**Ricochet Robots Solver**

Mitch Powell and Daniel Tilgner

*University of North Carolina Wilmington*

**Abstract**

*Ricochet Robots is a German board game that is played on a 16x16 square grid. There will be one randomly assigned destination square and four to five randomly placed robots on the board depending on the version of the game being played. We decided to concentrate on versions of the game with four robots. One of these robots will be the target robot and the objective of the game is to get the target robot to the destination square in as few steps as possible. In a game, players are given 30 seconds to find the best solution that they can. A step consists of moving any one robot in any possible cardinal direction until it comes into contact with either a wall or another robot at which point it stops. Our goal was to develop an algorithm that returns the path that requires the least number of steps to complete the objective as quickly as possible. We developed a breadth-first-search algorithm that employed varying degrees of optimization as we progressed. Also used was a branch-and-bound algorithm. We worked on creating both an iterative deepening algorithm and an A\* algorithm, but ended up abandoning them for different reasons.*

**Key Words:**  Breadth-first-search, Adjacency Matrix, Linear Search, Binary Search, Hash Search, Branch-and-bound, Iterative Deepening, A\*

**1. Introduction**

We decided on this project after discussing various board games that we might be able to solve. Some were abandoned because of complexity (e.g. Letters From Whitechapel), others because of too much human interaction involved (e.g. Runewars), and some because of a high degree of random elements (e.g. Cosmic Encounter). We settled on Ricochet Robots because it was a game that took out the element of human interaction, always has a best solvable solution, and can be absolutely calculated and optimized regardless of initial starting conditions, yet was inherently complex enough that the design of an algorithm to solve it would not be trivial.

There are a few initial conditions that are important to understanding the problem. There are 96 different board configuration permutations, 17 different destination square possibilities, and different configurations of robots. This gives us 4,851,184,366,080 possible board configurations. We limited our scope to just one of the 96 board configurations as the setup has no real bearing on our algorithms, but saves us time in setting up trials.

We considered a number of different possible algorithms before deciding which to implement. Since this is an optimization problem, the first things we looked at were breadth-first search, depth-first search, branch and bound, and exhaustive search. Later on in the project we discovered more complex candidate solutions like iterative deepening and A\*.

**2. Formal Problem Statement**

Given a directed graph G = (V , E); let V be the set of all vertices contained in the graph, each representing a square on the board and E be the set of all edges connecting V where each vertex w contained in V can be defined as a two-tuple

w = (*Occ. , Targ.*), where *Occ.* is a temporal variable describing the occupation of the square and can be either a 1 describing an occupied square or a 0 describing an unoccupied square and *Targ.* describes the target-status of the square and is either a 0 representing that the square is not the target of the object robot or a 1 representing that the square is the target of the object robot. Given a target robot occupying a vertex v in V such that v = (1,0) and a target square t in V such that t = (0,1) let a function P(i,j) describe a number of edge traversals on a path between two squares i and j in V. Find a P’(v , t) such that P’(v , t) ≤ P (v , w) for all P.

A constraint on the problem is that there exists more than one occupied square w in V such that w = (1,0). The robots occupying each square w where w ≠ v may traverse their own paths under the condition that the traversal of such a robot from its original square w to another square t means that w = (0,0) and t = (1,0). This change makes t unreachable.

**3. Methods**

The algorithms that we employed involved the generation of a tree of potential board states, where each state represented a movement of any robot in any direction, and the root of the tree was the initial board configuration. The first algorithm design technique that we employed was a breadth-first search of this resulting state tree. A breadth-first search ensured that the very first solution that the algorithm encountered would be one of the minimal solutions to the problem. While there was a possibility of extremely long searches, this seemed like a reasonable approach to our problem.

The second algorithm design that we looked at employed a depth-first search. We decided that this would be impractical due to the staggering number of possible moves that can be made. A depth-first traversal of the state generated state tree would almost always involve traversal to depths with move counts much greater than that of an optimum solution, and there would be no way to prove that the solution was optimal without evaluating the remainder of the tree. Due to the large branching factor of the state tree, the number of states that would need to be evaluated at such a depth would be immense. Additionally, this solution would generate recursive state cycles where two robots would be moved back and forth alternately forever and the algorithm would never finish. Under such a construction, there would be no maximum depth to the state tree unless additional logic was provided to prevent such constructions from occurring.

Initially, using the branch-and-bound algorithm seemed like a promising alternative to the depth-first search. Finding an initial case and then traversing through each branch until we reached the best case depth was simple on the surface. The difficulty surfaced when we realized that there was not a simple way to find a low-value initial best case.

