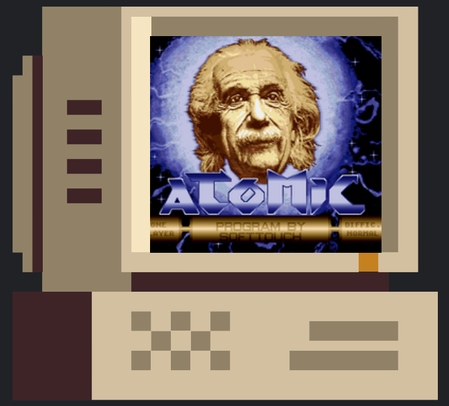
ONE PLAYER SOLITAIRE GAME

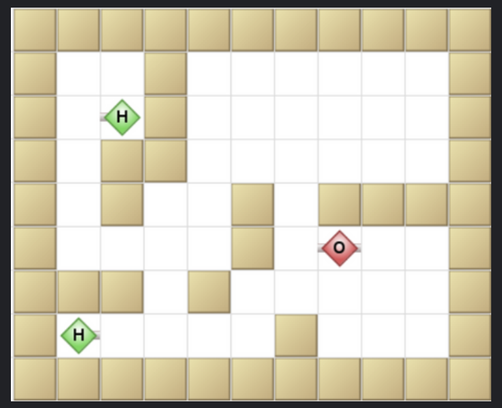
ATOMIX

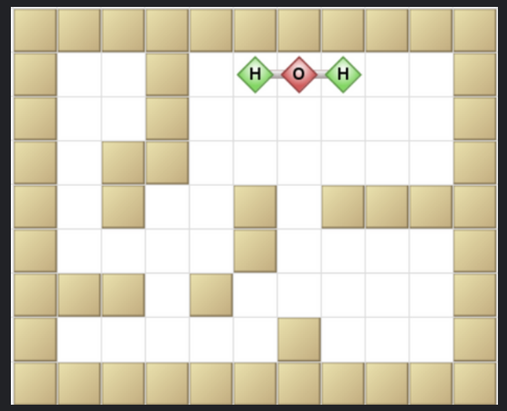
IART ASSIGNMENT 1



LARA MÉDICIS - UP201806762
VINÍCIUS CORRÊA - UP202001417

SPECIFICATION





Definition of the game:

- Atomix is a classic **puzzle game**.
- The objective of the game is to arrange atoms on a grid to create a specific molecule.
- The game involves a set of atoms of different types that can be moved around the grid in one of the 4 cardinal directions, sliding until finding a wall.

Optimization problem to be solved:

- Find the most efficient sequence of moves to form the molecule.
- Heuristic search, constraint satisfaction, and local search, to optimize the solution.

RELATED WORK

ARTICLES

Alex Zoch Gliesch (2015). Solving Atomix Exactly

Falk Huffner (2002). Finding Optimal Solutions to Atomix

WEBPAGES

Atomix (video game). In Wikipedia, Retrieved February 2023

SOURCE CODE

bishoybassem (2014). atomix [Java]. GitHub.

