Slovak Universiry of Technology in Bratislava
Faculty of Informatics and Information Technologies

Artificial Intelligence

Assignment 1

Exploring the resolution of the 8-Puzzle game employing the A* algorithm, while conducting a comparative analysis of approximation heuristics, specifically focusing on the Manhattan distance heuristic and the heuristic based on the count of misplaced tiles (Hamming).

Dávid Truhlář ID: 120897 xtruhlar@stuba.sk

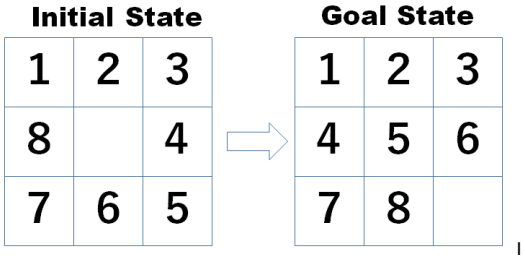
Table of Contents

The assignment 3
Implementation 4
Definition of A* algorithm and basic functions 4
Node and State 4
Operators 5
Heuristics 5
1. Hamming or Misplaced Tiles 5
2. Manhattan distance or the sum of distances of each tile from its goal position 5
A* Algorithm 6
Flowchart of the algorithm 6
Code 7
Diagram 8
Testing and evaluating 9
Methodology 9
Inputs and outputs 10
Results 10
Conclusion 11
User guide 12

The assignment

Defining Problem

Our task is to solve the 8-puzzle, a game involving 8 numbered tiles and an empty space. Tiles can be moved in four directions, but only if an empty space allows it. Each puzzle has an initial and target position, and the objective is to find a sequence of moves to transition from one to the other.



[1]

Implementation

When solving this problem using state-space search algorithms, we need to specify certain concepts:

State

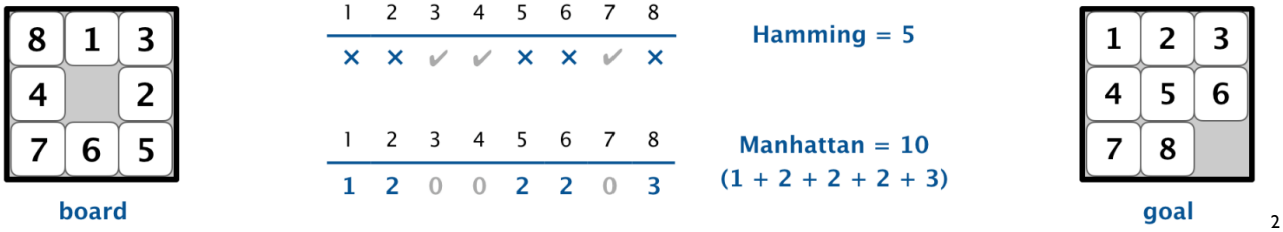
A state represents the current tile arrangement. The initial state can be represented as ((1 2 3)(4 5 6)(7 8 m)) or (1 2 3 4 5 6 7 8 m), each with its own advantages.

Operators

There are four operators: RIGHT, DOWN, LEFT, and UP. Operators transform a state if possible. All operators have equal weight.

Heuristic Function

Some algorithms require additional information in the form of a heuristic, estimating the distance from the current state to the goal state. Several heuristics are available, like tile count not in place or sum of distances of each tile from its goal. Avoid including the empty space in these calculations.



[2]

Node

To visualize the path, we create a graph from the state space, typically a tree. States become nodes with attributes such as state representation and parent reference. Nodes can also store operator history, depth, path cost, and estimated cost to the goal.

ALGORITHM

1. Create an initial node and place it among the generated but unprocessed nodes.
2. If there are no generated and unprocessed nodes, terminate with failure - no solution exists.
3. Select the most suitable node from the generated and unprocessed nodes, label it as the current node.
4. If this node represents the goal state, terminate with success - output the solution.
5. Create successors of the current node and add them to the processed nodes.
6. Sort the successors and store them among the generated but unprocessed nodes.
7. Go to step 2.

