Project AI in Games: Ricochet Robots

Linus Diepold
902911

1 Introduction

Ricochet Robots is a single-agent board game, that is suited to motivate and illustrate the use of artificial intelligence in games. The original board game is a multiplayer game. Every turn the players would be presented with a task that involves moving robots to a reach a goal. After a certain time each player has to present the best solution he could find. The solution that require the least amount of moves wins. Since there are a limit amounts of possibilities for robots to move creating computer generated solutions via search algorithm provides an intuitive approach. There are other approaches as well [3] however the following work will focus on creating and optimizing search algorithms. A generic search algorithm is able to find optimal solutions but requires a search tree that grows exponentially. Engels Et. Al. [2] indeed show that Ricochet Robots is NP-hard. Since there is a time limit, blind search can become insufficient for a human player as well as for computer generated solutions. In the same way humans develop strategies to guide their search to fit the time limit, we can use techniques like heuristic values and search space reduction to guide searching algorithms [1]. Applying such strategies can result in sub-optimal solutions as a trade off for faster computation. In the following work we search for algorithms that minimize the amount of used moves while being restricted in computation time.

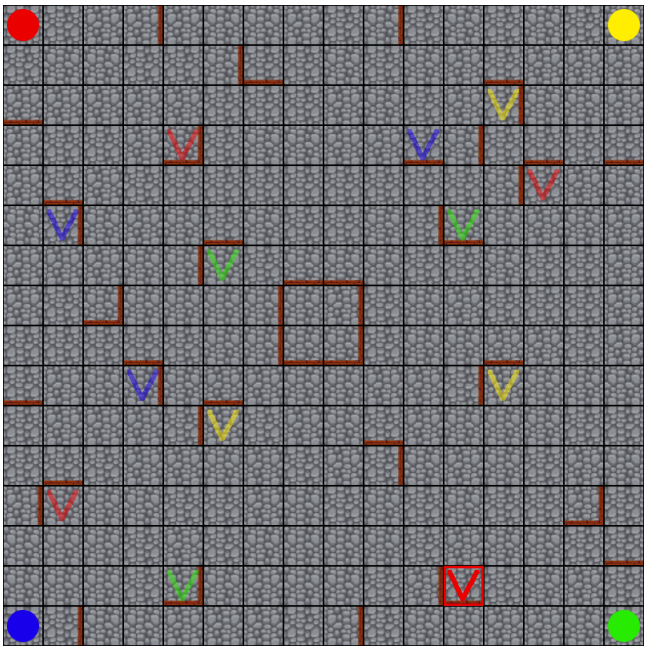


Figure 1: A ricochet robot board

2 Game

As depicted in Figure 1 a ricochet robot board contains 16x16 fields. On four of these fields are robots in four different colors (red, green, yellow and blue). At the beginning of the Game these robots are located at the corners of the board as shown in **??**. The board contains some obstacles between those fields where the robots can not pass. Each color has some spawnpoints for victorypoints. At the beginning of each round *one* of the spawnpoints(randomly selected) will turn into a victorypoint. The color of the victorypoint is equal to the color of the spawnpoint and dictates which robot has to reach that point. After revealing the victorypoint each of the players will search for a sequence of moves to reach the victorypoint with the corresponding robot. With one move a robot can go in every direction until they hit an obstacle, another robot or the end of the board. A robot can not stop otherwise. A solution is defined as a sequence of moves that will result in the victorypoint being reached by the robot of his respective color. The smaller the sequence the better the solution. A player can move robots of any color and therefore use other robots as artificial obstacles. Every move is weighted equally regardless of the color of the robot or the distance traveled. After a certain amount of time each player has to present the best solution that he could find. The player that presents the best solution will win the round and score a point. Afterwards a new victorypoint will be randomly selected, while the positions of the robots stay the same. This process creates

quite unique and unpredictable problems for the players to solve. If two players have solutions with equal amount of moves, the player who found his solution first will get the point (In the board game version people have to announce if they have a solution in mind). Therefore the quality of a solution can be measured by computation time and length of the sequence.

