Artificial Intelligence

Match the Tiles - Sliding Puzzle Game

Heuristic Search Methods for One Player Solitaire Games

João Dossena - UP201800174
João Rosário - UP201806334
João Sousa - UP201806613
MIEIC, Class 8, Group 14

Match the Tiles Game

The goal of this project was the implementation of a solitaire game: Match the Tiles, a sliding puzzle game, not only to be played by a human but also by the computer.

The game itself is made up by many different levels of varying complexity with no time restriction and can be played on a 4x4 board, and has undo move support.

On each level, the aim is to get the colored tiles to the spots of the respective color. Up, down, left and right moves are the only actions allowed and each move affects all existing (movable) tiles. Therefore, desynchronizing the tiles is necessary, resorting, for that effect, to their interaction with fixed (or obstacle) tiles.

Search Problem

State Representation

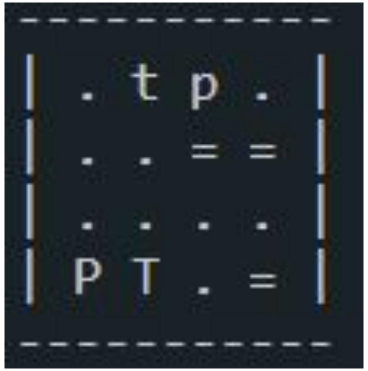
The board can be represented by a 16 elements 1D array (corresponding to 4x4) with characters representing the different tile possibilities: empty, obstacle, colored (piece(s) that can be moved) and respective final spot(s). Instead of having colors, we have lowercase letters representing the movable pieces, and uppercase letters representing their destination tiles.

Initial State

Board composed by N movable pieces, N destination tiles, obstacles and free tiles.

Objective Test

The level ends when all of the movable pieces are in their respective desired spots (all of the colored final tiles are occupied by the correct colored tiles, in this case, lettered tiles).



```
-----------
| . t p . |
| . . = = |
| . . . . |
| P T . = |
-----------
```

lowercase - movable pieces
uppercase - destination tiles
“=” - obstacle, “ . “ - free tile

Search Problem

Operators

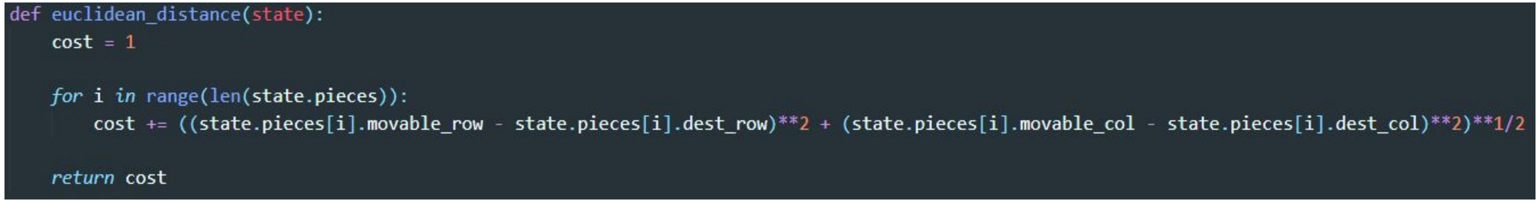
Name	Precondition(s)	Effects	Cost
up	$\exists_{\text{ColoredTile}}$ ColoredTile.(y - 1).isFree()	All colored tiles move up until the edge of the board or until they are adjacent to an obstacle tile.	1
down	$\exists_{\text{ColoredTile}}$ ColoredTile.(y + 1).isFree()	All colored tiles move down until the edge of the board or until they are adjacent to an obstacle tile.	1
left	$\exists_{\text{ColoredTile}}$ ColoredTile.(x - 1).isFree()	All colored tiles move left until the edge of the board or until they are adjacent to an obstacle tile.	1
right	$\exists_{\text{ColoredTile}}$ ColoredTile.(x + 1).isFree()	All colored tiles move right until the edge of the board or until they are adjacent to an obstacle tile.	1

Search Problem

<u>Heuristics/evaluation function</u>

We implemented two heuristic functions: euclidean_distance and min_string.

euclidean_distance: the sum of the Euclidean distances between movable pieces and their destination cells.

