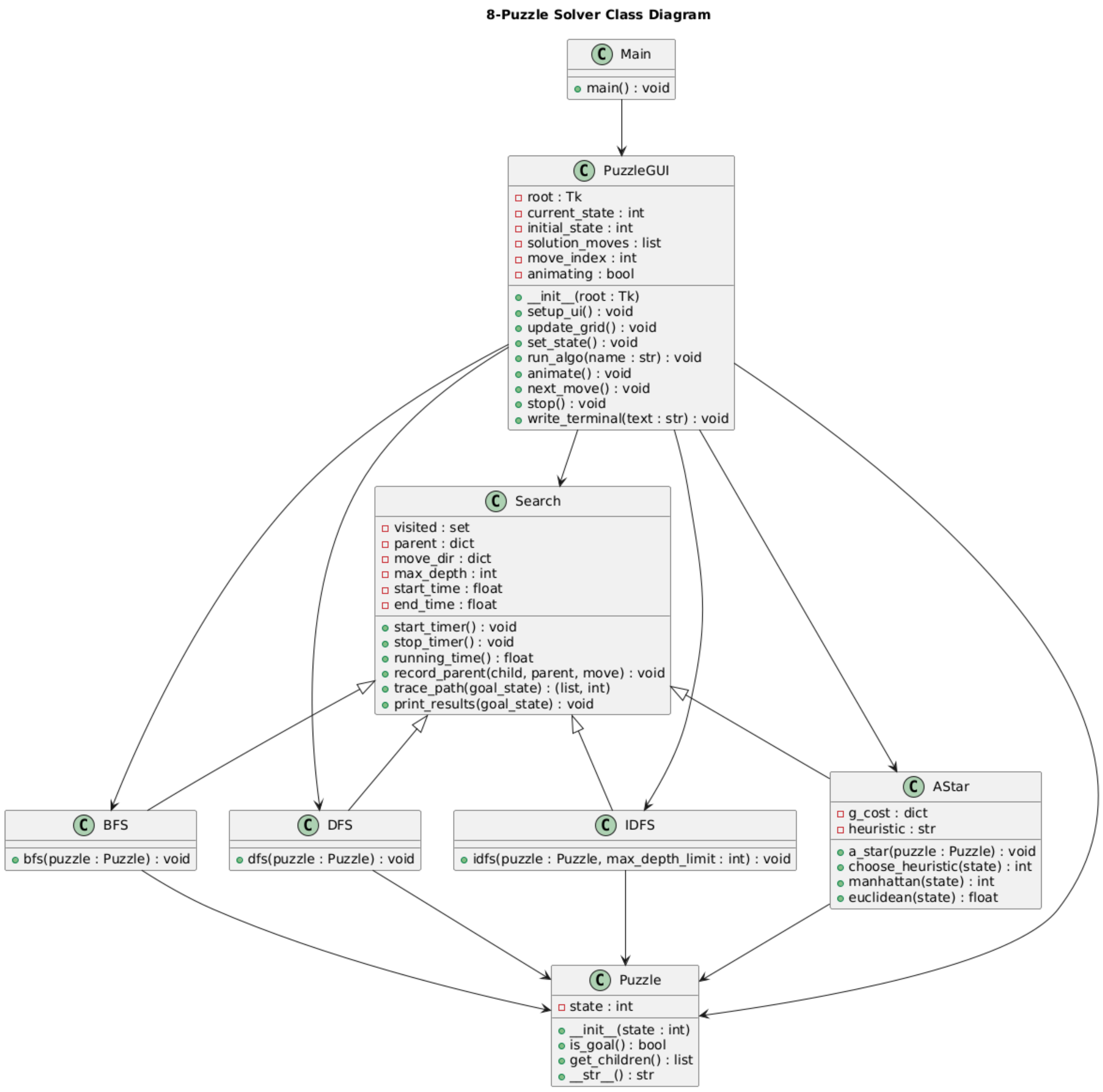


Figure 2.0.1: UML Diagram

3 Breadth First Search

BFS explores the search space level by level, ensuring that all nodes at a given depth are expanded before moving on to deeper levels. This guarantees that the first time the goal state is reached, the path found is the shortest possible in terms of the number of moves. **The algorithm uses a First-In-First-Out (FIFO) queue**, implemented using Python's collections.deque, as its main data structure to manage the frontier. Each node dequeued from the front represents the next state to expand, while its unvisited children are enqueued at the back.
Features:

- Completeness : Guaranteed — BFS will always find a solution if one exists.
- Optimality : Guaranteed — when all step costs are equal, as it finds the shortest path first
- Space Efficiency: Poor — requires storing all generated nodes, causing exponential memory usage.

