8-puzzle game

ID	NAME
6338	Ahmed mohamed ahmed ali
6672	Mostafa mohamed tawfik
6160	Mazen medhat farid
6670	Mohamed zayton

Algorithms:

BFS:

```
package app.algorithms;

import app.IntState;
import java.util.*;


public class BFS {

    private int maxDepth;
    private Queue<Integer> frontier = new LinkedList<>();
    private HashSet<Integer> explored = new HashSet<>();
    private HashMap<Integer, Integer> parentMap = new HashMap<>();

    public List<Integer> BFS(int initialState) {
        frontier.clear();
        explored.clear();
        parentMap.clear();
        HashMap<Integer, Integer> depth_map = new HashMap<>();
        IntState intState = new IntState();
        frontier.add(initialState);
        parentMap.put(initialState, initialState);
        this.maxDepth = 0;
        depth_map.put(initialState, 0);
        boolean goalFound = false;
        int currState;
        while (!frontier.isEmpty()) {
            currState = frontier.poll();
            if (explored.contains(currState))
                continue;
            else if (intState.isGoalState(currState)) {
                goalFound = true;
                break;
            }
            explored.add(currState);
            List<Integer> neighbors = intState.getNeighborIntStates(currState);
            int dep = depth_map.get(currState);
            for (int n : neighbors) {
                if (explored.contains(n))
```

```
                continue;
            depth_map.put(n, dep+1);
            if (dep + 1 > this.maxDepth)
                this.maxDepth = dep+1;
            frontier.add(n);
            parentMap.put(n, currState);
        }
    }

    return goalFound ? AlgorithmsBackTrack.backTrackPath(parentMap,
intState.getGoalState()) : null;
  }

  public int getNumberOfExpanded(){
    return this.parentMap.size();
  }

  public int getMaxDepth() {
    return maxDepth;
  }
}
```

DFS:

```
package app.algorithms;

import app.IntState;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Stack;

public class DFS {

    private int maxDepth;
    private Stack<Integer> frontier = new Stack<>();
    private HashSet<Integer> explored = new HashSet<>();
    private HashMap<Integer, Integer> parentMap = new HashMap<>();

    public List<Integer> DFS(int initialState) {
        frontier.clear();
        explored.clear();
        parentMap.clear();
        HashMap<Integer, Integer> depth_map = new HashMap<>();
        IntState intState = new IntState();
        frontier.add(initialState);
        parentMap.put(initialState, initialState);
        depth_map.put(initialState, 0);
        boolean goalFound = false;
        int currState;
        this.maxDepth = 0;
        while (!frontier.isEmpty()) {
            currState = frontier.pop();
            if (explored.contains(currState))
                continue;
            else if (intState.isGoalState(currState)) {
                goalFound = true;
                break;
            }

            explored.add(currState);
            List<Integer> neighbors = intState.getNeighborIntStates(currState);
            int dep = depth_map.get(currState);
            for (int n : neighbors) {
```

```
            if (explored.contains(n))
                continue;
            depth_map.put(n, dep+1);
            if (dep + 1 > this.maxDepth)
                this.maxDepth = dep+1;
            frontier.add(n);
            parentMap.put(n, currState);
        }

    }

    return goalFound ? AlgorithmsBackTrack.backTrackPath(parentMap,
intState.getGoalState()) : null;
  }

  public int getNumberOfExpanded(){
    return this.parentMap.size();
  }

  public int getMaxDepth() {
    return maxDepth;
  }
}
```

A_star:

```
package app.algorithms;

import app.IntState;

import java.util.*;

public class A_STAR {

  private int maxDepth;
  private HashMap<Integer, Integer> parentMap = new HashMap<>();
  private class StateHeuristicHolder {
    private int state;
    private byte g;
    private byte h;

    private StateHeuristicHolder(int state, byte g, byte h) {
      this.state = state;
      this.g = g;
      this.h = h;
    }

    public int getState() {
      return state;
    }

    public byte getG() {
      return g;
    }

    public byte getH() {
      return h;
    }

    public byte getHeuristic() {
      return (byte) (h + g);
    }
  }

  private class HeuristicComparator implements Comparator<StateHeuristicHolder> {
    @Override
```

```
    public int compare(StateHeuristicHolder o1, StateHeuristicHolder o2) {
      return o1.getHeuristic() - o2.getHeuristic();
    }
  }


  public List<Integer> AStar(int initialState, IntState.HeuristicsType heuristicsType) {
    HeuristicComparator comparator = new HeuristicComparator();
    PriorityQueue<StateHeuristicHolder> frontier = new
PriorityQueue<StateHeuristicHolder>(comparator);
    HashSet<Integer> explored = new HashSet<>();
    IntState intState = new IntState();
    frontier.add(new StateHeuristicHolder(initialState, (byte) 0, (byte) 0));
    parentMap.put(initialState, initialState);

    boolean goalFound = false;
    StateHeuristicHolder currStateHeuristicHolder;
    int currState;
    this.maxDepth = 0;
    while (!frontier.isEmpty()) {
      currStateHeuristicHolder = frontier.poll();
      currState = currStateHeuristicHolder.getState();
      if (explored.contains(currState))
        continue;
      else if (intState.isGoalState(currState)) {
        goalFound = true;
        break;
      }

      if (this.maxDepth < currStateHeuristicHolder.g){
        this.maxDepth = currStateHeuristicHolder.g;
      }
      explored.add(currState);
      List<Integer> neighbors = intState.getNeighborIntStates(currState);
      for (int n : neighbors) {
        if (explored.contains(n))
          continue;

        frontier.add(new StateHeuristicHolder(n, (byte)
(currStateHeuristicHolder.getG() + 1), (byte) (intState.getHeuristics(n,
heuristicsType))));
```

```
            parentMap.put(n, currState);
          }
        }

        return goalFound ? AlgorithmsBackTrack.backTrackPath(parentMap,
intState.getGoalState()) : null;
    }

    public int getNumberOfExpanded(){
        return this.parentMap.size();
    }

    public int getMaxDepth(){
        return this.maxDepth;
    }


}
```

