Assignment 1 Artifitial Intelligence .

Project name : Using Informed and Uninformed Search Algorithms to Solve 8-Puzzle.

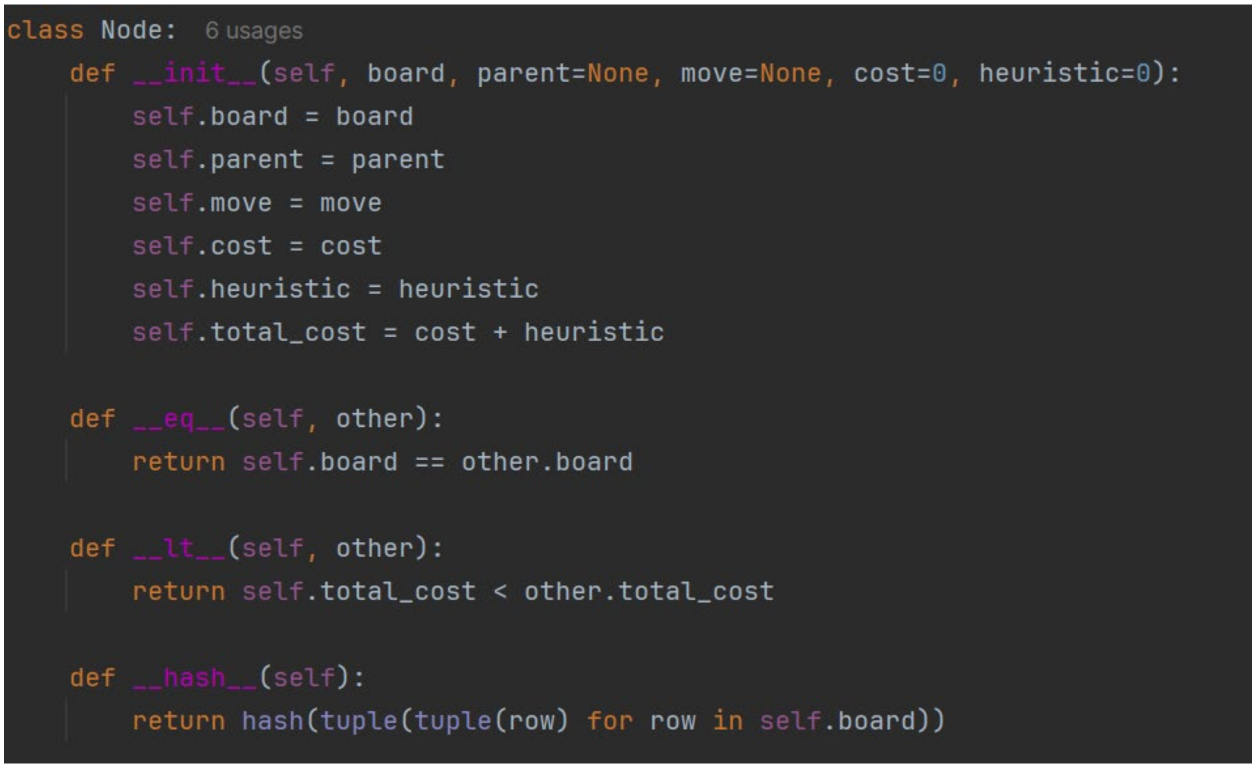
Karim Mohamed	8237
Ahmed maher	8017
Aly essam	8021

Introduction :

So in this project we discuss how to solve a 8 puzzle game by using Informed and Uninformed Search Algorithms and our solution methods for this game are 5 kinds that we implemented in the code which are BFS (Breadth-First Search, DFS (Depth-First Search), IDDFS (Iterative Deepening Depth-First Search) , A*, and heuristic strategies and in our code implementation we use 3 classes.