3 AI

As previously mentioned tree search algorithms provide an intuitive approach for solving ricochet robots. Tree search algorithms generally create a tree structure with every node containing a possible solution. Applied to ricochet robots nodes can represent sequences of moves. A nodes can be expanded by creating edges to other game states these will represent a specific robot move. Every tree originates at the root node that represents the initial game state with no moves made. Therefore the depth of the tree structure is equal to the amount of moves used. Theoretically in every position there are 16 possible moves(16=4 robots * 4 possible moves per robot), resulting in a quite large branching factor. In practise there are 8-12 possible moves since robots that are already besides a wall cant move in the direction of this wall. Whether a node is a solution can be easily verified in polynomial time. The difficulty of a ricochet robot round is determined by the length of the solutions. With growing length of the solutions, the amount of nodes inside of the search tree grow exponentially. Therefore finding an optimal solution in polynomial time with search trees is not unlikely. Worth mentioning is that every game has a solution and most games are solved in less then 15 moves. The time limit in the board game is about 1 minute. This leaves the tree search with the challenge of expanding a approx. maximum of $12^{15} = 15.4 * 10^{18}$ within a minute.

3.1 Search space vs complexity

Since there is no theoretical limit to the length of solutions(as the robots can use an infinite amounts of moves) the search space is infinite. As a consequence there are also an infinite number of solutions to a round of ricochet robots. In order to win a round rather than finding every possible solution, only one optimal solution needs to found. A solution is considered to be optimal if and only if every other solution either needs a similar amount or more moves. The amount of optimal solutions is limited and a ricochet robot algorithm can be considered complete if it will find every optimal solution. Reducing the search space (e.g. searching for solutions with a max of 15 moves) will violate the completeness of the algorithm, yet is a necessary step to reach acceptable computation times. Therefore strategies are needed that provide a reasonable trade off between the reduction of search space and loss of optimal solutions.

3.2 Basic tree search algorithms

Tree search algorithms in general expand and verify nodes within a tree structure. The challenge is to find the best nodes to expand/verify first in order to quickly find good solutions. The two most basic strategies are: breadth/depth first search. Depth first search will expand the node with the highest length first. Applied to ricochet robots this algorithm will consider long solutions first. Since there is no theoretical limit to the amount of moves this strategy requires a depth limit to properly traverse the whole tree. This strategy seems not fitting at all since we prefer to find smaller solutions and traversing the whole tree(even with a depth limit) is hard to achieve. Breadth first search will expand all nodes at the current depth prior to moving to the next depth level. This strategy does require more memory as depth first search but has the huge upside of finding (guaranteed) optimal solutions first. Because of this property the breadth first principle will be present in further discussed strategies.

3.3 Single color search

Every round the victorypoint has a fixed color that will be referred to as the victorycolor. The moves of the robot with the victorycolor is the most important one since his position determines whether the sequence is successful. By solely using moves of the robot with the victorycolor(= *victoryrobot*) we can already solve a lot of rounds. These solutions can be computed very efficiently since the branching factors is fairly low. Since the robot can not stop without a wall, every robot has at least one adjacent wall. This leaves at maximum 3 options, if the robot is in a corner there are only two. If the robot already made a move previously, there will be a move that counteract this move(e.g. if one robot moves right and left consecutively). These moves can be cut out reducing the branching factor to something between 1-2. Since the branching factor is small breadth first search can be used for optimal(w.r.t. the limitation of only using one color) solutions. In conclusion a search tree that solely contains moves of the victoryrobot is neither guaranteed to find a solution nor does it guarantee a solution to be optimal, however it can be computed very efficiently.