HEURISTICS:

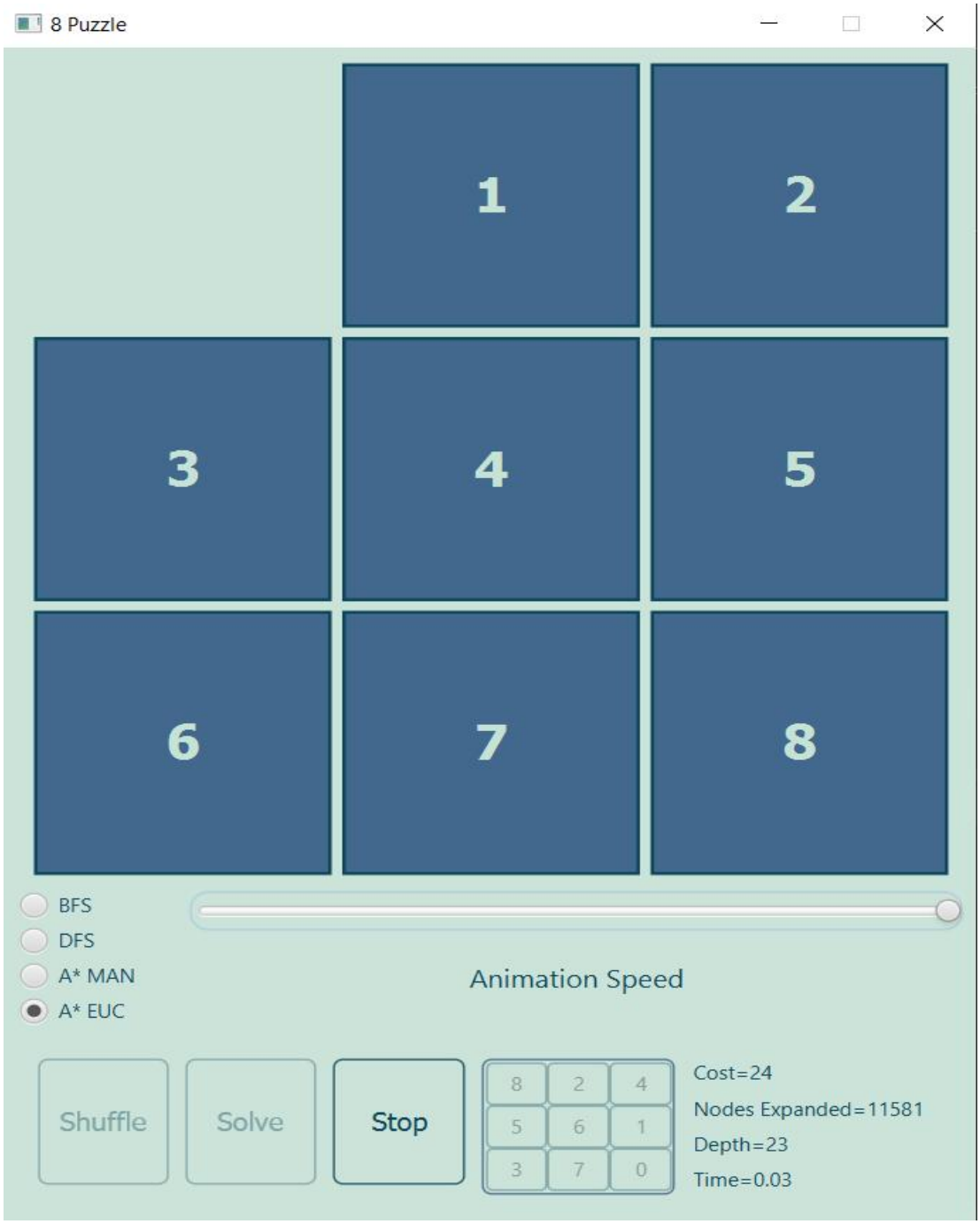
```
package app;

public class Heuristics {

  public enum HeuristicsType {
    NONE, MANHATTAN, EUCLIDEAN;
  }

   public int getHeuristics(State state, HeuristicsType heuristicsType) {
      if (heuristicsType == HeuristicsType.NONE)
         return 0;
      else if (heuristicsType == HeuristicsType.MANHATTAN)
         return calManhattan(state);
      else if (heuristicsType == HeuristicsType.EUCLIDEAN)
         return calEuclidean(state);
      else
         return 0;
   }


   private int calManhattan(State state) {
      int boardRowsNum = (int) Math.sqrt(state.getBoardSize());
      int emptySlotNum = state.getEmptySlotNum();
      int h = Math.abs(boardRowsNum - 1 - emptySlotNum/boardRowsNum) +
Math.abs(boardRowsNum - 1 - emptySlotNum%boardRowsNum);

      for (int i = 0, actualRow, actualCol, slotNum, currRow, currCol; i <
boardRowsNum; i++) {
         actualRow = i/boardRowsNum;
         actualCol = i%boardRowsNum;
         slotNum = state.getValSlot((byte) (i + 1));
         currRow = slotNum/boardRowsNum;
         currCol = slotNum%boardRowsNum;
         h += Math.abs(actualRow - currRow) + Math.abs(actualCol - currCol);
      }

      return h;
   }
```

