Assignment1

8-Puzzle Game

Report

Abdelrahman Amr Salah 8016
Ahmed Mahmoud Aly 8062
Youssef Mohamed Attia 8164

Overview:

our project is multi-module project consists of Main, Solver and Analyze modules as both Main and Analyze imports main module which contain needed functions and classes

1. **main**
    - NPuzzleGame Class: Manages the core game logic by integrating with the Puzzle and Solver classes, Handles the configuration of the game board by setting the initial and goal states, and managing the moves
    - NpuzzleUI Class: Implements the graphical user interface (GUI) for the N-Puzzle game using Pygame, manages interactions such as shuffling, solving, resetting the puzzle and choosing between different solving algorithms.
2. **Solver**
    - Node Class: Represents a state of the puzzle.
    - Puzzle Class: Defines the puzzle's properties and methods to interact with the puzzle.
    - Solver Class: Implements the different search algorithms to solve the puzzle.
3. **Analyze**
    - compares the performance of different search algorithms (BFS, DFS, A* with Manhattan distance, and A* with Euclidean distance) as number of movements and time taken is recorded and the results are visualized in a plot, comparing the efficiency of each algorithm in solving the problem.

Data Structures Used:

1. **Node Class (Attributes):**
    - board: 2D array representing the current state of the puzzle.
    - parent: Reference to the parent Node.
    - move: Tuple representing the move that led to this state.
    - g: Integer representing the cost from the start node to the current node.
    - h: Integer representing the heuristic cost from the current node to the goal node.
    - f: Integer representing the total cost, calculated as f = g + h.
2. **Puzzle Class (Attributes):**
    - initial_state: 2D array representing the initial puzzle.
    - goal_state: 2D array representing the goal.
    - goal_positions: Dictionary that maps each tile value to its position in the goal state.
3. **Queue:**
    - Used in the BFS algorithm to manage the frontier, allowing efficient popping from the front and appending to the back.
4. **List (frontier):**
    - Used in the DFS algorithm to manage the frontier, acting as a stack where nodes are appended to the end and popped from the end.
5. **Priority Queue:**

- Used in the A* search algorithm to manage the frontier, giving higher priority to nodes with the lowest f value (total cost).

6. **Set:**
    - Used to store explored nodes in all three algorithms (BFS, DFS, A*) to avoid revisiting already explored states.

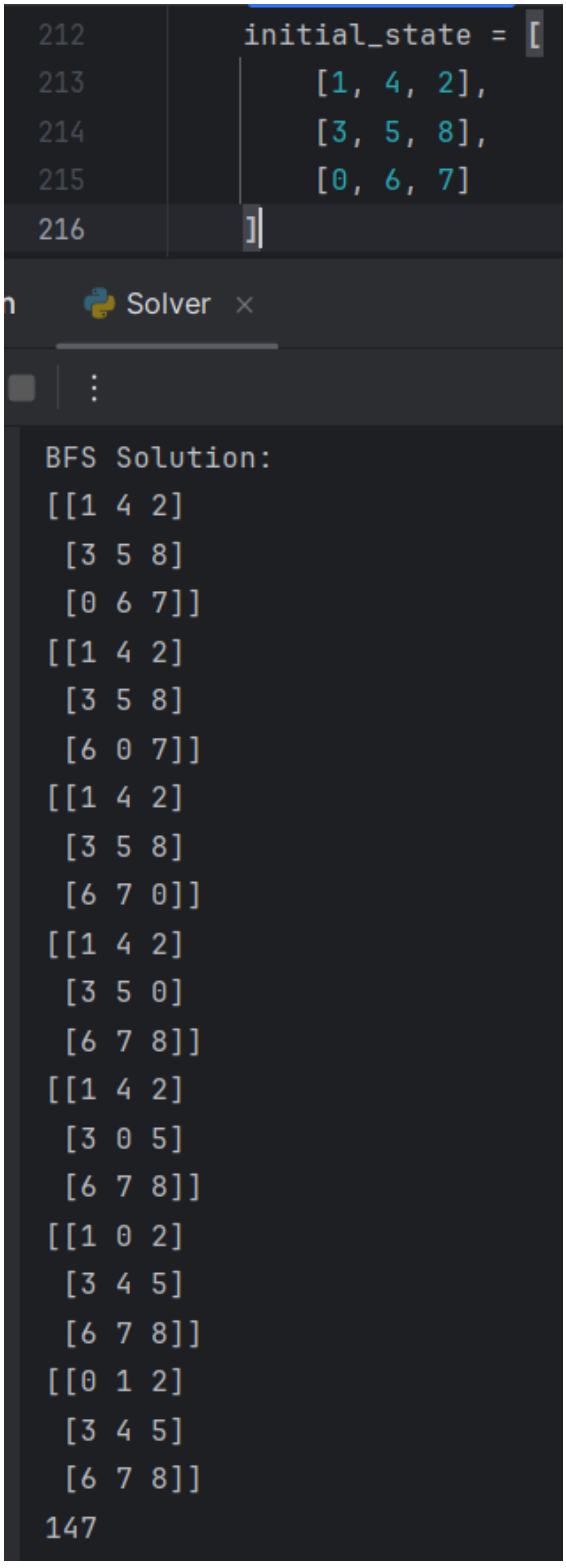
algorithms used:

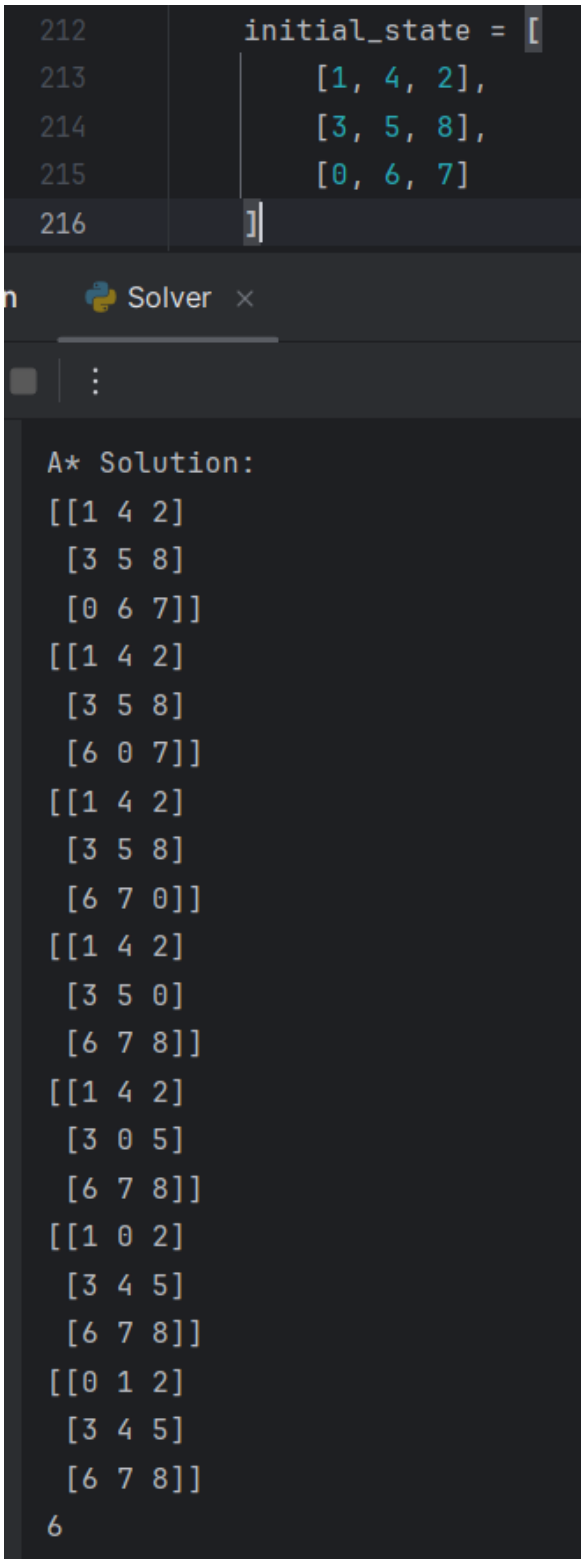
1. **Breadth-First Search (BFS):**
    - Explores all nodes at the present depth level before moving on to nodes at the next depth level.
    - Uses a queue for the frontier.
2. **Depth-First Search (DFS):**
    - Explores as far as possible along each branch before backtracking.
    - Uses a stack (implemented as a list) for the frontier.
3. **A* Search Algorithm:**
    - Combines the cost to reach the node (g) and the estimated cost to the goal (h) into a total cost f = g + h.
    - Uses a priority queue for the frontier to always expand the node with the lowest f value first.
    - Manhattan Distance: The sum of the vertical and horizontal distances each tile is from its goal position.
    - Euclidean Distance: The straight-line distance for each tile to its goal position.