Listing 1: Breadth First Search Implementation

```python
while queue:
            current, depth = queue.popleft()  # FIFO     ensures level-order traversal
            self.max_depth = max(self.max_depth, depth)

            current_puzzle = Puzzle(current)
            if current_puzzle.is_goal():      # Goal test
                self.stop_timer()
                return

            if current in self.visited:       # Skip already expanded nodes
                continue
            self.visited.add(current)

            # Expand neighbors and enqueue them
            for neighbor, move in current_puzzle.get_children():
                if neighbor not in self.visited and neighbor not in self.parent:
                    self.record_parent(neighbor, current, move)
                    queue.append((neighbor, depth + 1))
```

4 Depth First Search

DFS explores the search space by expanding the deepest unvisited node first before backtracking, effectively diving down one path as far as possible before exploring alternatives. This search pattern is **implemented using a Last-In-First-Out (LIFO) stack**, represented in Python by a simple list where new states are appended and the most recent one is removed using the pop() operation. Each node popped from the stack represents the next state to expand, and its unvisited children are pushed onto the stack, ensuring that newly discovered nodes are explored before older ones. Features:

- Completeness : Not guaranteed — DFS may get stuck exploring deep paths in infinite or cyclic search spaces, but it will find a goal if the state space is finite and reachable.
- Optimality : Not guaranteed — DFS does not necessarily find the shortest path to the goal. It may find a solution that is longer than the shortest path because it explores deeply before considering alternative shorter paths.
- Space Efficiency: Excellent — DFS stores only the path from root to current node and unexpanded siblings in the stack.

Listing 2: Depth First Search Implementation

```python
while stack:
    current, depth = stack.pop()                    # LIFO      explores
       deepest node first
    self.max_depth = max(self.max_depth, depth)

    current_puzzle = Puzzle(current)
    if current_puzzle.is_goal():                    # Goal test
        self.stop_timer()
        return

    if current in self.visited:                     # Skip already expanded
       states
        continue
    self.visited.add(current)

    # Expand neighbors (children) and push them onto the stack
    for neighbor, move in current_puzzle.get_children():
        if neighbor not in self.visited:
            self.record_parent(neighbor, current, move)
            stack.append((neighbor, depth + 1))
```

5 Iterative Depth First Search

(IDFS) algorithm combines the advantages of both Depth-First Search (DFS) and Breadth-First Search (BFS). It performs a series of depth-limited DFS iterations, gradually increasing the depth limit with each iteration until the goal is found. This approach ensures the memory efficiency of DFS while maintaining the completeness and optimality of BFS for problems with uniform step costs. **The algorithm uses a stack as its main data structure**, exploring nodes in a depth-first manner within the current limit, and restarting the search with a greater depth limit if the goal is not reached. This repeated exploration of shallow nodes allows IDFS to find the shallowest and most optimal solution without consuming excessive memory.

Features:

- Completeness : Guaranteed — IDFS will always find a solution if one exists, since it systematically increases the depth limit.
- Optimality : Guaranteed — when all step costs are equal, IDFS finds the shallowest (shortest) solution path.
- Space Efficiency: Excellent — similar to DFS, it uses linear space with respect to the maximum search depth.

Listing 3: Iterative Depth First Search Implementation

```python
for depth_limit in range(max_depth_limit):
    self.depth_limit = depth_limit
    stack = [(puzzle.state, 0)]
    self.visited.clear()
    self.parent.clear()
    self.move_dir.clear()
    self.parent[puzzle.state] = None
    self.move_dir[puzzle.state] = None
    self.max_depth = 0

    while stack:
        current, depth = stack.pop()
        self.max_depth = max(self.max_depth, depth)

        current_puzzle = Puzzle(current)
        if current_puzzle.is_goal():          # Goal test
            return

        if current in self.visited:
            continue

        if depth >= depth_limit:              # Depth cutoff
            continue

        self.visited.add(current)

        # Expand children and push to stack
        for neighbor, move in current_puzzle.get_children():
            if neighbor not in self.visited:
                self.record_parent(neighbor, current, move)
                stack.append((neighbor, depth + 1))
```

6 A Star

The A* search algorithm is an informed search strategy that uses knowledge about the problem domain to guide the exploration toward the goal more efficiently than uninformed methods such as BFS or DFS. It uses problem-specific knowledge, in the form of a heuristic, to efficiently guide the exploration toward the goal. At every iteration, A* selects the node with the smallest total estimated cost given by the evaluation function

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

where g(n) represents the exact cost from the start state to the current node n, and h(n) is the heuristic function estimating the remaining cost from n to the goal. The algorithm prioritizes expanding the node with the lowest f(n) value, thereby balancing exploration (via g(n)) and goal directed-ness (via h(n)).