1) node :

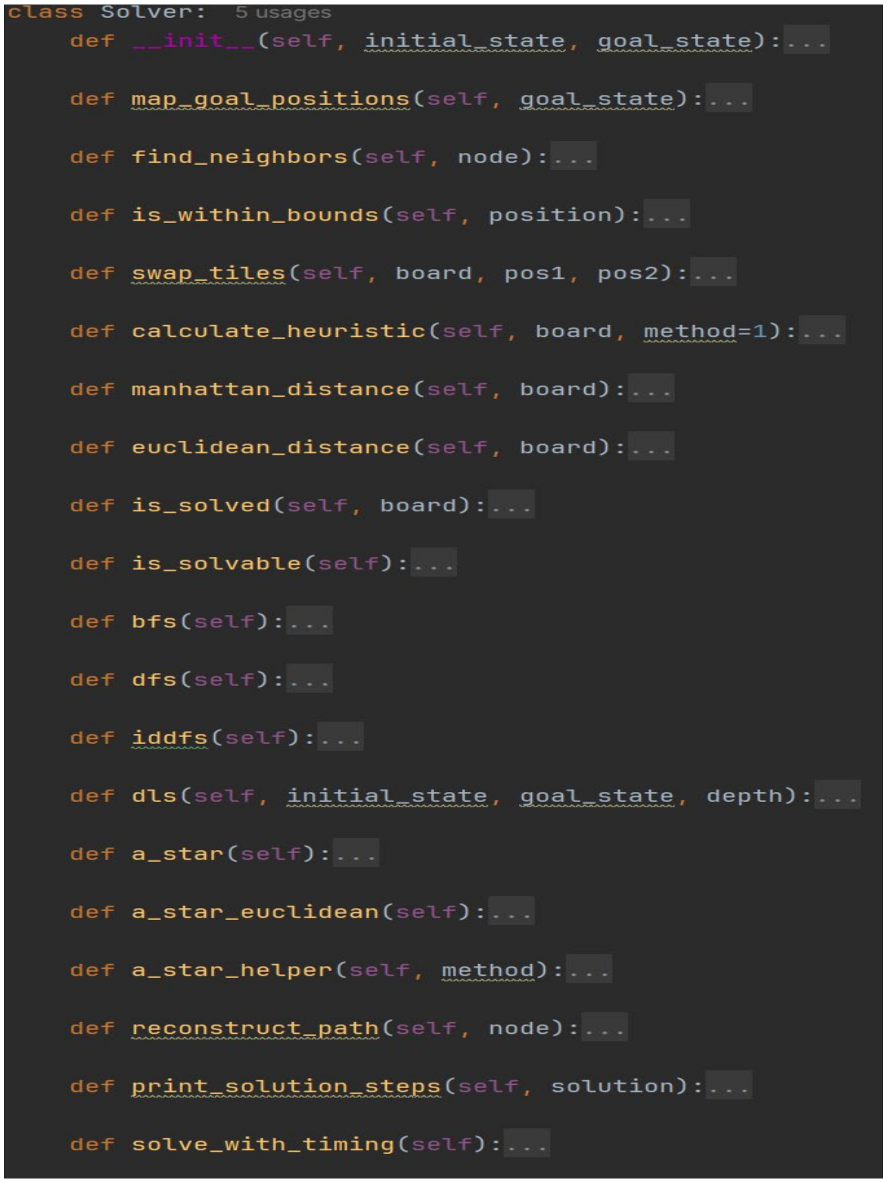
which is used to define the puzzle board and information about the board configuration, cost, heuristic, and parent node.



```
class Node:  6 usages
    def __init__(self, board, parent=None, move=None, cost=0, heuristic=0):
        self.board = board
        self.parent = parent
        self.move = move
        self.cost = cost
        self.heuristic = heuristic
        self.total_cost = cost + heuristic

    def __eq__(self, other):
        return self.board == other.board

    def __lt__(self, other):
        return self.total_cost < other.total_cost

    def __hash__(self):
        return hash(tuple(tuple(row) for row in self.board))
```

2) Solver:

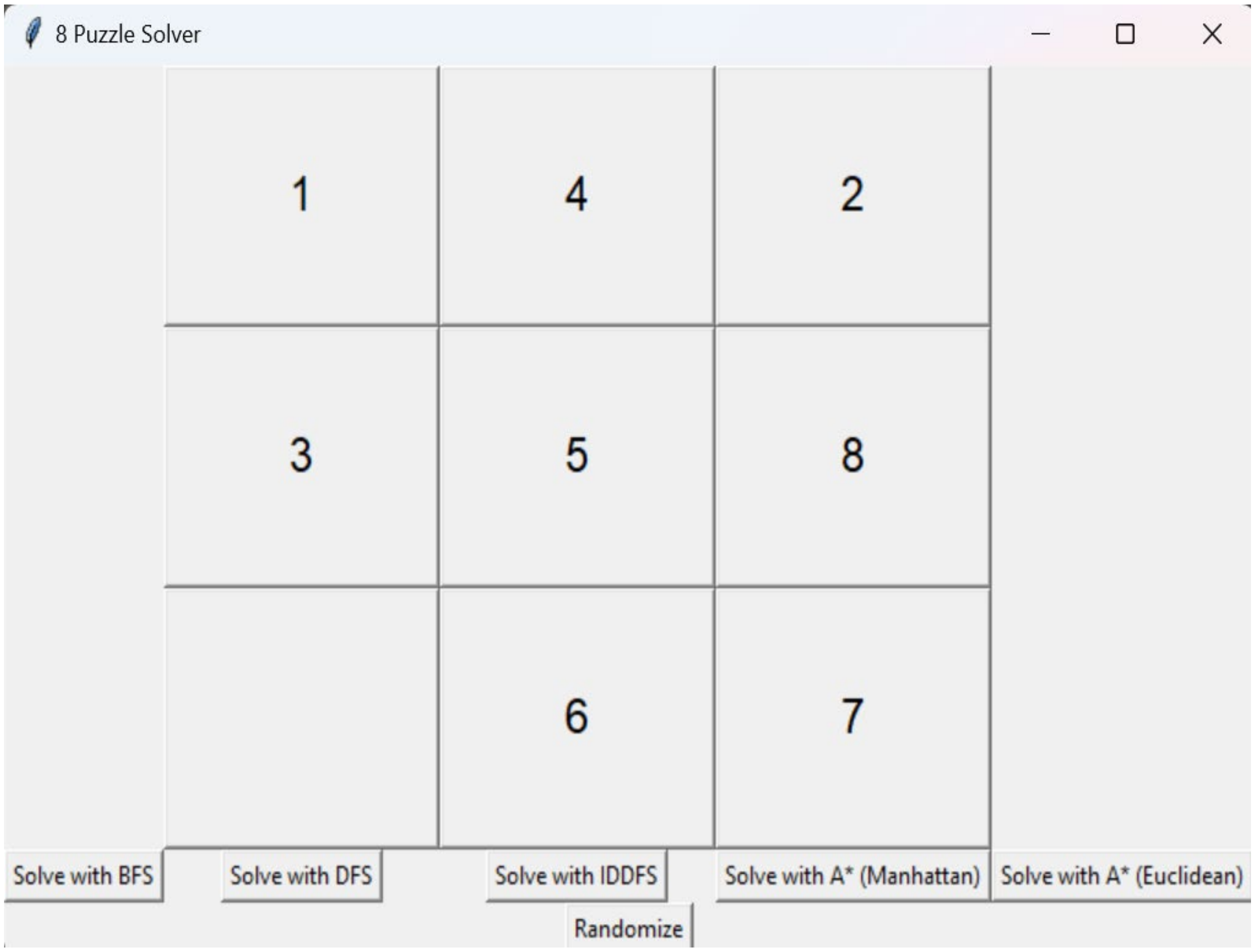
Contains the core algorithms to solve the puzzle, including BFS, DFS, IDDFS, A* (using Manhattan and Euclidean distances as heuristics) and Includes methods like figure below and heuristic calculations Also has methods to print, reconstruct, and visualize the solution paths and timing results.



```
class Solver:  5 usages
    def __init__(self, initial_state, goal_state):...

    def map_goal_positions(self, goal_state):...

    def find_neighbors(self, node):...

    def is_within_bounds(self, position):...

    def swap_tiles(self, board, pos1, pos2):...

    def calculate_heuristic(self, board, method=1):...

    def manhattan_distance(self, board):...

    def euclidean_distance(self, board):...

    def is_solved(self, board):...

    def is_solvable(self):...

    def bfs(self):...

    def dfs(self):...

    def iddfs(self):...

    def dls(self, initial_state, goal_state, depth):...

    def a_star(self):...

    def a_star_euclidean(self):...

    def a_star_helper(self, method):...

    def reconstruct_path(self, node):...

    def print_solution_steps(self, solution):...

    def solve_with_timing(self):...
```

