2023

8-PUZZLE

Search Algorithms

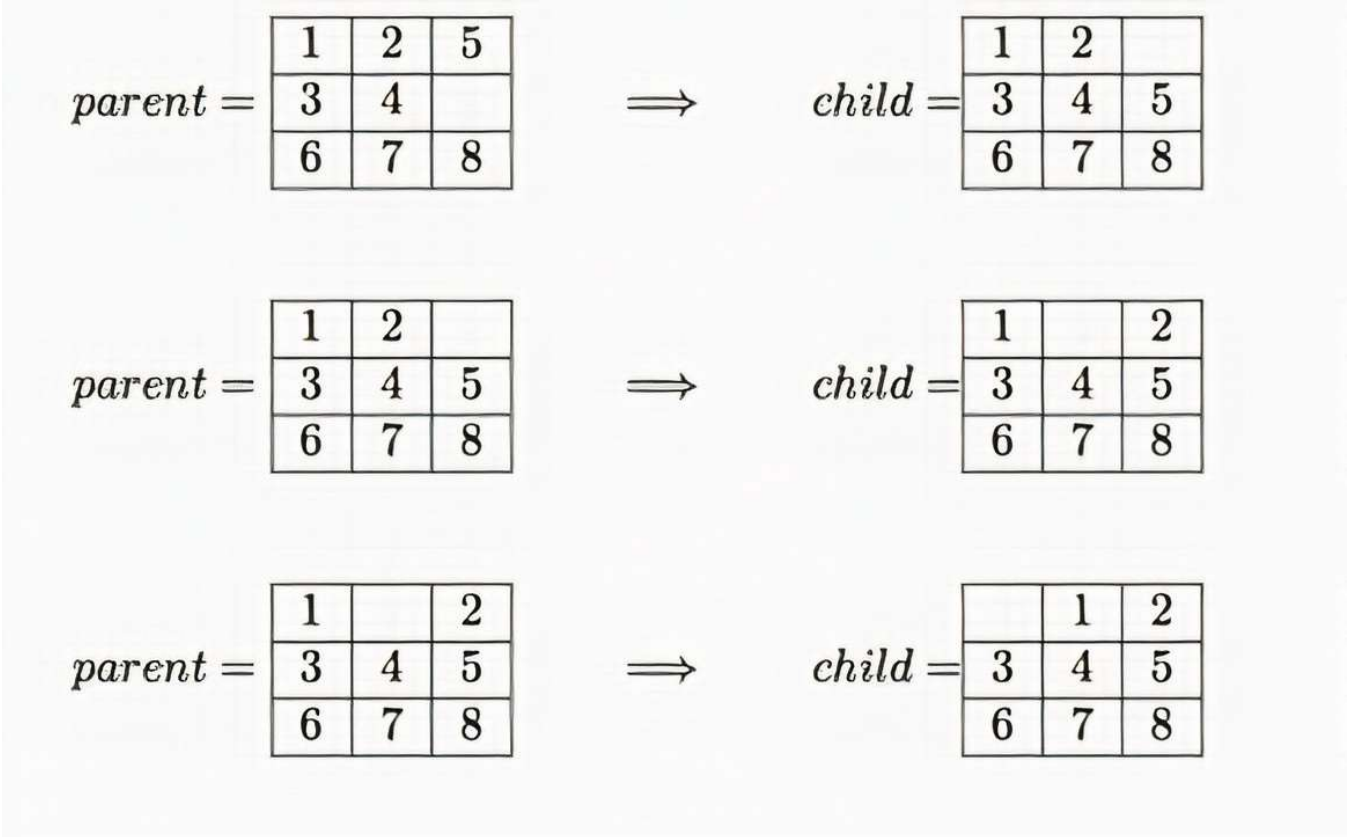
Team

Hussein Mohamed
Abdelrahman Wael
Ahmed Hesham Ahmed
Mohamed Raffeek

Overview

This problem appeared as a project in the edX course ColumbiaX: CSMM.101x Artificial Intelligence (AI). In this assignment an agent will be implemented to solve the 8-puzzle game.

