

Eight Puzzle Game Engine

Introduction to Artificial Intelligence Assignment 01

Team Members

Name	ID	Tasks
Amin Mohamed Amin Elsayed	21010310	DFS, IDS
Andrew Safwat Fawzy Gerges	21010314	Astar
Omar Al-dawy Ibrahim Al-dawy	21010864	GUI, BFS



Table Of Content

- 1. How To Play?**
- 2. Different Solving Algorithms**
- 3. Explanation of Algorithms**
- 4. Pseudocodes**
- 5. Search Tree Parameters**
- 6. Data Structures used**
- 7. Analysis of Algorithms**
- 8. Heuristic**
- 9. GUI Design**
- 10. Test Cases**



Table Of Content

1. **How To Play?**
2. **Different Solving Algorithms**
3. **Explanation of Algorithms**
4. **Pseudocodes**
5. **Search Tree Parameters**
6. **Data Structures used**
7. **Analysis of Algorithms**
8. **Heuristic**
9. **GUI Design**
10. **Test Cases**



1. How To Play?

- The **8-puzzle game** consists of a **3x3 grid containing 8 distinct movable tiles** and **one empty space**, represented by the number 0. **The objective is to arrange the tiles in ascending order from 0 to 8**, with 0 representing the empty space. **Players can swap the empty space with any tile adjacent to it (up, down, left, or right) to achieve the goal configuration.**
- The **search problem** involves **finding a sequence of legal moves that transforms a given initial configuration** into the **goal state**. The cost of each move is uniform and equals one, meaning the total cost to reach the goal is the number of moves made. The challenge lies in exploring the search space of possible states and efficiently determining the optimal sequence of moves.