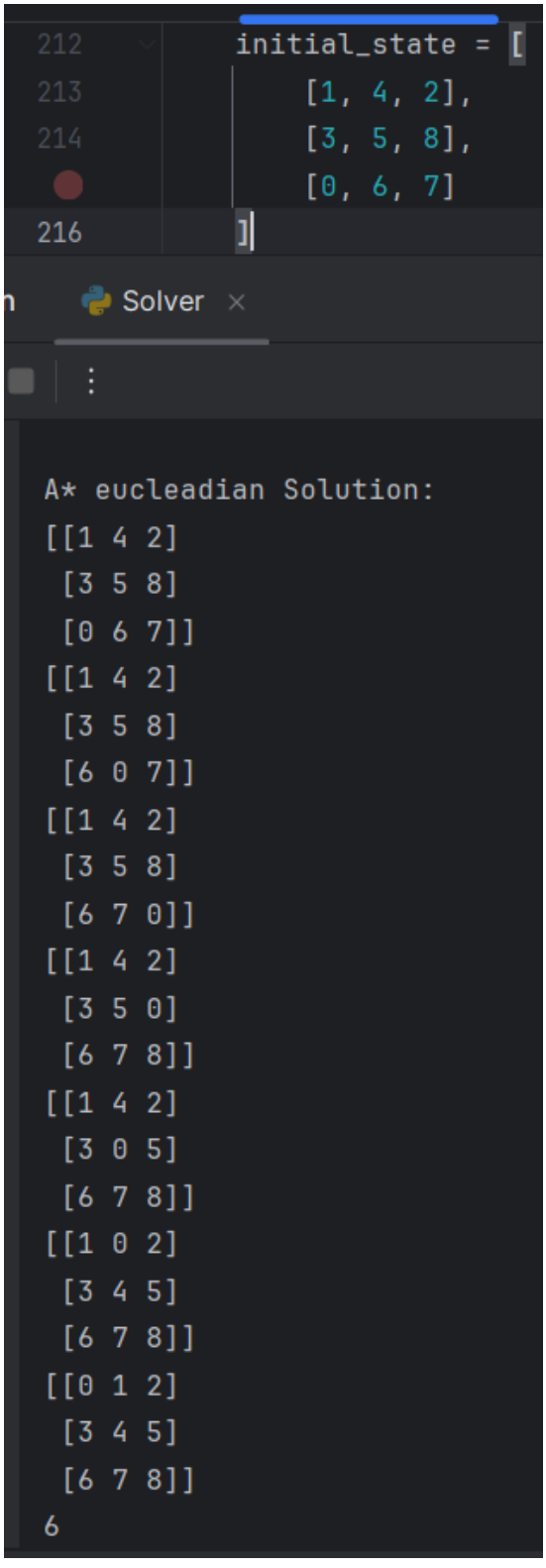
Sample runs:

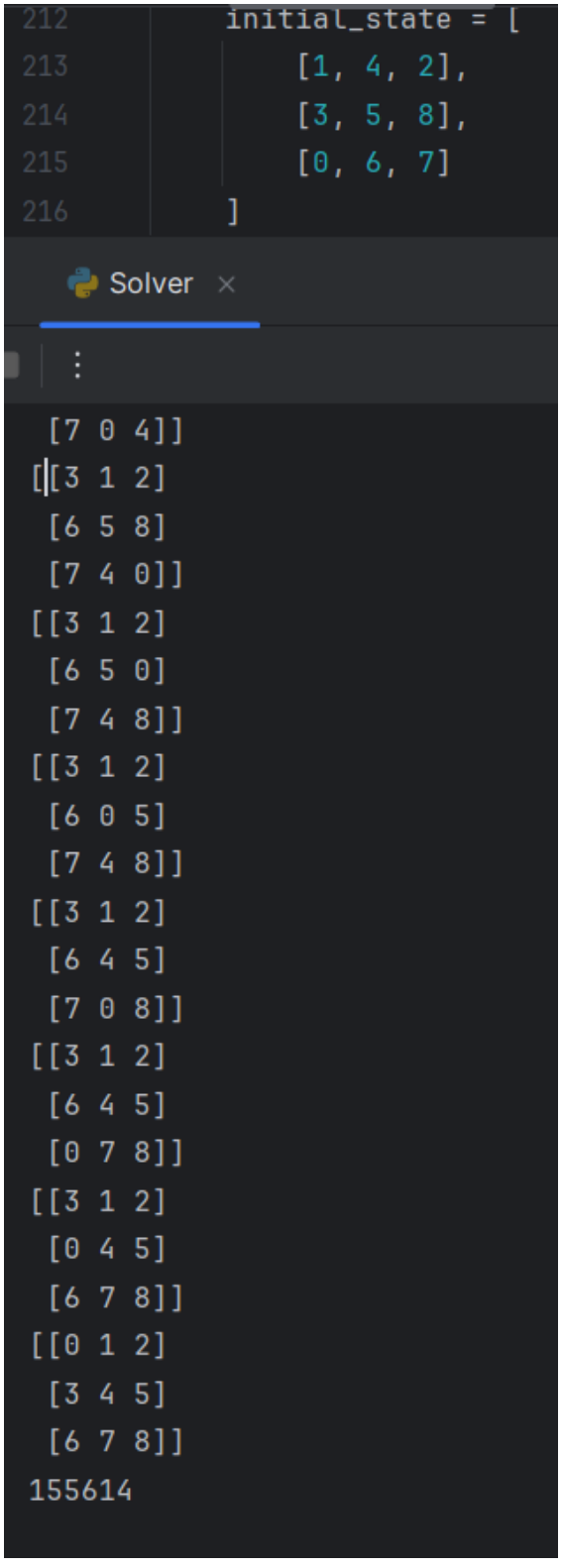
Those are paths and number of explored nodes using each algorithm

But for DFS only last couple moves are showed with the explored count






212 initial_state = [
213 [1, 4, 2],
214 [3, 5, 8],
[0, 6, 7]
216]
Solver
A* eucleadian Solution:
[[1 4 2]
[3 5 8]
[0 6 7]]
[[1 4 2]
[3 5 8]
[6 0 7]]
[[1 4 2]
[3 5 8]
[6 7 0]]
[[1 4 2]
[3 5 0]
[6 7 8]]
[[1 4 2]
[3 0 5]
[6 7 8]]
[[1 0 2]
[3 4 5]
[6 7 8]]
[[0 1 2]
[3 4 5]
[6 7 8]]
6


212 initial_state = [
213 [1, 4, 2],
214 [3, 5, 8],
215 [0, 6, 7]
216]
Solver
[7 0 4]]
[[3 1 2]
[6 5 8]
[7 4 0]]
[[3 1 2]
[6 5 0]
[7 4 8]]
[[3 1 2]
[6 0 5]
[7 4 8]]
[[3 1 2]
[6 4 5]
[7 0 8]]
[[3 1 2]
[6 4 5]
[0 7 8]]
[[3 1 2]
[0 4 5]
[6 7 8]]
[[0 1 2]
[3 4 5]
[6 7 8]]
155614

Provided below 3 test cases with the algorithms comparison


```
initial_states = [[
    [
        [6, 4, 2],
        [1, 3, 7],
        [0, 5, 8]
    ],
    [
        [0, 8, 3],
        [2, 1, 6],
        [4, 5, 7]
    ],
    [
        [1, 4, 2],
        [3, 5, 8],
        [0, 6, 7]
    ],
```

```
C:\Users\ADMIN\AppData\Local\Programs\Python\Python311\python.exe "C:\Users\ADMIN\OneDrive\Desktop\Uni\Summer 7\Ai\Project 1\analyze.py"
BFS path cost,time,no of explored for initial states respectively: [(23, 3.4804513454437256, 107796), (15, 0.14054465293884277, 5734), (25, 5.021486282348633, 145440)]
DFS path cost,time,no of explored for initial states respectively: [(58141, 4.862426042556763, 161869), (62499, 3.1778528690338135, 120816), (58369, 2.8850536346435547, 109916)]
A* Manhaten path cost,time,no of explored for initial states respectively: [(23, 0.8852262496948242, 495), (15, 0.018547058105468875, 43), (25, 3.2793912887573242, 940)]
A* Eucleadean path cost,time,no of explored for initial states respectively: [(23, 1.6552951335906982, 676), (15, 0.010732650756835938, 44), (25, 12.568846111297607, 1816)]
```