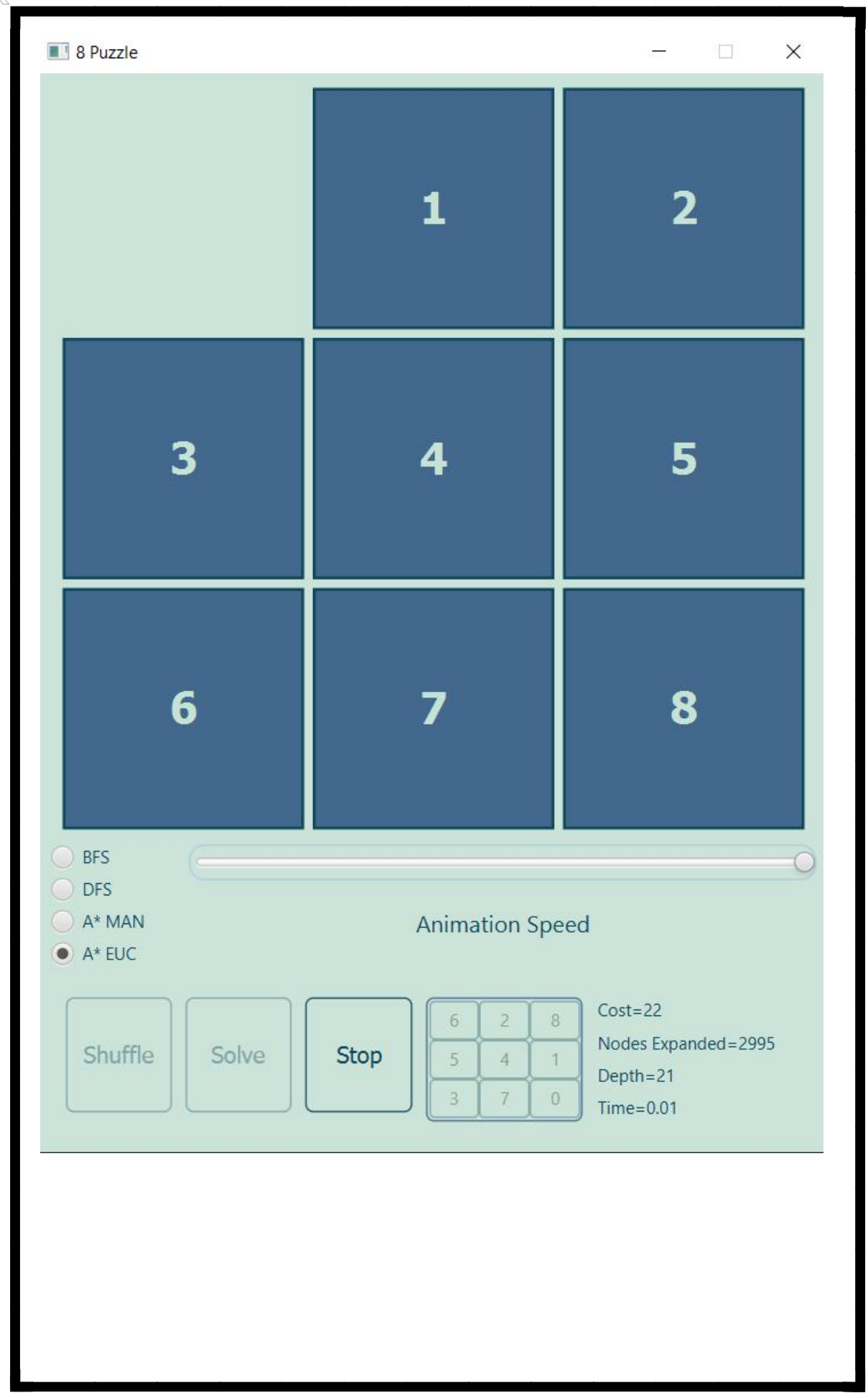
```
    private int calEuclidean(State state) {
        int boardRowsNum = (int) Math.sqrt(state.getBoardSize());
        int emptySlotNum = state.getEmptySlotNum();
        int h = (int) Math.sqrt(Math.pow((boardRowsNum - 1 -
emptySlotNum/boardRowsNum), 2) + Math.pow((boardRowsNum - 1 -
emptySlotNum%boardRowsNum), 2));

        for (int i = 0, actualRow, actualCol, slotNum, currRow, currCol; i <
state.getBoardSize(); i++) {
            actualRow = i/boardRowsNum;
            actualCol = i%boardRowsNum;
            slotNum = state.getValSlot((byte) (i + 1));
            currRow = slotNum/boardRowsNum;
            currCol = slotNum%boardRowsNum;

            h += Math.sqrt(Math.pow((actualRow - currRow), 2) + Math.pow((actualCol
- currCol), 2));
        }

        return h;
    }

}
```