3.4 Precomputing setups

In some rounds a solution requires the use of multiple robots. Since robots with a non-victorycolor do not need to reach a specific field, their job is to create artificial obstacles for the victoryrobot to stop at strategic fields. In the vast majority of solutions the non-victoryrobots are moved before the victoryrobot to setup a path. In the following a *setup* is defined as a sequence of moves from non-victoryrobots. Those setups can be used as root node of a single color search. If a subsequent single color search is successful a setup can be considered a partial solution. Since a single color search can be done fairly efficient, verifying whether a setup is a partial solution can also be done efficiently. Verifying every possible setup is still a hard task since there are a huge amount of possible setups

depending on the length of the (optimal) solutions. Even if all possible setups will be considered the resulting algorithm will still be incomplete and optimal solutions can not be guaranteed. However it does significantly extend the search space and allows for more optimal solutions, by setting limits to the length of a setup and the length of the overall solution such a search can terminate in a reasonable time frame. A setup length is defined by the combined amount of moves from every robot except the victoryrobot. Since there are 3 robots to consider searching for a setup will result in an search tree with a fairly high branching factor(ca. 5-8). Therefore the depth limit for the setup search will be significantly lower than the depth limit of the single color search. Therefore a search using precomputed setups does yield a more complex algorithm than single color search that does also result in more and better solutions. A weakness of this strategy are situations where the victoryrobot has to execute moves before the setup. Such a situation is depicted in Figure 2 and 3. In Figure 2 is a situation where the red robot provides a setup for the blue robot to reach the victorypoint within 6 moves. In Figure 3 is a situation where the blue robot moves first so the green robot can provide a better setup than the red one, allowing the blue robot to reach the victorypoint within just 5 moves.

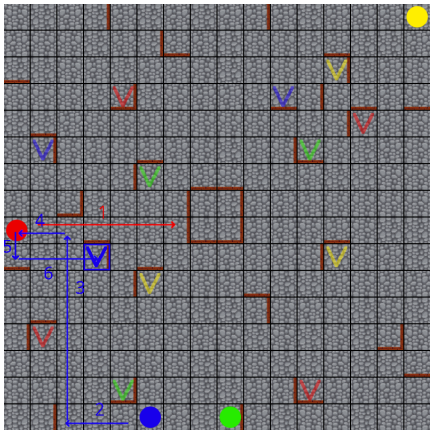


Figure 2: Example solution for a single color search with a setup which requires one extra move

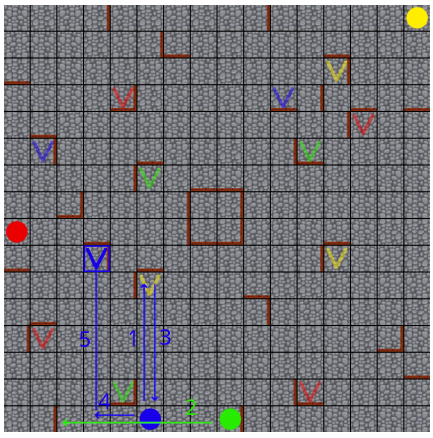


Figure 3: The optimal solution for the problem from figure 2

3.5 Critical positions

Since most of the computation time is spend with verifying setups an efficient way to reduce complexity is to apply heuristics to the setup search space. For this purpose a metric that describes the quality of a setup is needed. The length of the best solution that is based on that setup would be the most precise metric. Since this would require a full single color search a easier to compute heuristic is needed. For this purpose the *critical paths* are defined. A critical path consists of all fields that can directly(without help of another robot) access the victorypoint. Since there are a lot of such fields and some are closer from the victorypoint than others, those paths can be attributed with a *degree*. The degree of a field within a critical path is defined as the amount of moves required to reach the victorypoint starting from that field.

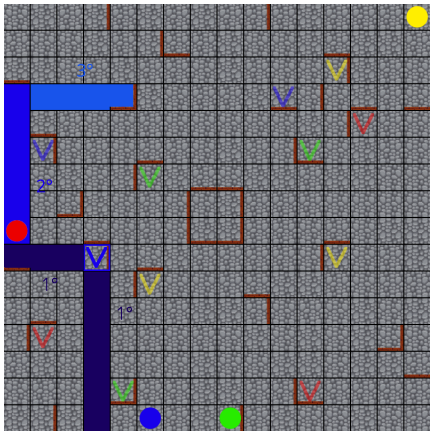


Figure 4: A visualization of the critical paths up to the third degree

The fields adjacent to the critical paths are called critical positions also attributed with the degree of the related path. If a non-victoryrobot occupies a critical position(in the context of a setup) it might allow the victoryrobot to stop on a critical path. Therefore the quality of a setup increases with the amount of critical positions that are occupied. The occupation of lower degree critical positions is also more valuable. Also setups with a lower amount of moves are preferred. This factors combined result in the *position score*.

