Danmarks Tekniske Universitet

DTU

BOARD GAME ASSIGNMENT

Authors

Chang Hui Simone Lin - s232963
Lucas Emil Borlund Orellana - s234122
Dimitrios Tsogias - s233536
Eleni Papageorgiou - s240148

March 18, 2024

Contents

1 Introduction

The purpose of this report is to provide a thorough examination of our efforts in implementing an AI player for the board game Ricochet Robots. Throughout this report, we delve into various aspects of our implementation strategy, including the intricacies of the game rules, the analysis of its mechanics, the estimation of its state space, the detailed description of game elements, the representation of states and moves, the selection and justification of AI methods and algorithms, the development of heuristics and evaluation functions, the meticulous parameter adjustment and bench-marking process, and the identification of potential areas for future work and improvement.

2 Game Map, Rules and Elements

Ricochet Robots unfolds on a square grid 16x16 board populated with obstacles, targets, and maneuverable robots. Players control these robots, each distinguished by a unique color, and take turns moving them across the board. The primary objective is to guide a specific robot to its corresponding target square in the fewest possible moves. Robots move in straight lines until they encounter an obstacle, another robot, the board's edge or walls. The strategic placement and utilization of mirrored walls play a pivotal role in optimizing movement trajectories, adding layers of complexity and intrigue to the gameplay experience. Briefly the rules are explained below:

- Initial State (s_0): The game board setup with robots and targets.
- Players: Multiple players taking turns to move robots.
- Actions: Players can choose to move a robot in any direction until it hits a wall or another robot.
- Results: The new state of the game after a player's move.
- Terminal-Test: The game ends when a player successfully navigates a robot to the target.
- Utility: Players aim to minimize the number of moves required to reach the target.

For the sake of simplification, we utilized a common target shared by all robots. Specifically, the target represents a singular entity with a distinct color separate from that of the robots. Consequently, all robots have the ability to reach this target square, regardless of their individual colors. An indicative game state and it's solution is shown below:

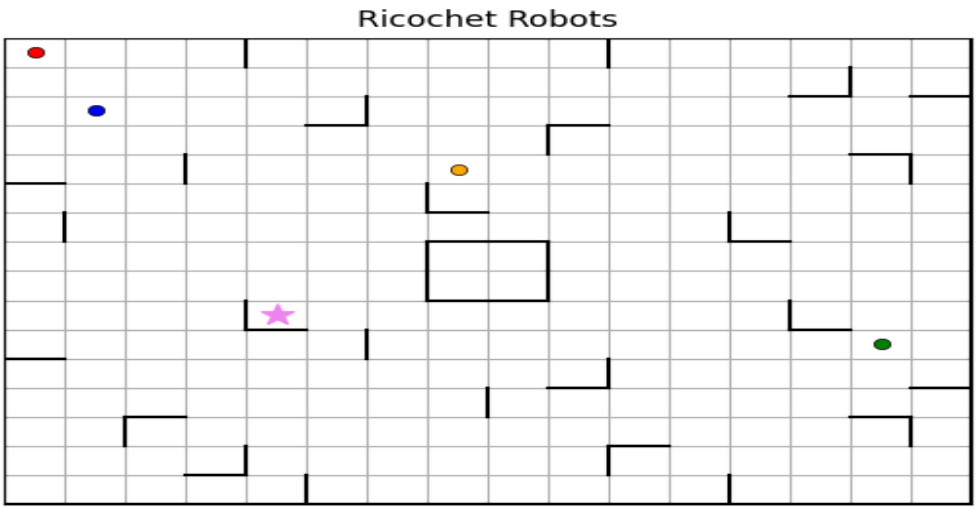

Figure 1: Indicative position of target and robots for an initial game state s_0

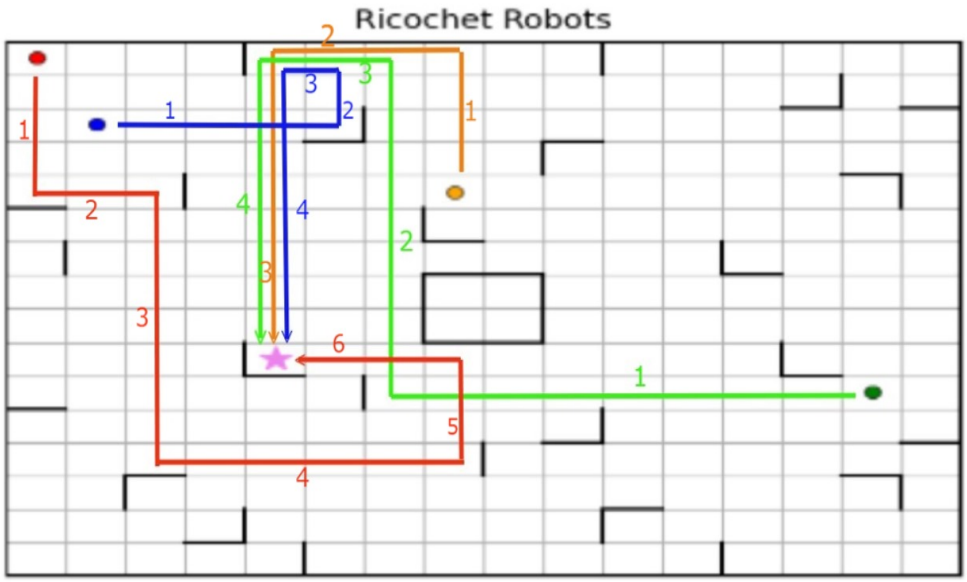

Figure 2: Indicative solutions for the initial game state of figure 1

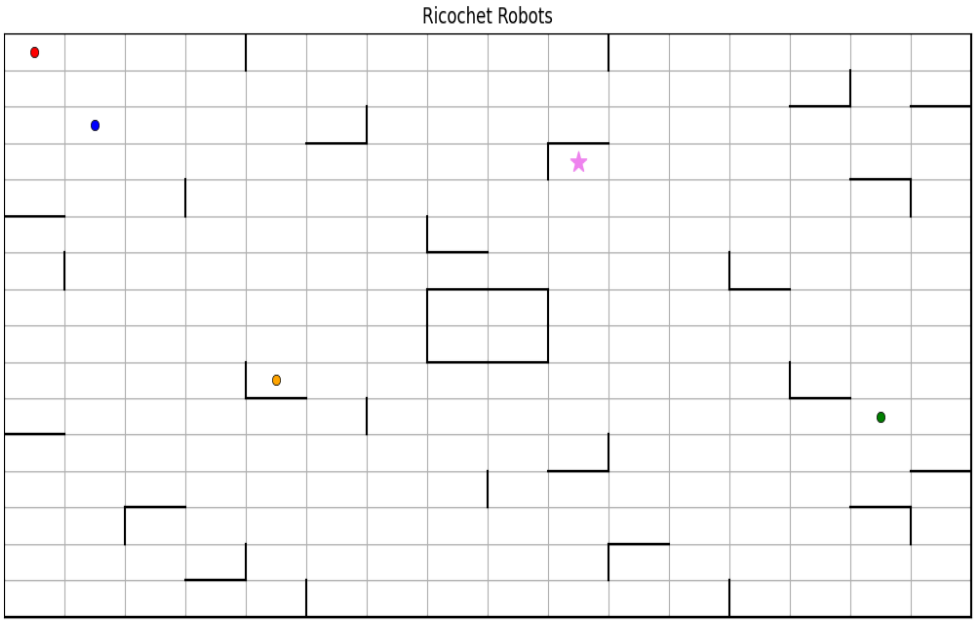

Figure 3: New initial game state s'_0

As derived from figure 2 the solution of the initial game state (s_0) depicted in figure 1 is the yellow robot with 3 moves, which is the optimal solution. In contrast with the yellow robot, all the other one require more than 3 moves in order to reach the target square. In addition, figure 3 illustrates the new initial game state (s'_0).

Richochet robots is a competitive, multi player, turn-based, game. In our implementation, however, the ai plays alone, controlling only a single robot at a time, and is therefore only "competing" with obtaining as good a possible solution for the given position. It is a perfect information game, with full observability of both the goal, board obstacles, and other players. As there are no elements of randomness, it is a deterministic game, especially when considering only one robot, as we don't have to consider other robots movements. Considering the fact that it's a perfect information game, search algorithms that utilize this fact, like heuristic functions, will be a great fit for this particular game.