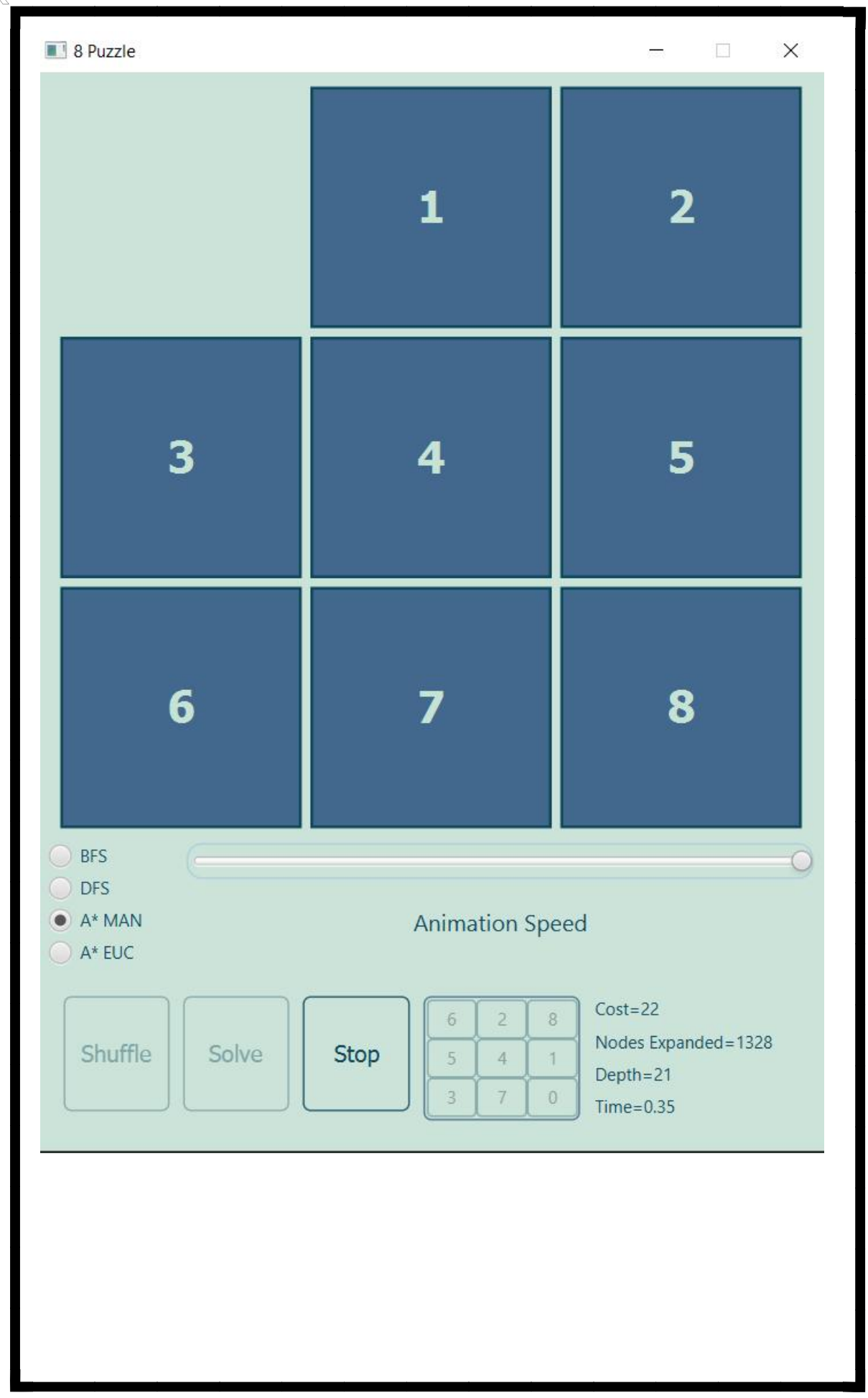
Data structures :