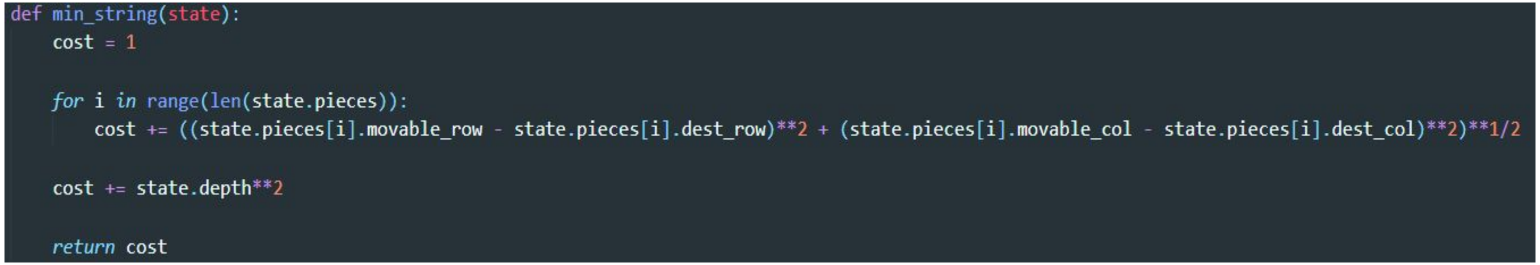


```python
def euclidean_distance(state):
    cost = 1

    for i in range(len(state.pieces)):
        cost += ((state.pieces[i].movable_row - state.pieces[i].dest_row)**2 + (state.pieces[i].movable_col - state.pieces[i].dest_col)**2)**1/2

    return cost
```

min_string: the same as euclidean_distance, with an addition of the node depth squared.



```python
def min_string(state):
    cost = 1

    for i in range(len(state.pieces)):
        cost += ((state.pieces[i].movable_row - state.pieces[i].dest_row)**2 + (state.pieces[i].movable_col - state.pieces[i].dest_col)**2)**1/2

    cost += state.depth**2

    return cost
```

Implementation

We have implemented the game in Python with simple text editors to be played on the terminal.

There are two main classes, each one with a respective file: State and Piece.
State stores information related to a game state (a tree node): board, parent state, current move sequence, depth, cost, key if applicable and a list of the pieces of the board.
In its turn, each object of the Piece class has data related to the movable piece (representation symbol and position coordinates) and to the destination tile of that movable piece (position and coordinates).

The data structures used are an one dimensional array (to store game states) that is utilized as a stack, and a set used to store the already explored states. For the search algorithms we also use sets, deques, lists and heaps.

The project is composed by a main and utilities files, two class files (Piece and State classes), a file where the levels are stored (levels.py), game logic, search algorithms and heuristics files, and a global variables file.

Experimental Results

Search Algorithm	Number of moves	Solution String	Execution Time (ms)	Nodes Expanded	Memory Used (KiB)
Breadth-first search	10	ldruluruld	32	46	256
Depth-first search	17	druluruldrulurul d	27	21	176
Iterative deepening search	10	ldruluruld	205	357	208
Greedy search	15	dldruldruldruld	19	15	192
A Star	10	ldruldruld	37	46	224

(results for level 25)

Experimental Results

Search Algorithm	Number of moves	Solution String	Execution Time (ms)	Nodes Expanded	Memory Used (KiB)
Iterative Deep	4	rdru	46	57	160
A Star	4	rdru	13	10	144

(lower complexity - level 5, not taking into account the last move direction, i.e. four possible moves from each state)

Search Algorithm	Number of moves	Solution String	Execution Time (ms)	Nodes Expanded	Memory Used (KiB)
Iterative Deep	10	ldruluruld	13124	8916	208
A Star	10	ldruluruld	66	47	256

(higher complexity - level 25, not taking into account the last move direction, i.e. four possible moves from each state)

Experimental Results

Search Algorithm	Number of moves	Solution String	Execution Time (ms)	Nodes Expanded	Memory Used (KiB)
Iterative Deep	4	rdru	15	11	72
A Star	4	rdru	10	9	112

(lower complexity - level 5)

Search Algorithm	Number of moves	Solution String	Execution Time (ms)	Nodes Expanded	Memory Used (KiB)
Iterative Deep	10	ldruluruld	216	357	172
A Star	10	ldruluruld	39	46	304

(higher complexity - level 25)

Conclusions

With the results shown previously we can conclude that any method that uses some kind of heuristic algorithm is much faster than all the others, especially when the complexity of the level increases.

With our A* implementation we always get the shortest solution to the problem with minimal to no downside in performance (most of the times we even get a faster solution).

We really enjoyed doing this project, in our heads it is really rewarding seeing the solution for a problem we couldn’t do ourselves appear right in front of our eyes, we even used our own application to add more levels (the game we based our levels on required completion of a level to unlock a new one).

Bibliography

- Google Play Website for the Match the tiles - Sliding Puzzle Game https://play.google.com/store/apps/details?id=net.bohush.match.tiles.color.puzzle&hl=pt_PT&gl=US
- Lectures “2a: Search Problems” and “2b: Solving Search Problems” Slides https://moodle.up.pt/pluginfile.php/185347/mod_resource/content/0/IART_Lecture2a_Search.pdf https://moodle.up.pt/pluginfile.php/185348/mod_resource/content/0/IART_Lecture2b_SolvingSearch.pdf
- Puzzle Algorithm Comparisons https://github.com/theprogrammer05/15-Puzzle-Algorithm-Comparisons
- 8-puzzle-solver https://github.com/speix/8-puzzle-solver