Department of Applied Mathematics and Computer Science

02180 Introduction to Artificial Intelligence

Board Game Assignment

Authors:

Dario Cannistrà 194830

Stella Maria Saro 196563

Konstantinos Spyrikos 200240

Ghassen Lassoued 196609

Federico Crespo 200161

Course responsible:
Andrés Occhipinti Liberman

April 13, 2020

1 Introduction

For this report an AI solver has been implemented for the game Ricochet Robots: a multiplayer board game in which you have four robots to move in order to make one of them reach a goal position on the board. The board consists of walls and squares, which display the various positions.

Each possible goal is represented on a different card. The player has to pick a card, in order to select the current goal. The color of the goal also indicates the robot that needs to reach the goal.

When the goal and therefore the robot are selected, the player has to find the way to the goal that requires the least possible steps, but not less than two. Moving the robot, it has to be kept in mind that: every robot can only move in a straight line and once it starts moving, it can only stop when it reaches a wall, a different robot or the goal. Finally, any robot can be moved in order to create an 'obstacle' for the desired robot (these moves are also included in the total number of moves).

In our implementation of the solver,the rules have been simplified by removing the goal 'cosmic vortex', which can be reached with any robot. The board could also be modified, in order to alter the positions of the walls, but this would not affect the functionality of the AI. In the actual board game, extensions of the board can be found with different wall positions, that form the entire board. For further details see rules attached in the appendix A.

1.1 The representation of the board

The game is played on a board of 16×16 squares. The board contains the random initial positions of the robots and the positions of the goals, while the positions of the walls are always fixed between adjacent squares, as you can see in figure 1.

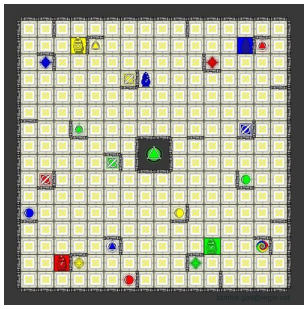

Figure 1: Board with walls, the positions of the robots displayed are just an example, to be taken at random before the start of the game, the possible goals positions are fixed

For the representation of the board we used an existing graphic implementation[1]. Ricochet robots is a fully observable, zero-sum game . It is a multiplayer game that can be played by two or more players that compete by trying to figure out the solution faster than the others. It is deterministic and and real-time.

[1] https://github.com/sharonzhou/ricochet

1.2 The representation of the states

The sizing of the state space of the game is defined by the dimension of the board and the number of robots on it. There are 4 robots that can be positioned on any square, so the state space S consists of the product between the states defined by the possible positions of the robots [2]

$$dim(S) = (16 \cdot 16 - 4) \cdot ((16 \cdot 16 - 4) - 1) \cdot ((16 \cdot 16 - 4) - 2) \cdot ((16 \cdot 16 - 4) - 3) = 3.937437 \cdot 10^9 \approx 3.94 \cdot 10^9 \quad (1)$$

Another factor making the game harder to be solved is the branching factor. This factor depends on the state given, but to give a logical order it can be said that generally, each one of the four robots has an obstacle which is the case most of the time. That gives a branching factor of 12.

The initial state is generated randomly before the game starts and it's defined by the initial positions of the robots on the board.

$$s_0 = \{\text{position robot 1, position robot 2, position robot 3, position robot 4}\}$$

To simplify the representation let's consider a relevant subset of our state space, the one generated by the positions of the robot that has to reach the goal.

$s_0 = \{\text{robot position}\}$
Player(s) = {the player that claims to have found the lowest number of moves}
Actions(s) = {move up, move down, move left, move right}
Results(s, move up) = {position under the first wall encountered while moving up}
Results(s, move left) = {position to the right of the first wall encountered while moving left}
Results(s, move right) = {position to the left of the first wall encountered while moving right}
Results(s, move down) = {position above of the first wall encountered while moving down}
Terminal-Test(s) = {Check if the robot needing to reach the goal is in the goal position }

These are all the possible moves and results, but the robots can move in four directions only from their initial positions if they are not close to a wall or to another robot. For the other states the robots can move maximum in three directions.

1.3 Computer representation

It is possible to visualize each one of the 4 robot positions and the current goal on the console window of python. The initial state is generated randomly, as previously stated, following the constraints to not generate robots on the targets. After the goal has been reached, the robots will return back to the original initial state. Remember that the user can generate a new random position whenever he/she desires, this in order to give the possibility to restart the game or to avoid some easy configuration.

