# 8-PUZZLE-SOLVER

Ahmed Abdallah (4)
Ahmed Mohamed Elzeny (6)
Jamaleldin Ahmed Khalaf (15)

# Problem Statement:

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 number0. 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.

Suppose the program is executed starting from the initial state 1,2,5,3,4,0,6,7,8 as follows:

$$parent = \begin{array}{|c|c|c|}\hline 1 & 2 & 5 \\ \hline 3 & 4 & \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|}\hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

$$parent = \begin{array}{|c|c|c|}\hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|}\hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

$$parent = \begin{array}{|c|c|c|}\hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|}\hline & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

# Data Structure:

We made a data structure called Node to represent a state in DFS & BFS.

Each Node holds 3 variables

1) The value of state in a list of integers
2) The depth of Node
3) A reference to its parent

Another data structure was made called Entry to represent a state in A*.

Each Entry holds 4 variables

1) The value of state in a list of integers
2) The depth of Entry
3) A reference to its parent
4) A key that holds the estimated cost of the Entry

A comparator was implemented in Entry data structure to compare each Entry according to its key so in priority queue we can get the lowest key Entry.

The visited states were represented in a Hash Set of lists of integers HashSet<List<Integer>> so that we can lower the cost of searching for the value of visited state.

# Algorithms:

```
BFS (initial_state):

    frontier→Queue.new (new Node (initial_state))
    visited→HashSet.new

    While not frotier.isEmpty ():
        state = frontier.pop()
        visited.add (state)

        if goalTest (state)
            return path (state)

        for neighbor in state.neighbors()
            if neighbor not in (frontier & visited)
                frontier.add (neighbor)

    return failed
```

**Notes:**

- We get the path by back track from the goal through the parents to the head node that has no parents
- Searching in the frontier costs O (n)
- Searching in the visited costs O (1)
- Goal test takes O (n)

```
DFS (initial_state):

    frontier→Stack.new (new Node (initial_state))
    visited→HashSet.new

    While not frotier.isEmpty ():
        state = frontier.pop()
        visited.add (state)

        if goalTest (state)
            return path (state)

        for neighbor in state.neighbors()
            if neighbor not in (frontier & visited)
                frontier.push (neighbor)

    return failed
```

**Notes:**

- We get the path by back track from the goal through the parents to the head node that has no parents
- Searching in the frontier costs O (n)
- Searching in the visited costs O (1)
- Goal test takes O (n)

```
AStar (initial_state, h):

    frontier→PriorityQueue.new (new Entry (initial_state, cost (h)))
    visited→HashSet.new

    While not frotier.isEmpty ():
        state = frontier.deleteMin()
        visited.add (state)

        if goalTest (state)
            return path (state)

        for neighbor in state.neighbors()
            if neighbor not in (frontier & visited)
                frontier.insert (new Entry (initial_state, cost (h)))
            else if neighbor in (frontier)
                frontier.decreaseKey(neighbor)

    return failed
```

**Notes:**

- We get the path by back track from the goal through the parents to the head node that has no parents
- Searching in the frontier costs O (n)
- Searching in the visited costs O (1)
- Goal test takes O (n)
- Decreasing key takes O (n)
- h is the wanted heuristic to measure the cost of the node

# Sample Runs & Comparison:

We run the application using the 3 algorithms on these 3 initial states

State_1 = 5, 1, 2, 3, 4, 8, 6, 7, 0

State_2 = 1, 2, 5, 3, 4, 0, 6, 7, 8

State_3 = 1, 8, 2, 0, 4, 3, 7, 6, 5

- Comparing with number of moves

| States | DFS | BFS | A*_manhtn | A*_euclidean |
|---|---|---|---|---|
| State_1 | 55114 | 12 | 12 | 12 |
| State_2 | 3 | 3 | 3 | 3 |
| State_3 | 55791 | 21 | 29 | 23 |

- Comparing with total number of visited nodes

| States | DFS | BFS | A*_manhtn | A*_euclidean |
|---|---|---|---|---|
| State_1 | 63994 | 1831 | 254 | 287 |
| State_2 | 4 | 19 | 7 | 7 |
| State_3 | 65377 | 64322 | 15035 | 1286 |

- Comparing with time taken to solve in milliseconds

| States | DFS | BFS | A*_manhtn | A*_euclidean |
|---|---|---|---|---|
| State_1 | 125194 | 160 | 45 | 50 |
| State_2 | 2 | 5 | 3 | 3 |
| State_3 | 133270 | 42416 | 4189 | 141 |

- Comparing with depth of the goal

| States | DFS | BFS | A*_manhtn | A*_euclidean |
|---|---|---|---|---|
| State_1 | 55114 | 12 | 8 | 8 |
| State_2 | 3 | 3 | 3 | 3 |
| State_3 | 55791 | 21 | 11 | 9 |

- Comparing with max depth reached

| States | DFS | BFS | A*_manhtn | A*_euclidean |
|---|---|---|---|---|
| State_1 | 55114 | 12 | 10 | 9 |
| State_2 | 3 | 3 | 3 | 3 |
| State_3 | 55791 | 21 | 17 | 13 |