3) PuzzleGUI:

Creates a graphical interface for interacting with the puzzle and it Provides buttons for each solving method (BFS, DFS, IDDFS, A*(Manhattan and Euclidean), tile movement, randomization, and displays the solution steps and provides user feedback through message boxes and Introduce Tkinter as a tool for creating GUIs in Python and its relevance in this puzzle application.



Algorithms:

Breadth-First Search (BFS): Describe the BFS implementation, it use a queue for level-order traversal and guaranteeing the shortest path in terms of the number of moves.

Depth-First Search (DFS): Explain DFS with a stack to explore deeper paths first, which may reach the goal but not necessarily in the shortest path.

Iterative Deepening Depth-First Search (IDDFS): it Describes how IDDFS repeatedly calls dls with increasing depth until it finds the solution, combining DFS depth control with BFS-like completeness.

Manhattan and Euclidean Heuristics: Describe A* as a search strategy using cost and heuristic to prioritize paths leading closer to the goal and Explain how Manhattan distance counts steps in a grid-like format, while Euclidean provides diagonal distances.

cost of path:

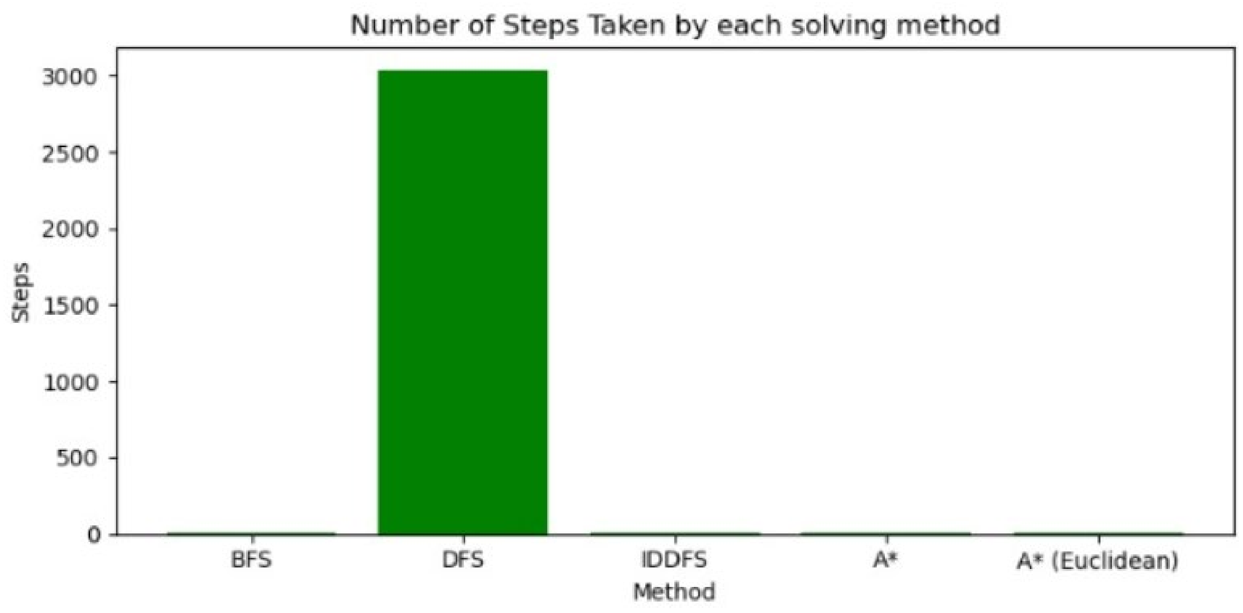
Breadth-First Search (BFS): 21.