[2] Positioning the first robot on the board we have $(16 * 16) - 4$ positions available , for the second robot we have the entire board minus the spot occupied by the first robot, and so on for the other robots.

After calling a class, where there are organized all the possible targets with the associated robots, a random target is picked up and through the move robots function a new state is created. This movement function always abide by the rules of the game as declared. Finally the robot will return to the originally initial state and the game will continue until the quit button is pressed. Dealing with a fully observable environment, the choice to use dictionaries and classes has been very useful to enhance the smoothness of the process.

2 Choosing the algorithm

From the algorithms considered in the course we could potentially use tree/graph BFS/DFS algorithms and A*.

Using a depth-first search (DFS) on this type of problem with such a great dimension of the state space would require an enormous amount of time because it would expand an entire branch of the tree before moving to the next. Since the depth we can get in a direction is effectively infinite some cutoff in the number of move would have been needed.

Because of this it had been decided to first use a breadth-first graph search (BFS) which explores the different levels of the tree and it's guaranteed to find the best solution first.

Knowing BFS algorithm is effective but not particularly efficient, A* has also been implemented. In order to implement an heuristic is also needed. The algorithm is identical to uniform-cost-search except that A* uses $g + h$ instead of g where $g(n)$ is the cost to reach the node and $h(n)$ is the heuristic: the cost to get from the node to the goal. Since the heuristic is consistent, A* is optimal. A* is also complete.

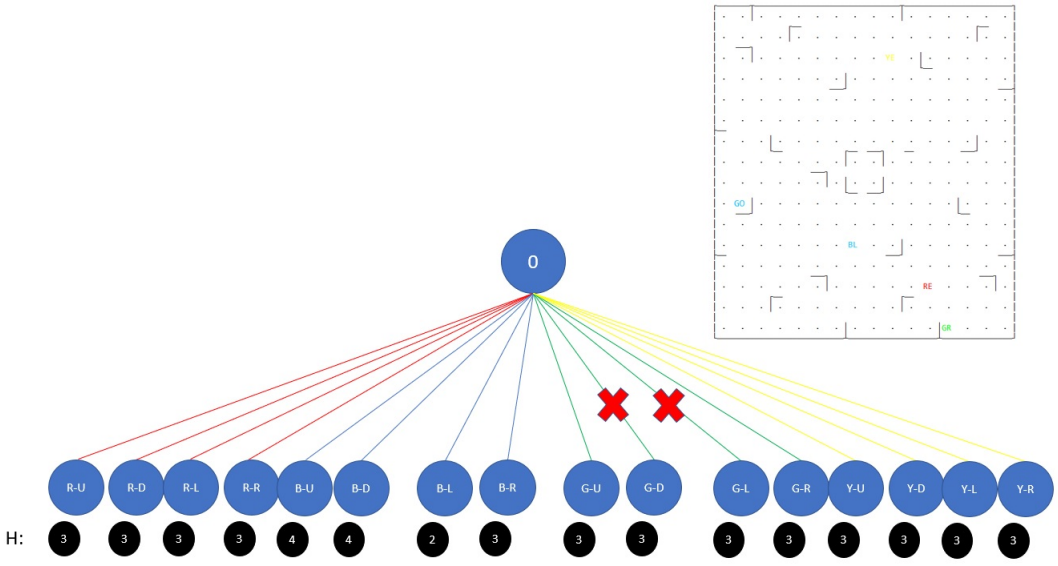

Figure 2: Tree representation of one action being performed on the Initial State

As seen in 2 the algorithm will try every possible move for every state. BFS after performing this action on

the initial state, it would keep performing this for every new state that are now in the frontier.

There are two states 'G-D','G-L' that are not added to the frontier, this is because they are already in the expanded nodes (as they are the initial state since nothing was changed).

For A*, after performing this initial action the frontier is ordered by F (the sum of the number of steps (cost function) and the heuristic). In the case in 2, the new state to be expanded would be 'B-L' and after expanding 'B-L' the new frontier has to be ordered again according to F.