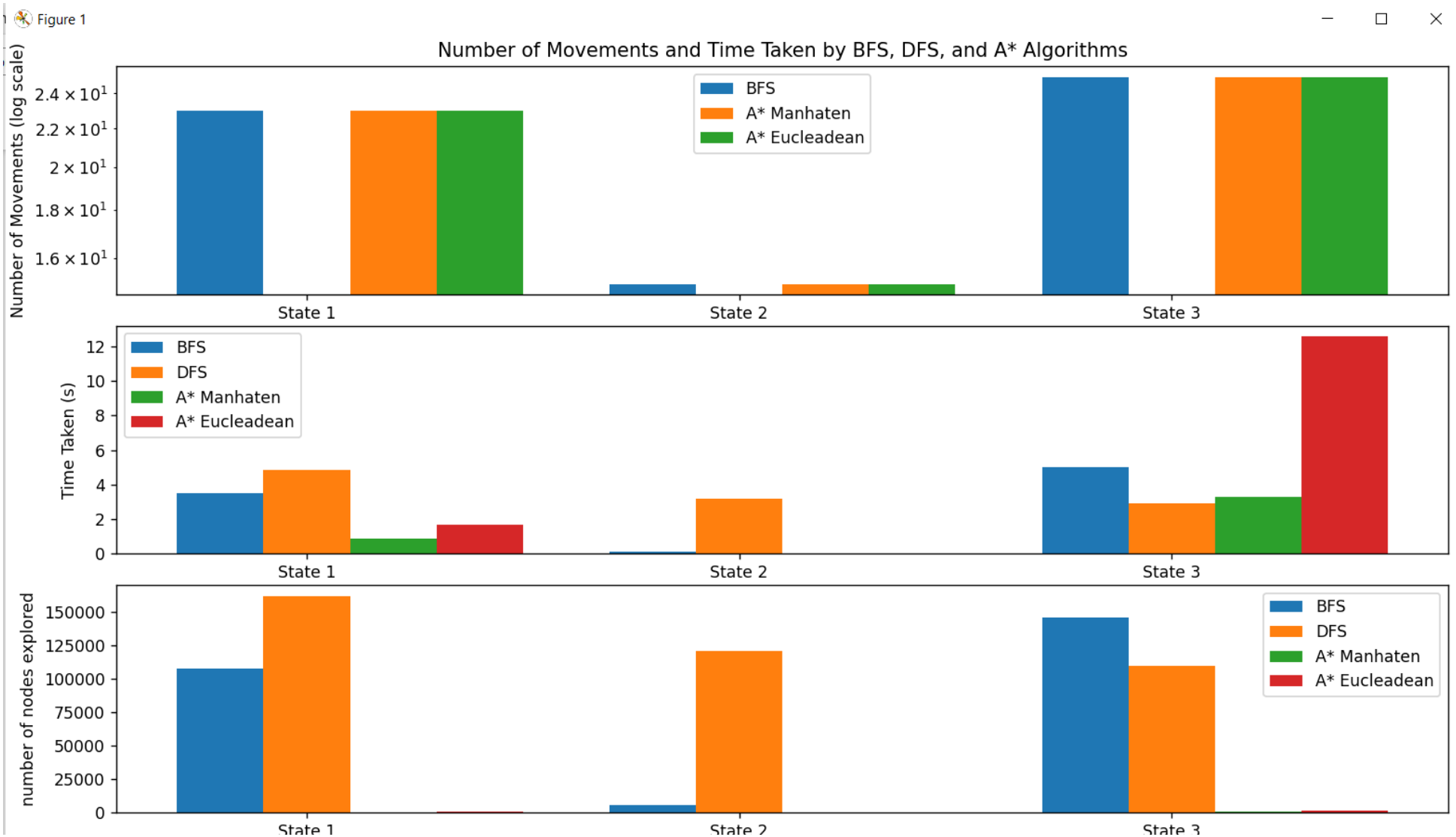


Figure 1
Number of Movements and Time Taken by BFS, DFS, and A* Algorithms
Number of Movements (log scale)
BFS
A* Manhaten
A* Eucleadean
State 1
State 2
State 3
Time Taken (s)
BFS
DFS
A* Manhaten
A* Eucleadean
number of nodes explored
BFS
DFS
A* Manhaten
A* Eucleadean

GUI with a sample run before and after solving and navigating to the end of the solution path