---

[1] Ando, Ruo & Takefuji, Yoshiyasu. (2020). A new perspective of paramodulation complexity by solving massive 8 puzzles. 10.13140/RG.2.2.24649.57441. Accessed 2. October 2023 at [link]

[2] Princeton Archive, COS 226 Programming Assignment, 2020. Accessed 2. October 2023 at [link]

Implementation

Definition of A* algorithm and basic functions

It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It remains a widely popular algorithm for graph traversal.[3]

Node and State

At the begging the new *class* is created to represent in what state the algorithm is.

```
class Node:
    def __init__(self, state, parent=None, operator=None, g=0, f=0):
        self.state = state
        self.parent = parent
        self.operator = operator
        self.g = g
        self.f = f
        self.h = 0
        self.n = 0
```

In the context of the A* algorithm, three essential variables must be defined: f, g, and h. The variable 'f' represents the sum of 'g' and 'h.' Here, 'g' signifies the distance between the root stage (the starting point) and the current stage, while 'h' corresponds to the result obtained when computing the Manhattan distance. The variable ‘n’ represents the number of misplaced tiles.

```
h += abs(i - goal_row) + abs(j - goal_col)
```

The 'parent' parameter plays a pivotal role in retaining a record of the most recent stage, thereby serving as a crucial mechanism for preventing cyclic solutions.

To represent the actual board, the 2D array filled with numbers is created using tuple:

```
start = '123456780'  # The initial state can be loaded as input from user
goal = '523461078'  # The goal state can be loaded as input from user
# (' ', ' ', ' '),
# (' ', ' ', ' '),
# (' ', ' ', ' ')) tuple representation of 2D array

# This function converts a string representation to tuple representation above
def tuplestring(str):
    # conversion to list of lists [[ , , ]…]
    board_list = [[int(str[i:i + 3][j]) for j in range(3)] for i in range(0, 9, 3)]
    # conversion into tuple of tuples (( , , )… )
    return tuple(tuple(row) for row in board_list)


# Initial and goal states of the puzzle in the tuple format
initial_state = tuplestring(start)
goal_state = tuplestring(goal)
```

The number ‘***0***’ represents the empty tile.

[3] Ravikiran A S, A* Algorithm Concepts and Implementation, Aug 9, 2023. Accessed 2. October 2023 at [Link]

Operators

For the sake of simplification, this document includes an illustrative example of leftwards movement. It's worth noting that the implementation of the other movement directions follows the same programming approach:

```
def move(state, operator):
    # Find the position of the empty space (represented by 0)
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                empty_row, empty_col = i, j

    # Create a copy of the current state to modify
    new_state = [list(row) for row in state]

    # Perform a move based on the specified operator ('L' for left, 'R' for right, 'U'
for up, 'D' for down)
    if operator == 'R':
        # Move the empty space left if possible
        if empty_col > 0:
            new_state[empty_row][empty_col], new_state[empty_row][empty_col - 1] = \
                new_state[empty_row][empty_col - 1], new_state[empty_row][empty_col]
            return tuple(tuple(row) for row in new_state)
…
    return None
```

Heuristics

1. Hamming or Misplaced Tiles

To compute the Hamming or misplaced tiles heuristic, a straightforward function is employed. This function essentially compares each tile of the current stage with the corresponding tile on the goal stage board. When a tile is found to be in the wrong position, the count 'n' is incremented accordingly.

```
def hamming(self, goal_board):
    n = 0
    for x in range(3):
        for y in range(3):
            if self.board[x][y] != goal_board[x][y]:
                n += 1
    return n
```

2. Manhattan distance or the sum of distances of each tile from its goal position

This code computes the Manhattan distance by measuring the sum of horizontal and vertical displacements for each tile on the board, comparing its position in the current state to its position in the goal state. The empty space (represented as 0) is excluded from the calculation.