A key property required of the heuristic function h(n) is itsadmissibility, which ensures that it never overestimates the true cost to reach the goal. Formally, a heuristic is admissible if

$$h(n) \leq h^*(n)$$

for all nodes n, where h*(n) is the actual minimum cost from n to the goal. This property guarantees that A* will always find an optimal solution, provided that all step costs are non-negative.

In the implementation, **A* uses a priority queue, implemented with Python's heapq module, to store and retrieve nodes based on their f(n) values.** The algorithm repeatedly extracts the node with the lowest total cost, checks if it represents the goal, and expands its neighbors by computing new g(n), h(n), and f(n) values.

If the heuristic is admissable:

- Completeness : Guaranteed — A* will always find a solution if one exists, provided all step costs are non-negative
- Optimality : Guaranteed — when the heuristic function is admissible, A* returns the optimal path to the goal.
- Space Efficiency: Moderate — A* stores all generated nodes in memory, which can grow exponentially with search depth.

Listing 4: A Star Search Implementation

```python
open_list = []
g = 0
h = self.choose_heuristic(puzzle.state)
f = g + h
heapq.heappush(open_list, (f, g, puzzle.state))

while open_list:
    current_f, current_g, current = heapq.heappop(open_list)

    if current_g > self.g_cost.get(current, float('inf')):
        continue

    current_puzzle = Puzzle(current)

    if current_puzzle.is_goal():            # Goal test
        self.stop_timer()
        return

    if current in self.visited:
        continue
    self.visited.add(current)

    self.max_depth = max(self.max_depth, current_g)

    # Expand neighbors
    for board, move in current_puzzle.get_children():
        new_g = current_g + 1
        new_h = self.choose_heuristic(board)
        new_f = new_g + new_h

        # Push neighbor if unvisited or better path found
        if board not in self.visited and (board not in self.g_cost or
           new_g < self.g_cost[board]):
            heapq.heappush(open_list, (new_f, new_g, board))
```

```
                self.record_parent(board, current, move)
                self.g_cost[board] = new_g
```

6.1 Manhattan Heuristic

The Manhattan heuristic is one of the most widely used and effective admissible heuristics for the 8-puzzle problem. **It estimates the cost to reach the goal by summing the vertical and horizontal distances each tile is away from its correct position on the grid.**

For each tile i, the algorithm computes its current coordinates (x_i, y_i) and its goal coordinates (x_i^*, y_i^*). The heuristic cost for that tile is then the absolute difference in their row and column positions:

$$h(n) = \sum_{i=1}^{8} (|x_i - x_i^*| + |y_i - y_i^*|)$$

This sum represents the minimum number of moves required to move each tile to its goal position, assuming there are no obstacles. Since the Manhattan heuristic never overestimates the actual cost (it ignores blocking tiles), it is always admissible, and therefore A* using this heuristic remains optimal.