Depth-First Search (DFS): **94587.**

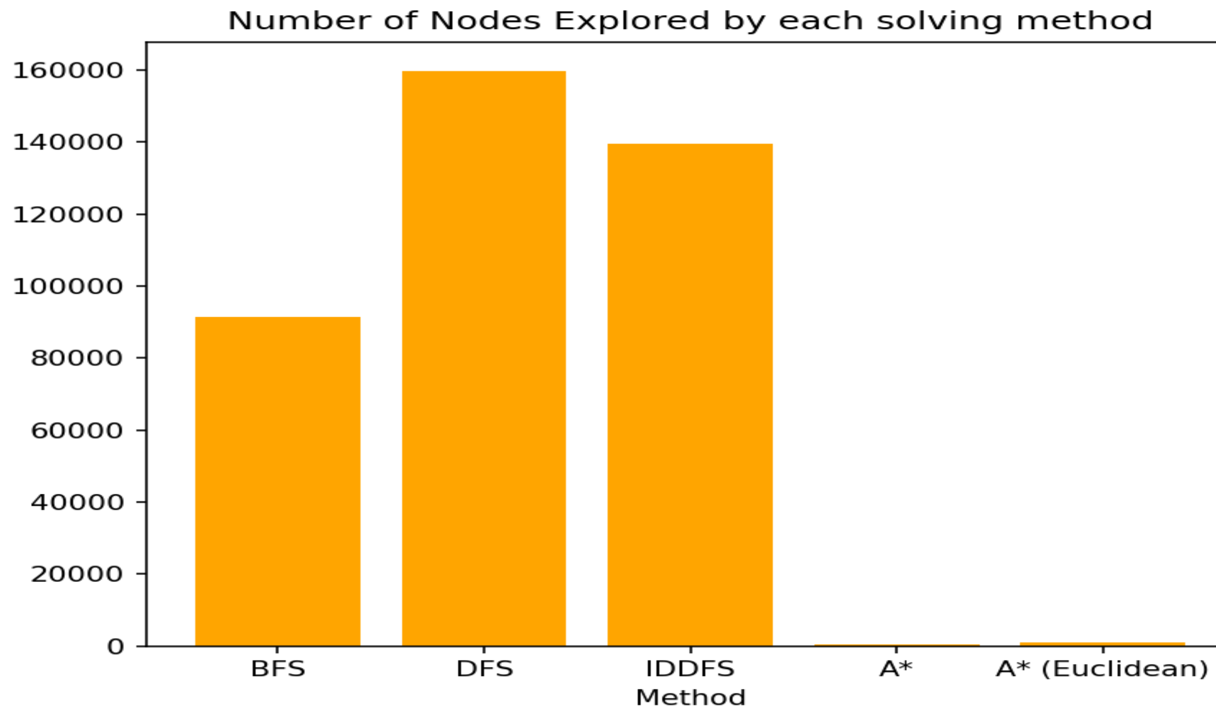
Iterative Deepening Depth-First Search (IDDFS): 23.

A*: 21.

A* (Euclidean): 21.