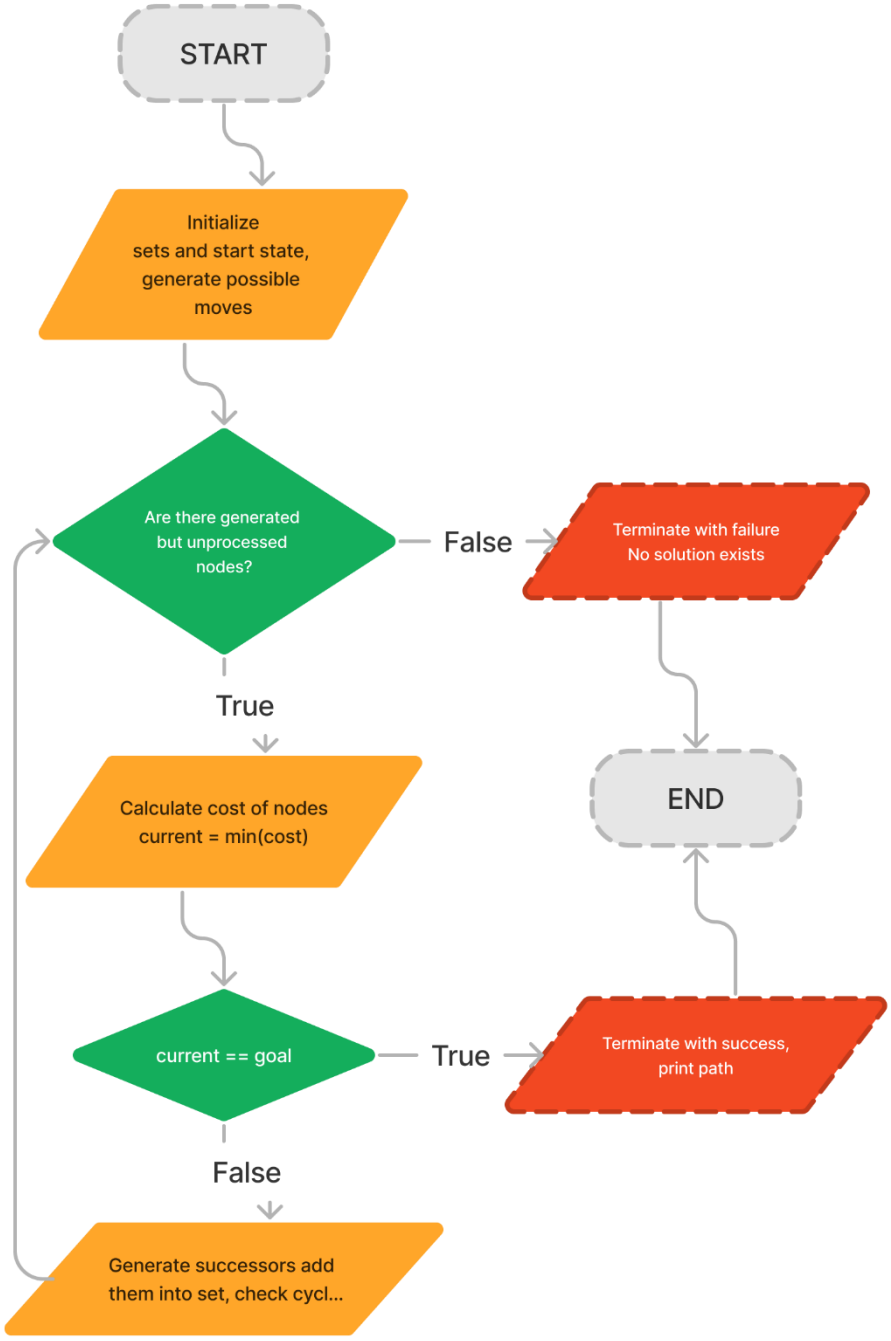
```
def manhattan_distance(state, goal):
    h = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                # Find the coordinates of the current tile in the goal state
                goal_row, goal_col = find_tile(goal, state[i][j])
                # Calculate the Manhattan distance for the current tile
                h += abs(i - goal_row) + abs(j - goal_col)
    return h
```

A* Algorithm

The steps from the assignment:

1. Create an initial node and place it among the generated but unprocessed nodes.
2. If there are no generated and unprocessed nodes, terminate with failure - no solution exists.
3. Select the most suitable node from the generated and unprocessed nodes, label it as the current node.
4. If this node represents the goal state, terminate with success - output the solution.
5. Create successors of the current node and add them to the processed nodes.
6. Sort the successors and store them among the generated but unprocessed nodes.
7. Go to step 2.