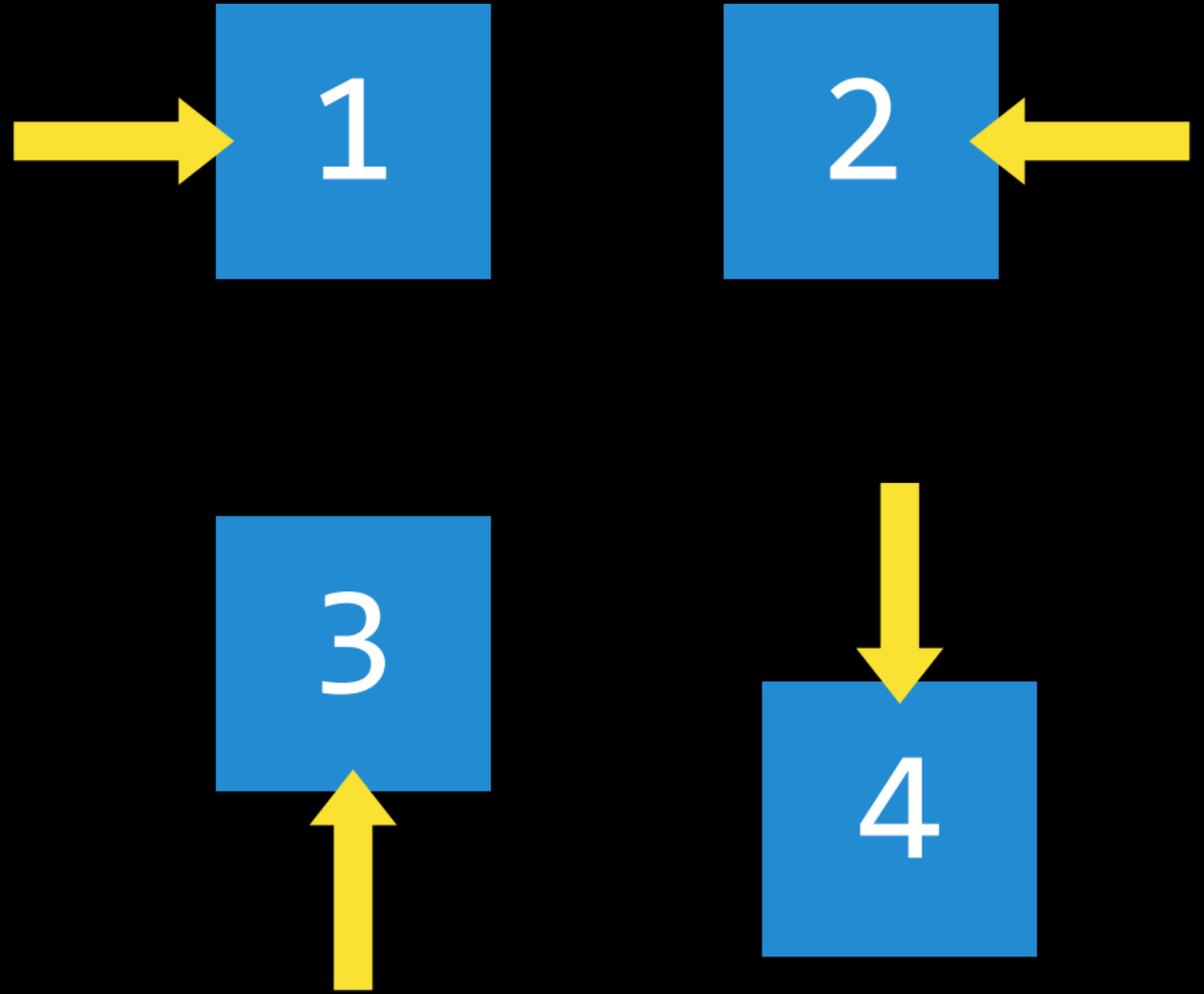


Search Problems

initial state

2	4	5	7
8	3	1	11
14	6		10
9	13	15	12

actions



RESULT(



2	4	5	7
8	3	1	11
14	6	10	12
9	13	15	

,  ) =

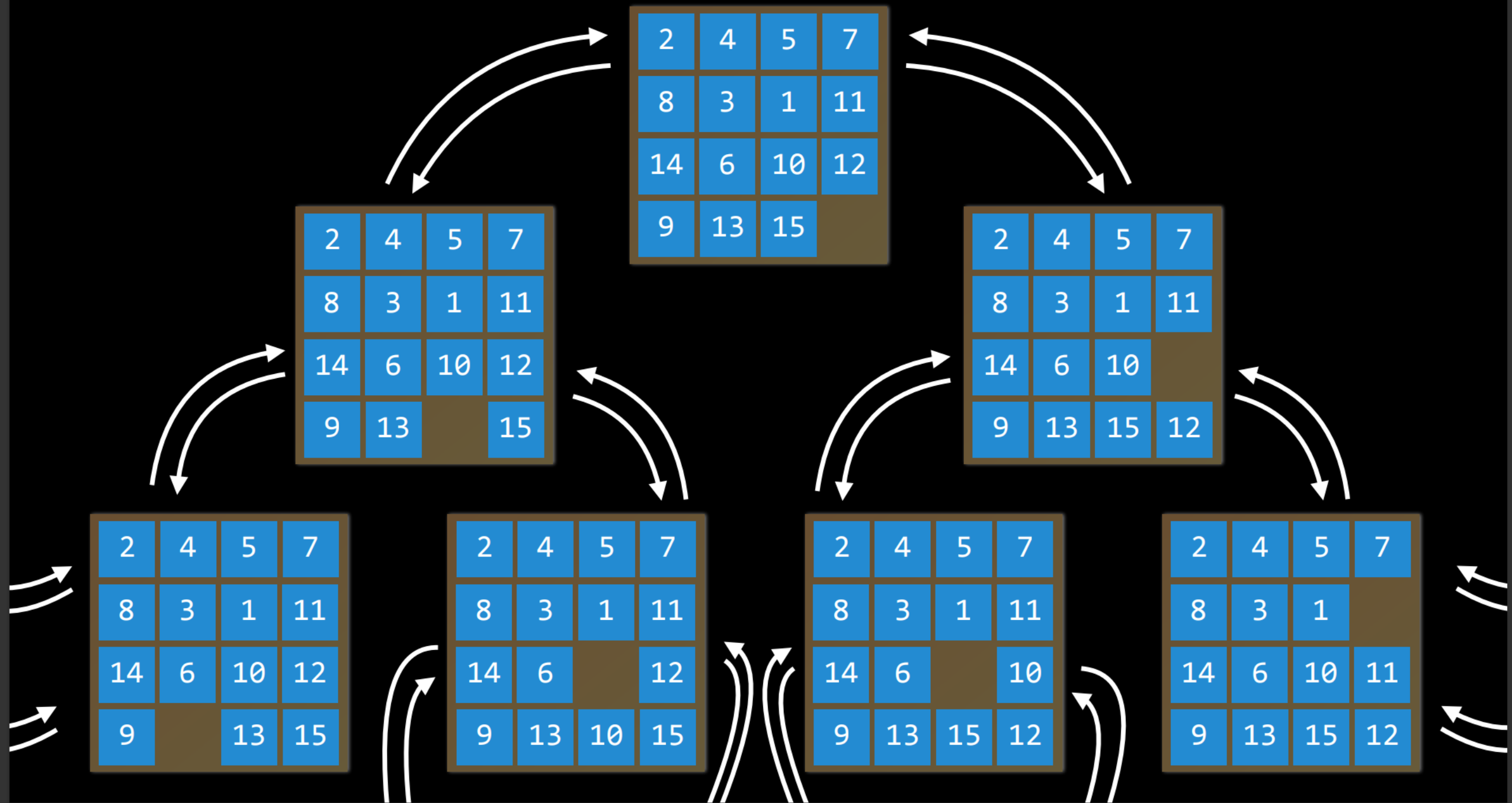
2	4	5	7
8	3	1	11
14	6	10	12
9	13		15

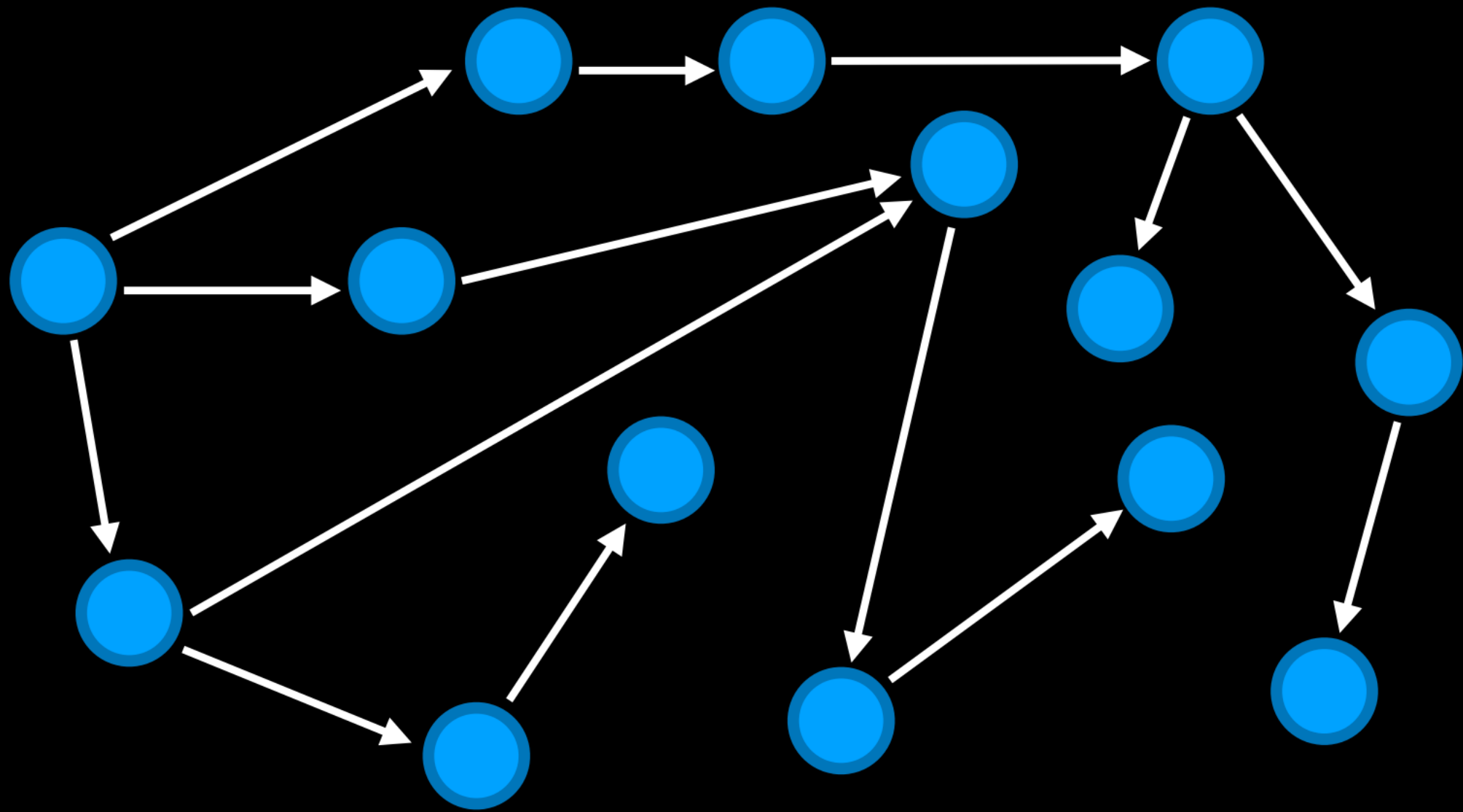
RESULT(

2	4	5	7
8	3	1	11
14	6	10	12
9	13	15	

,  ) =

2	4	5	7
8	3	1	11
14	6	10	
9	13	15	12





solution

a sequence of actions that leads from the initial state to a goal state

optimal solution

a solution that has the lowest path cost
among all solutions

Table Of Content

1. How To Play?
2. Different Solving Algorithms
3. Explanation of Algorithms
4. Pseudocodes
5. Search Tree Parameters
6. Data Structures used
7. Analysis of Algorithms
8. Heuristic
9. GUI Design
10. Test Cases



2. Different Solving Algorithms

1. **Depth-First Search (DFS)** Explores as far down a branch as possible before backtracking.
2. **Breadth-First Search (BFS)** Explores all nodes at the present depth before moving on to nodes at the next depth level.
3. **Iterative Deepening Search (IDS)** Combines DFS's space efficiency and BFS's completeness by incrementally deepening the search depth.
4. **Depth-Limited Search (DLS)** A variant of DFS that limits the depth of exploration.



2. Different Solving Algorithms

- 5. **Uniform Cost Search (UCS)** Explores the least costly path first, suitable for weighted graphs.
- 6. **Greedy Best-First Search (GBS)** Selects the node that appears to be closest to the goal based on a heuristic.
- 7. **A* Search** Combines UCS and GBS by using both the cost to reach a node and an estimate of the cost to the goal.



Table Of Content

1. How To Play?
2. Different Solving Algorithms
3. Explanation of Algorithms
4. Pseudocodes
5. Search Tree Parameters
6. Data Structures used
7. Analysis of Algorithms
8. Heuristic
9. GUI Design
10. Test Cases



3. Explanation of Algorithms

- **General Idea of search algorithms:**
General idea is that we have a frontier list and visited list, frontier list contains at first the start state, then that element is removed from the frontier list and check if it is the goal state that we want. If not, that state is put into the visited list so that we don't go back to it again, and neighbor states of this state are found and put into the frontier list (if they are not already in it). Repeating the previous step will eventually get us the solution (if a solution exists).
- **DFS (Depth-First Search):** Frontier list is a stack, where state at the top is always popped out of the frontier list, so that the search tree will expand vertically till reaching the end of a path (Dead end or a Solution).
- **IDS (Iterative Depth Search):** Same as DFS but there is a limit for the depth that starts from depth 0 and keeps incrementing till it reaches a certain depth that a solution is found at.