An instance of the 8-puzzle game consists of a board holding 8 distinct movable tiles, plus an empty space. For any such board, the empty space may be legally swapped with any tile horizontally or vertically adjacent to it. In this assignment, the blank space is going to be represented with the number 0. Given an initial state of the board, the search problem is to find a sequence of moves that transitions this state to the goal state; that is, the configuration with all tiles arranged in ascending order 0,1,2,3,4,5,6,7,8 .The search space is the set of all possible states reachable from the initial state. The blank space may be swapped with a component in one of the four directions 'Up', 'Down', 'Left', 'Right', one move at a time. The cost of moving from one configuration of the board to another is the same and equal to one. Thus, the total cost of path is equal to the number of moves made from the initial state to the goal state.



Data structures

- Tuples: The data structure responsible for the storage of frontiers is a tuple of heuristic score and the node itself.
- Hashset: A helper data structure to store each occurrence for the frontiers for faster access to check for any replication of the nodes in constant time O(1).
- Hashmap: In order to store the parents of each node on our path to the goal node.
- MinHeap: Works as our priority queue for the frontiers to get the node with the least manhattan/euclidean distance.

User-defined classes

- Node: Represents the state as an object with string attribute and depth
- SolverCommand: The base class for the solver functions
- EightPuzzle: Class containing the functions that operate on the puzzle state

Assumptions

- A board state is an object with two attributes: data and depth

  data: string representing the 8 numbers and 0 as the blank space

  depth: is the distance from the initial state to the current node in search tree

- The goal state is 012345678

Algorithms

- **BFS:**

BFS searching algorithm works by:

- **Starting at the initial state**
- **If the current state is the goal state, the algorithm stops**
- **If not, it starts exploring all the neighbors (possible next states) one by one and so on until reaching goal**
- **States are stored in queue data structure which grantees that the sequence of exploration is level by level**

BFS search

```
function BREADTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Queue.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.dequeue()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.enqueue(neighbor)

    return FAILURE
```

- **DFS:**

DFS searching algorithm works by:

- **Starting at the initial state**
- **If the current state is the goal state, the algorithm stops**
- **If not, it starts exploring certain neighbor (one of the possible next states) one by one and so on until reaching goal or until reaching an end state (state with no next state) then the algorithm rolls back**
- **States are stored in stack data structure which grantees that the sequence of exploration is branch by branch**

DFS search

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Stack.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.push(neighbor)

    return FAILURE
```

Taken from the edX course ColumbiaX: CSMM.101x Artificial Intelligence (AI)

- A*:

A* searching algorithm works by:

- Starting at the initial state keeping in mind the cost to reach the current state (which is 0 for initial state)
- The cost for moving from a state to another in this problem is 1
- If the current state is the goal state, the algorithm stops
- If not, it starts exploring the minimum cost state stored and so on until reaching goal
- States are stored in priority queue data structure which grantees that the sequence of exploration is the smallest cost state

A* search

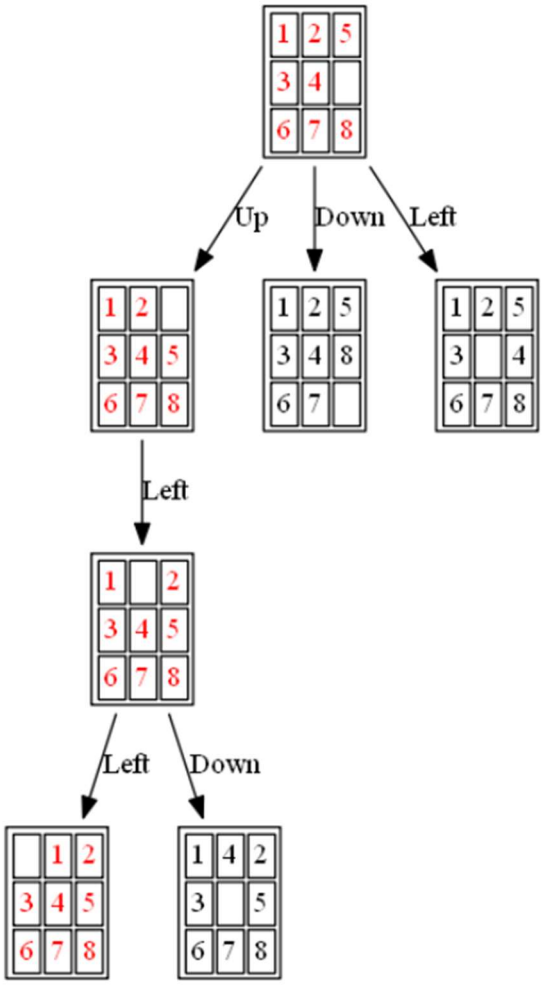
Taken from the edX course ColumbiaX: CSMM.101x Artificial Intelligence (AI)

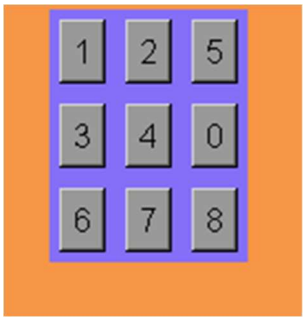
```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