No of nodes expanded:



Search depth:

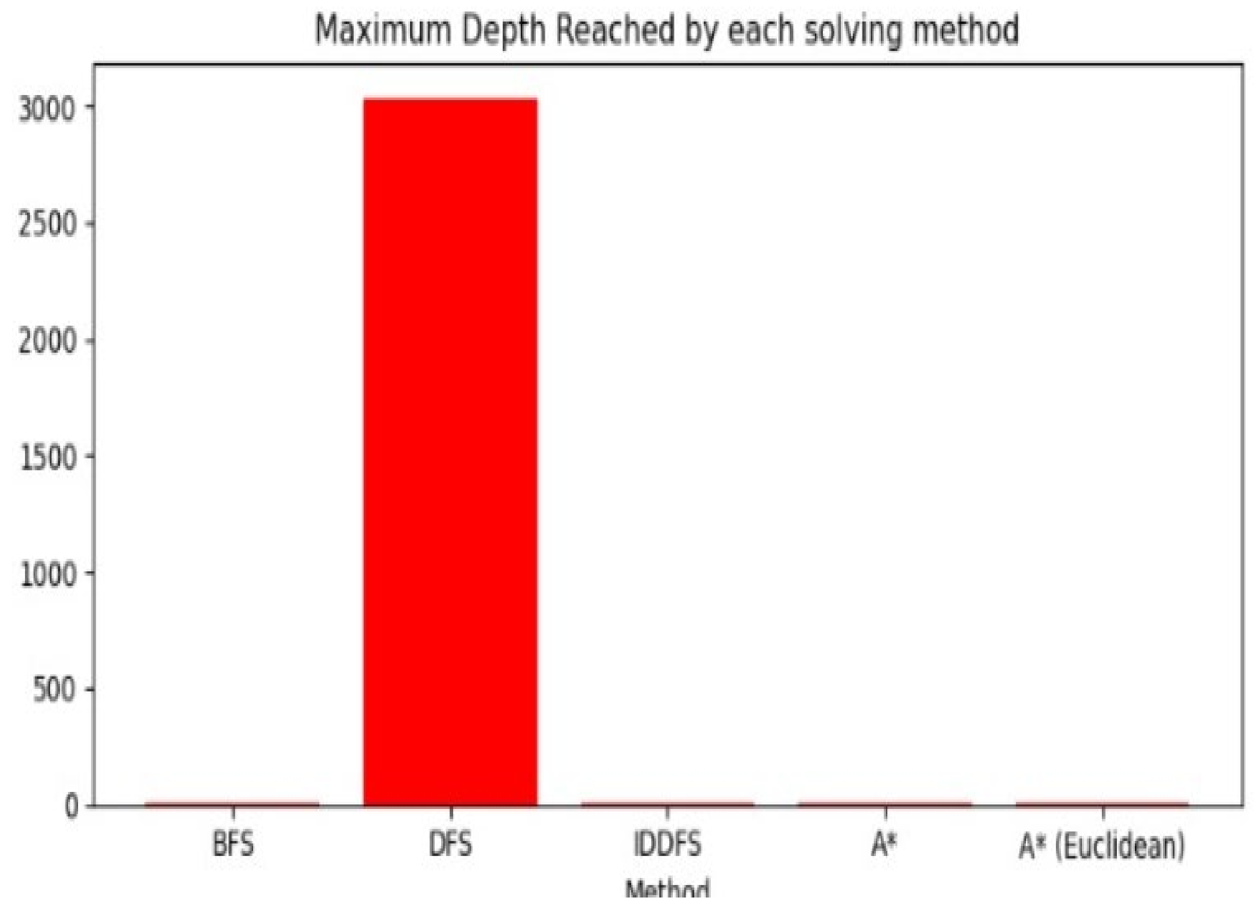
Breadth-First Search (BFS): 21.

Depth-First Search (DFS): 114821.

