Assignment 1 Report

Using Informed and Uninformed Search Algorithms to Solve 8-Puzzle

CS 482: Artificial Intelligence

Team

Aya Ashraf Saber	2
Khaled Barie Mostafa	21
Merit Victor Ageeb	69

Problem Statement

It is required to implement an agent to solve the 8-puzzle game. 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 which should correspond to the following board.

	1	2
3	4	5
6	7	8

Overview

The implementation was done using Java 11 and is divided into the following packages and modules:

- **algorithms**
  This package contains implementation of the required algorithms. It consists of the following classes:
  - **AbstractTreeSearch**
    An abstract class that implements common functionalities and acts as a contract for the implemented algorithms. Has the following methods:

    1. search: Abstract method implemented by each algorithm class
    2. runtimeMillis
    3. getSearchDepth
    4. getPathToGoal
    5. getGoal

  - **BreadthFirstSearch**
  - **DepthFirstSearch**
  - **AStarSearch**

- **puzzle**
  Contains classes related to the logic of the 8-puzzle and computing legal moves/neighbors of each game state. Consists of the following classes
  - **PuzzleState**
    Encapsulates the state of the board.
  - **PuzzleStateNode**
    Actual tree nodes traversed by the searching algorithm. Each node encapsulates a state and maintain a reference to their parent as well as their depth within the search tree
- **heuristics**
  This package contains classes heuristics used in A* search. Contains the following:
  - **HeuristicEvaluator**
    An interface implemented by heuristic evaluator so they can be passed at runtime when executing A* search
  - **ManhattanHeuristic**
  - **EuclideanHeuristic**

Algorithms & Data Structures

The algorithms used in each search technique closely resemble the pseudocode provided in the assignment description. The game has a maximum branching factor (b) of 4 and a maximum depth (m) of 31 as any solvable configuration should be possible to solve in 31 moves or less.

1. **Breadth First Search**
   Uses an ArrayDequeue as the fringe.
2. **Depth First Search**
   Uses a stack as the fringe

3. **A* Search**
   Uses a priority queue as the fringe. Since Java's priority queue does not support the decreaseKey operation, it is simulated by removing the object and reinserting it with a different key.

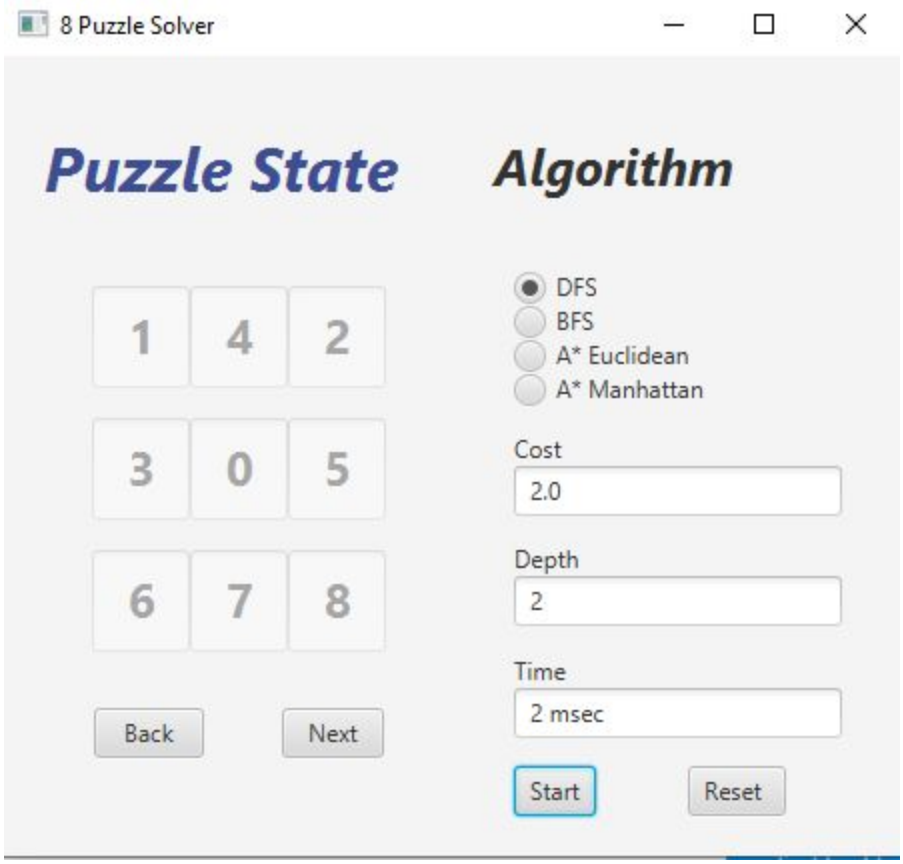
A hashset was used to implement the explored set in all 3 algorithms.