3 Defining the heuristics

In order to enhance the speed of the solver we defined an heuristic starting from the goal. The idea was to define the heuristics as the distance from the goal for each position on the board. The first idea that was inspected was that of using the Manhattan distance (distance points measured along axes at right angles) but the actual physical distance in this game does not represent properly the distance between positions because when moving the robots it does not matter how many squares they cross but just how many times they turn. So it has been decided to define the distance as the minimum number of straight lines that are needed to go from the goal to the point.[3] [4]

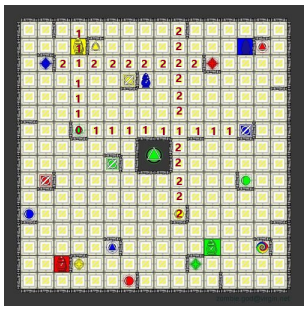

Figure 3: Board with written heuristics for 1 move and an example for the second move

Being n a general node and g(n) the cost of reaching n from the initial state in the search, but the Step-Cost (cost of a single action a) has been chosen as:

$$Step - Cost(n, a) = 1 \qquad \forall(n, a) \tag{2}$$

g(n) is then the number of moves to get to n from the initial state. The implemented heuristic function respects

[3] To make sure this heuristic was a good choice it has been searched online to compare it to different solutions. A lot of material has been found in particular the same solution has been analized in a conference (2015,MountainWest RubyConf (**?**)).

[4] Starting from the goal (h(goal) = 0) it has been assigned an h=1 to every position moving along the horizontal axe of the goal until a wall was hit. Each of those positions has then been used as a starting position from which, moving vertically it has been assigned to every spot a h = 2 and so on alternating the horizontal and vertical movement.The same procedure has been repeated starting with a vertical movement from the goal and substituting the heuristics of a spot with the newest found if this was smaller than the old one.

the admissibility property, in fact being h*(n) the minimum amount of steps to reach the goal from the node n:

$$h(n) = h^*(n) \qquad \forall n \tag{3}$$

The heuristics respects the consistency property too, if n and m are two nodes and a the action needed to reach m from n:

$$h(n) \leq c(n, a, m) + h(m) \tag{4}$$

where c(n, a , m) is the cost of the action. These two properties guarantee that the search is optimal.

To take into account the rule, described in chapter 1, for which a robot cannot reach a goal in less than two moves and it has to turn at leas ones, it has been implemented a loop that verifies if the heuristic for the initial position of the robot is h = 1 and if so, before beginning to solve the game, it moves the robot in a direction that increases the heuristic. This way the goal is reached in at least 3 moves for this particular case.

4 Results

To evaluate the performance of the A* it has been compared to a Breadth-First search.The first version of the BFS stored the information into dictionaries, the second one was implemented replacing most of the dictionaries with lists. In Table 1 the times to execute the different algorithms compared solving the same game, the procedure has been repeated for three different games.

Table 1: Comparing executing times for different algorithms

Algorithm	moves	expanded nodes	frontier nodes	time(s)
BFS with dictionaries	3	44	60	7.56
	4	235	218	11.92
	7	4665	2737	88.13
BFS	3	44	60	6.69
	4	235	218	10.25
	7	4665	2737	30.41
A^*	3	5	43	0.005
	4	23	132	0.02
	7	485	1560	1.587
Human player	3	X	X	~ 2
	4	X	X	~ 8
	7	X	X	~ 45

As you can see from Table 1 the difference between the BFS with dictionaries and the BFS with lists is of one or few seconds in games that require a little amount of moves and it grows rapidly when the number of moves is a little higher.
The A* performs well compared to the BFS, it has been compared to the human user and it solves the problem much faster than the player. Testing repeatedly for this board we found a particular position in which the A*, for how it has been designed, solves the game in a number of moves which is not the minimal amount, a more

detailed description will be given in chapter 5.

After the AI had been successfully implemented with the heuristic described in chapter 3, an idea implemented by the group to try to optimize the process has been to change the heuristics for every movement of the other robots(not the one that has to reach the goal) considering each of them as four moving walls forming a squared barrier. The results this procedure gave where very slow. One call of the heuristics took 0.2 seconds so calling it for every move is not possible, the program does not find the solution in a reasonable time.

5 Further improvements

If we were not to change neither the data structures nor the algorithms, a possible improvement would be finding a way to load the heuristics at every move of the other robots but making the process less time consuming. This could solve some performance issues when the path towards the solution requires multiple 'increases' (because we are not taking into account other robots) of the heuristic.