Flowchart of the algorithm



Code

```
# A* search algorithm to find the solution path from initial board to the goal board
def a_algorithm(initial, goal_board):
    open_set = []  # priority queue of nodes
    closed_set = set()  # set of visited states

    # Create the initial node and set the initial heuristic
    initial_node = Node(initial)
    if heuristic == 1:
        initial_node.h = manhattan_distance(initial, goal_board)
    elif heuristic == 2:
        initial_node.n = hamming(initial, goal_board)
    else:
        print("Invalid input.")
        return None

    open_set.append(initial_node)
    times = timer()

    while open_set:
        # Find the node with the lowest priority (lowest g + h or n)
        if heuristic == 1:
            current = min(open_set, key=lambda node: node.g + node.h)
        elif heuristic == 2:
            current = min(open_set, key=lambda node: node.n)

        if current.board == goal_board:
            return reconstruct_path(current)  # Goal board reached

        open_set.remove(current)
        closed_set.add(current.board)

        # Generate successor nodes and add them to the open set
        for operator in ['L', 'R', 'U', 'D']:
            next_state = move(current.board, operator)
            if next_state is None or next_state in closed_set:
                continue
            successor = Node(next_state, current, operator, current.g + 1, current.f + 1)
            if heuristic == 1:
                successor.h = manhattan_distance(next_state, goal_board)
            elif heuristic == 2:
                successor.n = hamming(next_state, goal_board)
            open_set.append(successor)
        # Check the time limit
        timee = timer()
        if (timee - times) > time_limit_seconds:
            return None

    return None
```

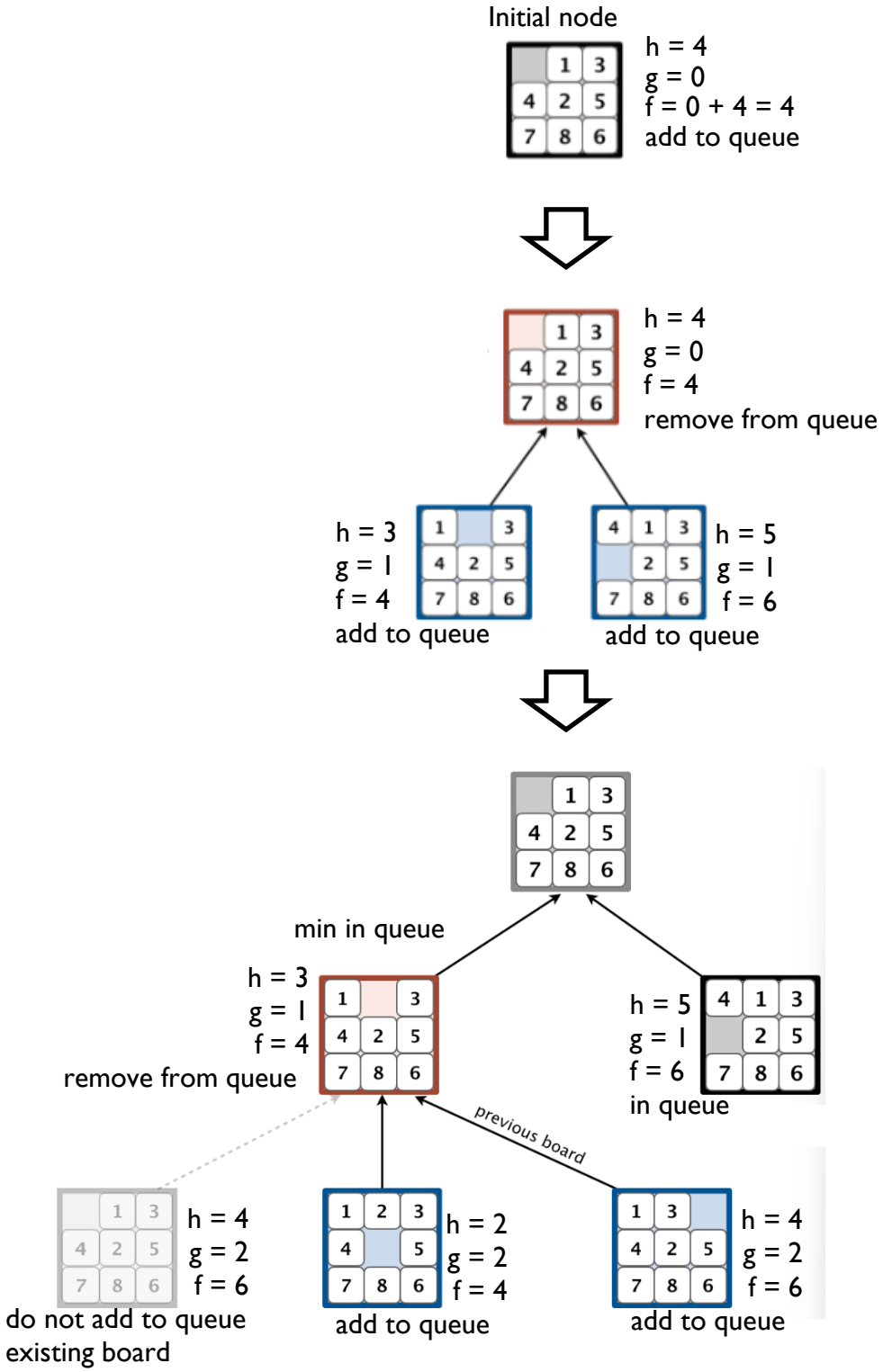
With the 'open_set' and 'closed_set' initialized, the algorithm starts by creating the initial node and adding it to the 'open_set'. It then enters a while loop to drive the algorithm's progress.