Assumptions

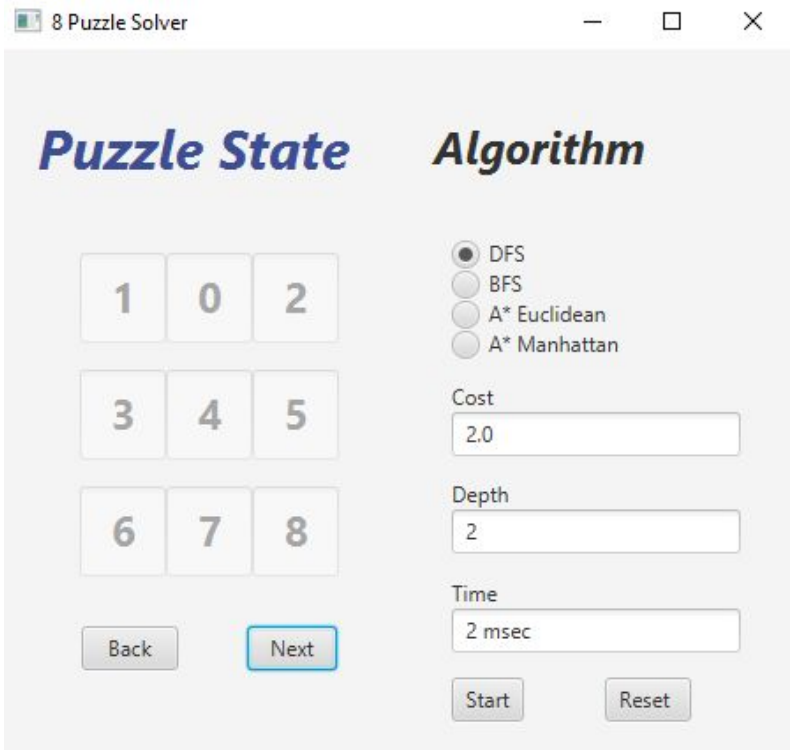
1. The given initial configuration is solvable
2. The goal state is fixed as the one given in the assignment description.

Sample Runs

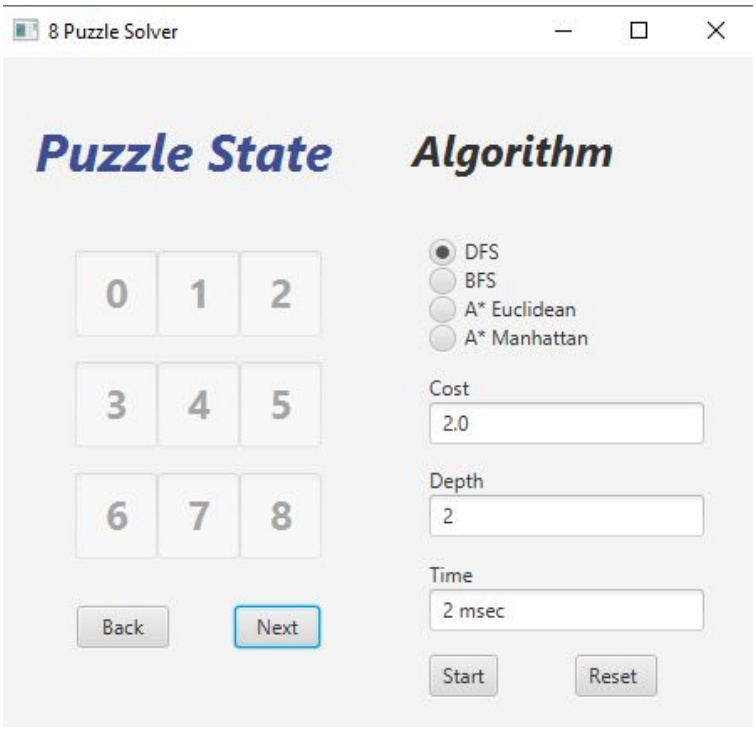
DFS:



Step 1:

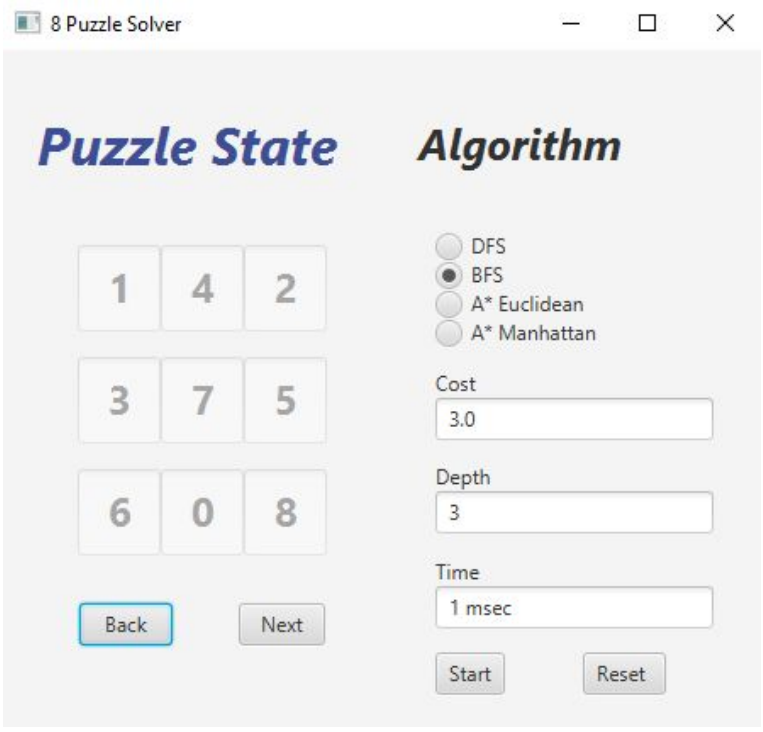

8 Puzzle Solver
Puzzle State
Algorithm
1 0 2
3 4 5
6 7 8
DFS
BFS
A* Euclidean
A* Manhattan
Cost
2.0
Depth
2
Time
2 msec
Back
Next
Start
Reset

Step 2:

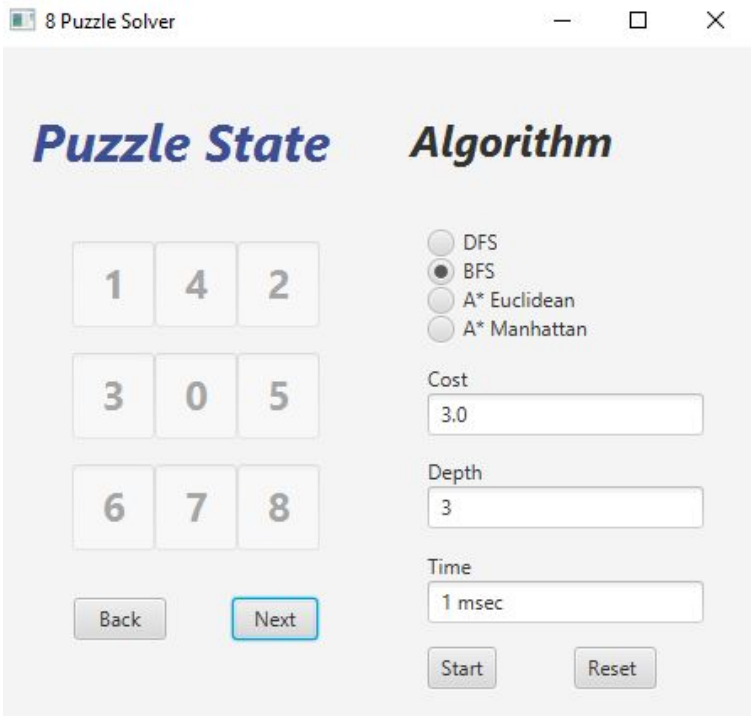

8 Puzzle Solver
Puzzle State
Algorithm
0 1 2
3 4 5
6 7 8
DFS
BFS
A* Euclidean
A* Manhattan
Cost
2.0
Depth
2
Time
2 msec
Back
Next
Start
Reset