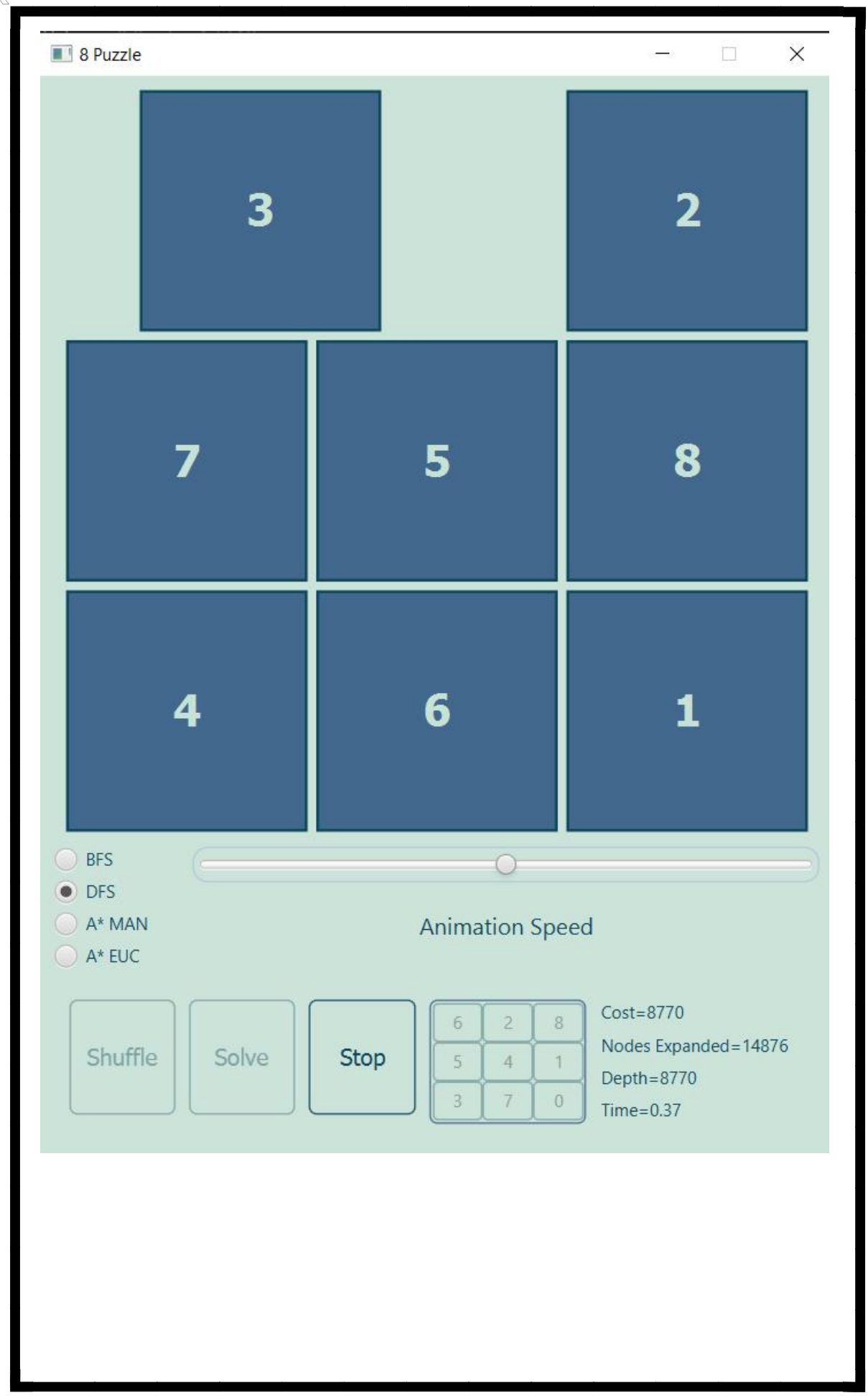
- STACK
- QUEUE
- HASH MAP
- PRIORITY QUEUE
- HASH SET
- VECTOR

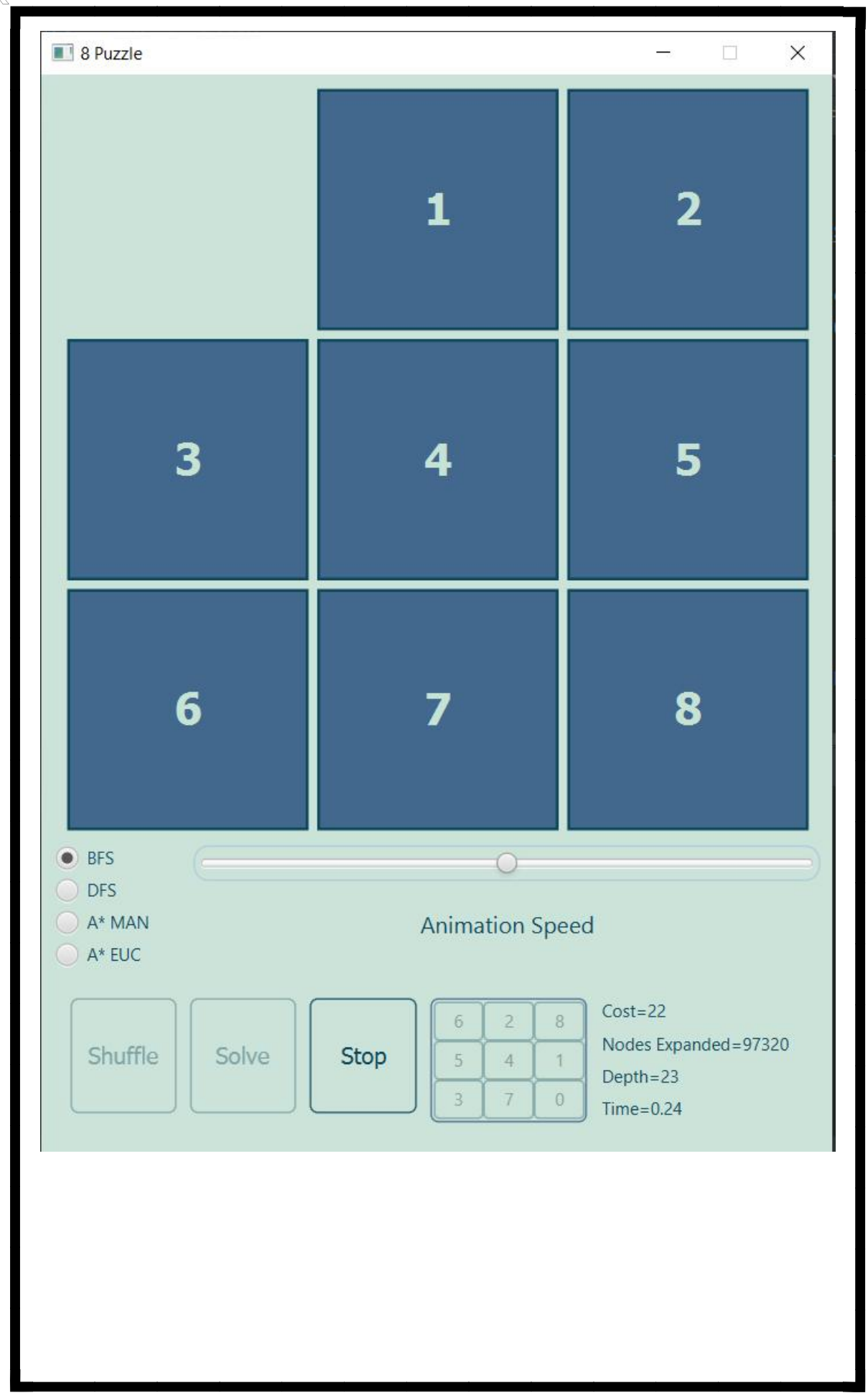
Sample Run:

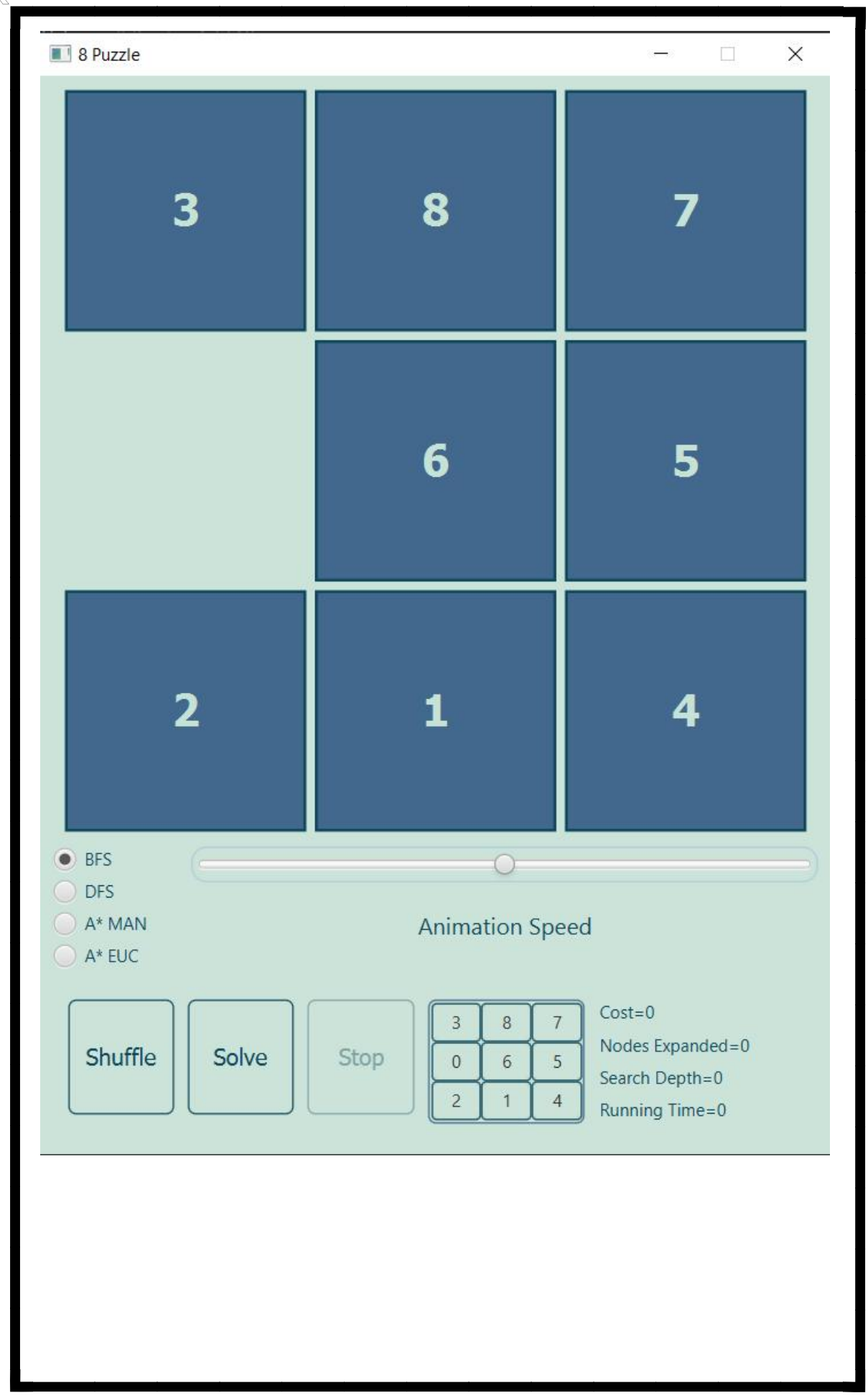

8 Puzzle
1
2
3
4
5
6
7
8
BFS
DFS
A* MAN
A* EUC
Animation Speed
Shuffle
Solve
Stop
8 2 4
5 6 1
3 7 0
Cost=24
Nodes Expanded=11581
Depth=23
Time=0.03