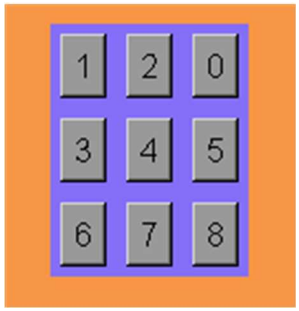
Sample Runs

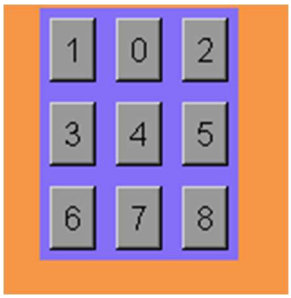
1) Path to goal on Tutorial example:

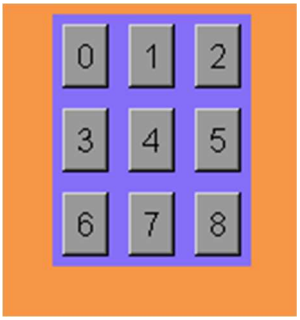
In this example the 4 algorithms go through the same path









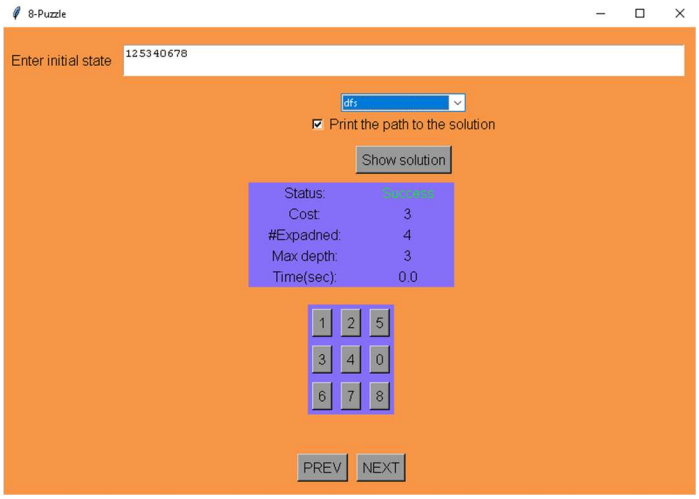


2) Easy test case: 125340678

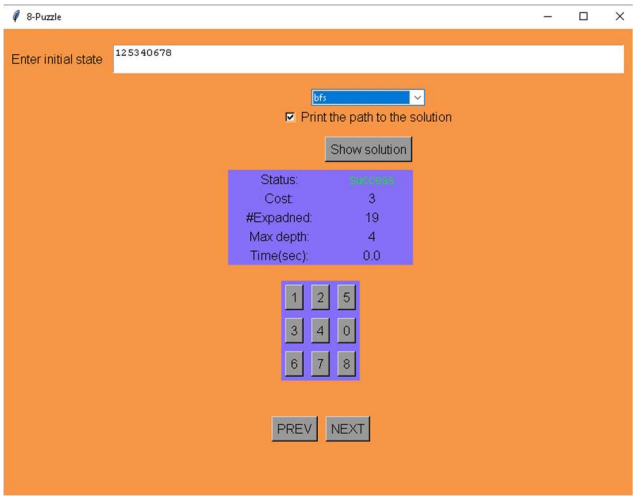
Goal state on the left end of the search tree