$$\text{Position score} = \frac{\sum_{\text{Occupied critical positions}} \text{Critical degree}^{-1}}{|\text{Setup length}|}$$

By reducing the setup search space to setups with a position score greater than 0 and expanding the nodes with the best position score first we can massively reduce the computation time.

3.6 Cross block positions

Critical positions are important since their occupation might lead to the victoryrobot being allowed to stop on a critical path. Some critical positions are far more likely to fulfill this purpose. In order to find out which critical position are useful for the victoryrobot, it is important to find out where the victoryrobot can cross a critical path. This can be done efficiently during an initial single color search E.g. when the robot does cross a critical path the position that would block the cross is added to a list of cross block positions. A cross block position is defined by its position, the degree of the adjacent critical path and the amount of moves needed from the victoryrobot to make the cross. Equivalent to the position score a *cross block* score can be defined:

$$\text{Cross block score} = \text{Victoryrobot moves} + \text{Critical Degree} + \text{Setup moves}$$

In contrast to the position score every occupied cross block position has its own score with the lowest score, defining the score of the whole setup. In the case of no occupied cross block position the cross block score of the setup would be 0. If a setup contains a cross block position it is likely that there is a solution with an equal length to the cross block score. Reaching such a solution can be done in three steps. First the cross block setup needs to established requiring |Setup moves| amount of moves. Then the victoryrobot needs to repeat the moves from the single color search(which were used to determine the cross block position in the first place) requiring |Victoryrobot moves| amount of moves. After that the victoryrobot should be on the critical path besides the occupied cross block position. This might not always be the case since the single color search did not account for the changes that have been made by the setup. E.g. the robot that occupies the cross block position was needed to lead the victoryrobot to the cross position in the initial single color search. If the victoryrobot is indeed on a critical position besides the cross block position it can follow the critical path to the victory point requiring an amount of moves

equivalent to the Critical Degree. Therefore the Cross block score does provide a good estimate of the quality of the setup. Important to note is that every setup with a positive cross block score also has a positive position score. Therefore using cross block score instead of position score will result in a additional setup search space reduction.

3.7 Examples

The setup used in Figure 5 has a position score of $\frac{1}{4}$, since it covers one critical position of degree 1 and requires a setup length of 4.
The setup in Figure 5 also has a cross block score of 6. The first 4 moves are used for the setup. The 5th move is the victoryrobot move to reach the critical path. The final move follows the critical path with a degree of 1. Resulting in a solution with 6 moves.

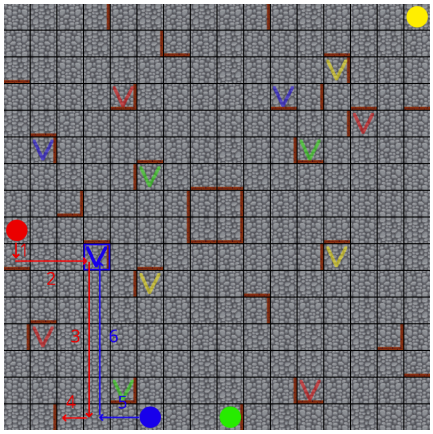


Figure 5: A solution that utilizes a setup with 4 moves with a position score of $\frac{1}{4}$ and a cross block score of 6

Figure 6 is an example of a setup with 0 cross block score and a positive position score. Therefore algorithms that are using cross block score as search space reduction would not be able to compute the solution in Figure 6. This example shows some problems of using position score as search space reduction. The occupied critical position of the green robot does not help the victoryrobot at all yet is necessary to validate the setup(otherwise the position score would be 0 and the setup would be ignored). Also the red robot(before applying the setup) already is located on a critical position but needs to be moved away to make space for the victoryrobot. Therefore every setup that does not move the red robot has a positive position score and has to be computed. For those reasons the cross block score provides a refined search space reduction by eliminating such scenarios.