3 Size of the State Space of the game

Lower Bound Estimate:

- Each of the four robots can be in one of the 256 squares on the 16x16 grid.
- The target location can also be in one of these 256 squares.

- Considering these, the lower bound estimate for the number of reachable states from an initial state would be $256^4 \times 256$, which is approximately 4.3×10^{11}.

Upper Bound Estimate:

- The upper bound considers all possible arrangements of robots and the target on the board.
- For each of the 256 squares, there can be a robot or the target, giving 2^{256} possibilities.
- This results in a total upper bound estimate of approximately 1.16×10^{77}.

The actual number of reachable states is likely to fall between these bounds, closer to the lower bound.

Implications for Algorithms:

- The estimated state space size suggests that traditional brute-force search algorithms might be feasible, albeit on the upper end of computational complexity.
- Algorithms such as GreedyBFS, A* or DFS search would be suitable for efficiently exploring the state space and finding optimal solutions.
- Techniques like pruning and symmetry detection can further optimize search algorithms in navigating the state space effectively.

4 States and Moves represented in the computer

Representation of States:

- Each game state can be represented as a snapshot of the board configuration at a particular point in time.
- For a 16x16 board with four robots and one target location, the state representation includes the positions of each robot and the location of the target on the grid.
- This information can be stored in a data structure such as a 2D array or a matrix, where each cell represents a square on the grid, and the presence of a robot or the target is indicated by the value stored in the cell.

Representation of Moves:

- Moves are represented as actions that can be taken by the player to move the robots on the board.
- Each move specifies the direction (up, down, left, or right) in which a particular robot should move.
- These moves can be encoded as commands or instructions that manipulate the positions of the robots on the grid.

Belief States:

- In the context of AI for Ricochet Robots, belief states may not be directly applicable since the game is deterministic and fully observable.
- Belief states are typically used in games with uncertainty or partial observability, where the exact state of the game is not known to the player or the AI agent.
- Since Ricochet Robots involves deterministic moves and the entire board state is visible to the player, representing states as belief states is not necessary for this game.

However, it's crucial to deliberate about the DFS approach concerning belief states. DFS (Depth-First Search) is an uninformed search strategy that explores as far as possible along each branch before backtracking. This approach might not be suitable for problems where uncertainty or partial observability is involved, such as those requiring belief states.

Moreover, it is important to highlight the differences between A* (A-star) and GBFS (Greedy Best-First Search). The discrepancy lies in the g(n) function, where one utilizes distance, and the other employs the number of moves. Both algorithms maintain values iteratively in the map and store the f function values in the map for efficient traversal and pathfinding. In addition, it is worth mentioning that the heuristic function in the implementation of Greedy BFS is developed according purely to the Manhattan distance in contrast with A_star that a cost function is also considered.

Below are presented the methods used for this assignment and the implementation of them in the computer:

1. **Greedy Best-First Search (GreedyBFS)**:

– In this algorithm, the map array is used to mark visited nodes. When a node is visited, its corresponding entry in the map array is set to -1.

– The map array is also utilized to store the heuristic values (i.e., estimated cost from the current node to the goal). If a node has not been visited yet, its entry in the map array represents the heuristic value.

– For unvisited nodes, the heuristic value is initialized to `np.inf` (infinity). This is done during initialization of the map array.

– By setting a node's value to -1, the algorithm knows not to visit it again during the search process.

2. **A-Star Search (A_star)**:

– Similar to GreedyBFS, the map array is used to mark visited nodes and store heuristic values.

– However, in A-star search, the map array also stores the actual cost from the initial state (`s0`) to each visited node. This cost is updated dynamically during the search process.

- The entries in the map array for unvisited nodes are initialized to `np.inf`.
- Additionally, in A-star search, the map array is used to store the total cost (`g-value` + `h-value`) associated with each visited node.

3. **Depth-First Search (DFS)**:

- In DFS, the map array serves a simpler purpose compared to GreedyBFS and A-star search.
- It is primarily used to mark visited nodes. When a node is visited, its corresponding entry in the map array is set to -1.
- The DFS algorithm does not utilize heuristic or cost values like A-star search, so the map array is used solely for tracking visited nodes.