DFS is the optimal in this case
BFS expands the most nodes
A* work as DFS

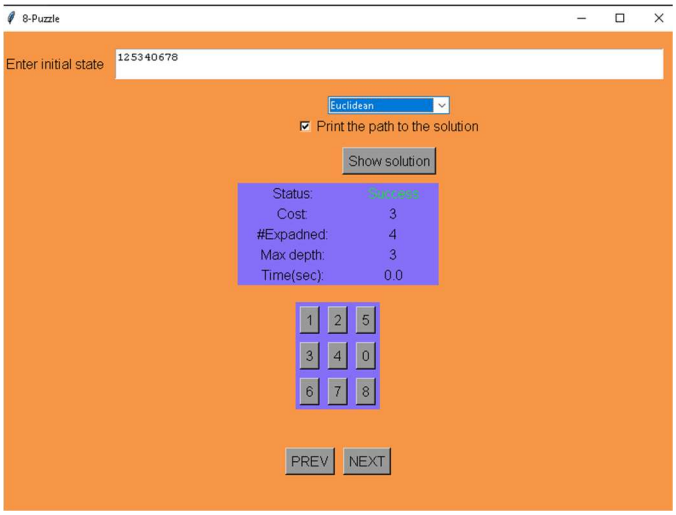
DFS



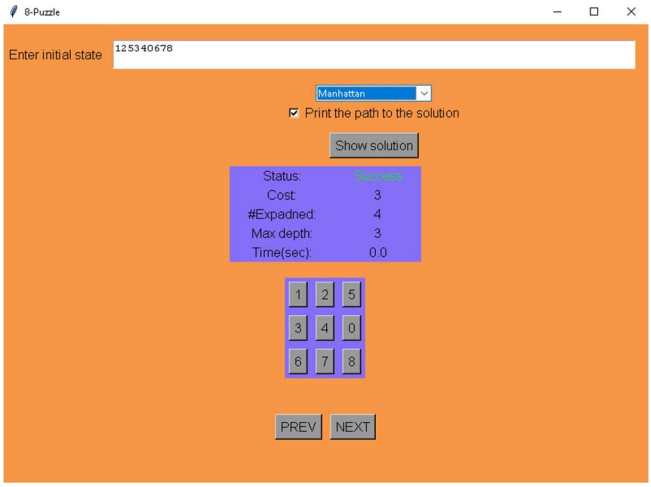
BFS



Euclidean



Manhattan

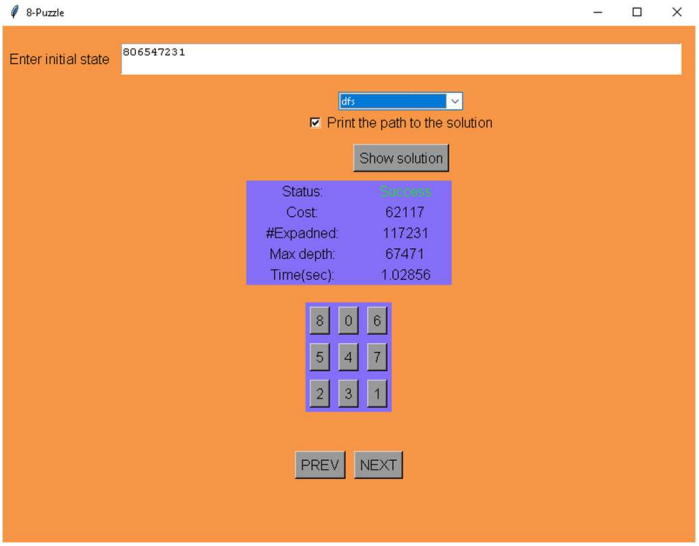


3) Hard Test case 806547231

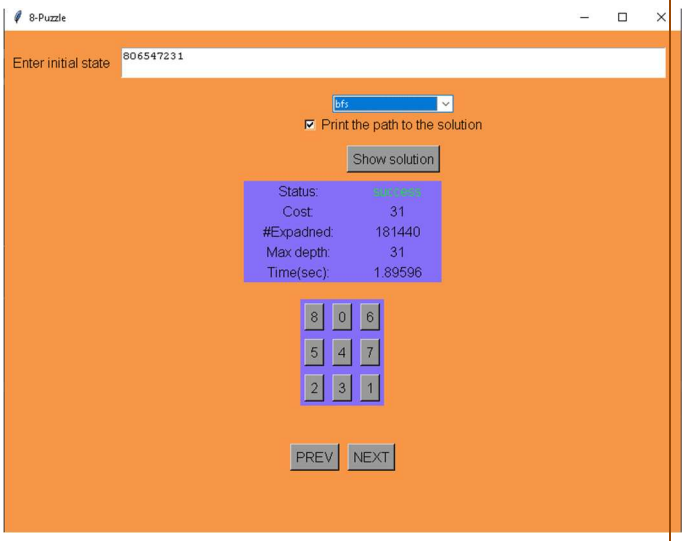
The goal state lies at the far bottom end of the search tree

BFS was the worst when solution is on the bottom right (Max time and nodes expanded)
DFS has the worst path cost
A* was the best choice as in less nodes expanded and time taken
Manhattan usually has better outcomes as how the heuristic is more suitable for the puzzle

