Match-the-Tiles
Artificial Intelligence - IART

Class 1 Group 12

Beatriz Costa Silva Mendes up201806551@fe.up.pt
Nuno Guilherme Amaral Santos up201405774@fe.up.pt
Telmo Alexandre Espírito Santo Baptista up201806554@fe.up.pt

Explanation of the project to be performed

The theme of this project is the game Match-the-Tiles, it consists of a board of N by M, where N, M is greater than 1, 1 or more colored blocks and equal number of goals, one for each block, matching in color. The objective of the game is to put each colored block on top of a goal with the same color. Along the board there are tiles (represented with color grey) that act as obstacles and do not move. Collision is enabled and can happen between any type of tile but no movement after said collision. Plus, every time the player moves a block in a direction (up, right, down, left) all colored blocks move the maximum distance in that direction as demonstrated in figure 1. The level ends when all colored blocks are in the same position of their designated goals.

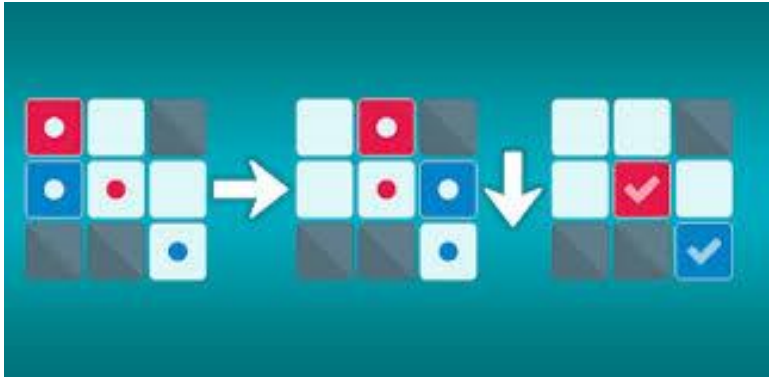


Figure 1 - Movement right and down.

The objective for this project is to find a way to evaluate a move and a game state so we can make consecutive moves and win the game with the least amount of movements possible. The evaluation of a move becomes more complex with the increase of colored blocks and size of the board, as well as the position of the goals, since it might require multiple colored blocks for one of them to stop in the goal.

Rules:

- All movable tiles (colored blocks) move as a group;
- You can only swipe up,down, left or right;
- Block and goal color must all match to beat the level;
- Tiles must all finish their movement in their designated goal to finish.

Formulation of the problem

State: GameState is represented by 4 main elements: a class with the common elements between all the game states (walls, goals and their corresponding coordinates), a list with the colored blocks and their coordinates, the move that originated the game state and number of moves between the initial state and itself
Initial state: Generated game.
Successor Operators: Generates valid states that result from execution.There are four actions (move the colored blocks right, left, up and down).
Objective Test: Checks whether the state corresponds to the objective configuration (all the colored blocks are in their designated goals).
Solution Cost: Each move (step) costs 1, the cost of the solution being the number of moves to solve the problem.

Operators:

1- Move Colored Blocks to the left

- **Preconditions**: the colored block that will be moved can't have a wall directly to its left;
- **Effects**: the colored blocks move to the left until they are stopped by a wall, the board border or another colored block, in the later case the colored block only gets stopped after the colored block to its left stops its movement;
- **Costs**: 1

2- Move Colored Tiles to the right

- **Preconditions**: the colored block that will be moved can't have a wall directly to its right;
- **Effects**: the colored blocks move to the right until they are stopped by a wall, the board border or another colored block, in the later case the colored block only gets stopped after the colored block to its right stops its movement;
- **Costs**: 1

3- Move Colored Tiles upwards

- **Preconditions**: the colored block that will be moved can't have a wall directly on top of it;
- **Effects**: the colored blocks move upwards until they are stopped by a wall, the board border or another colored block, in the later case the colored block only gets stopped after the colored block directly on top of it stops its movement;
- **Costs**: 1

4- Move Colored Tiles downwards

- **Preconditions**: the colored block that will be moved can't have a wall directly below it;
- **Effects**: the colored blocks move downwards until they are stopped by a wall, the board border or another colored block, in the later case the colored block only gets stopped after the colored block directly below it stops its movement;
- **Costs**: 1

Formulation of the problem - Heuristic

For this game, the heuristic function turned out to be very hard to find because a certain move might put the colored block near the goal, but at the same time might be turning the solution harder to find due to, for example, the existence of obstacles. To finish a level of the game, the player has to pick a strategy to move the block to the designated goal, or the opposite, chose a path from the goal to that block. With the adding of blocks with same/different colors, it becomes difficult to evaluate the chosen strategy move by move.