3. Explanation of Algorithms

- **BFS (Breadth-First Search):** Frontier list is a Queue, where oldest state is always dequeued out of the frontier list, so that the search tree will expand level-by-level till reaching a Solution.
- As an optimization, the tree is being traversed level by level so if I met a state once I do not need to add it again to the frontier as I found it early with less cost so instead of looping the frontier to check if I have a state like the current one I can use one set -lets name it visited set- to check in constant time, in addition to this an integer is used to count the number of explored nodes until reaching the goal and use its value instead of the explored set size.

3. Explanation of Algorithms

- **A* (A star):** Frontier list is a minimum heap, where it sorts according to value computed from $g(x) + h(x)$ for each state, where $g(x)$ is the cost from start state to that state and $h(x)$ is the estimated cost from that state to the goal state, so that states are explored according to their estimated distance to the solution state, with minimum cost explored first. $h(x)$ is a heuristic function, by which we remove some constraints of the problem to get a better estimate than real cost, there are 3 heuristic functions we implemented:
 - **Misplaced tiles:** Where we assumed there is no borders on the board and tiles can be removed and placed at right place immediately using 1 step.
 - **Euclidean distance:** Where we assumed there is only one tile on board and want it to go to its right place, so it can move (Euclidean distance from its current place to its destination) steps to get to its destination.
 - **Manhattan distance:** Where we assumed there is only one tile on board and want it to go to its right place, so it can move (horizontal + vertical) steps to get to its destination. Note that this is the closest to real cost.

Table Of Content

1. How To Play?
2. Different Solving Algorithms
3. Explanation of Algorithms
4. Pseudocodes
5. Search Tree Parameters
6. Data Structures used
7. Analysis of Algorithms
8. Heuristic
9. GUI Design
10. Test Cases



1. Depth-First Search

Explanation

The Depth-First Search (DFS) algorithm begins at the initial state of the problem. It checks whether the current state is the goal state; if it is, the algorithm terminates successfully. If not, DFS explores one of the neighboring states (one of the possible next states) at a time, continuing this process until it either reaches the goal or encounters an end state (a state with no further neighbors). When an end state is reached, the algorithm backtracks to explore alternative branches. States are stored in a stack data structure, which ensures that exploration occurs branch by branch, allowing DFS to delve deeply into each path before moving on to the next.

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 \cup explored:

 frontier.push(neighbor)

return **FAILURE**

2. Breadth-First Search

Explanation

The Breadth-First Search (BFS) algorithm begins at the initial state of the problem. It first checks if the current state is the goal state; if so, the algorithm terminates successfully. If the current state is not the goal, BFS explores all neighboring states, which represent possible next states, in a systematic manner. States are stored in a queue data structure, ensuring that exploration occurs level by level, moving outward from the initial state. This approach allows BFS to guarantee that the shortest path to the goal state is found, if one exists.

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  $\cup$  explored:  
                frontier.enqueue(neighbor)
```

```
    return FAILURE
```

3. Iterative Deepening Search

Explanation

Iterative Deepening Search (IDS) is a search algorithm that merges the principles of depth-first search (DFS) and breadth-first search (BFS). It begins with a depth limit of zero and performs a depth-limited search, exploring nodes to that limit. If the goal state is found, the algorithm stops. If not, the depth limit is incremented, and the process is repeated. This approach allows IDS to explore all possible paths while using less memory than traditional DFS, as it systematically backtracks and continues searching for alternative paths at increasing depths. Ultimately, IDS ensures a complete and optimal search in large spaces.

4. Astar Search

Explanation

The A* search algorithm begins at the initial state, keeping track of the cost to reach each state, which is 0 for the initial state. In this problem, the cost of moving from one state to another is consistently 1. The algorithm first checks whether the current state is the goal state; if it is, the algorithm terminates successfully. If the current state is not the goal, A* explores the state with the lowest total cost (the sum of the cost to reach the state and an estimated cost to the goal) and continues this process until the goal is reached. States are stored in a priority queue data structure, ensuring that exploration is conducted based on the smallest cost, facilitating an efficient pathfinding approach.

A* search

```
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
```

Table Of Content

1. How To Play?
2. Different Solving Algorithms
3. Explanation of Algorithms
4. Pseudocodes
5. Search Tree Parameters
6. Data Structures used
7. Analysis of Algorithms
8. Heuristic
9. GUI Design
10. Test Cases



4. Search Tree Parameters

1. **Maximum Branching Factor (b)** This parameter represents the maximum number of children each node in the tree can have.
2. **Maximum Depth (m)** This denotes the maximum number of edges in the longest path from the root node to the deepest leaf node. It defines the height of the tree.
3. **Goal Depth (d)** This indicates the depth of goal node within the tree.



Table Of Content

1. How To Play?
2. Different Solving Algorithms
3. Explanation of Algorithms
4. Pseudocodes
5. Search Tree Parameters
6. Data Structures used
7. Analysis of Algorithms
8. Heuristic
9. GUI Design
10. Test Cases



5. Data Structures used

Common Data Structures:

1. **solution_path**: list of Integers.
2. **explored_set**: set of Integers.
3. **frontier_set**: set of Integers.

Note: We used Integer representation of States for improving efficiency of computation instead of using strings and matrices.

The image shows a PyCharm IDE interface. The top toolbar includes icons for PC, menu, EP (Eight Puzzle), master branch, solver_factory, and various development tools like play, bug, and search. The left sidebar shows the Project view with a tree structure: Eight Puzzle (C:\Users\Yousef\Desktop\Eight Puzzle) > .venv library root > GUI > 8-Game-Window.ui. Below this is the Run view showing the command: "C:\Users\Yousef\Desktop\Eight Puzzle\.venv\Scripts\python.exe" "C:\Users\Yousef\Desktop\Eight Puzzle\Logic\solver_factory.py". The main editor displays the solver_factory.py file with the following code:

```
55 # print(depth, result[depth])
56 # print("Number of Nodes:", result['num_nodes'])
57 # print("Cost:", result['cost'])
58
59 print(sys.getsizeof(123456780))
60 print(sys.getsizeof("123456780"))
61 print(sys.getsizeof([[1, 2, 3],
62                      [4, 5, 6],
63                      [7, 8, 0]]))
64
65
```

The bottom status bar shows the file path: Eight Puzzle > Logic > solver_factory.py, and the encoding: 64:1 CRLF UTF-8 4 spaces Python 3.12 (Eight Puzzle). The Windows taskbar at the very bottom shows the search bar and several application icons.

5. Data Structures used

DFS & IDS:

1. **Stack: Stack of tuples (State: Integer, Depth: Integer)**
 - **Child_Parent_map: To store parent of each state tuple (State: Integer, Depth: Integer)**

5. Data Structures used

BFS:

1. Queue: Queue of states: Integer.
2. Parent_map: To store parent of each state.

A*:

1. map_of_costs: Maps each state: Integer to its total cost: Integer which is $g(x) + h(x)$.
2. G_cost: Maps each state: Integer to its real cost: Integer which is $g(x)$ only.

Table Of Content

1. How To Play?
2. Different Solving Algorithms
3. Explanation of Algorithms
4. Pseudocodes
5. Search Tree Parameters
6. Data Structures used
7. Analysis of Algorithms
8. Heuristic
9. GUI Design
10. Test Cases



6. Analysis of Algorithms

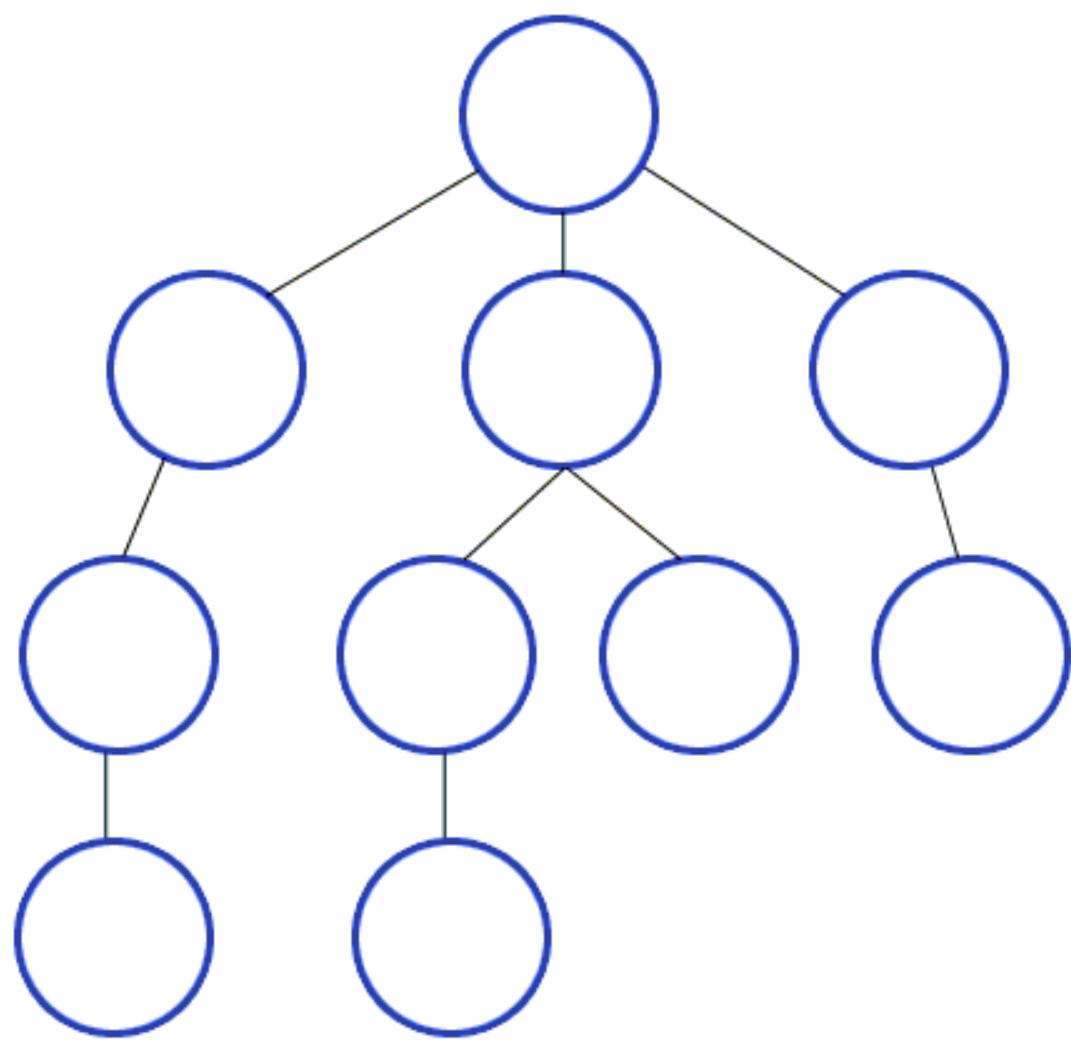
1. **Depth-First Search (DFS)**
2. **Breadth-First Search (BFS)**
3. **Iterative Deepening Search (IDS)**
4. **A* Search**



6. Analysis of Algorithms

1. **Depth-First Search (DFS)**
2. **Breadth-First Search (BFS)**
3. **Iterative Deepening Search (IDS)**
4. **A* Search**





Analysis of DFS

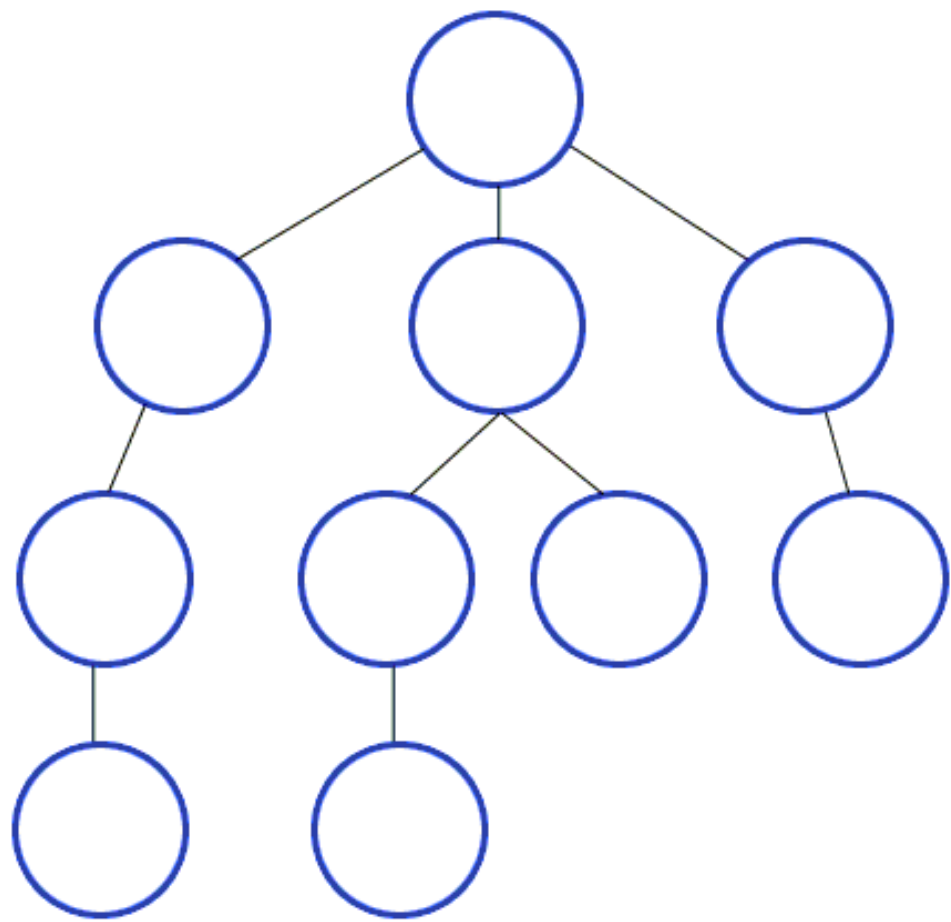