We also considered employing an iterative deepening version of the depth-first search, where the root node was evaluated, and if no solution was found, the algorithm would start again, evaluating all of the solutions up to a depth of one, and increasing the depth of the search until a solution was found. This proved to be very similar to our breadth-first search in theory, however, so we decided not to implement it. The complexity of the algorithms would be very similar, except for a tradeoff of the larger memory requirements of the breadth-first search for the repeatedly evaluated lower-depth nodes of the iterative-deepening, depth-first search.

The last algorithm that we looked at was the A\* algorithm, a kind of “best-first search” which was actually designed for path traversal to begin with. The A\* algorithm traverses the graph and builds up a tree of partial paths. The leaves are stored in a priority queue that uses a cost function to order them, and this is where we started to realize that it was not possible for our problem. The cost function uses the cost (usually distance) to get from one node to another; in Ricochet Robots, that cost is always 1. It is always just 1 step to move to an adjacent location. In addition, a heuristic estimate of the cost to get to the destination could not be found for the same reason. Therefore the A\*, while used for path traversal, was not applicable to Ricochet Robots.

**4. Breadth-First Search**

The breadth-first search algorithm design proved to be the most useful in solving the problem; but it required some optimization before it became viable. Given a board configuration where none of the robots were on squares right next to one another or next to a wall, the state tree would have an initial branching factor of 16: each of the four robots could move in four different directions. From the depth of one on, things improved slightly as there was a guarantee that at least one robot was up against a wall or another robot as per the rules of the game, however, a branching factor of 15 is not a lot more tractable than a branching factor of 16.

The initial breadth-first search algorithm proved to not be very useful for precisely this reason; the number of states to be visited at the next depth became very large, very quickly. It became clear that we would have to borrow the concept of memoization from dynamic programming and not revisit states in order to make this algorithm fast. To do this, we did two things. First, we stopped distinguishing between the non-object robots on the board. As far as the logic of the game goes, if a square is occupied by a non-object robot, it does not really matter if the robot is green or blue; the game plays out the same. This also results in a much smaller number of potential states to be evaluated; there are drastically fewer ways to position three non-distinct robots and one distinct robot on a board than there are to position four distinct robots on a board.

Second, we had to think of a compact way to represent the state of the board. If our algorithm was going to be keeping track of all of the different states that had already been visited, it was going to take a lot of memory. The target square and the list of adjacencies of the board were static for a given initial board configuration, so the addition of this information into a particular state was unnecessary; all we really needed to keep track of was which robots were where. Since we were no longer distinguishing between the non-object robots, this was fairly straight forward. We made a construct where each board state was concisely represented by an 8 byte string, consisting of the row and column values of the locations of each of the four robots. Since Ricochet Robots is played on a 16 by 16 board, the row and column values were given by a hexadecimal character. The position of the object robot was given first, followed by a sorted list of the positions of the other robots since it does not matter which robot is which.

With all of this in place, it was simple to maintain a list of all of the states that had been visited, we simply created an array and scanned through it each time a new state was generated to make sure that it had not been visited before. This drastically reduced the number of states that had been visited; however the searching of this list became the new bottleneck in the algorithm.

The linear search of such a list can be executed in O(n) time where n is the size of the list; at worst the algorithm would have to scan through every item in the list. On the surface, this complexity seems fairly reasonable. However, this is an operation that has to happen every single time a new state is evaluated. The average Ricochet Robots board can be solved in just around 6.3 moves, and involves searching through around 40,000 different states. For such a case, the linear search did well, however due to the large branching factor of the state tree it became very slow for solutions with 10 or more moves.

To improve on this, we began to maintain an order in our list of states. Since the visited states are represented by 8 byte strings, it was simple to maintain the list in the order given by ASCII character code. The slight overhead that was involved in inserting each new state into its proper, ordered position of the list enabled us to use a binary search algorithm to search it. This new way to check our list meant our algorithm’s bottleneck was now reduced to O(log(n)) time, an appreciable improvement for the cases that involved dramatically more states.

However, for extreme solutions – ones that involved 12 steps or more – even this complexity proved to be very slow. A solution with around 15 steps can involve the visitation of nearly 3,000,000 states, and even with the binary search, this could take around ten minutes. We decided to use a hash set to keep track of how many different board states had been visited. This improvement meant we could search through our list of visited states in O(1) time. Again, the slight overhead of creating a construct like a hash set paid off dramatically for extreme cases. A similar solution involving nearly 3,000,000 visited states could be executed in roughly 20% of the time – just under two minutes.