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

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

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

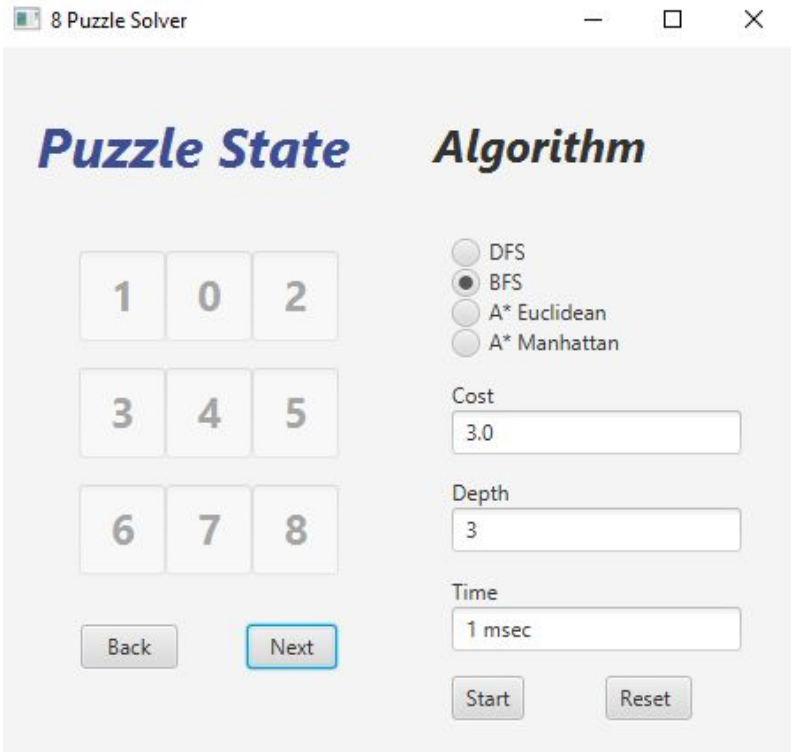
BFS



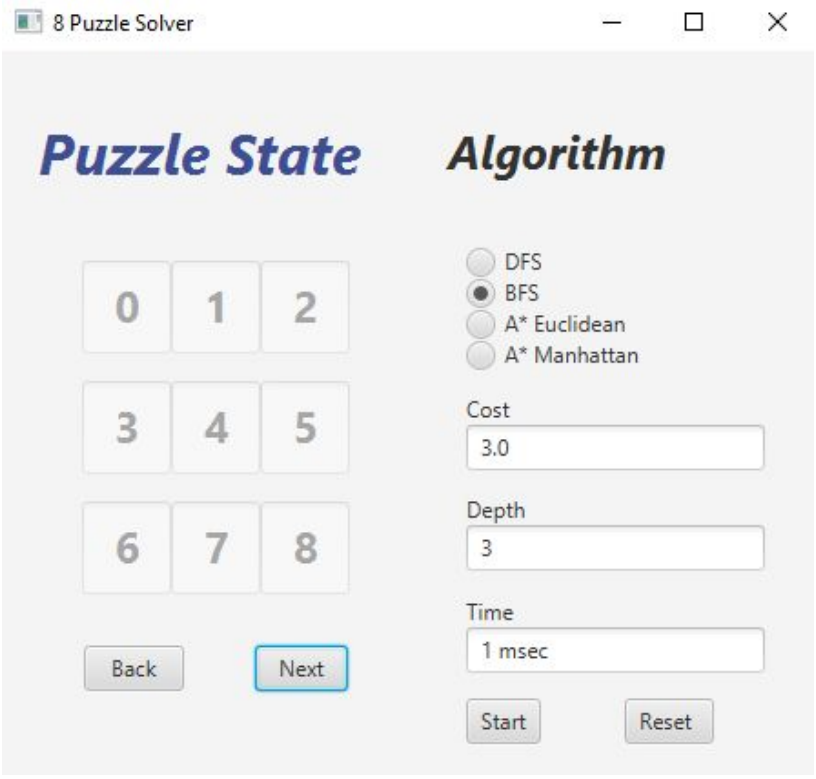
Step 1:


8 Puzzle Solver
Puzzle State
Algorithm
1 4 2
3 0 5
6 7 8
Back
Next
DFS
BFS
A* Euclidean
A* Manhattan
Cost
3.0
Depth
3
Time
1 msec
Start
Reset

Step 2:


8 Puzzle Solver
Puzzle State
Algorithm
1 0 2
3 4 5
6 7 8
Back
Next
DFS
BFS
A* Euclidean
A* Manhattan
Cost
3.0
Depth
3
Time
1 msec
Start
Reset

Step 3:

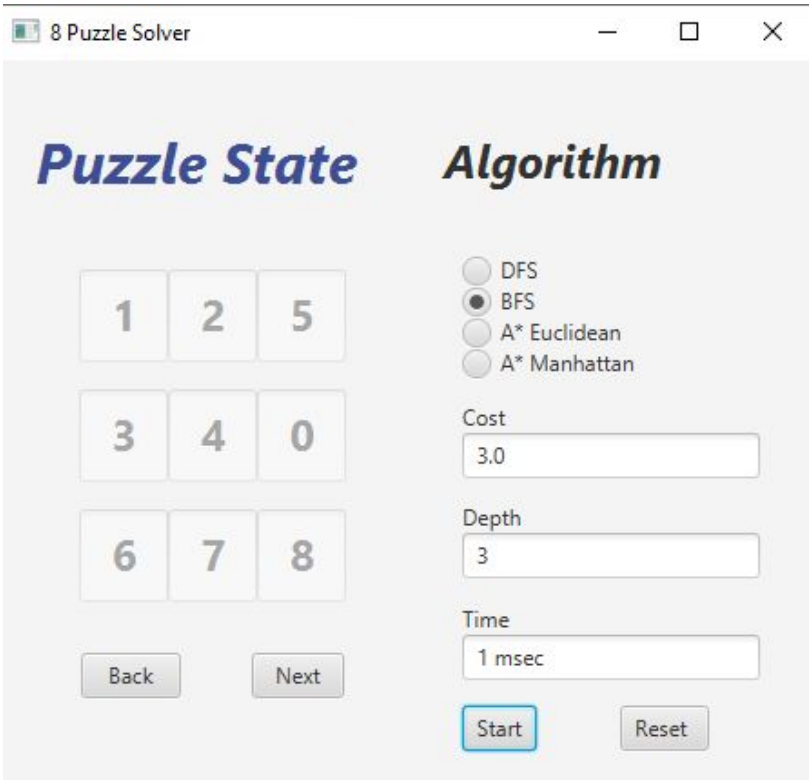

8 Puzzle Solver
Puzzle State
Algorithm
0
1
2
3
4
5
6
7
8
Back
Next
DFS
BFS
A* Euclidean
A* Manhattan
Cost
3.0
Depth
3
Time
1 msec
Start
Reset