1. **Number of Nodes Expanded (Time Complexity)** = $(1 + b + b^2 + \dots + b^m) - (b + b^2 + \dots + b^{m-1}) = \mathbf{O(b^m)}$
2. **Space Complexity** = $(b - 1) + (b - 1) + (b - 1) + \dots + (b - 1) = m(b - 1) = \mathbf{O(bm)}$
3. **Completeness** : If the maximum depth m is finite, DFS will find the goal state **(complete)**
4. **Optimality** : DFS is **not optimal**, it may explore paths that exceed the depth of the optimal solution leading to suboptimal results.



6. Analysis of Algorithms

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)
3. Iterative Deepening Search (IDS)
4. A* Search





Analysis of BFS

1. **Number of Nodes Expanded (Time Complexity)** = $(1 + b + b^2 + \dots + b^d) = O(b^d)$
2. **Space Complexity** = $b^d = O(b^d)$
3. **Completeness** : BFS is **complete**, it will always find the goal state if exists
4. **Optimality** : BFS is **optimal**. It ensures finding the least costly path to the goal, **assuming all edges have equal cost**.



6. Analysis of Algorithms

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)
3. Iterative Deepening Search (IDS)
4. A* Search



Analysis of IDS

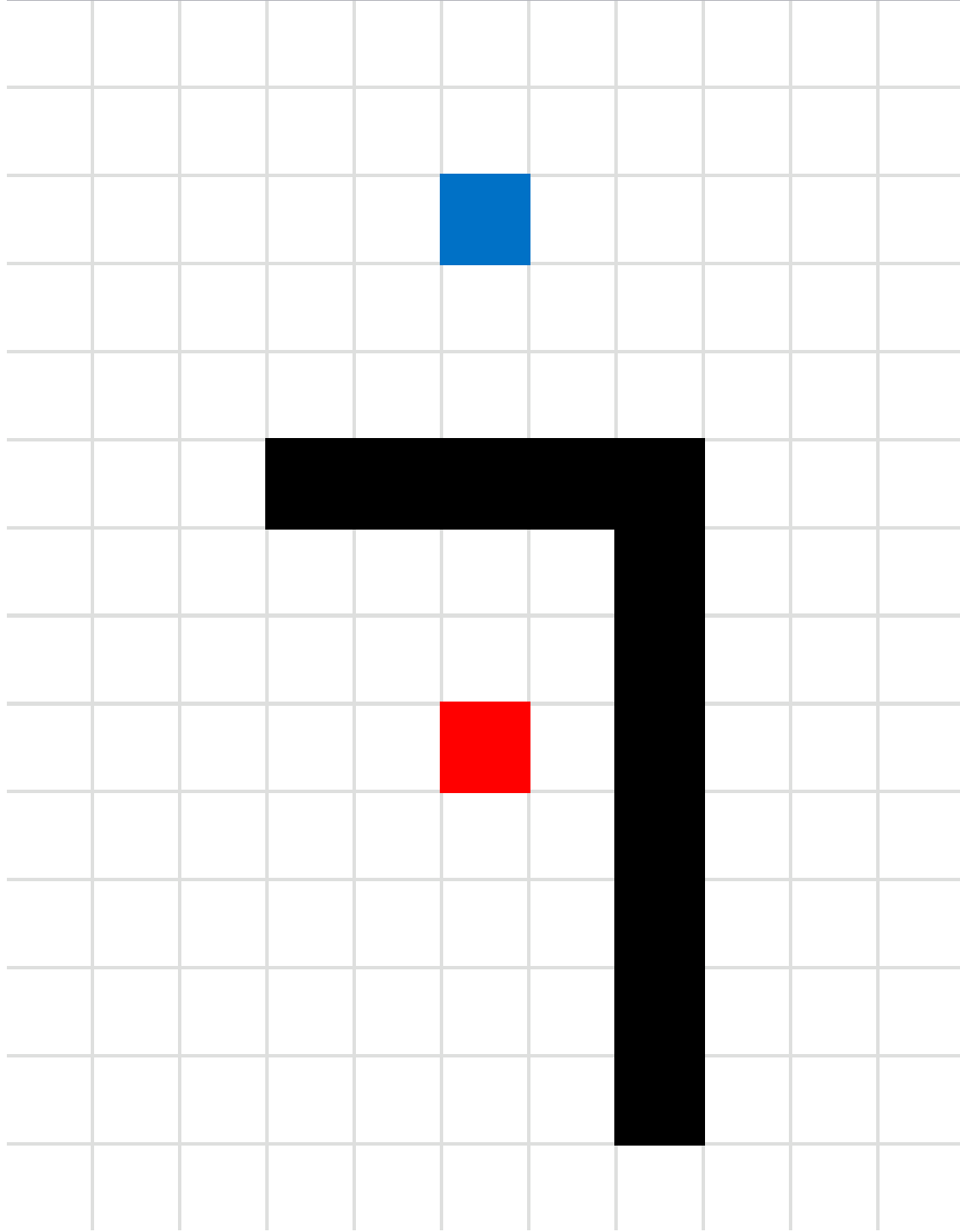
1. **Number of Nodes Expanded (Time Complexity)** = $1 + (1 + b) + (1 + b + b^2) + \dots + (1 + b + b^2 + \dots + b^d) = O(b^d)$
2. **Space Complexity** = $db = O(bd)$
3. **Completeness** : IDS is **complete**, it will always find the goal state if exists
4. **Optimality** : IDS is **optimal**. It guarantees finding the least costly path to the goal in terms of the number of edges, **assuming all edges have equal cost**.



6. Analysis of Algorithms

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)