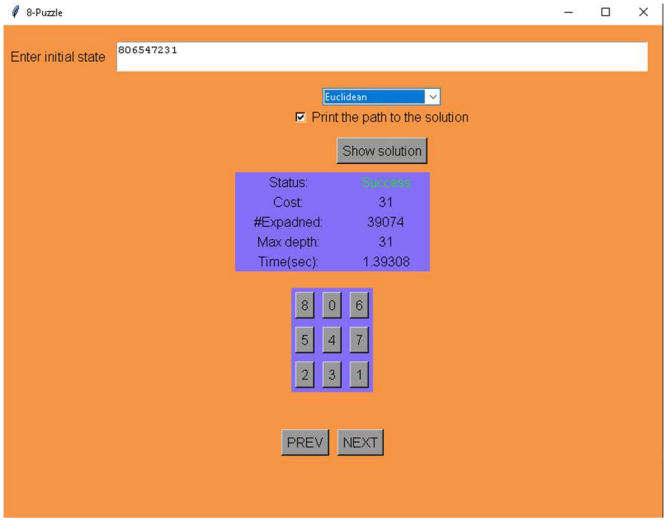
DFS



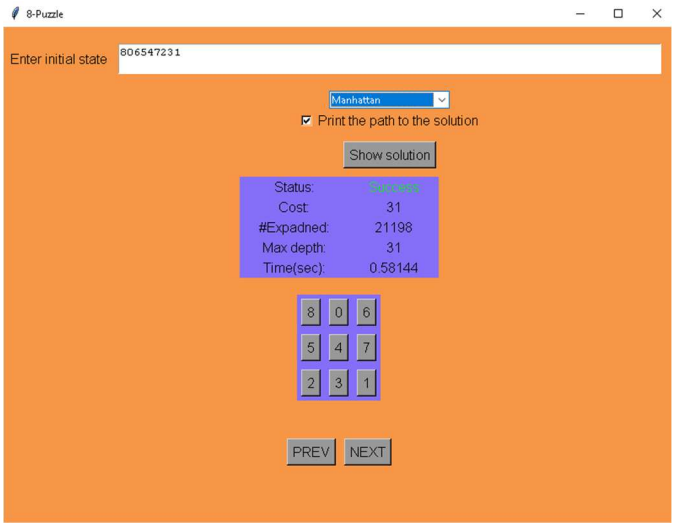
BFS



Euclidean



Manhattan

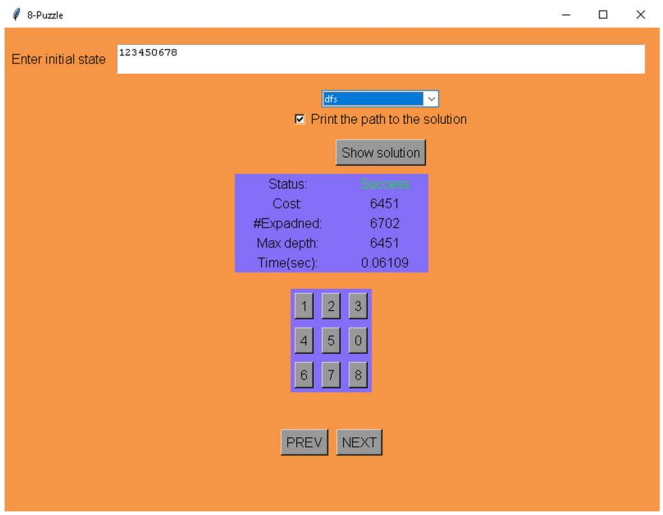


4) Medium Test case: 123450678

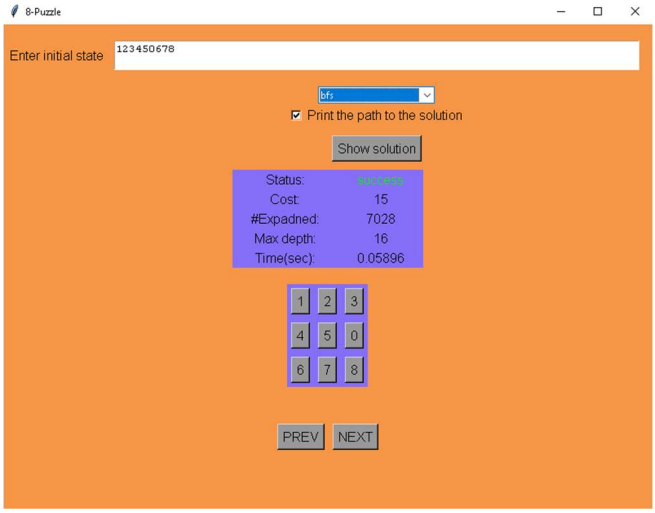
The goal state at the middle of the search tree

DFS performed the worst
A* is the best performing
Manhattan slightly better than Euclidean