To lessen the problems of efficiency related to the heuristic for this game we chose to come up with several functions to be compared between themselves and added together to form the ideal heuristic function.

- **Heuristic Function 1 - Euclidean/Manhattan Distance**: the distance between a block and the corresponding goal, being able to apply layers of functions to calculated distances.
- **Heuristic Function 2 - Collinear and non-collinear**: based on estimating the moves left from the position of the block, checking if the block is collinear with the corresponding goal, blocked by a wall or not collinear.
- **Heuristic Function 3 - Value on collision**: on pre-processing, the walls are assigned values based on how many orientation changes it would take to get to it from the goal, so the block's move is valued based on which wall it collides after said move.

Implementation

For this project, we chose Python as the programming language, using Pygame for the graphic interface.

As data structures we will use *CommonGameState*, that saves the information immutable from move to move (wall and goal position, color), and *GameState* that contains the data for blocks and their position and the move that originated the state, so *GameState* provides functions to change the game state (*swipe_down*, *swipe_up*, *swipe_left*, *swipe_right*) that calculate the positions of each block after the movement and then calls the function *move* in order to change the values on the blocks positions. It also provides the function *is_game_over* that evaluates if the current game state represents the objective state.

The *tile_data file* contains several classes to save the coordinates of all types of cells(goals, walls and blocks).

We created a class named *Graph*, where we implemented the algorithms that will explain later on. This class has the functionality (using the class *Node*) of building a tree and saving the path it was found to the solution.

Our project is concentrated in the following files:

- matchthetiles.py - contains the main loop of the game and is responsible for the graphic interface
- game_state.py - contains the *CommonGameState* and the *GameState* class
- tile_data.py and utils.py - contains several classes designed to aid in setting and accessing coordinates
- graph.py - contains the graph class
- node.py - contains the node class
- utils.py - contains information about coordinates
- statistics.py - contains functions used to generate statistics and test all the heuristics and algorithms
- options.py - data class containing the heuristics options used by GameState
- solver.py - contains the interface that is called to solve a puzzle with a certain algorithm
- reader.py - contains functions used to read the levels and create game states

Approach

To obtain the best possible results, we combined all the heuristics that we created (euclidean/manhattan distance, collinear and non-collinear, value on collision)

- **Euclidean/Manhattan distance -** returns the distance between a block and its corresponding goal after applying a function to the list of all distances of that block to its corresponding goals. After it calculates for all blocks, it is applied another function to return the final value of the distance.
- **Collinear and non-collinear -** returns the estimate of the moves needed to reach the goal based on the position of a block and its corresponding goal, returning the minimal value of 0 if they overlap, minimum of 1 if they're collinear with no walls in the way, minimum of 2 if they're not collinear and minimum of 3 if they're collinear but have a wall in between.
- **Value on Collision -** return the value of a move based on the wall the block collided to. The value of said wall is calculated on loading the level.

In addition, we created an evaluation function that is used to evaluate a game state. This function (*eval_node()*) returns the function f(), which is equal to g() + h() depending on the algorithm that is used.

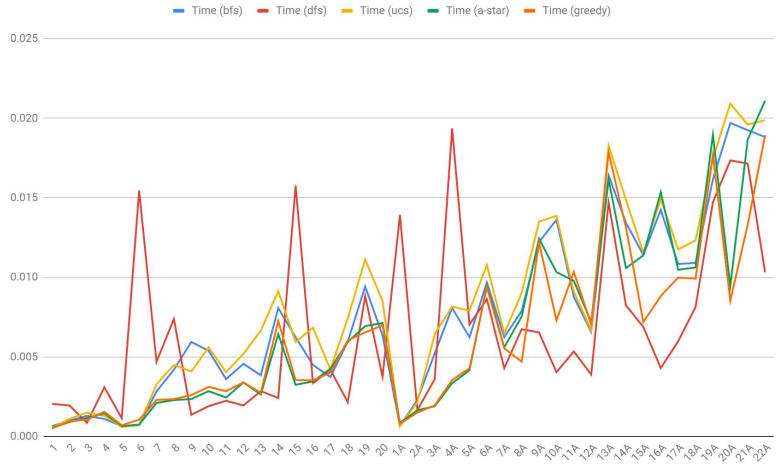
The operators are the same that are specified in the Formulation of the Problem

