8-Puzzle Search

By

Amr Mohamed Nasr (47)
Michael Raafat (57)
Mohammed Deifallah (59)

OVERVIEW

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.

TERMINOLOGY USED

- b: Branching factor, which is up to 4 for that problem.
- d: The depth of the solution.
- Search tree: The tree whose root is the initial random state, the internal nodes are the states you can reach with only one move from the current node, and the leaves are the goal state and dead ends.
- m: The maximum depth in the search tree.
- Actual Cost: The number of moves applied to reach the current state from the initial one.
- Heuristic Cost: The approximate optimistic cost to reach the goal state from the current one.
- Total Cost: The sum of the previous two costs.
- h*(x): The optimal heuristic, the actual cost from the state x to the goal state.

ALGORITHMS USED

It's important to mention that all the algorithms, implemented in that project, are following the same *interface*, except for the data structure used to apply the search section in each one. So, all common data structures will only be described, in detail, in the first algorithm.

1) BFS

I. ALGORITHM

As shown in the figure below, *BFS* algorithm searches in all possible paths, following *FIFO* approach.

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
```

II. DATA STRUCTURE

A. **Queue** of states to apply *FIFO* approach.

B. **Search Result** that have the following attributes:

```
public List<State> expanded_list;
public List<State> goal_path;
public int goal_cost;
public int search_depth;
public long search_time;
```

C. **State** that define the current status of the puzzle. It contains the following attributes and pointers:

```
/**
 * String containing the puzzle under the format all
 * rows concatenated to each other.<br>
 * [ 1   2   3 ]<br>
 * [ 5   7   8 ] ----->  "123578460"<br>
 * [ 4   6   - ]
 */
private String puzzle;
/**
 * Index of empty tile.
 */
private int empty_tile;
/**
 * Cost of reaching this state.
 */
private int actual_cost;
/**
 * Heuristic cost of reaching the goal.
 */
private double heuristic_cost;
/**
 * Children states that can be reached from this state.
 */
private List<State> children;
/**
 * The parent state of this state.
 */
private State parent;
```

2) DFS

I. ALGORITHM

As shown in the figure below, *DFS* algorithm searches in only one path till its end, then gets back to other possible paths, following *LIFO* approach.

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
```

II. DATA STRUCTURE

Stack of states to apply *LIFO* approach.

3) A*

I. ALGORITHM

As shown in the figure below, *A** algorithm takes the path with the least cost, guaranteeing optimality, using an admissible consistent heuristic function.

