Block Space Game AI

Final Delivery - Group 85

IA 22/23

Lia Vieira - up202005042

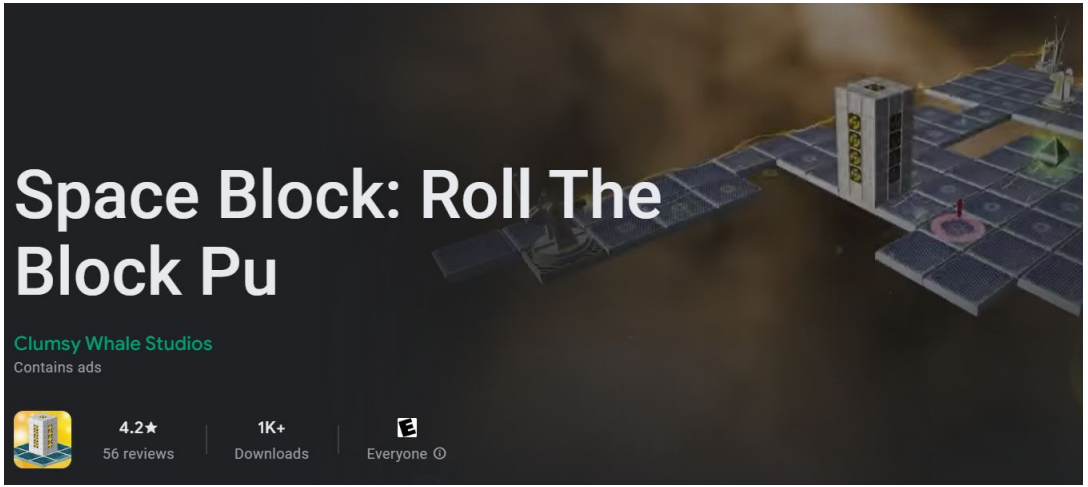
Marco André - up202004891

Ricardo Matos - up202007962

Specification of the work to be performed

For this project, we developed an adaptation of the “Space Block” game, in order to study Adversarial Search Methods for a 1-player game.

The game has several levels with different maps and the goal is to move the block controlled by the player to the exit portal (where it should be placed vertically) without hitting any obstacles or falling off the map before the timer ends.



Formulation of the problem as a search problem

State Representation:
BlockState(x, y, x2, y2) → static maze (with starting and ending positions) represented as a 2D matrix.

- if X and X2 are equal and Y and Y2 are equal than the block is standing.
- Otherwise, the block is on its side with (x, y) and (x2, y2) representing, the two cubes of the block, respectively.

Objective State: The block is standing and in END_NODE
A move is invalid if one or both positions of the block are on an invalid position.

Heuristics/evaluation functions:

- Informed Algorithms: Greddy, A*, Weighted A* …
- Uninformed: DFS, BFS, Iterative Deepening …
- Heuristic evaluation notes:
  - should aim for the shortest solution.
  - a good heuristic function should try to get closer to the solution node
  - Chebyshev, tries to go on a diagonal to the the solution.
  - Manhattan, gives same heuristic evaluation values to solutions that biforked themselves in the same way (as a analogy to the place that gives its name)

Formulation of the problem as a search problem

Operators	Effect	Cost
Left	• Standing: x2 -= 1 && x -=2 • else if vertical: move left • else: standing (x = x2 = --x)	1
Right	• Standing: x2 += 1 && x +=2 • else if vertical: move right • else: standing (x = x2 = ++x)	1
Up	• Standing: y2 -= 2 && y -=1 • else if horizontal: move up • else: standing (y = y2 = --y)	1
Down	• Standing: y2 += 2 && y +=1 • else if horizontal: move down • else: standing (y = y2 = ++y)	1

Note: The player doesn't have any preconditions but the AI **has as precondition moves** that don't lead to **losing/dying**