Algorithms Implemented

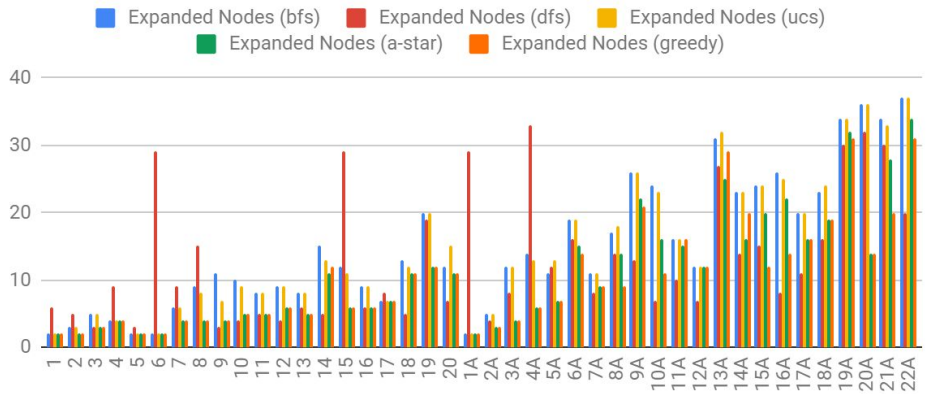
We implemented two **blind search** algorithms (breadth-first search and depth-first search) and multiple algorithms of **directed search** (uniform cost, greedy search, A* algorithm).

- We chose to not implement the iterative deepening algorithm because we realized that it would not be easy to define a proper limit for the depth level for our search, and would not be appropriate for our problem.

Statistics - Algorithm analysis



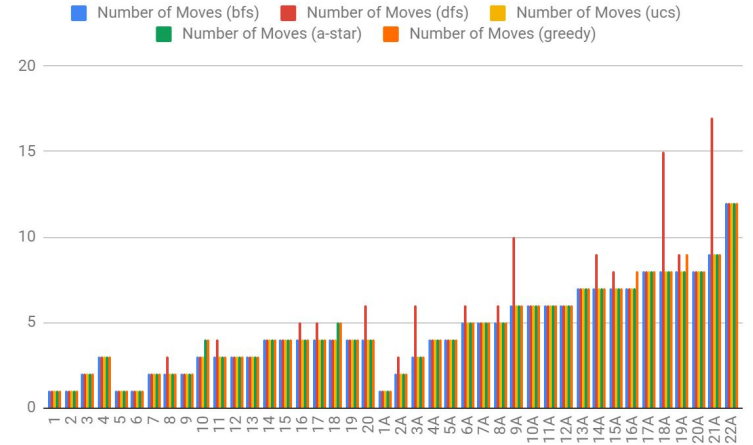
Graph 1. Execution time per level of each algorithm



Graph 2. Expanded nodes per level of each algorithm

Analysing the results obtained for each algorithm, we observed that:

- BFS always obtains the optimal number of moves;
- Greedy is the fastest in execution time;
- A* has, on average, the lowest number of expanded nodes, which results on less memory used;
- DFS is the most inconsistent algorithm, as expected, since it's not that appropriate for our problem due to its nature.



Graph 3. Number of moves per level for each algorithm

Statistics - Heuristic analysis

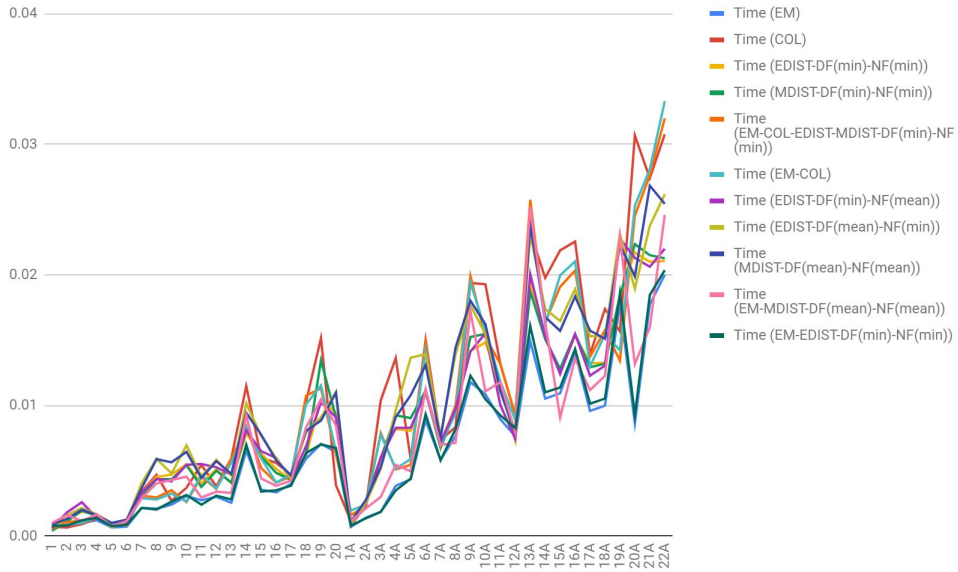


Chart 4. Execution time per level of each combination of heuristic using A*

Analysing the results obtained for each heuristic, we observed that:

- Using the function *min* in the distance based heuristic lead to the best results in terms of optimal path (in both DF (function applied to the group of distances of all blocks) and NF (function applied to the distances of a single block));
- Collinear and non-collinear (EM) heuristic is the fastest in terms of execution and memory usage;
- Using the *mean* function in the distance based heuristics lead to the worst results.

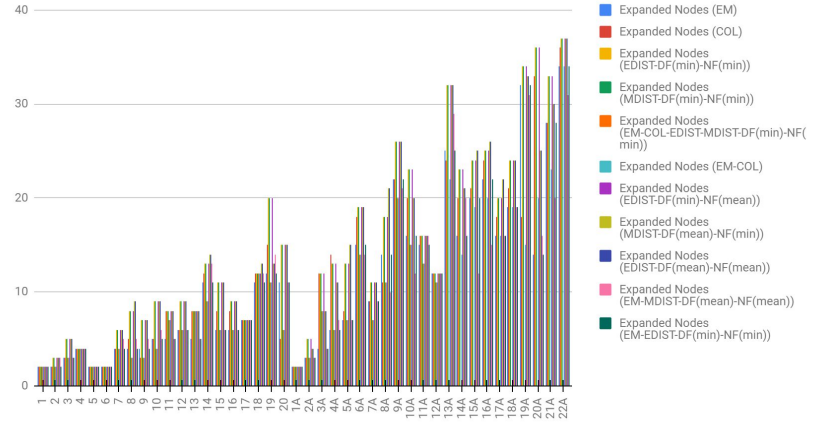


Chart 5. Expanded nodes per heuristic (memory usage)