3. Iterative Deepening Search (IDS)
4. A* Search





Analysis of A*

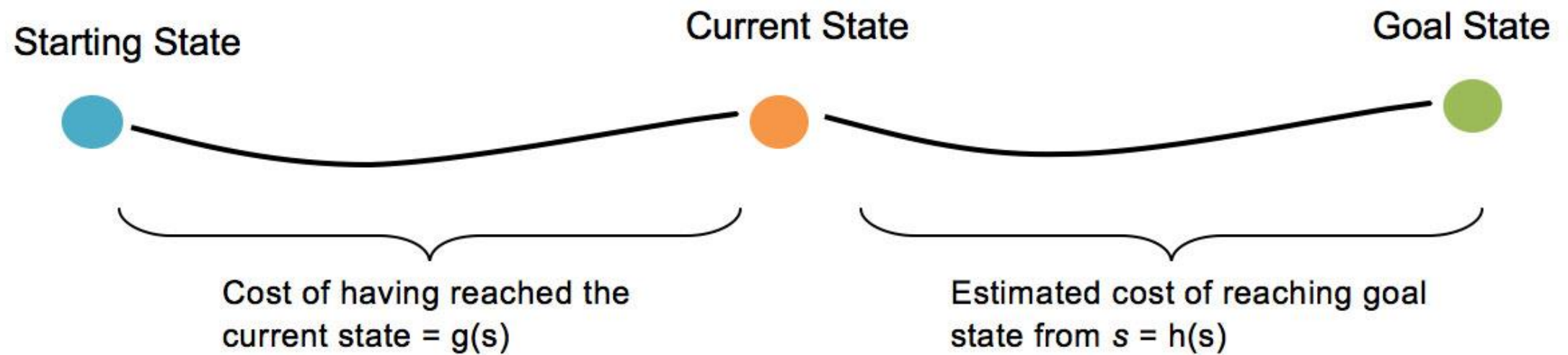
1. **Number of Nodes Expanded (Time Complexity) = $O(b^d)$**
2. **Space Complexity = $O(b^d)$**
3. **Completeness** : A* is **complete**, it will always find the goal state if exists
4. **Optimality** : A* is **optimal** when the heuristic used is **admissible** (it never overestimates the true cost to reach the goal)



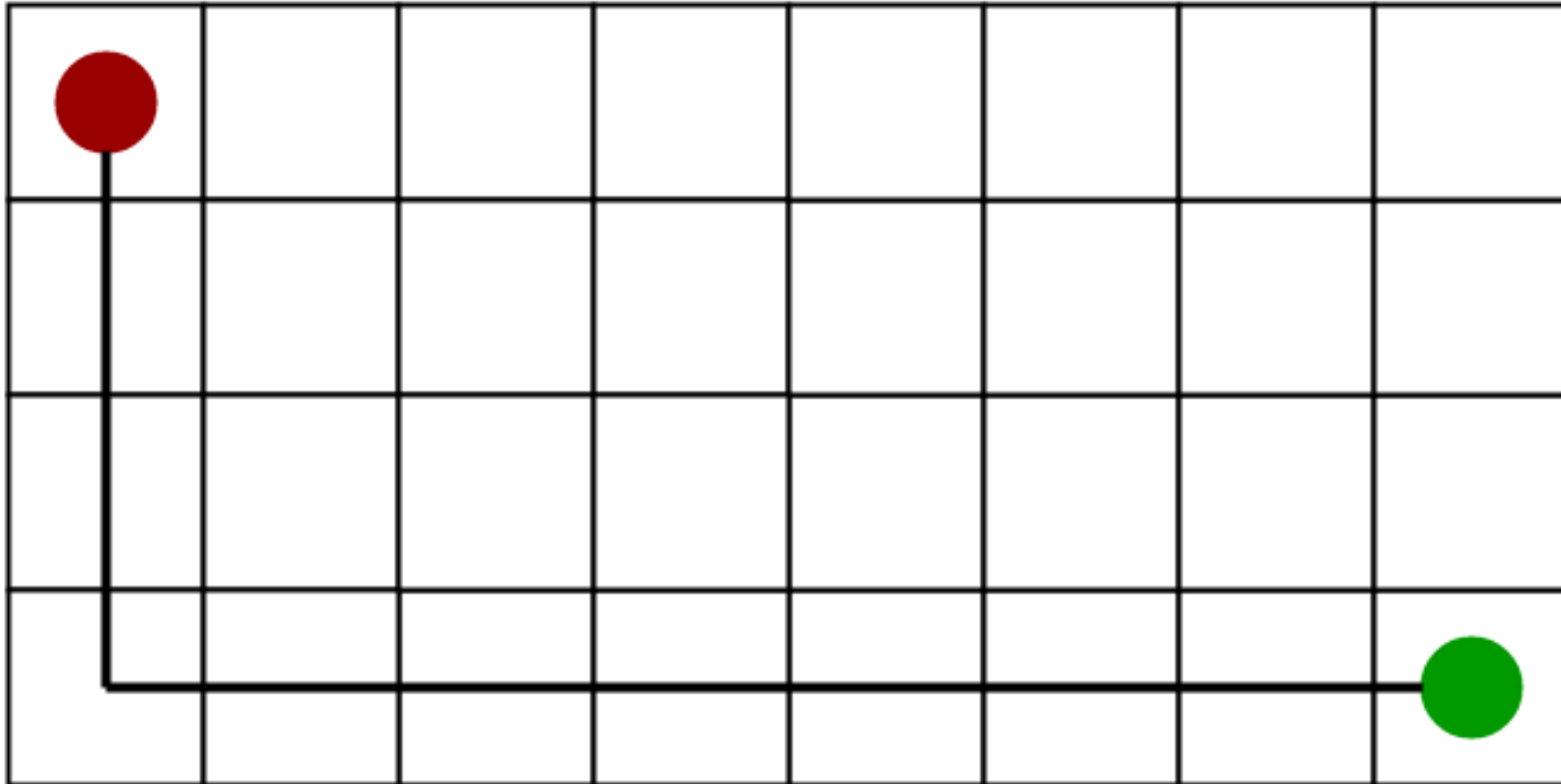
Table Of Content

1. How To Play?
2. Different Solving Algorithms
3. Explanation of Algorithms
4. Pseudocodes
5. Search Tree Parameters
6. Data Structures used
7. Analysis of Algorithms
8. Heuristic
9. GUI Design
10. Test Cases

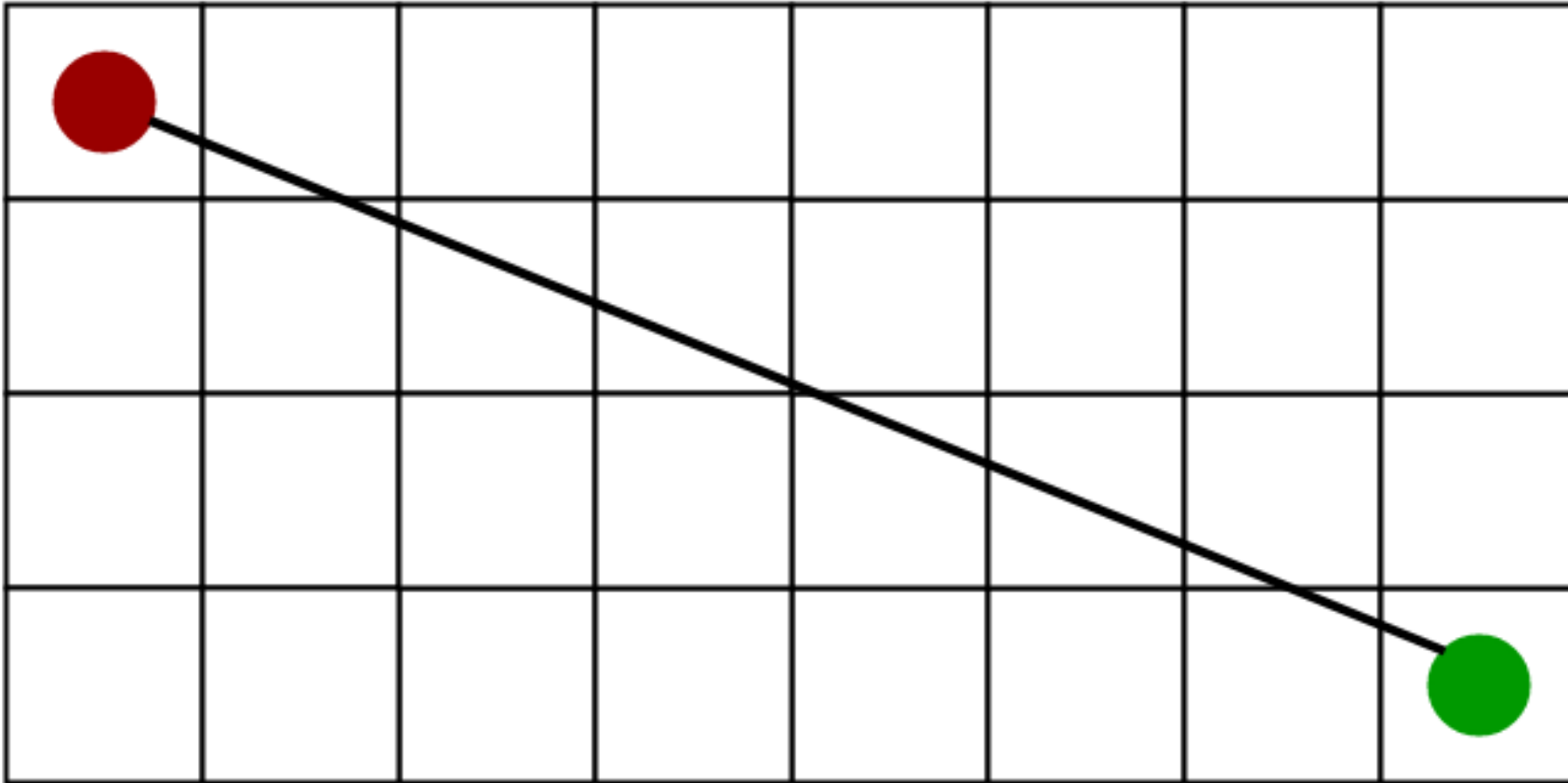




Manhattan Distance



Euclidean Distance



7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

Misplaced Tiles



Admissibility and Estimation for the 8-Puzzle

- Both the Manhattan and Euclidean distances are admissible heuristics in the context of the A^* search algorithm. Admissibility means that these heuristics never overestimate the true cost to reach the goal state, ensuring that A^* can find the optimal path.
- However, when applied to the 8-puzzle problem, the **Manhattan distance is generally the better estimation**. The 8-puzzle involves moving tiles on a grid where only horizontal and vertical moves are allowed. The Manhattan heuristic effectively calculates the total distance each tile must move to reach its goal position, reflecting the number of moves required. In contrast, the Euclidean heuristic, while still admissible, considers diagonal distances that are not possible in the 8-puzzle, leading to less accurate estimates of the moves needed (more relaxed hence no great accuracy).