6.2 Euclidean Heuristic

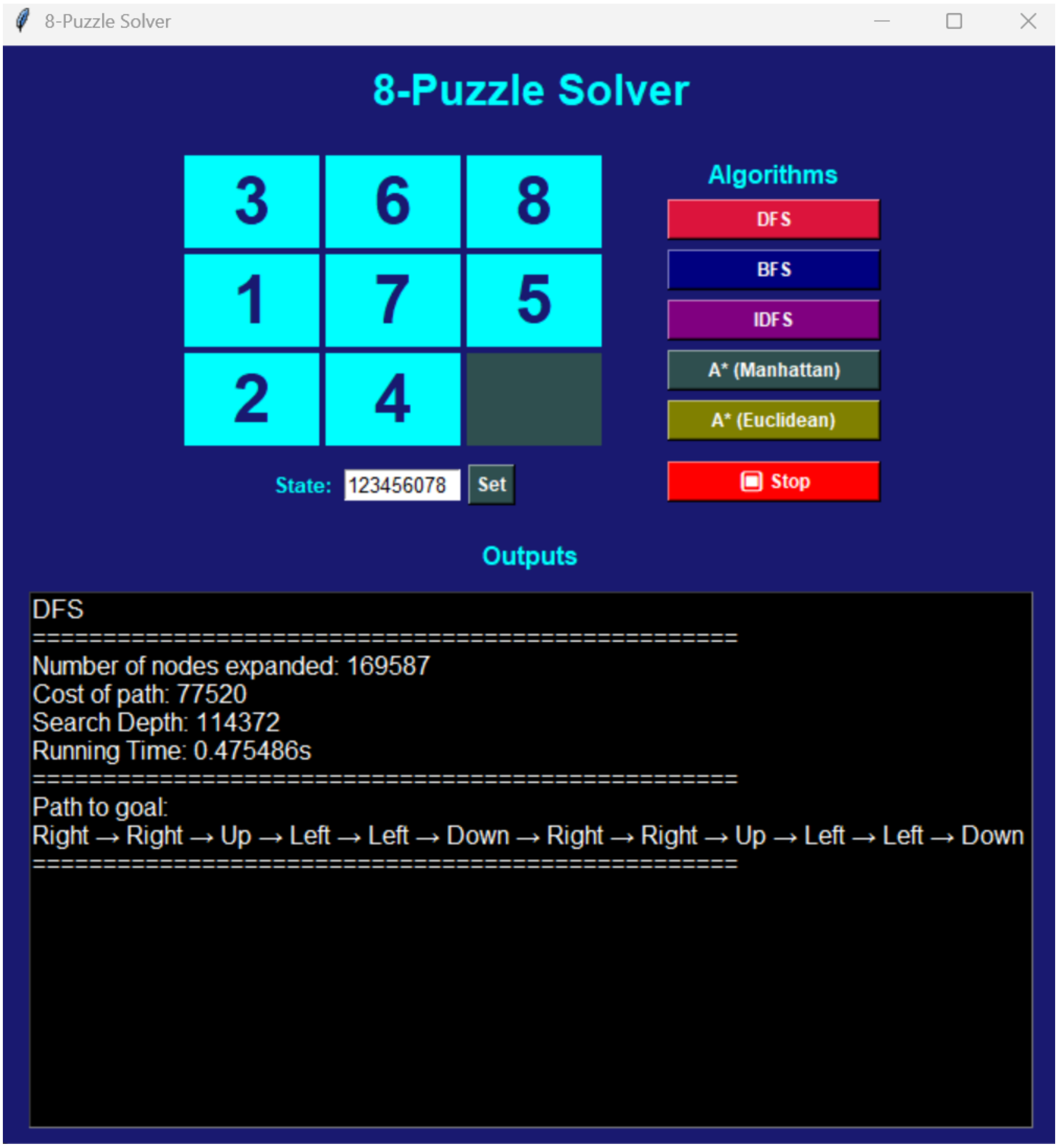
The Euclidean heuristic is another admissible heuristic used in the A* search algorithm for the 8-puzzle problem. Unlike the Manhattan heuristic, which measures only horizontal and vertical distances, **the Euclidean heuristic estimates the cost to the goal by considering the straight-line (diagonal) distance between each tile's current position and its target position on the grid.**

For each tile i, the algorithm computes its current coordinates (x_i, y_i) and its goal coordinates (x_i^*, y_i^*), and applies the Euclidean distance formula as follows:

$$h(n) = \sum_{i=1}^{8} \sqrt{(x_i - x_i^*)^2 + (y_i - y_i^*)^2}$$

This heuristic provides a more "natural" geometric estimate of how far each tile is from its correct location. However, because tiles in the 8-puzzle can only move horizontally or vertically, the Euclidean heuristic tends to underestimate the true cost more strongly than the Manhattan heuristic. **This makes it less informed but still admissible**, ensuring that A* remains complete and optimal when using it.

7 Simulation



```
DFS
===================================================
Number of nodes expanded: 169587
Cost of path: 77520
Search Depth: 114372
Running Time: 0.475486s
===================================================
Path to goal:
Right → Right → Up → Left → Left → Down → Right → Right → Up → Left → Left → Down
===================================================
```

Figure 7.0.1: Running Simulation of DFS