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

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

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

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

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

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

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

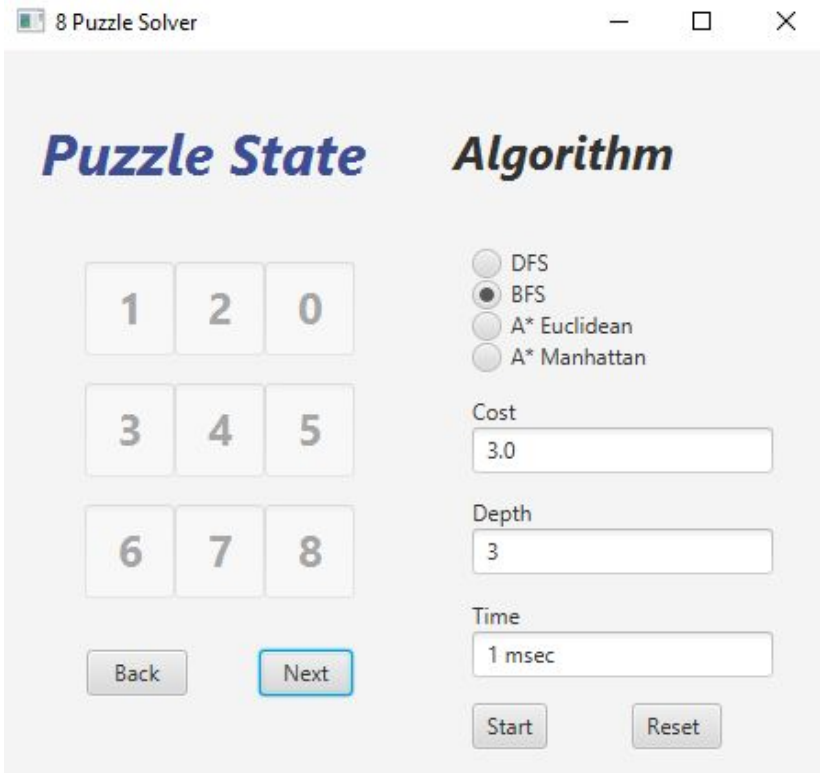
```
[1, 4, 2]
[0, 3, 5]
[6, 7, 8]

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

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

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

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

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

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

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

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

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