Admissibility and Estimation for the 8-Puzzle

Test Case	Euclidean T	Nodes	Manhattan T	Nodes	Misplaced T	Nodes
725310648	0.00619s	151	0.00211s	74	0.00659s	307
35428617	0.00335s	13	0.001066s	13	0.00108s	17
641302758	0.004857s	117	0.002020s	91	0.00345s	217
158327064	0.0033602s	21	0.0027706s	14	0.0030654s	45
328451670	0.0025472s	33	0.00217869s	26	0.0016019s	67
123456780	0.1398593	1665	0.025213s	1111	0.15406449s	6763
806547231	2.41201s	34825	0.11809s	5132	7.617s	119933

All get the same path but with different numbers of nodes expanded

Table Of Content

1. How To Play?
2. Different Solving Algorithms
3. Explanation of Algorithms
4. Pseudocodes
5. Search Tree Parameters
6. Data Structures used
7. Analysis of Algorithms
8. Heuristic
9. GUI Design
10. Test Cases



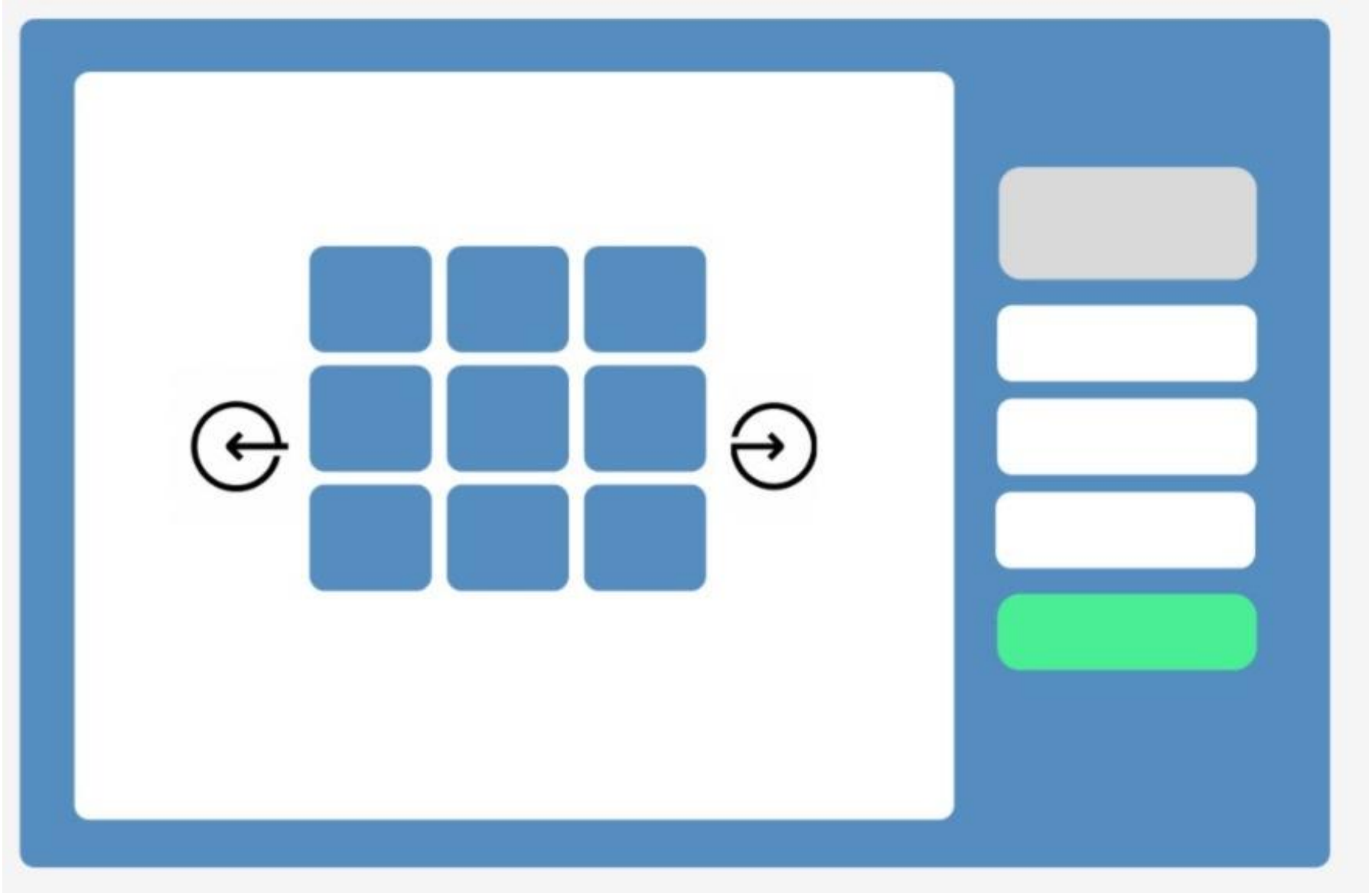
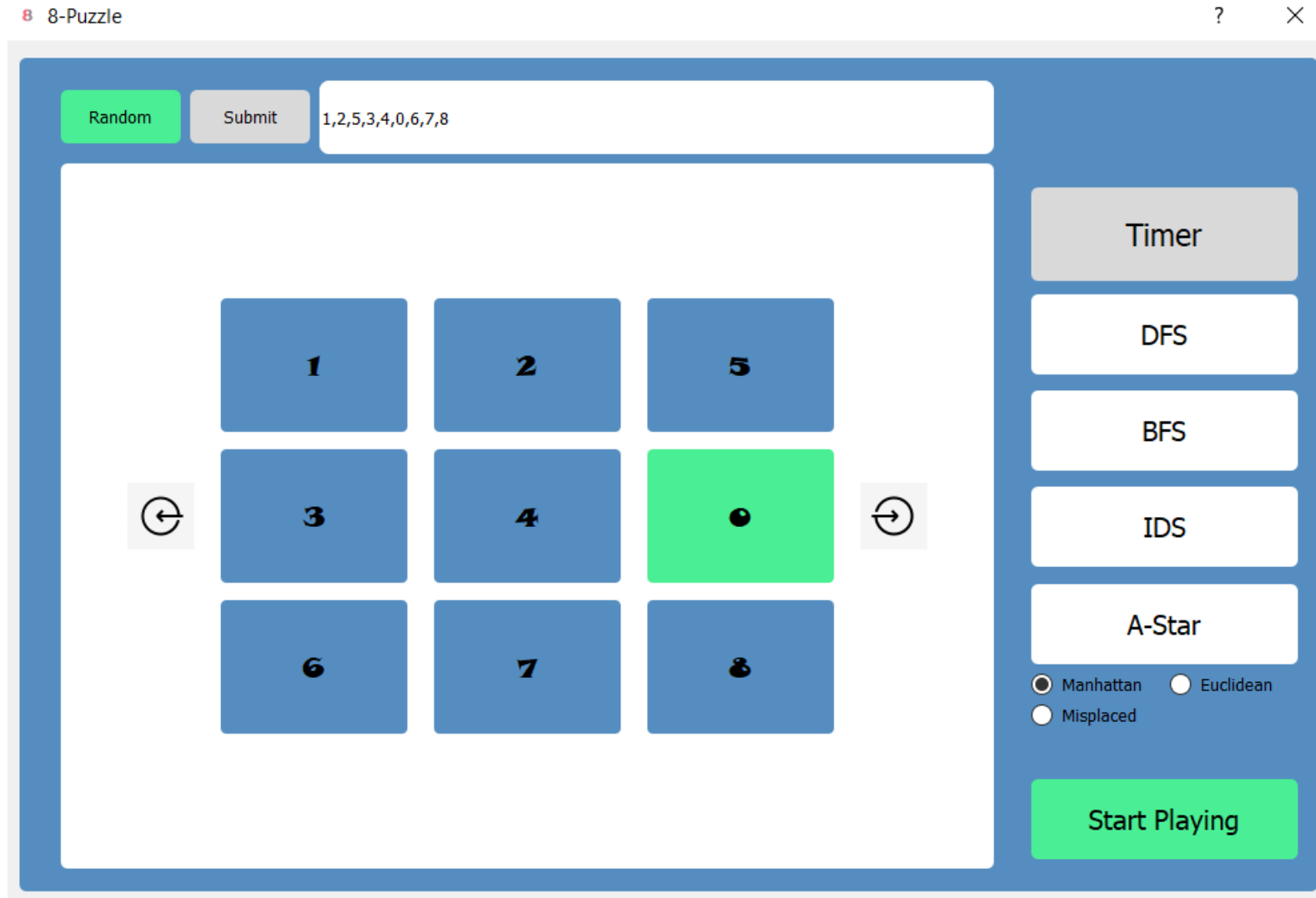


Table Of Content

1. How To Play?
2. Different Solving Algorithms
3. Explanation of Algorithms
4. Search Tree Parameters
5. Data Structures used
6. Analysis of Algorithms
7. Heuristic
8. GUI Design
9. Test Cases



Test Case #1: Normal GUI



Test Case #1: DFS

8 8-Puzzle ? X

Random Submit 1,2,5,3,4,0,6,7,8

1	2	5
3	4	0
6	7	8

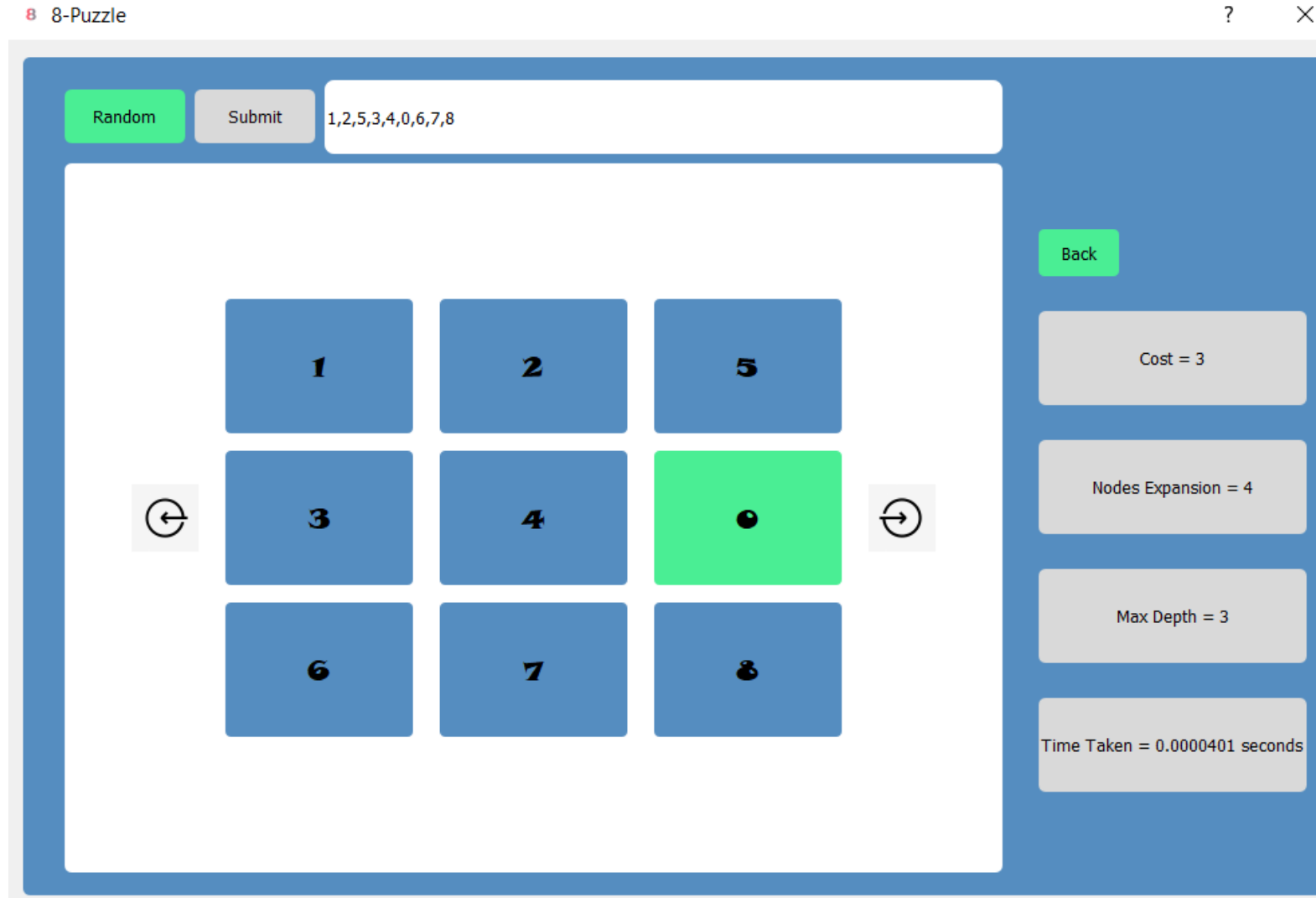
Back

Cost = 3

Nodes Expansion = 4

Max Depth = 3

Time Taken = 0.0000401 seconds



Test Case #1: BFS

8 8-Puzzle ? X

Random Submit 1,2,5,3,4,0,6,7,8

1	2	5
3	4	0
6	7	8

Back

Cost = 3

Nodes Expansion = 10

Max Depth = 3

Time Taken = 0.0000409 seconds



Test Case #1: IDS

8 8-Puzzle ? X

Random Submit 1,2,5,3,4,0,6,7,8

1	2	5
3	4	●
6	7	8

Back

Cost = 3

Nodes Expansion = 19

Max Depth = 3

Time Taken = 0.0000588 seconds