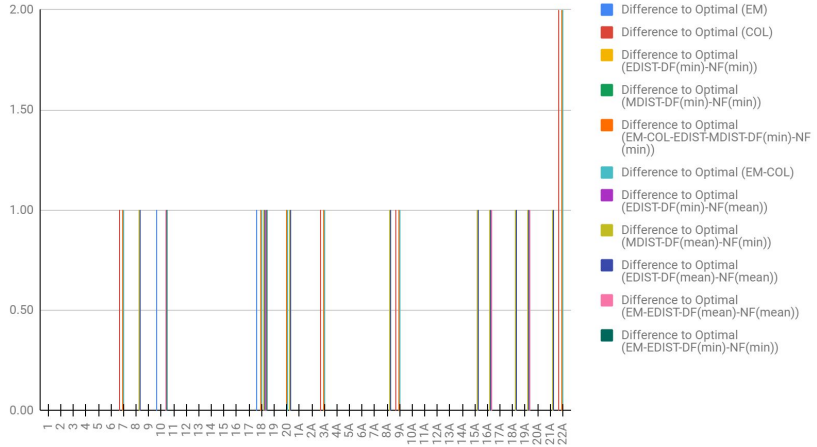


Chart 6. Difference in moves to the optimal number of moves per heuristic

Statistics - Error analysis

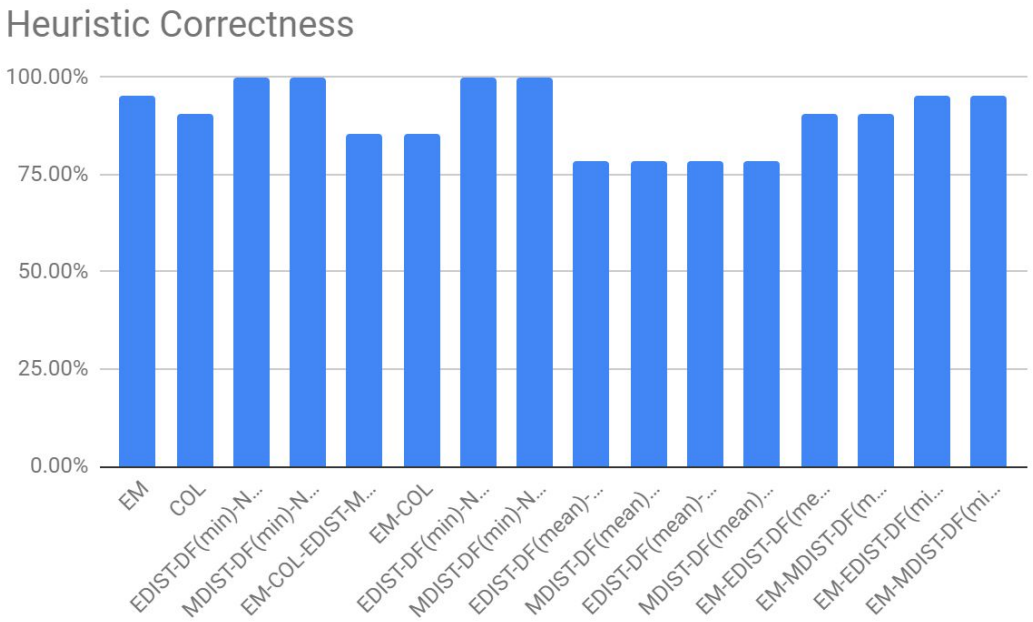


Chart 7. Percentage of correctness of each heuristic (number of times it obtained optimal result)

Since BFS algorithm always gives the optimal number of moves for each level, we used its results to evaluate the level of correctness (how many levels the heuristic got with optimal results) in order to choose the heuristic that lead to the most accurate results.

Observing the results obtained we concluded that the distance based heuristics (both Manhattan (MDIST) and Euclidean (EDIST)) were the most accurate with a correctness of 100% when using the function layers (*DF=min, NF=min* or *DF=min, NF=mean*), but obtained the worst results when both layers were the function *mean*.

The heuristic that estimates the number of moves needed to reach the goal based on collinearity of the blocks and goals (EM) obtained the second best results with a correctness of 95.24% while being the fastest one to execute as seen in earlier charts. The combination of these two previously mentioned heuristics also obtained a correctness of 95.24% and slightly improving the execution time.

Conclusions

After completing our project and analyzing all the results obtained, we realized that some algorithms were substantially better than others and the same happened with the combination of heuristic functions used.

We can observe in the data collected that, for example, the depth-first search algorithm is the worst since it might expand the tree in a certain way and not find the optimal solution. This is caused by the nature of our problem that is based per level and not depth. In contrast, the breadth-first search algorithm always gives the optimal result for our problem for the same reason explained above. A* algorithm is overall the best algorithm, since it's the most balanced between execution time, memory usage (number of nodes expanded) and results obtained being optimal (further looked into below).

Regarding the study of the heuristics, we concluded that, against all expectations, the functions that involved the calculation of distances between the blocks and the goals were the ones that obtained better results, in terms of correctness, if using the function *min* to operate on the list of distances obtained for the levels implemented by us. Though it was also observed that the same heuristic was the worst one, in terms of correctness (78.57%), when using the function *mean* to operate on the intermediate list of distances obtained. In addition to these functions, we also verified that the heuristic involving the collinearity and non-collinearity of blocks and goals combined with the euclidean distance was the heuristic function that obtained solutions quicker and spent less memory space (less node expansion). Although, this same function does not generate the optimal solution every time it was called (it has an accuracy of 95.24% in our tests, contrary to pure distance based heuristic that obtained 100% correctness with on average slower execution times and more memory usage). This last combination of heuristic function was the one chosen as default in our solver.

In sum, we can state that we reached fairly positive results and we were able to find heuristic functions with significant high accuracy. However, we underline that the chosen problem has a sizable difficulty added regarding the creation of heuristic functions different to the distance calculation, leading us to believe that a major part of the time spent on the development of this project was on the creation, improvement and correction of our functions.

References

- **Google Play - Match the Tiles**
  - https://play.google.com/store/apps/details?id=net.bohush.match.tiles.color.puzzle&hl=pt_PT&gl=US
- **Match Tiles - Sliding Puzzle Game**
  - https://www.youtube.com/watch?v=_6Yzx5eVVUs
- **Heuristic Search, Chris Amato Northeastern University**
  - http://www.ccs.neu.edu/home/camato/5100/heuristic_search.pdf
- **Heuristics, from Amit's Thoughts on Pathfinding**
  - http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
- **Pygame Documentation**
  - https://www.pygame.org/docs/
- **Statistics Data Spreadsheet**
  - https://docs.google.com/spreadsheets/d/1CTq4l7y3i_pXWlr7ymcHxUJ3RZwiFxsYHrUPwLuIZkU/edit?usp=sharing