Inside the loop, the algorithm identifies the node with the lowest cost as the 'current' node. It checks whether this 'current' node matches the goal state. If it does, the algorithm proceeds to call a function responsible for reconstructing and displaying the path from the initial node to the goal node.

If the 'current' node doesn't match the goal state, the algorithm generates all possible moves and checks if the resulting board configuration is already in the 'closed_set'. If no match is found, it creates successor nodes, each with new board configurations based on the previous step. Cost computations are performed, and the while loop continues to execute.

Diagram

Diagram illustrates the game tree after each of the first three steps of running the A* search algorithm on a 3-by-3 puzzle using the Manhattan priority function.[4]



[4] Princeton Archive, COS 226 Programming Assignment, 2020. Accessed 2. October 2023 at [link]

Testing and evaluating

Methodology

In the process of evaluating the program's performance, I initiated the creation of a file named 'inputs.txt,' housing a compilation of 20 distinct character strings that represent the initial board states. The generation of these inputs was facilitated by an *8-puzzle solver*[5]. Subsequently, a 'for' loop was employed to iterate through each input, leading to the generation of the following output:

```
10:180437625
┌───┬───┬───┐           ┌───┬───┬───┐
│ 1 │ 8 │   │           │ 1 │ 2 │ 3 │
├───┼───┼───┤   ---->   ├───┼───┼───┤
│ 4 │ 3 │ 7 │           │ 4 │ 5 │ 6 │
├───┼───┼───┤   ---->   ├───┼───┼───┤
│ 6 │ 2 │ 5 │           │ 7 │ 8 │   │
└───┴───┴───┘           └───┴───┴───┘
Solution Path: ['R', 'U', 'U', 'L', 'D', 'D', 'R', 'U', 'U', 'R', 'D', 'L', 'L', 'U', 'R', 'D', 'L', 'U', 'R', 'D', 'R', 'U', 'L', 'L']
Number of nodes in path: 24
Number of nodes created:  317
Time taken: 13.82619992364198 ms

---------------------------
```

In the context of this illustration, the 10th input, denoted as '180437625,' yielded a successful solution. The path to this solution consisted of 24 nodes. The final piece of information conveyed in the output pertained to time, which, in this particular instance, was measured at 13.56 milliseconds.