Test Case #1: A* (Manhattan)

8 8-Puzzle ? X

Random Submit 1,2,5,3,4,0,6,7,8

1	2	5
3	4	0
6	7	8

Back

Cost = 3

Nodes Expansion = 4

Max Depth = 3

Time Taken = 0.0000495 seconds



Test Case #1: A* (Euclidian)

8 8-Puzzle ? X

Random Submit 1,2,5,3,4,0,6,7,8

Back

Cost = 3

Nodes Expansion = 4

Max Depth = 3

Time Taken = 0.0000632 seconds

← 1 2 5 3 4 0 6 7 8 →



Test Case #1: A* (Misplaced Tiles)

8-Puzzle

Random Submit 1,2,5,3,4,0,6,7,8

1	2	5
3	4	0
6	7	8

Back

Cost = 3

Nodes Expansion = 4

Max Depth = 3

Time Taken = 0.0000427 seconds



Test Case #2: DFS

8 8-Puzzle ? X

Random Submit 8,0,6,5,4,7,2,3,1

8	0	6
5	4	7
2	3	1

Back

Cost = 62117

Nodes Expansion = 117231

Max Depth = 67471

Time Taken = 0.3814761 seconds



Test Case #2: BFS

8 8-Puzzle ? X

Random Submit 8,0,6,5,4,7,2,3,1

Back

8	0	6
5	4	7
2	3	1

Cost = 31

Nodes Expansion = 181428

Max Depth = 31

Time Taken = 0.9202910 seconds



Test Case #2: IDS

8 8-Puzzle ? X

Random Submit 8,0,6,5,4,7,2,3,1

8	0	6
5	4	7
2	3	1

Back

Cost = 31

Nodes Expansion = 5958165

Max Depth = 31

Time Taken = 19.3641113 seconds



Test Case #2: A* (Manhattan)

8-Puzzle ? X

Random Submit 8,0,6,5,4,7,2,3,1

8	0	6
5	4	7
2	3	1

Back

Cost = 31

Nodes Expansion = 5132

Max Depth = 31

Time Taken = 0.0636315 seconds



Test Case #2: A* (Euclidian)

8 8-Puzzle ? X

Random Submit 8,0,6,5,4,7,2,3,1

8	0	6
5	4	7
2	3	1

Back

Cost = 31

Nodes Expansion = 34825

Max Depth = 31

Time Taken = 0.8780038 seconds



Test Case #2: A* (Misplaced Tiles)

8 8-Puzzle ? X

Random Submit 8,0,6,5,4,7,2,3,1

8	0	6
5	4	7
2	3	1

Back

Cost = 31

Nodes Expansion = 119933

Max Depth = 31

Time Taken = 3.8058315 seconds



Thank YOU!