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

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

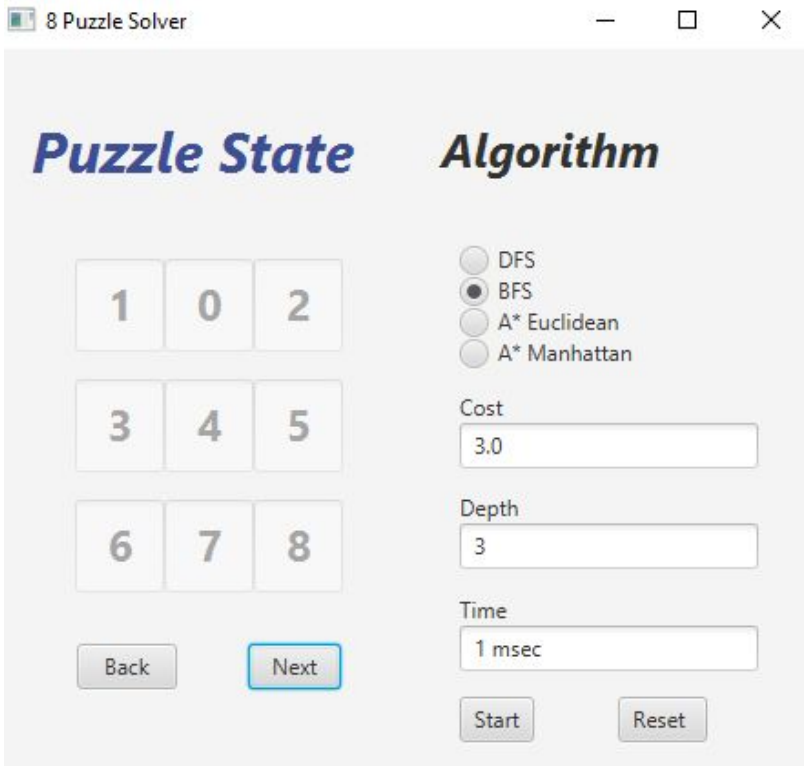
A*


8 Puzzle Solver
Puzzle State
Algorithm
1 2 5
3 4 0
6 7 8
Back
Next
DFS
BFS
A* Euclidean
A* Manhattan
Cost
3.0
Depth
3
Time
1 msec
Start
Reset

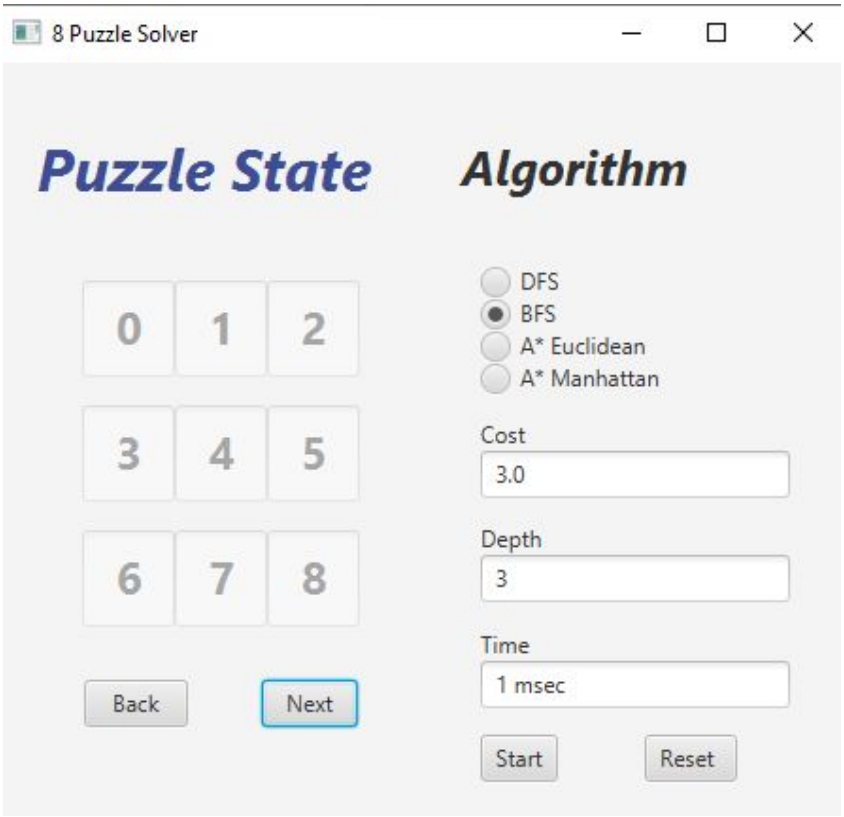
Step 1:


8 Puzzle Solver
Puzzle State
Algorithm
1 2 0
3 4 5
6 7 8
Back
Next
DFS
BFS
A* Euclidean
A* Manhattan
Cost
3.0
Depth
3
Time
1 msec
Start
Reset

Step 2:


8 Puzzle Solver
Puzzle State
Algorithm
1 0 2
3 4 5
6 7 8
Back
Next
DFS
BFS
A* Euclidean
A* Manhattan
Cost
3.0
Depth
3
Time
1 msec
Start
Reset

Step 3:


8 Puzzle Solver
Puzzle State
Algorithm
0 1 2
3 4 5
6 7 8
Back
Next
DFS
BFS
A* Euclidean
A* Manhattan
Cost
3.0
Depth
3
Time
1 msec
Start
Reset

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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