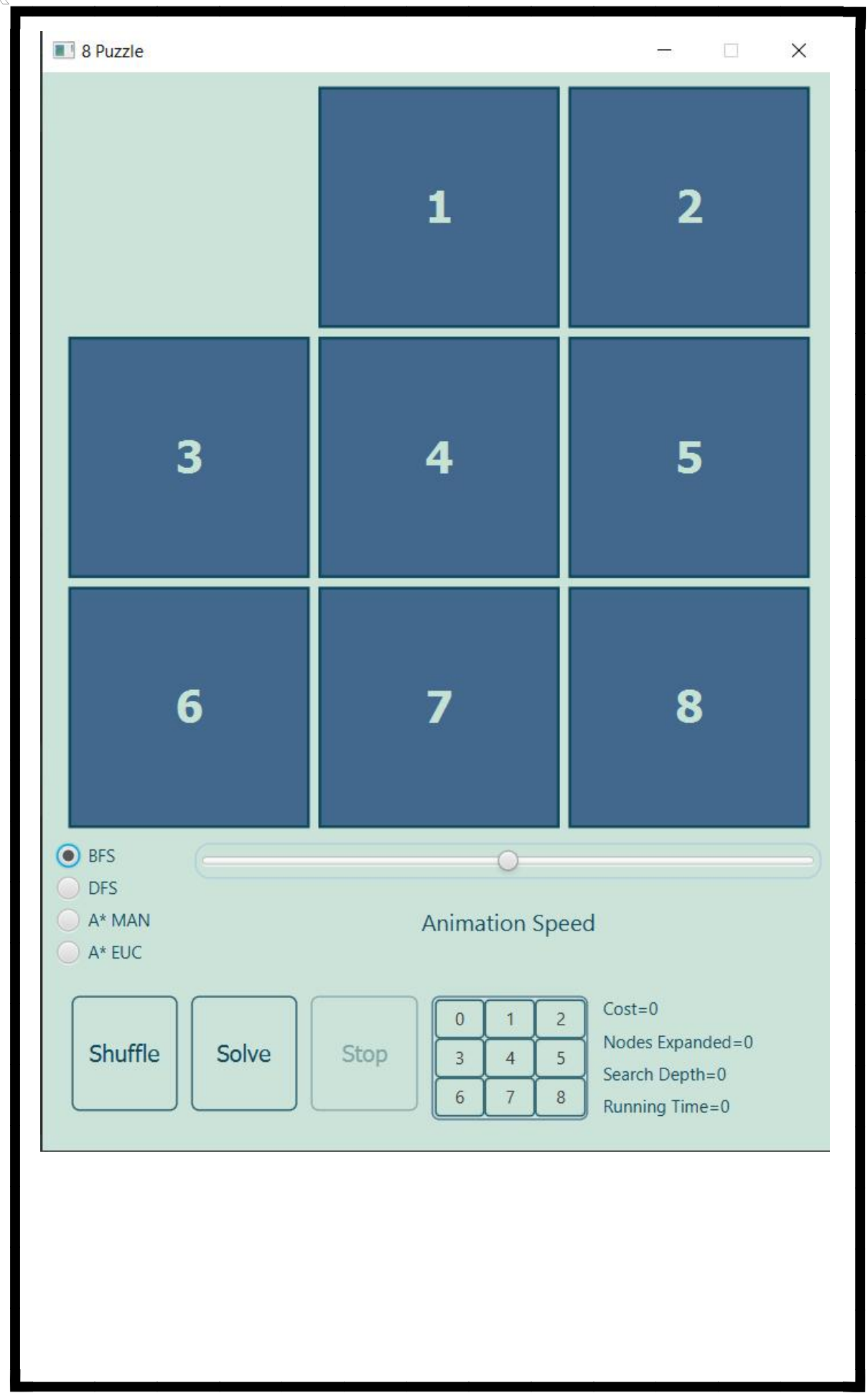

8 Puzzle
1
2
3
4
5
6
7
8
BFS
DFS
A* MAN
A* EUC
Animation Speed
Shuffle
Solve
Stop
6 2 8
5 4 1
3 7 0
Cost=22
Nodes Expanded=2995
Depth=21
Time=0.01


8 Puzzle
1
2
3
4
5
6
7
8
BFS
DFS
A* MAN
A* EUC
Animation Speed
Shuffle
Solve
Stop
6 2 8
5 4 1
3 7 0
Cost=22
Nodes Expanded=1328
Depth=21
Time=0.35