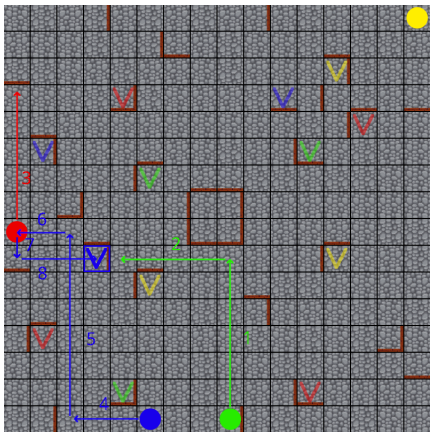


Figure 6: An example of a setup that has a positive position score but with 0 cross block score

4 Results

While the usage of search space reduction techniques will result in faster computation time there is still a trade off with solution quality. To put these factors into perspective different algorithms using those techniques have been tested. For those tests a standard 16*16 ricochet robot board with random victorypoints has been used. There are a few testing parameters that will vary the results and favor certain algorithms. The most important ones being the depth limits for both single-color search and setup search. Also computation time should be limited since it is limited in the original board game. Higher depth limit favors algorithms with higher search space reduction since they will be able to find solutions in the computation time limit. The following table features some of the collected results when executed in our implementations of those algorithms. These are the used algorithms:

- Algorithm 1 is a basic breadth first search.
- Algorithm 2 is a basic depth first search.
- Algorithm 3 is a breadth first search that utilizes precomputed setups.
- Algorithm 4 is a depth first search that utilizes precomputed setups.
- Algorithm 5 is a breadth first search that utilizes precomputed setups with positive position score
- Algorithm 6 is a breadth first search that utilizes precomputed setups with positive cross block score

The used Parameters are: Single color depth limit 10 with 40 tested rounds. The Board is initialized with the robots starting in their corner as depicted in Figure 1. The victorypoint will always be selected randomly. At the end of a round(if the victorypoint is reached or the algorithm exceeds the time limit) a new victorypoint will be determined. If the victorypoint is reached the new positions of the robots will serve as starting position for the next rounds. If the algortihm exceeds the time limit the robot positions will stay the same and a new victorypoint will be determined. For the setup depth limit and the time limit three different set of parameters are used.

- Parameter set 1: Setup depth limit = 4, time limit = 10s
- Parameter set 2: Setup depth limit = 5, time limit = 20s
- Parameter set 3: Setup depth limit = 6, time limit = 30s

Algorithm	Avg time	Avg moves	failed attempts	timeouts	solved
Algorithm 1	2.13 sec	4.166	0	34	6
Algorithm 2	1.82 sec	9.23	0	23	17
Algorithm 3	1.77 sec	6.147	2	4	34
Algorithm 4	0.69 sec	9.0	0	2	38
Algorithm 5	1.06 sec	6.27	3	1	36
Algorithm 6	0.03 sec	7	4	0	36

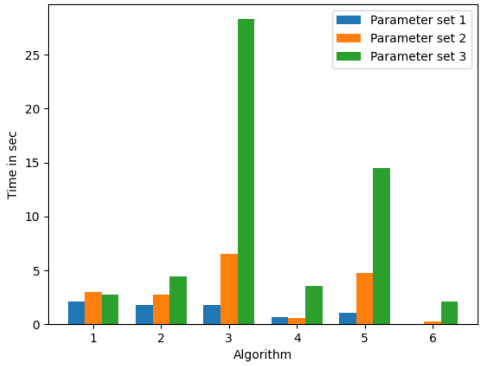
(a) Test results with parameter set 1

Algorithm	Avg time	Avg moves	failed attempts	timeouts	solved
Algorithm 1	2.98 sec	4.625	0	32	8
Algorithm 2	2.73 sec	9.23	0	23	17
Algorithm 3	6.56 sec	5.42	0	14	26
Algorithm 4	0.59 sec	9.0	0	3	37
Algorithm 5	4.77 sec	6.86	3	1	36
Algorithm 6	0.24 sec	7.97	4	0	36

