Danmarks Tekniske Universitet

DTU

Board Game Assignment - Report: Ricochet Robots

02180 - Introduction to Artificial Intelligence

Authors

Andrei Farcas - s242999
Lorenz Helde - s250008
Oliver Lamine Thiam - s250017

March 24, 2025

Contents

1 Introduction

The project implements a Riccochet Robots puzzle game where a human player interacts with a graphical interface to move robots towards a target place through a maze. The goal of the game is to achieve this in as few moves as possible. The board game is a 16x16 grid that contains squares which can be occupied by robots or walls. The game allows the user to either solve the puzzle manually or use an AI to find the optimal solution. The puzzles are automatically generated with random robot and walls placements.

The goal of this report is to document the game mechanics, computational complexity, and AI implementation using A* search. The report will also analyze heuristic design, performance, and potential improvements.

2 Game Description

2.1 Rules of the Game

The game consists of four robots (red, green, blue and yellow) that are placed at random positions on a 16x16 grid. The board has walls that create a maze-like structure, constraining the freedom of the robots. Also at random, a target of a certain color is chosen, and the matching robot must reach the position of the target. To do so, any robot can be moved in straight lines until it hits an obstacle. The game is solved when the correct robot stops on the target position, and this must be achieved in a minimal number of moves.

No significant modifications of the rules of the game (as we have found them) was made. The only change is that we have considered the AI to be used as a tool that can provide (and play) the optimal solution of any generated game at the request of the users.

2.2 Game Type and Implications

The game is considered to be either a single player (through individual puzzle solving) or a multi-player (multiple users can use the interface to try and solve the puzzle before allowing the AI to do it). However, from the AI's perspective, this is a puzzle-solving task in which all the states are fully observable. The moves do not have any random components so the game is deterministic, making it suitable for solving with search-based algorithms like A* and Breadth-first search (BFS) algorithm.

2.3 State Space Complexity

The state space complexity depends on the board size, the number of robots, their degrees of freedom and walls that are restricting some positions. A rough estimate of the upper bound of states for a game is obtained considering 16x16 = 256 positions per robot, resulting in a size of $256^4 = 2^{32} = 4.3 * 10^9$. This size suggests that brute-force approaches like an exhaustive BFS search are infeasible because they would require too many resources

for some puzzles. Instead, we have decided to use a heuristic function to efficiently choose the states that are explored.

3 Game Representation and AI approach

3.1 Game Representation

3.1.1 State Representation

Each state needs to fully describe a certain point of a game and therefore contains the following elements: a 16x16 board represented as a 2D grid, considered common for all possible states of the same game, where each cell can contain walls that restrict movement, four robots, and a target containing one of the colors of the robots. All of these elements are stored in dictionaries.

Each robot is assigned a unique color and is represented by a dictionary entry that stores its current position as a tuple of row and column. A similar approach is used for the target. In order to represent the "quality" of the solutions, we need to keep a move counter that is just a variable that gets incremented after every move.

Each state in the Ricochet Robots game contains the following components, structured using efficient data structures:

- **Board**: A 16×16 grid implemented as a 2D array of dictionaries. Each cell dictionary contains a `walls` set specifying which directions (N, S, E, W) have walls:

```
board[x][y] = {"walls": {"N", "W"}} # Cell with N and W walls
```

- **Robots**: A dictionary mapping each robot's color to its position and selection state:

```
robots = {
    "red": {"pos": (3, 5), "selected": False},
    "green": {"pos": (12, 7), "selected": False},
    ...
}
```

- **Target**: A dictionary containing the color and position of the current target:

```
target = {"color": "red", "pos": (8, 10)}
```

For state representation during search, the positions of all robots are converted into a hashable structure using a tuple of sorted (color, position) pairs:

$$\text{state\_key} = \text{tuple}(\text{sorted}(\{(\text{color}, \text{position}) \text{ for all robots}\})) \quad (1)$$

This approach ensures efficient duplicate detection in the search's visited set, which is critical since the Ricochet Robots state space can grow exponentially.