Implemented work

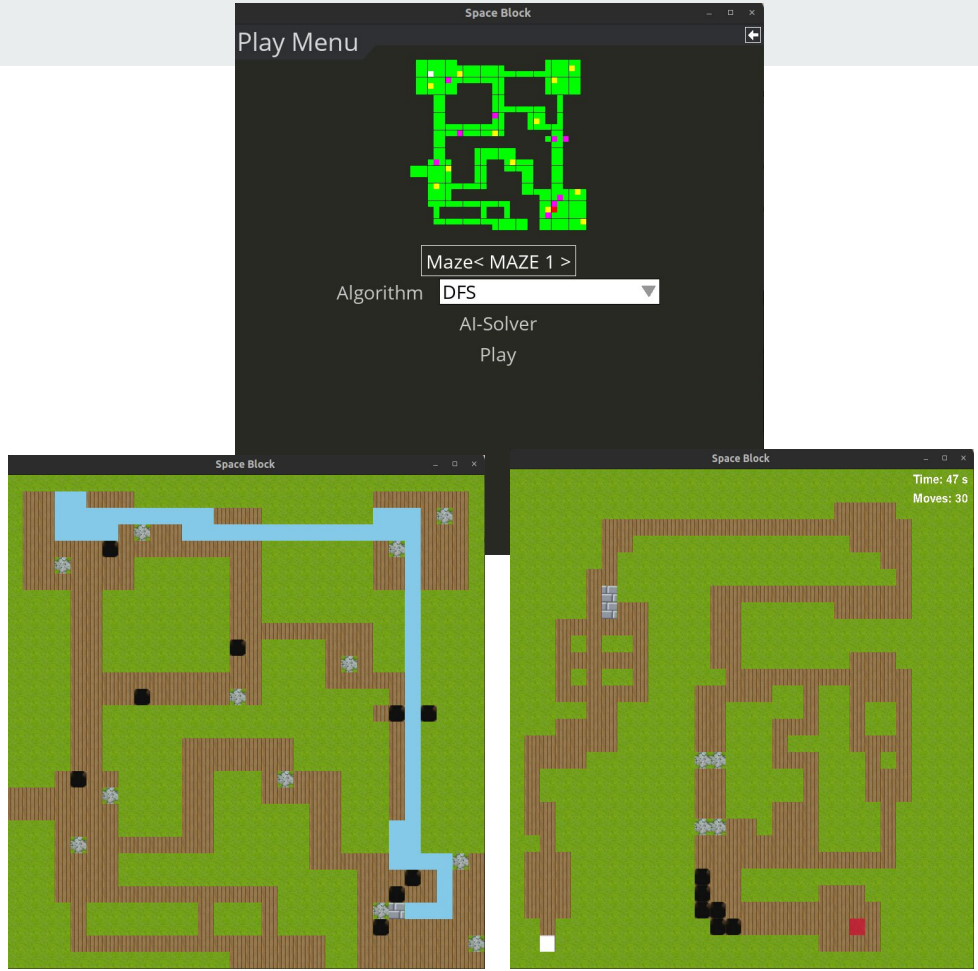
For this project, we opted to use the recommended programming language: Python 3.10.

We implemented all algorithms and heuristics (and more) with this project and achieved satisfying results.

To ensure we had clean and organized code, we followed a OOP path with the classic MVC architecture.

We added visual menus and game displaying with *Pygame* and *Pygame_menu* (the final GUI can be seen to the right).

The game can be played by the player with the option to request assistance from a chosen algorithm by pressing the **'H' button and a hint will be shown**. Alternatively, the maze can be solved entirely by a specific algorithm.



Implemented Algorithms

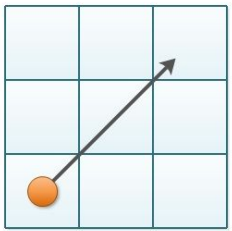
- **DFS:** Explores as far as possible along each branch before backtracking.
- **BFS:** Visits all the vertices at the same level before moving on to the next level.
- **Random DFS:** Chooses a random node and expands from it. Used in the **genetic** and **maze generator** algorithms
- **Iterative Deepening:** Combines the advantages of BFS and DFS. It starts with a shallow depth limit of 200 and incrementally increases it until the goal is found.
- **Greedy:** Selects the best option available at each step.
- **A*:** Informed search algorithm that uses heuristics (manhattan, chebyshev and euclidian) to guide the search towards the goal. It combines the advantages of BFS and Greedy search.
- **Weighted A*:** w=1.5. Reduces the number of nodes expanded in **A***, producing optimal (or not by far) solutions
- **Genetic:** Optimization algorithm that uses the principles of natural selection and genetics to find the optimal solution. It selects the fittest with a tournament and a roulette selection, mutates (generates a random dfs from a certain point) with a 15% chance. Crossovers using parts of the interception of the parents

Heuristics

- **Manhattan Distance**: well-known heuristic for pathfinding. Calculates the distance between two points on a grid by summing the absolute distances of their horizontal and vertical coordinates.
- **Chebyshev Distance**: another common heuristic, calculates the distance between two points by considering the maximum of the absolute differences.
- **Euclidean Distance**: uses the pythagorean theorem to calculate the heuristic value.

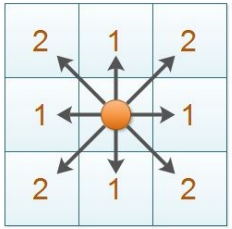
To guarantee **admissibility** in the **A*** algorithm we had to **divide all heuristics by 2.0**. This is due to the fact that when the block is by his side, the heuristic evaluation can give a value of 2, when the cost is actual 1 (depending on the block x,y and x2, y2 parameters). So, to keep the heuristic **admissible** we had to divide by two the heuristic cost. The **consistency** of the heuristic is also kept, as every move has a cost of one, keeping the **triangle inequality** property.