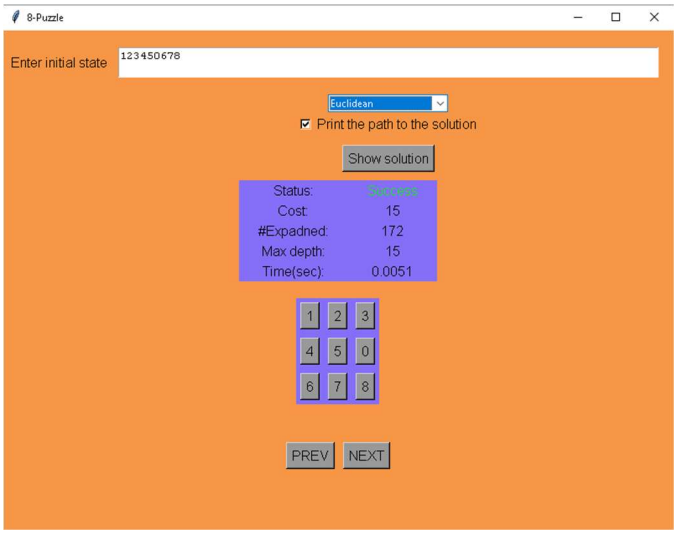
DFS



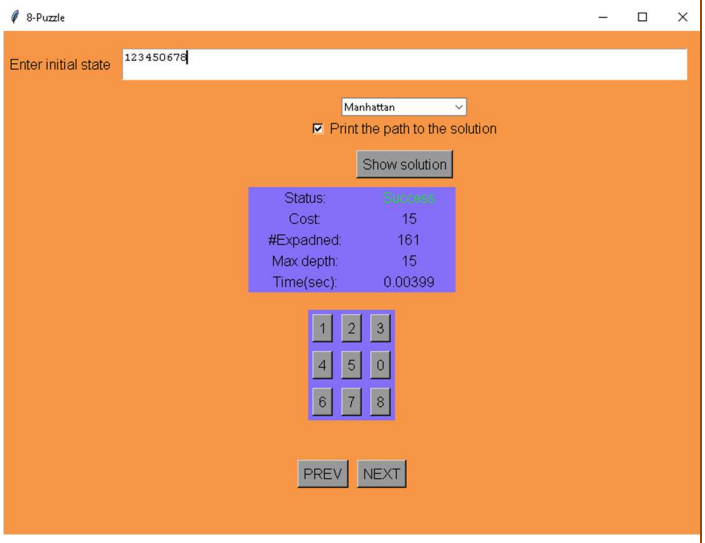
BFS



Euclidean

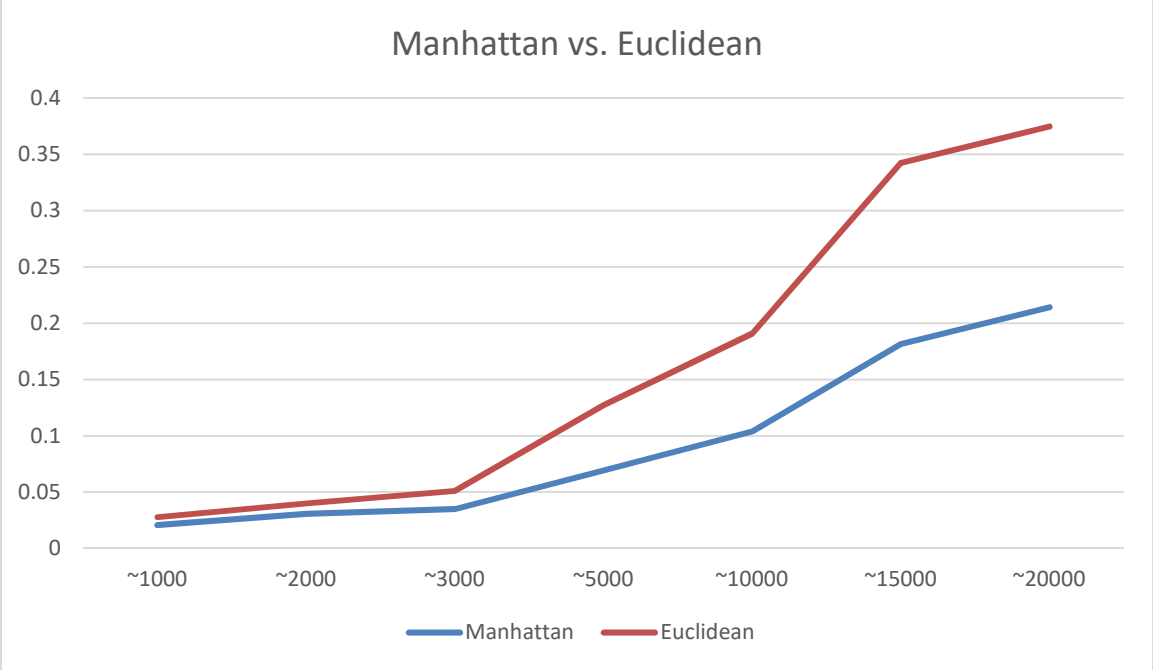


Manhattan



Manhattan vs. Euclidean

Expanded States	Average Time using Manhattan (sec)	Average Time using Euclidean (sec)
~1000	0.02077	0.02774
~2000	0.03077	0.04007
~3000	0.03479	0.05112
~5000	0.06926	0.12743
~10000	0.10388	0.19099
~15000	0.18154	0.34258
~20000	0.21436	0.37492



It is noticed that as the number of expanded states increase, the Manhattan heuristic performs better than the Euclidean heuristic.

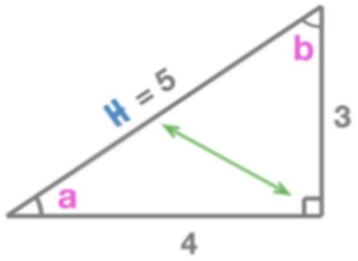
Why?

At first, we must prove that both heuristics are admissible:

- **Manhattan: it calculates the minimum number of moves to move a tile from its current position to its original position, therefore it is guaranteed to never overestimate the actual cost since it only considers horizontal and vertical movements, while diagonal movements are not allowed in the 8-puzzle problem. Therefore, the Manhattan heuristic is admissible.**
- **Euclidean: it is also an underestimate of the actual distance required to reach the goal state. This is because the Euclidean distance represents the shortest straight-line distance between two points, but in the 8-puzzle problem, only horizontal and vertical movements are allowed. Therefore, the Euclidean heuristic is also admissible.**

Now, we prove why is the Manhattan heuristic better, and for this we have two reasons:

1. The Manhattan heuristic is less admissible than the Euclidean heuristic, to prove this we can use the following triangle for illustration:



For any right angled triangle, the Manhattan distance between the two points a and b will always be greater than the Euclidean distance between the two points.
This is due to the nature of right angled triangles where the length of the hypotenuse is always less than the summation of the lengths of the two other sides.