The chosen data structures are a good option for efficient collision detection, movement validation, and state expansion during the AI search process. The dictionary based storage of robot positions enables fast lookup and modification, while the 2D list of walls ensures quick boundary checks. This structured representation allows the game logic to operate efficiently while also supporting our heuristic-based search algorithm.

3.1.2 State Generation

The state space is explored by generating successor states through robot movements by a calls of $\_move\_robot(self, positions, color, direction)$ function. Each move follows these rules:

1. A robot can move in four cardinal directions (N, S, E, W)
2. Once movement begins, the robot continues in a straight line until it encounters a wall, another robot or the edge of the board.
3. A move is considered valid only if it changes the robot's position

For each state in the search frontier, all possible moves for all robots are considered, generating up to 16 successor states (4 directions $\times$ 4 robots). The state generation function implements the rules of the game, ensuring that robots slide until stopped and updating positions.

This state representation and generation approach efficiently encapsulates the game mechanics while supporting the heuristic search algorithms used to solve the puzzles.

3.2 AI Algorithm: Breadth First Search (BFS)

Breadth-First Search (BFS) is an uninformed search algorithm that explores all possible states in increasing order of moves. This guarantees that the first solution found is optimal in terms of move count.

In our Ricochet Robots solver, we implemented BFS as a baseline method. The algorithm starts from the initial state and iteratively expands all possible successor states by generating all valid moves in four directions for each robot. New states are stored in a queue to be processed in the order they were discovered.

However, in practice, BFS exhibited significant scalability issues:

1. **Long computation times**

   Since BFS does not use heuristics, it searches the entire state space uniformly. Due to the exponential growth of possible states, this leads to excessive computation times for more complex puzzles.

2. **Limited to 20 moves**

   To keep computations feasible, we imposed a 20-move limit on the BFS solution. This restriction prevents BFS from solving more difficult puzzles, further limiting its scalability.

3. **High memory usage**

   BFS stores all visited states to avoid redundant calculations. As the state space grows, memory consumption becomes a bottleneck.

Due to these limitations, we found BFS to be impractical for solving Ricochet Robots efficiently. Instead, it served as a benchmark for comparison with the A* search algorithm, which, by leveraging heuristics, significantly improves search efficiency and resource usage.

3.3 AI Algorithm: A* Search

A* Search is a widely used algorithm for path finding and graph traversal, leveraging both cost-so-far (g-cost) and estimated cost-to-goal (h-cost) to efficiently find optimal solutions. In Ricochet Robots, A* is applied to determine the shortest sequence of moves that guides a designated robot to the target position.

A* operates by exploring states based on the function:

$$f(s) = g(s) + h(s) \tag{2}$$

where:
$g(s)$ represents the number of moves taken to reach state;
$h(s)$ is a heuristic estimate of the remaining moves to reach the goal;
$f(s)$ is the total estimated cost of reaching the goal.

3.4 Heuristic Design

An good heuristic design is critical for A*'s efficiency. The first heuristic function used is the Manhattan distance (sum of horizontal and vertical distances) between the target robot's position and the target position:

$$h(s) = |x_{\text{robot}} - x_{\text{target}}| + |y_{\text{robot}} - y_{\text{target}}| \tag{3}$$

This heuristic is admissible because the Manhattan distance is always less than or equal to the actual number of moves required, as it ignores walls and movement constraints. Also, it is consistent because the difference in Manhattan distance between adjacent states is at most 1, which is the cost of a single move.

The second solver uses a more informed heuristic based on pre-computed reachability. It performs a BFS search from the target position to generate a map of all positions reachable by direct robot slides, considering walls but ignoring other robots. For each position in the reachability map, the stored value represents the minimum number of robot slides needed to reach the target from that position. For positions not in the map (which occurs frequently

due to the fact that robots move until they meet an obstacle), the algorithm simply uses the Manhattan distance.

This reachability map is generated once at initialization. It is better than the Manhattan distance because it takes into consideration the fact that the robots move until they meet an obstacle and also considers the walls.