To obtain the path from the initial board state to the goal board state, a path reconstruction mechanism was employed, which involves traversing from the goal node through the parent parameter until reaching the initial node. At each step of this process, the operator responsible for the transition to the current state was appended to the path.

```python
# Reconstruct the path from the goal node to the initial node
def reconstruct_path(node):
    path = []
    while node is not None:
        path.append(node.operator)
        node = node.parent
    path.reverse()
    path.remove(path[0])  # Remove the initial None operator
    return path
```

The measurement of time was executed using the 'timer()' function from the 'timeit' library, which was determined to offer a more precise time recording compared to Python's native 'time()' function.

```python
from timeit import default_timer as timer

time_start = timer()
solution_path = a_algorithm(initial_state, goal_state)
time_end = timer()
```

Additionally, the program underwent testing with inputs that do not possess a viable solution, and the outcomes of such tests were documented in the 'err.txt' file.

```
This puzzle is not solvable.


---------------------------
```

[5] Github, 8-puzzle-solver by dgurkaynak at https://github.com/dgurkaynak/8-puzzle-solver

Inputs and outputs

Structure of inputs.txt

```
823416750
312456780
136245780
752314680
876543201
124037586
012345678
486752031
150764382
180437625
210475836
254803176
862157430
236504178
350174286
827403615
431506872
250846317
642105783
042863517
```

Structure of output-*[method]*.txt

```
1:823416750
 ┌─┬─┬─┐          ┌─┬─┬─┐
 │8│2│3│          │1│2│3│
 ├─┼─┼─┤  ---->   ├─┼─┼─┤
 │4│1│6│          │4│5│6│
 ├─┼─┼─┤  ---->   ├─┼─┼─┤
 │7│5│ │          │7│8│ │
 └─┴─┴─┘          └─┴─┴─┘
Solution Path: ['R', 'R', 'D', 'L', 'D', 'R', 'U', 'U', 'L', 'L', 'D', 'R', '
Number of nodes in path: 36
Number of nodes created:  366
Time taken: 18.773799994960427 ms

----------------------------

2:312456780
 ┌─┬─┬─┐          ┌─┬─┬─┐
 │3│1│2│          │1│2│3│
 ├─┼─┼─┤  ---->   ├─┼─┼─┤
 │4│5│6│          │4│5│6│
 ├─┼─┼─┤  ---->   ├─┼─┼─┤
 │7│8│ │          │7│8│ │
 └─┴─┴─┘          └─┴─┴─┘
Solution Path: ['R', 'D', 'D', 'R', 'U', 'L', 'U', 'L', 'D', 'D', 'R', 'R', '
Number of nodes in path: 30
Number of nodes created:  840
Time taken: 42.698899982497096 ms

----------------------------
```

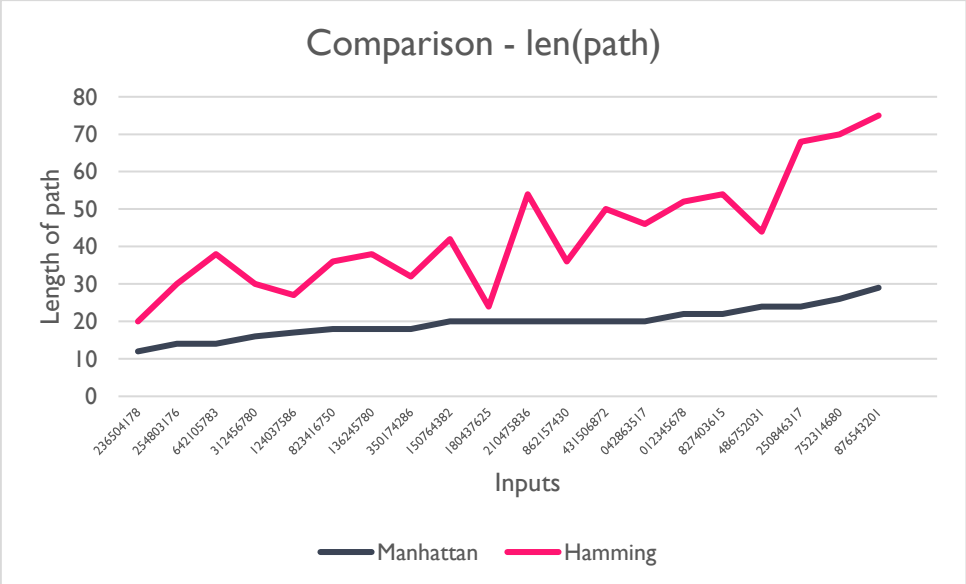
```
1:402315786

 ┌─┬─┬─┐          ┌─┬─┬─┐
 │4│ │2│          │1│2│3│
 ├─┼─┼─┤  ---->   ├─┼─┼─┤
 │3│1│5│          │4│5│6│
 ├─┼─┼─┤  ---->   ├─┼─┼─┤
 │7│8│6│          │7│8│ │
 └─┴─┴─┘          └─┴─┴─┘
This puzzle is not solvable.

----------------------------
```