Euclidean Distance



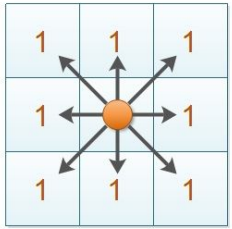
$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Manhattan Distance



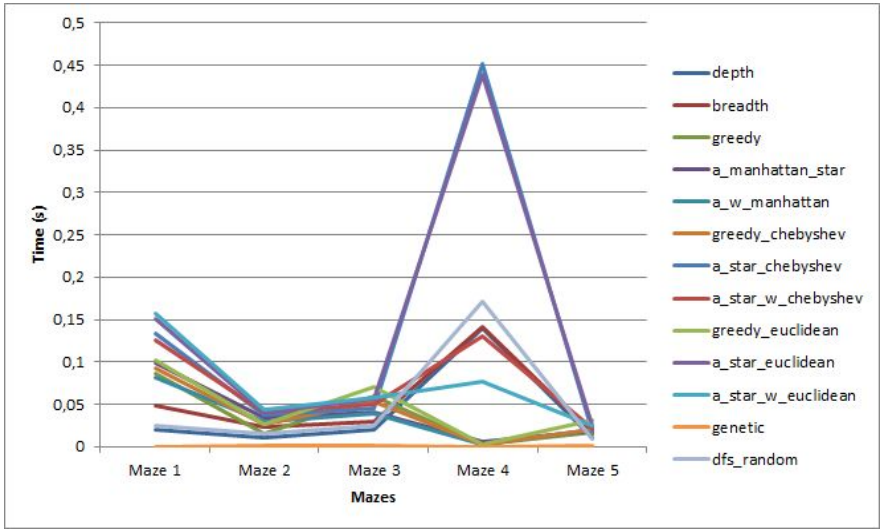
$$|x_1 - x_2| + |y_1 - y_2|$$

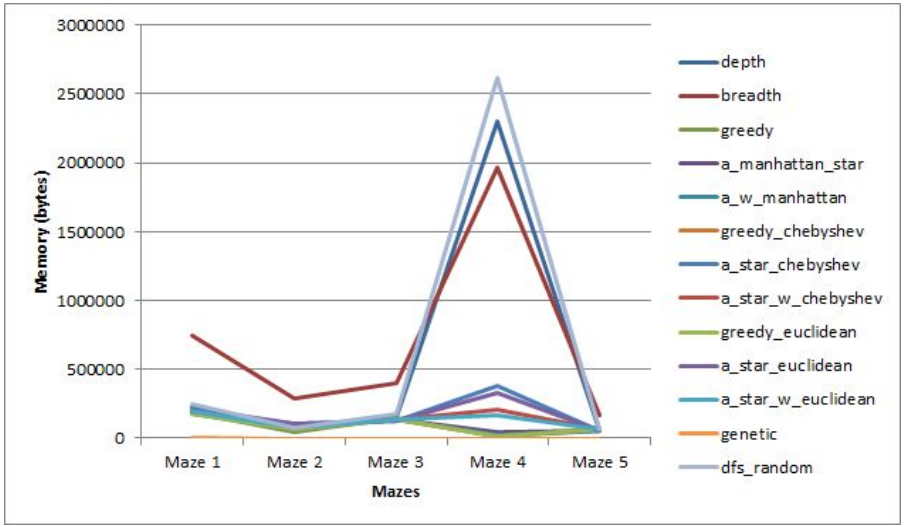
Chebyshev Distance



$$\max(|x_1 - x_2|, |y_1 - y_2|)$$

Experimental Results





As we can see, there is a huge improvement in the number of nodes expanded with the **A* weighted**, as shown in the second chart. The execution time also decreases. Note that we decided to omit the **iterative deepening** as it had terrible performance. The **greedy algorithms** are the **best** in terms of **memory use** and **execution time**, however their solution is not optimal in open mazes. There is also a huge impact on the maze layout, **strict mazes (with limit branching)** are good for greedy approaches as there is not a lot of alternative moves that can be made, however, **open mazes** as maze_4 are **costly in terms of time and space** and bad for the greedy algorithms optimality.

Conclusion

We started this project with limited knowledge about the practical applications of search algorithms and during the development of this game's AI we reached a better understanding of it.

The game itself is pretty simple and didn't take long to code so we had time to explore different algorithms and approaches to the problem in question. With that in mind, we were able to implement all suggested algorithms and implemented many different heuristics with different degrees of success.

Even if it wasn't a target or suitable algorithm for this project, we also explored the viability of a metaheuristic - genetic algorithm - and found it a very interesting approach even if it was far from optimal when compared with A*, for example.

With all that being said, when concerning the results emerging from the statistical analysis done by us to study the implemented algorithms, we noted the impact that different mazes had on the memory and time usage. An AI solver should try to find an optimal solution that does not represent a bottleneck for performance. In our own experience, weighted A* represents a good solution for larger mazes, although not being optimal.

As a final note, the maze sizes aren't that big and so almost every optimal algorithm represents a good solution.

References

During the development of this project, the course slides were of major inspiration. Moreover, when concerning the IA algorithms to be used on this project, we already had previous knowledge of some of them, from previous curricular units like Algorithm Design and use that course materials as well.

Furthermore, we, in the spirit of the course, used AI tools to aid us in the creation of the project:

- Chat GPT → Helped coding the statistics generator
- Github Copilot → Helped with code documentation

The heuristics image present on the 7th slide was taken from here.