Introduction to Artificial Intelligence

Andrea Onorato, Enrico Tuda, Jorge Arias Cuesta, Sergio Monzon

March 2024

1 Game Rules

The game Ricochet is a board game and the main objects present on the board are walls, robots, and a goal. The game has to be played by at least 2 players. The game it's a multi-player game, where there is a timer and every participant has to say a number which is the length of the path to the solution that they thought about and when the timer stops the player with the shortest path in mind has to show it. The agents are the robots. Each player aims to find the shortest path to move a robot of a certain color to its goal, i.e. goal with the same color as the robot, in the fastest way. There must be a maximum of one goal. The robots can be moved in 4 different directions (up, down, left, right) and they have to go in that direction until they don't find an obstacle or reach the border of the map. To create a solution you are allowed to move also other robots (not only the main one) to create obstacles, every time that you move any robot it counts as 1 move in the length of your final solution.

2 What kind of game is it?

Ricochet is a multi-player board game, involving two or more players competing against each other. It is a competitive game where each player aims to find the shortest solution to the goal in the shortest time. The game is not zero-sum, because there are not any gains by one player directly balanced by the losses of the other players. Ricochet is often played turn-based, where players take turns to make their moves. Each turn consists of a player moving the robots across the board. Ricochet involves perfect information, where players have full knowledge of the current state of the board, including the positions of all pieces and obstacles. Ricochet is deterministic, meaning that the outcome of any action is fully determined by the initial state of the game and the choices made by players. There are no random elements involved. Ricochet's turn-based nature makes it suitable for algorithms that can explore the game tree deeply during each player's turn, analyzing potential future moves and their consequences making DFS and BFS valid solutions.

3 State space of the game

To determine the size of the game's state space, we consider a 16x16 grid with 50 walls. With 4 robots also present, there are 202 free squares. To find all possible states of the game, we apply the formula for combinations without repetition.

$$\frac{202!}{4!(202-4)!} = 67331650 \approx 10^7$$

Therefore, it is not computationally possible to explore the entire solution space to find the optimal solution, because it would require too much time to explore the whole space, but it is necessary to rely on an algorithm specifically designed for the game to find even an acceptable solution in a reasonable time.

4 Description of elements

The initial state is determined by the level chosen, each level will contain different robots, walls, and goal positions. It is also possible to start with a random initial state, where robots, walls, and goal are placed pseudo-randomly on the board. The players can perform actions, which are selecting a robot or moving a robot up, down, left, and right until it doesn't hit an obstacle. The result of the action will be the new state containing the new position of the robot. The program Terminates when the main robot for the game has reached its goal, practically the coordinate of the robot must be equal to the one of the goal and that is checked every time after one action is performed (a result is created). The function Utility is used by A* and Greedy BFS, based on the Manhattan distance heuristic. The robot is invited to perform the action that will lead itself closer to the goal, so the satisfaction factor generated by the utility function will be indirectly proportional to the distance between the robot and the goal. The lower the distance the better. The higher the satisfaction factor, the better.

5 Representation of states

States are represented by the position of pieces on the board. A state contains the walls' position, robots' position, and the goal position. After a move ("up", "down", "left" or "right"), a new state is generated with the updated robot position. One robot is moved at a time, every time the robot moves the moves counter will be increased by 1. Design-wise, after an action (move) we decided to change only the robot's position and take the position of walls and goal the same as the initial state since the game does not imply changing of position of walls and goal during the game. We decided to store the robot in a map where we have the ID of the robot as a key (integer number starting from 0) and the coordinates of the robot's position as value, walls are represented by their coordinates and they are stored in a set, and the goal is stored in a tuple containing its coordinates.

6 Algorithms used by the AI

In the context of the Ricochet Robots game, various search algorithms have been developed for constructing artificial intelligence to find a path to the solution. When these algorithms fail to find a solution, another robot will be randomly selected to make a random move before attempting to calculate the correct path again. The algorithms used include: Depth-First Search (DFS), Breadth-First Search (BFS), A* Search, and Greedy Best-First Search (GBFS).