Results

To facilitate a comparative analysis of each heuristic's performance, the following table encompasses essential data pertaining to the 20 test inputs. Two output files were generated, namely 'output-manhattan.txt' and 'output-hamming.txt,' with each input subjected to testing using both the Manhattan distance and Hamming heuristics for subsequent comparison.

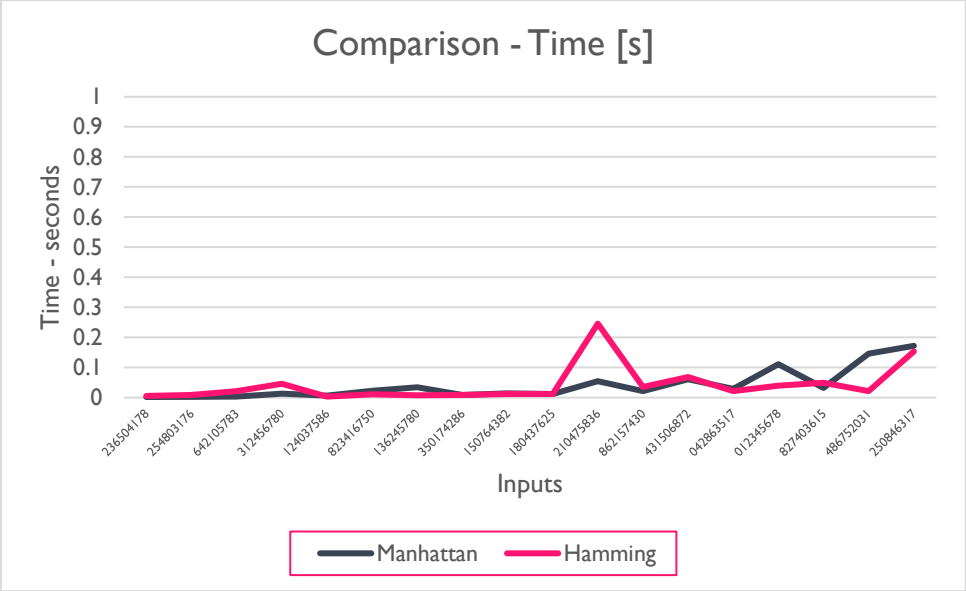
Path Length from Initial to Goal State

When scrutinizing the path length, it becomes evident that the heuristic utilizing Manhattan distance as its guiding principle consistently outperforms its Hamming counterpart. This signifies that the Manhattan distance heuristic requires fewer steps, on average, to navigate from the starting configuration to the desired goal state. The numerical differences between the two heuristics illuminate the effectiveness of the Manhattan distance in guiding the algorithm to its target with greater efficiency.