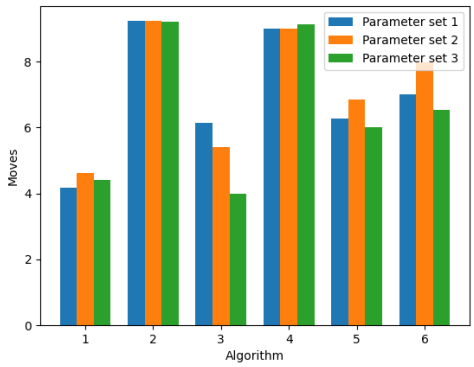
(b) Test results with parameter set 2

Algorithm	Avg time	Avg moves	failed attempts	timeouts	solved
Algorithm 1	2.76 sec	4.416	0	28	12
Algorithm 2	4.438 sec	9.22	0	18	22
Algorithm 3	28.3 sec	4	0	38	2
Algorithm 4	3.52 sec	9.13	0	3	37
Algorithm 5	14.5 sec	6	0	14	26
Algorithm 6	2.1 sec	6.527	4	0	36

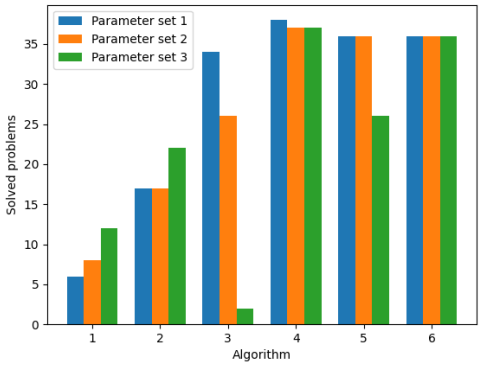
(c) Test results with parameter set 3



(a) Average computation time of the various algorithms



(b) Average moves needed of the various algorithms



l

(c) Amount of solved problems of the various algorithms

The testing data shows as expected a huge decrease in computation time with the usage of search space reduction. The basic breadth/depth first search algorithms struggle to solve problems within the time limit but have no failed attempts. This implies that these algorithms did never run out of search space. The data also shows that every step of search space reduction results in a increase of average moves needed. Therefore the data does depict the relation between search space reduction and the quality of output solutions. Also the depth first search algorithms in comparsion to the breadth first search algorithms show slightly lower computation time with the downside of drastically higher moves used.

5 Outlook

Finding an optimal solution in a round of ricochet robots is a difficult search problem since it is NP-Hard [2]. By limiting the size of the board, the length of the solution and with some efficient computing this can still be accomplished in a reasonable amount of time. Applying heuristics like cross block positions and position score can help to solve even harder problems by sacrificing the guarantee for optimal solutions. Depending on the problem at hand configuring parameters can further optimize computation time. Parameters like depth limits for the search tree or degree limits for critical paths can be changed. There are some further techniques for more efficient computations. E.g. cutting a search trees at nodes where the robot cant reach the goal within the depth limit. As an alternative approach the usage of neural networks might be useful to solve ricochet robots. But similar to the usage of search space reducing heuristics, neural networks are unlikely to provide optimal solutions. Our approaches yield decent results for boards of size 16x16 with solutions of lengths up to 15 moves. Solving harder problems can be done with different heuristics and more efficient computing.

References

[1] Nicolas Butko, Katharina A. Lehmann, and Verónica C. Ramenzoni. Ricochet robots-a case study for human complex problem solving. 2005.

[2] Birgit Engels and Tom Kamphans. Randolphs robot game is np-hard! *Electronic Notes in Discrete Mathematics*, 25:49–53, 2006. CTW2006 - Cologne-Twente Workshop on Graphs and Combinatorial Optimization.

[3] Filipe Gouveia, Pedro T. Monteiro, Vasco M. Manquinho, and Inês Lynce. Logic-based encodings for ricochet robots. In *EPIA*, 2017.