FORMULATION AS A SEARCH PROBLEM

State Representation	Represented by the board in that exact state.			
Initial State	The initial state is the starting position of the atoms on the board, which is pre-defined by the current level.			
Objective Test	Checks whether the state matches the target molecule.			
Operators	up(atom, X, Y)	down(atom, X, Y)	left(atom, X, Y)	right(atom, X, Y)
Preconditions	state[atom_X][atom_Y-1] != WALL	state[atom_X][atom_Y+1] != WALL	state[atom_X-1][atom_Y] != WALL	state[atom_X+1][atom_Y] != WALL
Effects	while(precondition) atom_Y -= 1 state[atom_X][atom_Y] = atom state[atom_X][atom_Y+1] = WALL atoms[atom] = (atom_X, atom_Y)	while(precondition) atom_Y += 1 state[atom_X][atom_Y] = atom state[atom_X][atom_Y-1] = WALL atoms[atom] = (atom_X, atom_Y)	while(precondition) atom_X -= 1 state[atom_X][atom_Y] = atom state[atom_X+1][atom_Y] = WALL atoms[atom] = (atom_X, atom_Y)	while(precondition) atom_X += 1 state[atom_X][atom_Y] = atom state[atom_X-1][atom_Y] = WALL atoms[atom] = (atom_X, atom_Y)
Costs	for each iteration, cost += 1	for each iteration, cost += 1	for each iteration, cost += 1	for each iteration, cost += 1
Heuristics/ Evaluation function	The evaluation function is estimated by the cost of reaching the target molecule from the current state. One possible heuristic is the Manhattan distance between the atoms in the current configuration and the atoms in the target molecule.			

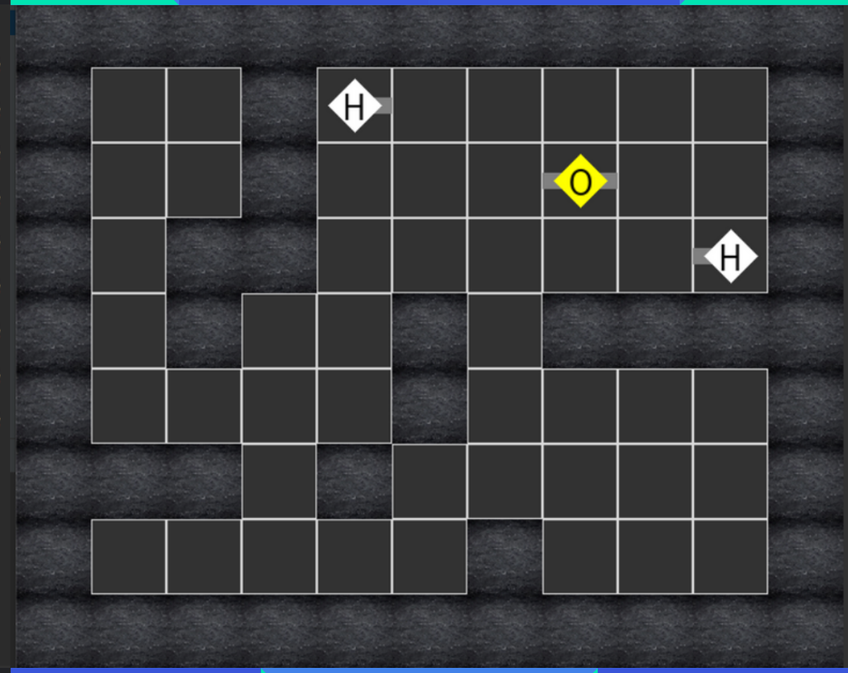
IMPLEMENTATION

- **Programming Language**
  - Being developed using Python programming language and Pygame framework.
- **Development Environment**
  - PyCharm by JetBrains
- **Data Structures**
  - **board**: a 2D array representing the game board, where each cell is a dictionary containing information about the atom in that cell, including its position, type, and connections.
  - **board_size**: a tuple representing the size of the game board (number of rows and columns).
  - **atoms**: a dictionary containing information about each type of atom, including its name, symbol, color, number of connections, and directions.
  - **selected_atom**: a string representing the currently selected atom type.
  - **game_over**: a Boolean value indicating whether the game is over or not.
- **Level Representation**
  - The figure on the right shows a diagram of a water molecule with different atoms represented by numbers 1, 2, and 3.
  - '#' represents the walls (boundaries) of the board and '.' means an empty cell.
  - The 123 means how the atoms must be disposed to form the molecule anywhere on the map.
  - Atom 1 is represented by a single H (Hydrogen) atom at the left of the molecule having one connection to the Right.
  - Atom 2 is represented by an O (Oxygen) atom located in the middle of the molecule, with one R (Right) and one L (Left) connection.
  - Atom 3 is represented by a single H (Hydrogen) atom located at the rightness of the molecule, with one L (Left) connection attached to it.

```
Water
##########
#..#1.....#
#..#...2..#
#.##.....3#
#.#..#.####
#....#....#
###.#.....#
#.....#...#
##########

123

1 H 1R
2 O 1R 1L
```