This new heuristic is also admissible because it calculates the minimum number of moves required considering walls but ignoring other robots, which can only increase (never decrease) the actual cost. The heuristic also accounts for walls, maintaining the triangle inequality property.

4 Performance and Evaluation

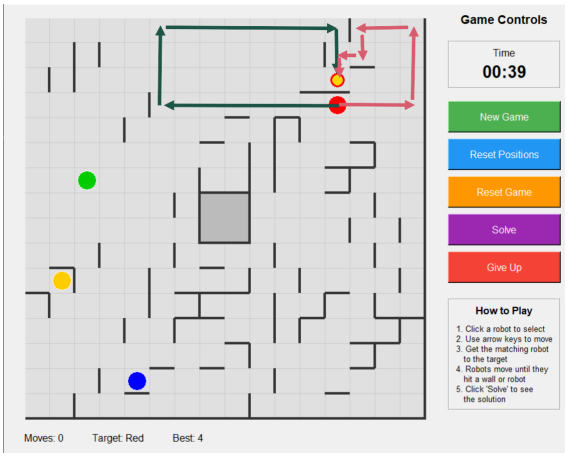

Figure 1: A* solution with Manhattan Distance vs Ideal Solution

While simple to compute, the Manhattan distance does not consider walls on the board or the specific movement rules of Ricochet Robots (robots move in straight lines until they hit an obstacle). This leads to optimistic estimates that may significantly underestimate the actual number of moves required, or it may force the algorithm to ignore a solution that requires the robot to move away from the target in order to get there in a smaller number of moves.

An example of such a case that was identified during our testing is presented in figure 1. The green arrows show the optimal way in which the robot can reach the target in 4 moves, which would be found by a full BFS search. However, the Manhattan distance heuristic influenced the decision of the algorithm to explore the solution shown by the pink arrows, which never gets far from the target but requires 2 extra moves.

Consequently, a new solver with an updated heuristic has been developed, that can perform better for cases like the one presented above. This heuristic implies computing a reachability matrix during the initialization of the class that implements the algorithm. This approach may add a short delay at the beginning, but it enhances the search phase of the algorithm and overall can be expected to find a solution faster.

5 Future Improvements and Conclusions

5.1 Future Improvements

1. **Hybrid Search Approach**

   Combining BFS and A* dynamically, depending on the complexity of the current state, could improve efficiency. Introducing iterative deepening A* (IDA*) could reduce memory consumption while still benefiting from heuristic-based search.

2. **Enhanced Heuristic Functions**

   Further refinement of the heuristic function could improve AI efficiency. A potential approach is incorporating machine learning techniques to dynamically adjust heuristics based on previously solved puzzles. Another possibility is using domain-specific heuristics that take into account common movement patterns observed in solved games.

3. **Graphical interface and user experience improvements**

   Enhancing the graphical interface with visualization of the AI decision-making process could help users understand the algorithm better. Adding step-by-step hints or a solution preview option could make the game more interactive and educational.

5.2 Conclusion

The implementation of Ricochet Robots using AI techniques has demonstrated the effectiveness of heuristic search algorithms in solving complex puzzles efficiently. The project highlights several key findings:

1. **Heuristic Selection Matters**

   The choice of heuristic greatly impacts the performance of the A* algorithm. While the Manhattan distance heuristic is simple, it sometimes leads to suboptimal solutions. The reachability-based heuristic improves accuracy and efficiency.

2. **Trade-offs Between BFS and A***

   While BFS guarantees finding the shortest solution, its exponential growth in state space makes it impractical for large boards. A* significantly reduces the search space by guiding exploration towards the goal.

3. **Scalability Challenges**

   The AI approach works well for moderately complex puzzles but struggles with very large search spaces. Future improvements, such as parallelization and hybrid approaches, could enhance scalability.

Overall, this project successfully demonstrates how AI search algorithms can be applied to real-world puzzle-solving scenarios. With further refinements, the system could be expanded to handle more complex puzzles and provide an even more intelligent and interactive experience for users.