For this given triangle, the Manhattan distance = 7, while the Euclidean distance = 5.
This proves that the Manhattan heuristic is less admissible than the Euclidean heuristic, which causes the Manhattan heuristic to **consistently have better estimations that are closer to the length of the actual path from the current state and the goal state.**

2. The Manhattan heuristic calculates the sum of the Manhattan distances between each tile's current position and its goal position. It only considers horizontal and vertical movements, which are the only valid moves in the 8 puzzle problem.
   On the other hand, the Euclidean heuristic calculates the Euclidean distance between each tile's current position and its goal position. It represents the shortest straight-line distance between two points, but in the 8-puzzle problem, only horizontal and vertical movements are allowed.
   Since the Manhattan heuristic considers only valid moves and does not account for diagonal movements, **it tends to provide a more accurate estimate of the actual number of moves required to reach the goal state**. In contrast, the Euclidean heuristic tend to be less accurate by considering diagonal movements that are not allowed in the 8-puzzle problem.

Optimality Analysis

Finally, we will discover which of the algorithms described above is the best for solving the 8-puzzle problem.

We wrote a program to run 1000 random initial states on each of the 4 algorithms (DFS, BFS, A* Manhattan, A* Euclidean), after it concluded, the following results were obtained:

	DFS	BFS	Manhattan	Euclidean
Average solving time (sec)	0.46486	1.03716	0.039	0.07744
Average number of expanded nodes	64661.043	95145.426	1738.922	2662.112
Average search depth	45606.816	23.095	22.096	22.096

At last, it was found that the most optimal algorithm to solve the 8-puzzle problem is the **A* Manhattan algorithm**, because it takes the least amount of time and expands the smallest number of nodes on average with a relatively low search depth.