Time elapsed

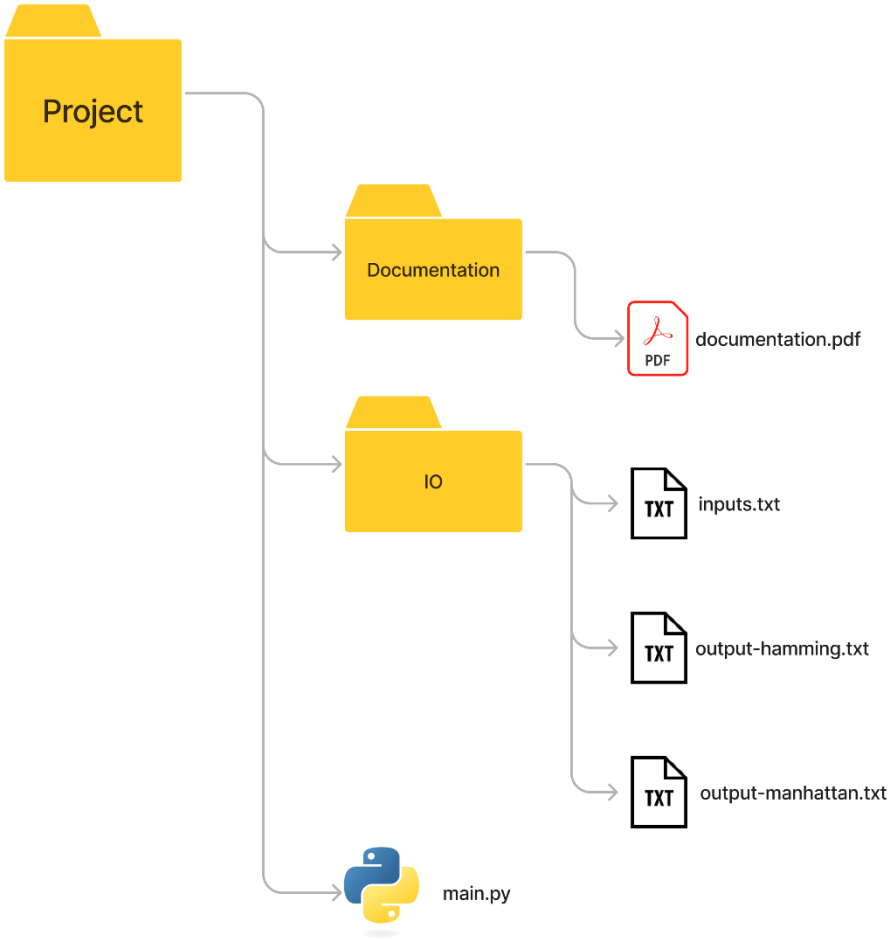
Delving into the time elapsed during the algorithm's execution, we note that distinguishing between the two heuristics in terms of time presents a more difficult challenge. While there are variations in the computational time taken by each heuristic, these distinctions are relatively subtle and might not significantly impact the overall performance in practical applications. It is essential to acknowledge that the choice between heuristics should primarily hinge on the specific requirements and priorities in this case the emphasis should be on memory and path length.



Conclusion

In conclusion, the comparative analysis reveals that the heuristic employing Manhattan distance consistently outperforms the Hamming heuristic in terms of the number of steps required to reach the goal state. However, the difference in computational time between the two heuristics is relatively small and not as discernible. The choice of heuristic should therefore be primarily guided by the specific requirements of the application, considering factors such as path length and computational efficiency.

User guide



For users interested in utilizing the program, the process begins by accessing the 'inputs.txt' file located at the '/IO/' directory. Within this file, inputs should be provided in the following format: '123456780.' Each string should consist of nine characters, comprising numbers ranging from 1 to 8, and the number 0, symbolizing an empty tile.

The implementation has established the 'goal_board' as follows: ((1, 2, 3), (4, 5, 6), (7, 8, 0)). If users wish to modify this configuration, they need to make alterations in the following line:

```
232     goal = '123456780'
```

To select the preferred heuristic function, whether it be Manhattan or Hamming, users should navigate to lines 235 through 239. The appropriate line should be uncommented based on the chosen heuristic:

```
235     # Manhattan distance
236     # heuristic = 1
237
238     # Hamming distance
239     heuristic = 2
```

Furthermore, it is advisable to customize the output file based on the selected heuristic. This can be achieved by modifying the following line:

```
227     with open("IO/output-hamming.txt", "w", encoding="utf-8") as output_file:
```

By following these instructions, users can tailor the program's behaviour to suit their specific requirements, making it a versatile tool for a wide range of applications.