- **Depth-First Search (DFS)**: explores as deep into the search space as possible before backtracking. The algorithm creates his children in order of directions, starting from north, down, left, and then right. That means that the DFS will first explore all the northern cells at first. This can be memory efficient and useful in scenarios where the search space is vast, but it does not guarantee finding the shortest path to the goal. In Ricochet Robots, where the objective is to reach a specific location optimally, DFS might not be the best choice due to its tendency to explore less promising paths.
- **Breadth-First Search (BFS)**: examines the nearest nodes to the starting point before moving on to further levels. That means that first explores the cells with distances 1 to the initial state, then 2, and so on. It ensures finding the shortest path in terms of the number of moves, which is advantageous for determining the optimal solution. However, its main drawback is intensive memory use, especially on large boards with many obstacles.
- **Greedy Best-First Search (GBFS)**: uses a heuristic to guide the search toward the goal, prioritizing nodes that seem to be closer to the goal. In our code, we have used the Manhattan distance heuristic. Although this method can be fast, it does not guarantee finding the shortest route since it tends to "lock into" suboptimal paths if the heuristic is not perfect.
- **A Search***: combines the strengths of BFS and GBFS, using both the cost so far (as in BFS) and a heuristic (as in GBFS) to determine which path to explore next. This not only guarantees that the shortest path is found but also significantly reduces the number of nodes explored compared to BFS, making it more efficient in terms of memory and time in many cases.

Explanation and differences using Level 2:

- Using **BFS**, the algorithm finds the optimal solution.
- The difference using **DFS** (compared to BFS) is that the algorithm first explores the cells that are in the northern part, since the order of the directions to generate children is up, down, left, and right. The optimal path goes from 2 to 3 using the direction "down", but DFS explores first the direction "up", not finding a solution.

- The algorithm **A-star** works in the following way: If a path to the goal is found, the algorithm reconstructs the path by tracing back through the recorded parent nodes from the goal node to the starting node. As we are using the heuristic, cells 7 and 8 are going to be generated before cells 16 and 17, and even 16 and 17 follow the optimal solution. With this example, we can see that some heuristics might also confused when choosing nodes.
- In this specific case, the algorithm **GBFS** generates the same number of nodes as A-star, but this might not be the case in other examples, since A-star takes into account the number of movements to reach that state (g(n)).

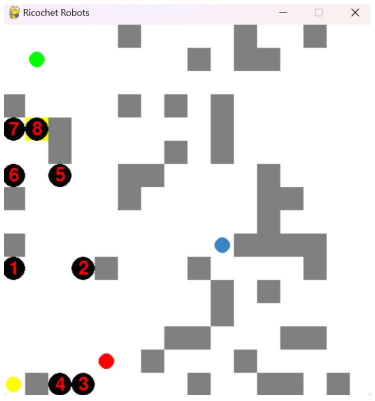

Figure 1: Solution path using BFS, explores all reachable states at a given depth level before moving on to the next possible move. DFS finds the same path.

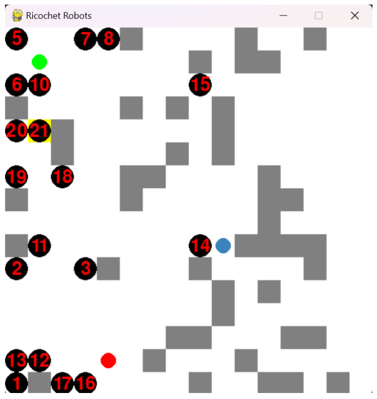

Figure 2: Generated states using A-star and GBFS. It shows the states generated instead of the solution path for a better understanding of the algorithm.

7 Heuristics

Our AI player uses the Manhattan heuristic for both A* and greedy best-first search algorithms estimating the remaining distance to the goal from each explored cell based on the Manhattan distance. The Manhattan distance heuristic calculates the distance between two points on a grid by summing the absolute differences in their x and y coordinates.
A* uses the evaluation function

$$f(n) = g(n) + h(n)$$

, where: g(n) is the cost of reaching the cell from the start. h(n) is the heuristic estimate of the remaining distance to the goal (Manhattan distance).