With this higher level of optimization in place, the breadth-first search algorithm became very viable. The algorithm could find the optimum solution for the approximately 99% of Ricochet Robots configurations with solutions of 11 or fewer moves in well under the 30 seconds provided by the rules of Ricochet Robots.

**5. Branch and Bound**

As stated previously, the difficulty in the branch and bound algorithm was in finding the initial best case solution. We employed an unsophisticated approach to this by only moving the target robot alternatively vertically and horizontally towards the destination square. If it could not move towards it because of a wall then it would move in the opposite direction for one step. This ensured that it would eventually reach its destination eventually. However, this did not prove to be a good starting best case. On average it took 24 steps to reach the destination in this manner.

The major drawback of the branch and bound with respect to the breadth-first search is that, unlike the breadth-first search, the branch and bound algorithm continues until it has run out of leaves to traverse. This means that the optimal solution could be found early on, but the algorithm will still continue to run regardless. The plus side is that if this is the case then then rest of the tree traversal will be optimally pruned, but it is still time wasted.

Because of such a large initial value for our best-first search branch and bound algorithm, we had higher numbers for average visited states at 7,261,149 and average time at 1,482,294ms. On average it was 14 times slower than our linear search.

**6. Results**

This chart shows the distribution of the number of moves required to solve a Ricochet Robots board configuration based on 50,000 randomly generated boards. The Y-Axis corresponds to the number of solutions that were found, while the X-Axis corresponds to the number of moves required.

Our algorithms were also run many times to generate statistics on their effectiveness. Due to the large difference in the execution speed of these algorithms, however, we were unable to run them all the same number of times. The Linear search was slow and could only reasonably be run 1000 times, the Binary search was run 25,000 times and the hash set search was run 1,000 times. The branch and bound was very slow and could only be run 20 times. This table shows the results of these trials.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | BFS Linear | BFS Binary | BFS Hash | B&B |
| Average Time (ms) | 191531 | 2160 | 828 | 1482294 |
| Average Visited States | 40580 | 47176 | 46894 | 7261149 |

Each of the algorithms was able to find the best solution, approximately 6 steps on average. The highest number we ran into was 16 and the lowest was 1, ignoring the iterations where the board was generated with the object robot on the target square.

With each iteration of the breadth-first search there were improvements made in respect to average time to run, but the number of states visited only changed due to the random nature of the trials. The branch and bound is the larger outlier here and performed considerably worse.

**7. Interpretation**

The breadth-first search clearly was the better solution for this particular problem given how we implemented the algorithms. It is possible that given a better starting best case solution, the branch and bound could out-perform the breadth-first search.

Improvements to the breadth-first search itself yielded very good results. The hash search was over 230 times faster on average than the initial linear search and 2.6 times faster than the heavily improved binary search. There is still possibly room for improvement with pre-computed maps, but it would be minimal. This was the best solution to our initial problem we could come up with.

The branch and bound algorithm could potentially be very useful for people playing the actual game to check their solutions with. If the initial best case could be set to whatever the player comes up with as their best solution, the branches that need to be traversed are dramatically smaller. This could cut the visited states down considerably and would likely yield a very large performance gain.

**8. Future Work**

Generating pre-computed maps for the hash search could yield small performance gains. We would ideally like to implement a GUI to show people the moves of the robots and to possibly be interactive and let people play the game themselves from within our solver. Additionally, the ability to put in different board configurations would allow for more interactivity and increase the number of solvable solutions for anyone playing the actual game. Other than that, we accomplished our goal of finding a fast solver that guaranteed an optimal solution.

**9. Questions**

1. What algorithm design technique made the breadth-first search approach viable?
   1. The breadth-first search approach was made viable by borrowing techniques from dynamic programming.
2. What is the worst case starting branching factor for a ricochet robots state tree.
   1. If the four robots begin in positions away from the walls and one another, the initial branching factor will be 16.
3. What algorithm would be well suited for a player inputting and testing their own best solution?
   1. The branch and bound algorithm would do well here; its biggest drawback is a lack of means to find a good starting solution. A human player’s innate heuristic problem solving ability would provide a good solution for the algorithm to work off of.

**10. References**

[1]

Fogleman, M. (2014). Ricochet Robot. Retrieved December 2, 2015, from <http://www.michaelfogleman.com/ricochet/>

[2]

Coulman, R. (2015, March 9). Solving Ricochet Robots. Retrieved December 2, 2015, from <https://speakerdeck.com/randycoulman/solving-ricochet-robots>

[3]

Ricochet Robots. (n.d.). Retrieved December 2, 2015, from https://boardgamegeek.com/boardgame/51/ricochet-robots