APPROACH

The heuristics used for the Greedy, A*, and Weighted A* algorithms took into account a number of variables:

- **Distance** - This is the sum of the shortest distances between all pairs of atoms that need to be bonded. Increasing the weight of this component may emphasize the importance of reducing the distance between atoms.
- **Openness** - This measures the number of atoms that can be connected to existing bonds. Increasing the weight of this component may emphasize the importance of making as many valid connections as possible.
- **Unbonded Pairs Distance** - This is the sum of the shortest distances between all pairs of atoms that are not yet bonded but need to be bonded. Increasing the weight of this component may emphasize the importance of connecting unbonded pairs quickly.
- **Pairwise Distance** - This measures the difference between the actual distance and the required distance between pairs of atoms that need to be bonded. Increasing the weight of this component may emphasize the importance of moving atoms closer to their correct relative positions in the molecule.

It first calculates the distance between all the atoms in the target bonds using the shortest_path function. Then, it calculates the number of open atoms that can be connected to the existing bonds using the openness function. It also calculates the distance between all unbonded atoms using the pairwise_distance function. Finally, it returns the sum of all these distances. The heuristic function is used to guide the search algorithm in finding the optimal solution to the problem.

IMPLEMENTED ALGORITHMS

Our initial approach involved selecting a group of algorithms for implementation, including BFS, DFS, IDDFS, UCS, Greedy, A*, Weighted A*. As we delved deeper into our research, we discovered that UCS was equivalent to BFS in cases where all edge costs were set to 1, which was the scenario we were dealing with. Consequently, we opted to discard the UCS algorithm. To streamline our implementation process, we categorised Greedy, A*, and Weighted A* as variants of each other, with A* being a variant of Greedy that accounted for the cost, and Weighted A* being a variant of A* that introduced a weight factor to the heuristic. This was achieved through inheritance. However, during our testing phase, we encountered issues with the DFS algorithm, which hit its maximum recursion depth. Despite our attempts to rectify the issue by implementing a non-recursive alternative, we continued to experience problems and thus concluded that DFS was not a feasible option.