Greedy BFS uses the evaluation function:

$$f(n) = h(n)$$

As it can be seen Greedy calculates only the heuristic to decide which path to follow. While the heuristic function employed in our AI system demonstrates limitations in certain scenarios, it nonetheless exhibits utility in specific level configurations. Across a diverse range of levels, the heuristic function occasionally proves beneficial by providing heuristic guidance toward the goal state, thereby aiding the search process.

8 Performance based on Benchmark

The AI can play using four different search algorithms: BFS, DFS, A*, and Greedy BFS. The heuristic and evaluation function for informed search algorithms like A* and Greedy BFS shows improvement in finding a solution in some levels where the steps to get to the solution are in direction towards the goal, while in other more intricate levels, BFS and DFS are faster into finding a solution and often the solution is optimal. This is regarding levels where you need to move only the main robots, treating the game as a single-agent system. If we want to move different robots, we rely on the random movements function and we get to a sort of dumb multi-agent system. Below are the benchmarks of our algorithm on different levels:

Algorithm	Level 1			Level 2			Level 3			Level 4		
	T	S	P	T	S	P	T	S	P	T	S	P
BFS	0.9	184	5	1.4	144	8	∞	∞	∞	2.9	148	8
DFS	0.1	20	5	1.2	204	8	∞	∞	∞	2.2	112	8
A-star	3	19	11	2.2	21	9	∞	∞	∞	1.0	9	9
GBFS	1.5	19	11	3.5	21	9	∞	∞	∞	0.1	9	9

Table 1: T: Time in milliseconds. S: states generated. P: solution length

It can be noticed that Level 4 is a level where the Manhattan heuristic works well since it takes less time to find a solution with respect to DFS or BFS. Moreover, Level 3 is not solvable, which is why the search never ends in that case.
Below are the graphs that show the comparison between our AI algorithm in different levels:

We can see that BFS and DFS in general found shorter solutions compared to A* and Greedy BFS, this is due to the non-optimal heuristic used. Because the heuristic is expensive to compute and it helps only at certain levels. In general, it can be concluded that the AI is faster than a human playing, so our research goal can be satisfied.

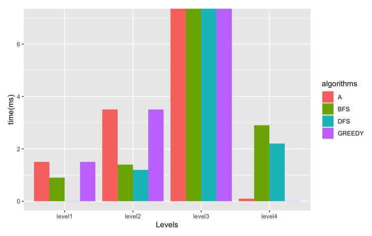

Figure 3: Time to find a solution comparison

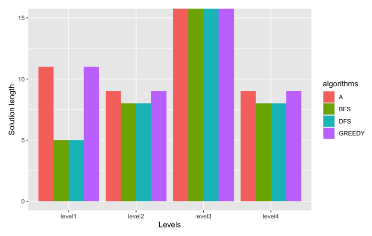

Figure 4: Solution length comparison

9 Future works

Minor improvements and optimizations can be achieved through the refinement of existing algorithms and data structures. For instance, fine-tuning the heuristics used in A* or enhancing the efficiency of data structures for storing game states might yield better performance. However, these optimizations will likely offer incremental improvements rather than addressing the core limitations related to complex scenarios and cooperative dynamics.

The current approach, utilizing algorithms like BFS, DFS, A*, and Greedy Best-First Search, has limitations in complex scenarios and lacks the cooperative dimension between robots which would be desirable. Multi-Agent AI, where multiple intelligent agents interact within the environment, could introduce dynamic problem-solving strategies, allowing for more efficient puzzle-solving in the game and solutions for more complex boards.

By leveraging Multi-Agent systems, the robots can be treated as individual agents with the capability to learn and adapt to the game environment. This method could enable the agents to coordinate their movements more strategically than static algorithms. Techniques such as cooperative learning, where agents share information and learn from each others, could be very beneficial.

In addition, leveraging Deep Learning (DL) presents another valuable improvement strategy. By training neural networks on game data, DL can uncover effective movement strategies and enhance decision-making. This method could work alongside multi-agent approaches, enriching the AI's capability to tackle complex puzzles with more sophisticated insights.