An example of the map array with the nodes used in the algorithms is shown below (-1 = visited nodes, ∞ = not visited):

$$\begin{bmatrix}
-1. & \infty & \infty & \infty & -1. & -1. & \infty & \infty & \infty & \infty & -1. & \infty & \infty & \infty & \infty & \infty \\
-1. & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & -1. & \infty & \infty & -1. & \infty & \infty & \infty \\
-1. & \infty & -1. & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & -1. & \infty & -1. & -1. \\
-1. & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & -1. & \infty & \infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & -1. & \infty & \infty & \infty & \infty & \infty & \infty & \infty & -1. & \infty & -1. \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
-1. & \infty & \infty & -1. & \infty & \infty & -1. & \infty & \infty & -1. & \infty & \infty & -1. & \infty & \infty & -1. \\
\infty & \infty & \infty & \infty & -1. & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & -1. & -1. & \infty \\
\infty & \infty & -1. & \infty & -1. & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & -1. \\
-1. & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & -1. \\
-1. & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & -1. \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & -1. & \infty & -1. & -1. \\
-1. & \infty & \infty & -1. & -1. & -1. & \infty & \infty & \infty & \infty & \infty & \infty & -1. & -1. & -1. & -1.
\end{bmatrix}$$

In summary, the map array in each algorithm is used to keep track of visited nodes and, in the case of A-star search, to store heuristic and cost values. The value -1 is used to indicate visited nodes, and `np.inf` represents uninitialized or unvisited nodes.

5 Methods and Algorithms

Search Algorithms: Search algorithms, such as Breadth-First Search (BFS), Depth-First Search (DFS), and A* Search, can be employed to explore the state space and find

optimal or near-optimal solutions. These algorithms traverse the search tree or graph to find paths from the initial state to the goal state.

Heuristic-Based Methods: Heuristic-based methods, particularly A* Search with an appropriate heuristic function, are commonly used for solving puzzles like Ricochet Robots. A* Search combines the advantages of both BFS and heuristic search, efficiently exploring the search space while guiding the search towards promising paths using a heuristic evaluation function.

Chosen Algorithm: Among these methods A* is the best choice for Ricochet Robots. That algorithm efficiently explores the search space while guiding the search towards solutions that are likely to lead to the target in minimal moves.

Illustrative Examples: Consider a simplified version of Ricochet Robots. With A* Search, the algorithm starts from the initial state and expands nodes in the search tree based on their estimated cost-to-go (heuristic value) plus the cost-so-far (number of moves). Nodes with lower total costs are explored first. The search progresses until a node representing a state where the target is reached is encountered. At each step, the algorithm dynamically adjusts its exploration strategy based on the heuristic guidance, efficiently navigating the search space towards the goal state. Whilst, the Greedy DFS starts at the initial state and explores as far as possible along each branch before backtracking and prioritize paths that appear promising based on a heuristic, such as moving towards the target location without fully considering the consequences. While this approach can sometimes find quick solutions, it may overlook more optimal paths by favoring immediate gains. Finally, Greedy BFS expands nodes level by level, prioritizing nodes with the lowest heuristic values. In Ricochet Robots, Greedy BFS explores all possible moves from the initial state, selecting the ones that seem closest to the target based on the heuristic. This method ensures that shorter paths are explored first, potentially leading to faster convergence towards the goal state compared to Greedy DFS.

6 Heuristics or evaluation functions

Heuristic Function for A* Search:

- A heuristic function estimates the cost from a given state to the goal state. In the case of Ricochet Robots, an appropriate heuristic should estimate the minimum number of moves required for a robot to reach the target from its current position.
- One commonly used heuristic for this purpose is the Manhattan distance heuristic, denoted as $h(s)$, which calculates the sum of the horizontal and vertical distances between the robot's position and the target's position.
- Mathematically, the Manhattan distance heuristic can be defined as: $h(s) = |x_{\text{robot}} - x_{\text{target}}| + |y_{\text{robot}} - y_{\text{target}}|$, where x_{robot} and y_{robot} are the coordinates of the robot's position, and x_{target} and y_{target} are the coordinates of the target's position.
- This heuristic provides a lower bound on the actual cost to reach the target, as it assumes that the robot can move directly towards the target without any obstacles.