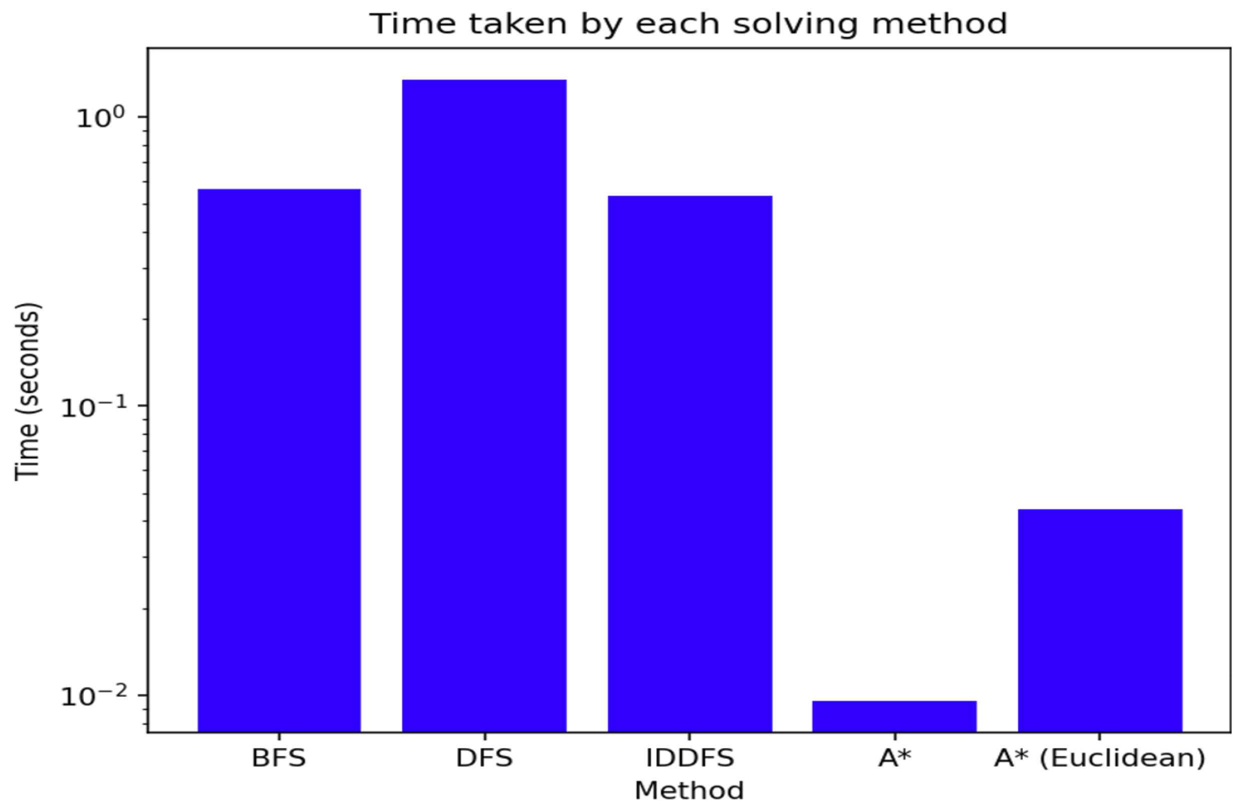
Iterative Deepening Depth-First Search (IDDFS): 23.

A*: 21.

A* (Euclidean): 21.



running time:



Data structure:

First of all, we used 2D list to represent the board and the goal state and it is a matrix 3x3 and each element in it holds a number to make it easy to access the row and column, then we used queue and stack in some functions in solver class like **solve_with_bfs** and **solve_with_dfs** the bfs and the BFS uses a queue to explore nodes level-by-level and the DFS uses a stack to explore nodes as far down a path as possible before backtracking and in **solve_with_astar**, **solve_with_astar_euclidean** we did use heaps and it’s used to maintain a priority queue where nodes with the lowest cost are processed first that’s easy to find shortest path and in this code we did implement tuples to represent moves in solution paths and it’s used in **animate_solution**, **print_solution_steps**