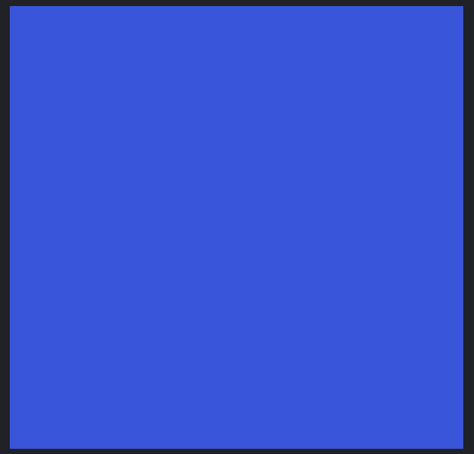
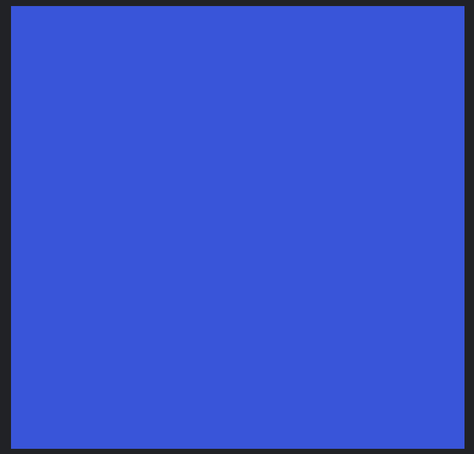
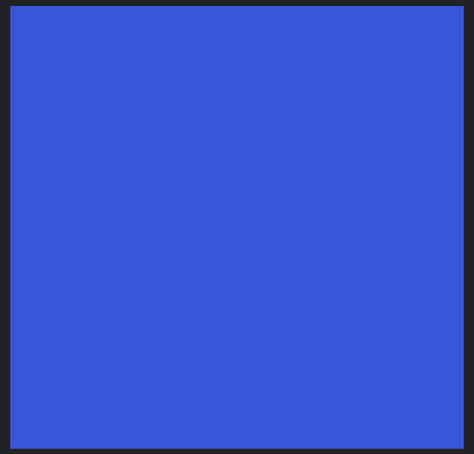
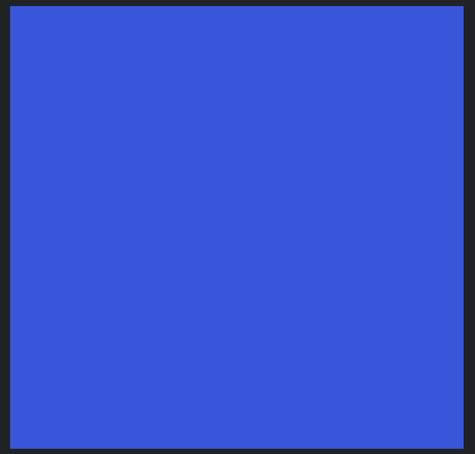
BFS - Uses a queue to explore all nodes at a given depth before moving to the next level.

IDDFS - The algorithm takes a node and a maximum depth as input, and performs a DFS search up to the specified maximum depth. If a solution is not found at this depth, the algorithm increases the depth limit and performs the search again, until a solution is found or the maximum depth is reached. The algorithm keeps track of the visited states to avoid exploring the same state twice. The function returns the moves required to reach the goal state if a solution is found, and returns None otherwise.

Greedy (A* & Weighted A*) - Uses heuristic described in previous slide The algorithm initialises a priority queue of states, starting with the initial node, and keeps track of visited states in a set visited. Then, it repeatedly pops the state with the lowest estimated distance to the goal from the priority queue and expands it by generating its children. If a child state has not been visited, it is added to the priority queue. The algorithm stops when it reaches a state where the is_molecule_formed() method returns True, indicating that the goal state has been reached.

IDA* - The IDA* algorithm implementation uses the heuristic function to search for the solution by iteratively increasing the threshold value until a solution is found. The search is performed recursively with the search function taking the current path, the g cost, and the threshold value as input. The algorithm maintains a set of visited states to avoid loops. If a solution is found, the function returns the solution path.

EXPERIMENTAL RESULTS

SOLVE TIME -

MAX MEMORY USED -

NUMBER OF OPERATIONS -

NUMBER OF MOVES -

OVERALL -

CONCLUSION

After conducting an analysis on the various algorithms used for solving Atomix, it became apparent that Weighed A* Search was the most effective algorithm overall.

However, when it came to implementing search algorithms in the context of solving games like Atomix, we encountered some significant challenges. Due to the fact that every board layout is a node on the graph and the search space consists of all possible layouts, the search space was enormous and continually expanded as we searched, making implementation considerably more difficult. This difficulty was particularly evident in informed search algorithms, as our research on heuristics revealed that most methods rely on pre-existing knowledge of the graph being searched.

During the execution of the DFS algorithm, we encountered persistent issues with a recurring maximum recursion error. After some investigation, we found that the issue was caused by our use of the deepcopy function from the copy library. We concluded that this function must have some level of recursion, which could be impacting the accuracy of the metrics we were collecting for the algorithms. Unfortunately, we were unable to change the core structure of AtomixState due to time constraints, and had to continue with the existing implementation.