Evaluation Function for State Evaluation:

- In addition to the heuristic function used by A* for guiding the search, an evaluation function can be employed to assess the quality of states encountered during the search process.
- For Ricochet Robots, the evaluation function $f(s)$ can be a combination of the cost incurred so far (the number of moves taken to reach the current state) and the estimated cost to reach the goal state (provided by the heuristic function).
- Mathematically, the evaluation function can be defined as: $f(s) = g(s) + h(s)$ where $g(s) = |x_{robot} - x_{start}| + |y_{robot} - y_{start}|$, represents the cost incurred so far to reach state s, and $h(s)$ is the heuristic estimate of the cost-to-go to reach the goal state from s.
- A* search uses this evaluation function to prioritize exploring states with lower total costs (lower $f(s)$ values), effectively guiding the search towards the goal state while considering the cost incurred so far.

7 Adjustable Parameters and Benchmarking

1. **Heuristic Function Parameters:** The choice of heuristic function and its parameters can significantly impact the efficiency and effectiveness of the AI. Experimenting with different heuristic functions can provide insights into which ones perform best for the given problem. Furthermore, since A* has the consistency property, the heuristic is crucial to output the best possible result
2. **Search Algorithm Parameters:** Parameters related to search algorithms, such as A* search, can be adjusted to fine-tune the behavior of the AI. These parameters may include the search depth, the weight given to the heuristic function, or any additional optimization techniques employed within the search algorithm.
3. **Evaluation Function Parameters:** If the AI employs an evaluation function to assess the quality of states, parameters related to the evaluation function, such as the weighting of the heuristic component versus the actual cost incurred, can be adjusted to influence the AI's decision-making process.
4. **Algorithmic Parameters:** Parameters specific to the algorithms used, such as the branching factor or pruning thresholds, can also be adjusted to optimize performance and efficiency.

Experimentation with different combinations of these parameters, as well as with various algorithms, search depths, heuristics, and evaluation functions, can provide valuable insights into the strengths and weaknesses of the AI. Benchmarking these different versions against each other can help identify the most effective approach for solving Ricochet Robots efficiently.

8 Solutions of Algorithms

In this section statistics of 100 different randomly generated game states and the solutions of the algorithms used are presented.

Algorithm	**Accuracy**	**Moves (mean value)**	**Time (Approx.)**
Greedy BFS	85%	12	9.8 ms
A*	85%	9	15 ms
DFS	70%	23	10.7 ms

Table 1: Performance Comparison of Algorithms

- **Accuracy:** It expresses if the algorithm managed to get to the solution. GBFS and A* have admissibility, therefore should be at 100%. However, some states are not reachable, therefore it's lower, but still higher than DFS.
- **Moves:** Expresses the number of moves each robot made until it reached the goal. A* is cost optimal, therefore the number of moves is the lowest, while GBFS might go off-topic since it doesn't take into account the cost from the start to the current position of the robot.
- **Time:** The time from the algorithm is started, to a solution is found, or the game is aborted. As we can see the A* is slower, due to being more computationally heavy ($(O(b^m))$), as it computes it's f-function for each searched field, as well as stores this value. The DFS is faster $O(b^d)$, however, is an algorithm which is very dependant on the starting conditions, and lucky/easy positions could skew the results. Greeedy BFS is somewhat of a middle ground, as it requires less computation than A*.

9 Future work

Our solutions can be improved in various ways:

1. The data structures used can be made more optimized, and capable of handling edge cases for example by implementing nodes as classes.
2. More efficiency is possible by using more the NumPy library, since it's written in C and, therefore, faster computationally speaking compared to Python.
3. In order to make the AI multiplayer compatible, adversial search algorithms can be used to make the robots compete to get to the goal. Furthermore, cooperation could be implemented in the early stages of the game, decreasing exponentially the unreachable states.

List of Figures

References

Introduction to Game AI 1st Edition by Neil Kirby , Chapters 1-6,pg. 10-113

Russell and Norvig: Artificial Intelligence – A Modern Approach. 4th edition. Chapters 3-5

https://boardgamegeek.com/boardgame/51/ricochet-robots