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
```

Please, note that there're two instances of that algorithms used, each of them applies a different heuristic function:

- *Manhattan Heuristic:* The following formula explains the calculation of *Manhattan Distance*.

$$h = abs(\texttt{current\_cell}.x - goal.x) + abs(\texttt{current\_cell}.y - goal.y)$$

- *Euclidean Heuristic:* The following formula explains the calculation of *Euclidean Distance*.

$$h = sqrt((\texttt{current\_cell}.x - goal.x)^2 + (\texttt{current\_cell}.y - goal.y)^2)$$

Now, it's obvious the more admissible heuristic is the second one, ***Euclidean Heuristic***. According to ***Triangle Inequality***, the hypotenuse (which is the shortest distance here) is always less than the sum of other two sides (which is the horizontal and vertical differences here).

II. DATA STRUCTURE

Priority Queue, which can be implemented by a **Heap** as mentioned in the pseudocode., where the states are sorted according to their corresponding costs.

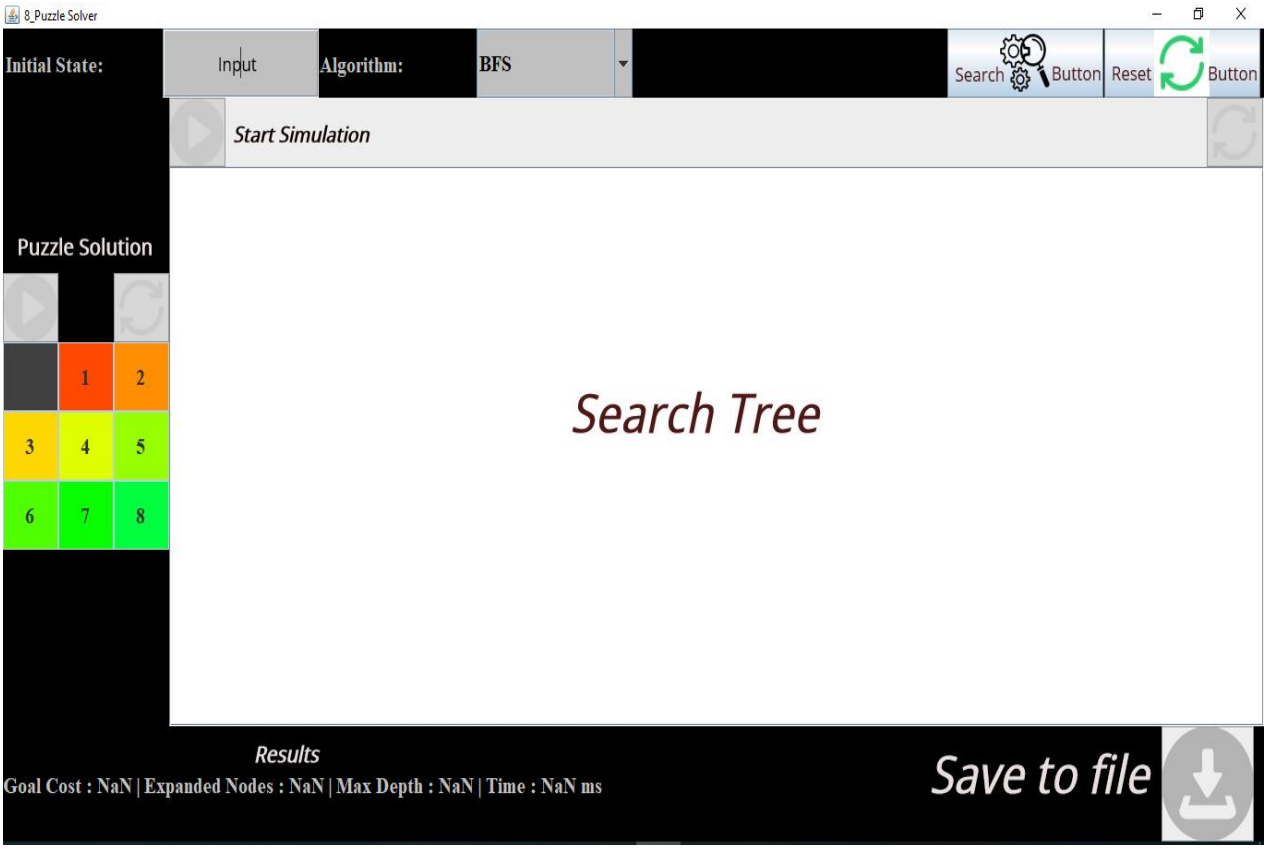
ALGORITHMS COMPARISON

Algorithm	Space Complexity	Time Complexity
BFS	$O\ (b^d)\ = O\ (4^d)$	$O\ (b^d) = O\ (4^d)$
DFS	O (b. d) = O (4.d) = O(d)	$O\ (b^m) = O\ (4^m)$
A*	$O\ (b^d)$	O (log$(h^*(x))$)

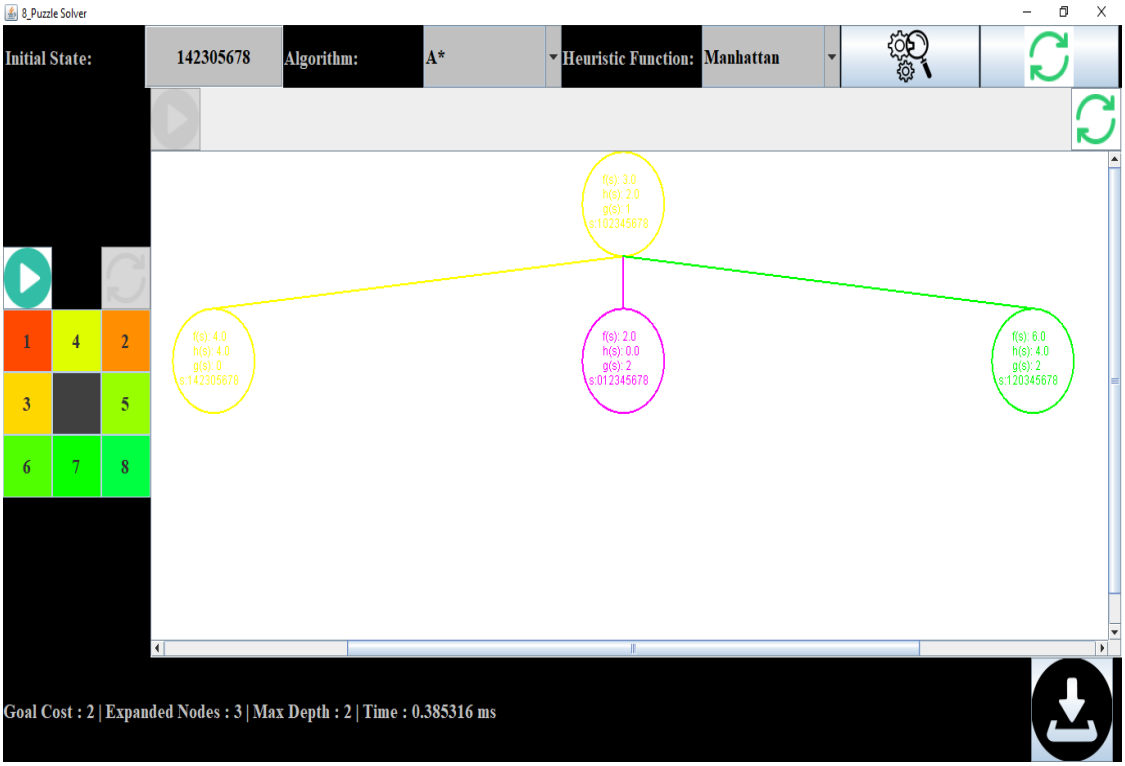
EXTRA WORK

We provide a GUI application, implemented in **Java Swing**, that enables the end user to solve his puzzle, showing him the actual steps to reach the correct end state, and helping him save those steps into a simple text file. Also, it helps the technicians measure the three provided algorithms, showing the search space, represented as a search tree, the running time for each one, and the total cost.

Here's a screenshot of the user-friendly interface for how to use it to solve the problem:



SAMPLE RUNS



And here's a screenshot to a part of its file:

```
Cost: 2.
Number of Expanded Nodes: 3.
Search Depth: 2.
Time Taken: 0.385316 ms.

Puzzle Path To Goal:
Actual Cost: 0
Heuristic Cost: 4.0
Total Cost: 4.0
---------------------------------
|       |       |       |
|   1   |   4   |   2   |
|       |       |       |
---------------------------------
|       |       |       |
|   3   |   0   |   5   |
|       |       |       |
---------------------------------
|       |       |       |
|   6   |   7   |   8   |
|       |       |       |
---------------------------------


To:
Actual Cost: 1
Heuristic Cost: 2.0
Total Cost: 3.0
---------------------------------
|       |       |       |
|   1   |   0   |   2   |
|       |       |       |
---------------------------------
```

ASSUMPTIONS & EXPLANATIONS

- The generation of the next states follows the sequence: *UP, Down, Left and Right.*
- Using a **HashSet** to keep track of explored nodes makes it O (1) to check duplicates.
- The nodes in the visual search tree are colored, as the following table shows:

State Type	Color
Unexplored	Black
Current	Red
Explored	Yellow
Frontier	Green
Goal	Purple/Magenta

EXPERIMENTAL COMPARISON

	BFS		DFS		A* (Manhattan)		A* (Euclidean)	
Testcase	Time (ms)	Max. Depth	Time (ms)	Max. Depth	Time (ms)	Max. Depth	Time (ms)	Max. Depth
528417036	72.23	18	6.97	4026	4.87	18	7.3	18
123405678	9.5	14	66.03	34838	1.1	14	2.65	14
142305678	.015	2	.011	2	.081	2	.044	2