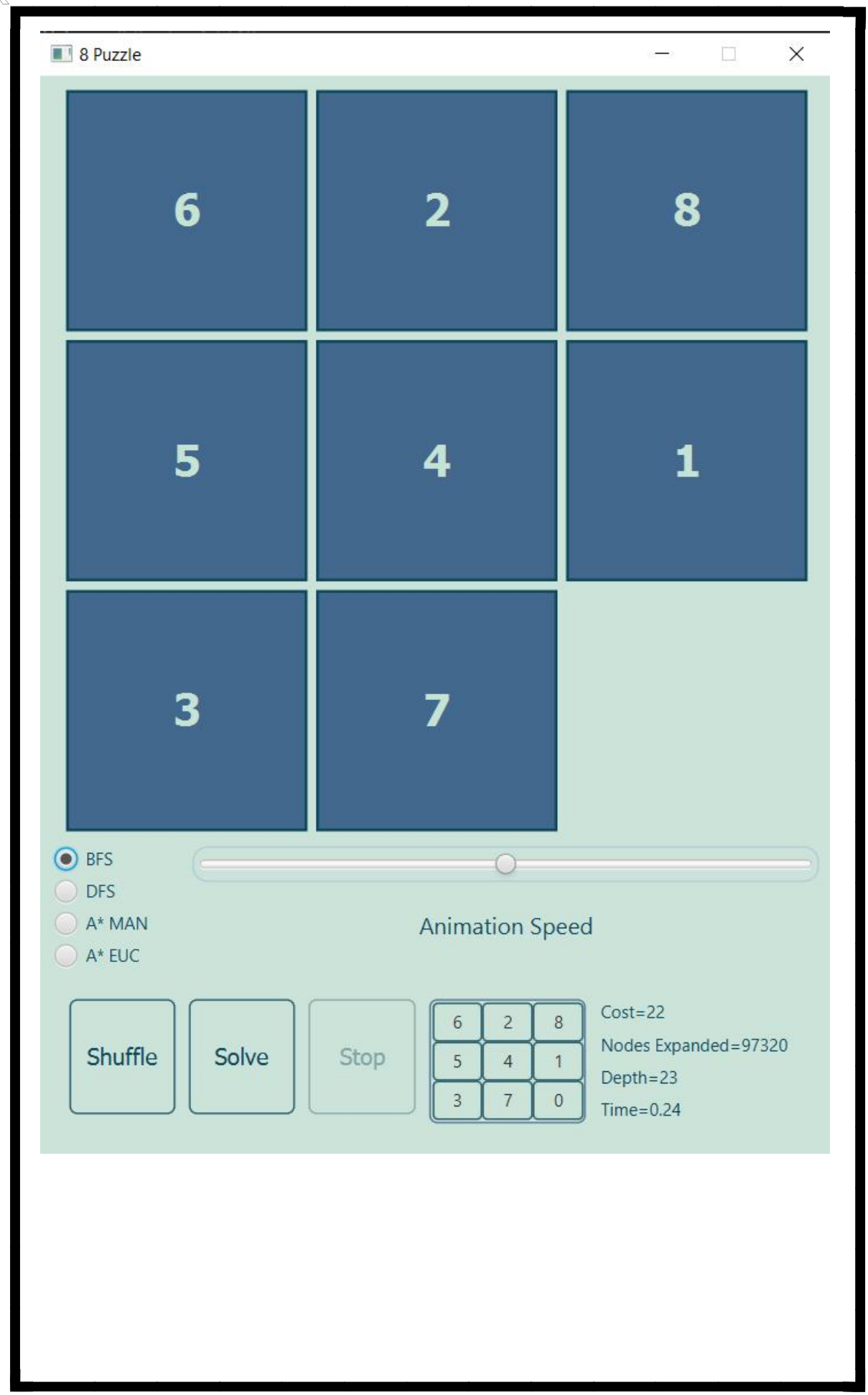

8 Puzzle
3
2
7
5
8
4
6
1
BFS
DFS
A* MAN
A* EUC
Animation Speed
Shuffle
Solve
Stop
6 2 8
5 4 1
3 7 0
Cost=8770
Nodes Expanded=14876
Depth=8770
Time=0.37


8 Puzzle
1
2
3
4
5
6
7
8
BFS
DFS
A* MAN
A* EUC
Animation Speed
Shuffle
Solve
Stop
6 2 8
5 4 1
3 7 0
Cost=22
Nodes Expanded=97320
Depth=23
Time=0.24


8 Puzzle
3
8
7
6
5
2
1
4
BFS
DFS
A* MAN
A* EUC
Animation Speed
Shuffle
Solve
Stop
3 8 7
0 6 5
2 1 4
Cost=0
Nodes Expanded=0
Search Depth=0
Running Time=0


8 Puzzle
1
2
3
4
5
6
7
8
BFS
DFS
A* MAN
A* EUC
Animation Speed
Shuffle
Solve
Stop
0 1 2
3 4 5
6 7 8
Cost=0
Nodes Expanded=0
Search Depth=0
Running Time=0

8


8 Puzzle
6
2
8
5
4
1
3
7
BFS
DFS
A* MAN
A* EUC
Animation Speed
Shuffle
Solve
Stop
6 2 8
5 4 1
3 7 0
Cost=22
Nodes Expanded=97320
Depth=23
Time=0.24