Another possibility would be to improve the movement function since it is checking more walls than the strictly needed ones.

Another interesting solution, changing the algorithm, would be to implement would be the backwards solution starting from the goal to get to the position of the robot.

An interesting improvement to A* was found by comparing it to human players. As previously mentioned, there is a particular case in the game with the considered board: when the goal is in position (3,9) (as seen in 3) and the robot we want to get to the goal is in the positions (4,9),(6,9),(9,9),(11,9) or (12,9) from which it would reach the goal in only one move and has no horizontal walls on his sides. The problem of reaching the goal in at least two moves has been solved as described in section 3, so the robot is moved up or down to increment the heuristics and then another way to the goal has to be found because not having horizontal walls to block it back in his initial position it cannot do the backwards movement to get to the goal in 3 moves. The solution is found by the human player with 3 moves in approximately 5 seconds without increasing the heuristic in the first move moving the robot right to position (13,9) then up then down returning to (13,9) and then left to the goal. The A* solves the problem in a fraction of a second but it solves it with a bigger number of moves.

6 Conclusion

Overall the results are promising since the AI solves the problem (in almost all cases) faster than a human being. Further improvements can be made in computing the moves and also an implementation with better GUI. For example, we could use Unity for the GUI while keeping the already implemented code in Python.

A good idea could also be to implement the hourglass and the bidding from the original game and different levels for the AI, in order to play against human beings (as if it was one of them).

References

[1] Andrés Occhipinti Liberman, Lecture notes

[2] Stuart Russell Peter Norvig Artificial Intelligence - A Modern Approach. 3rd edition, Global Edition.

[3] MountainWest RubyConf 2015 - Solving Ricochet Robots, https://www.youtube.com/watch?v=fvuK0Us4xC4

[4] Graphical Implementation, https://github.com/sharonzhou/ricochet/blob/master/render.py

Appendices

A Rules Appendix

-Distribute the 4 robots randomly on 4 spaces of the board not marked with a target symbol.
-Shuffle the 16 target chips face down an the table,pick up one chip from the table and place it face up on the center-piece; play may now begin. This was implemented as a random choice of a goal. A simplification was made here.
-Move the target robot(the robot whose color corresponds to the color of the chip) to the target space in a few moves as possible (the one with the chip's color and symbol). For reasons of simplicity, the chips were indicated by words for instance 'yellow triangle','yellow planet' .
- once a robot is set in motion, it cannot stop until it hits an obstacle: a wall or another robot.
-Each movement of a robot to the next obstacle counts as 1 move. The movement of all robots are counted.
-When a robot stops, it may move back the way it came only after another robot moves. (doubt about simplification here)
-If the chip in the center is the "cosmic vortex" (which has all colors), any robot, including the silver robot, may be considered the active robot and be brought to the cosmic vortex. This was simplified, so, the cosmic vortex, as well as the silver robot, were not taken into account. That explains that there are 16 targets spaces and not 17.
-On its way to the target space an active robot must hit and ricochet off (turn right or left) an obstacle at least once.
-When a robot stops, it may move back the way it came only after another robot moves.
-When a robot reaches a square with a diagonal colored wall, it passes through the wall if its color matches the wall and bounces off the wall at right angles if its color does not match the wall. When a robot reaches a wall opening, it passes through and appears, still moving, on the board exactly opposite the opening. In these special cases, the robot does not "stop" and continues to an obstacle, counting the entire move as just one move. This rule was also not taken into account while implementing the game.
-As soon as a player has found a solution, he may bid aloud the number of moves he thinks is required and set up the sand clock. The other players have now 1 minute to make their bids. Once a player has made a bid, he may not change it to a higher number.
-When the sand clock runs out, the lowest bidder plays out the round,if the number of moves he makes matches or is lower than the number he bid, he collects the chip and the round is over.If he fails, he must return the

robots he moved to their starting spaces and the turn passes to the next player. In case of equal bids, the player who has the fewer chips has precedence. This continues until a player succeeds. If no one succeeds, no one gets the chip, which is returned and reshuffled on the table.

-A 2-player game ends as soon as a player has won 8 chips; a 3-player game when a player has won 6 and a 4-player game when a player has won 5. If more than 4 are playing, continue until all the chips have been won. Of course, players are free to end in any other way they agree to before the start of the game.