Monte-Carlo Tree Search for Robocode

J. Grooss (jcgr@itu.dk), J. Melnyk (jmel@itu.dk)

Abstract— **The abstract goes here. Please try to make it less than 150 words. We suggest that you read this document carefully before you begin preparing your manuscript.**

This template is for LaTeX users of the Advanced AI in games class. Authors should use this sample paper as a guide in the production of their report(s).

I. Introduction

What problem are you trying to solve?

Why is this important?

II. Background

Most games attempt to engage the player by presenting a number of challenges for the player to overcome. Sometimes these challenges consist of precision, timing, execution speed and reaction time, while in other cases the challenge consists of making a strategic choice. When making these strategic choices, a player must consider not only the present state of the game, but also the actions taken by the adversary (either another player, an artificial intelligence or the game itself).

A. Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS)[1] is a searching algorithm that is based on the Monte Carlo method, which dates back to the 1940s. The idea behind the MCTS algorithm was explored in the 1980s and various implementations were written in the following years. It was not, however, until 2006 where a breakthough was made, which made AIs able to utilize MCTS to play games that had been considered too challenging until then, such as Go[2][3].

MCTS is, as the name implies, a tree search algorithm. Unlike other tree searches, it is a highly selective, best-first search that is able to figure out which parts of the search space that are the most promising and thus focus on that.

In order to determine the most promising part of the search space, MCTS runs a lot of *playouts*. *Playouts* are simulations of playing the game until an end condition is reached, with each move being chosen at random. The score of the game state at the end is based on the UCT and is used to update the weights of the tree in such a way that better nodes are more likely to be explored further.

The better the heuristic is at determining the value of a gamestate, the better MCTS will fare. An example of this is to factor in how many lives Pac-Man has left at an end state instead of only evaluating at the score.

B. MCTS in Partially Observable Games

MCTS works on the premise of information. The more it knows about what is going on, the better it performs. This means that in fully observable games, such as chess or Pac-Man, where it has access to all information about a game state, MCTS will perform well. Based on this fact, it is sensible to assume that MCTS will not perform as well in partially observable games.

Imagine a version of chess where you do not know the position of your opponent's pawns until you try to move a piece to their position or they are within one square of any of your pieces. In such a game, MCTS would not be able to properly evaluate various game states, as it does not have the necessary information to do so. It can assume that the opponent plays in a certain way and evaluate the search space based on that, but it will not be accurate at all times.

That a game is only partially observable does not mean that MCTS will not work. It simply means that algorithm will have to take into account that there are things it does not know and that it therefore must attempt to figure these things out.

Research indicates that is is possible to write MCTS implementations that works in a partially observable environment. The research has been done on very simple games (phantom tic-tac-toe)[4] and on systems where it is possible to build a history of previously encountered states and run statistics on that[5][6]. While they are not developed enough to work with most partially observable games, it does prove that MCTS can be adapted to such games, given the right methods.

III. Game Mechanics

The game we use is called "Robocode"[7]. It is a programming game where the players program robots to battle other players' robots.

Fights between robots take place on a battlefield, which is represented as a Cartesian coordinate system, with (0, 0) being in the lower left corner. Distances are measured in pixels and directions are measured in degrees, where north is 0 degrees, east is 90 degrees, south is 180 degrees and west is 270 degrees. Robots decide on what action they want to take and all robots' actions are then carried out simultaneously.

A robot starts with 100 energy, no idea of where the enemies are and can either move (ahead or back), turn its base, turn its gun, turn its radar or fire its gun during a turn. Moving and turning has no influence on the energy of the robot, but firing the gun and getting hit by bullets decrease

the energy of the robot. When a robot is out of energy, it dies. There are limits to how much a robot can move or turn during a single turn[8], as having no limits would break the game.

While a robot is limited to how much it can move or turn during a single turn, its API[9] makes the process easier. For example, if a robot wants to move 100 pixels, it call the method Ahead(100). This method does not return until it has moved the chosen distance, and will thereby prevent other methods from being called meanwhile. As a robot can move no more than 8 pixels per turn, that is, best-case, $\frac{100}{8} = 12.5$ turns in which it only moves, something that is important to keep in mind when designing a robot.

In order to figure out where the enemy is, the robot needs to use its radar. The robot's radar is constantly active and scanning in in a straight line. If an enemy is detected by the radar, which can happen when the radar turns[1], the robot's OnScannedRobot() method is called, even if the robot is moving or turning already. When the method returns, the robot resumes doing what it was doing.

When the OnScannedRobot() method is called, it is possible to get some information about the scanned robot. The information is limited to the following: The enemy robot's heading, velocity and energy, the bearing our gun has and the distance to the enemy. This means that a robot will have to calculate the other robot's position in order to get that information, and that it is impossible to know where another robot's gun is pointing. Due to this, it is difficult to keep track of what an enemy robot is doing.

Firing the gun is a bit different from moving and turning. Firing the gun can be done with a power level between 1.0 and 3.0. The power level determines how much energy the robot loses (at a 1:1 ratio), but also determines the damage and velocity of the bullet. The more power, the more damage it deals if it hits and the slower the velocity is. Firing the gun also produces gun heat, and a robot is unable to fire as long as its gun is hot.

The rules and the API mean that it is possible to write a simple robot easily, but developing a good robot is a challenge.

Why do you need AI in this game?

A. *Influence of Robocode mechanics on MCTS*

Robocode is, without a doubt, a partially observable game. A robot only knows its own information and occasionally knows where an enemy robot is and where it is heading. There is no way to get any information about the general game state either, so information about where bullets are and where they are heading is also unavailable.

These factors mean that MCTS has very little information to work with, both during its simulation and its evaluation steps. This will be a challenge, as MCTS will have to make assumptions about where the enemy robot is (even after scanning it, it will quickly move to a new position) and how the enemy behaves in order to be able to simulate a playout and evaluate the results of a playout. If the assumptions are off by just a little, it will have a major impact on how well MCTS does.

[1]The radar is connected to the gun, which is connected to the base of the robot. If one of those turn, the radar turns with them.

IV. Methods

How does your algorithm work? Describe in as much detail as you can fit into the report.

Also, how did you interface it to the game?

V. Results

Did it work?

How well? Provide some figures, and a table or two.

How much time does it take?

Remember to include significance values (remember the t-test?), variance bars Reread some of the papers from class and compare how they report their results.

VI. Conclusions

The conclusion goes here.

What are the strengths and shortcomings of your method? Why did you choose method X instead of Y? How well would it generalize to other game genres? How would you develop it further, if you had time?

References

[1] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012, [Online; accessed December 11, 2014]. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6145622

[2] S. Gelly and D. Silver, "Monte-Carlo tree search and rapid action value estimation in computer Go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011, [Online; accessed December 11, 2014]. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S000437021100052X

[3] G. Chaslot, "Monte-carlo tree search," Ph.D. dissertation, PhD thesis, Maastricht University, 2010, [Online; accessed December 11, 2014]. [Online]. Available: https://project.dke.maastrichtuniversity.nl/games/files/phd/Chaslot_thesis.pdf

[4] D. Auger, "Multiple tree for partially observable Monte-Carlo tree search," in *Applications of Evolutionary Computation*. Springer, 2011, pp. 53–62, [Online; accessed December 11, 2014]. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-20525-5_6

[5] D. Silver and J. Veness, "Monte-Carlo planning in large POMDPs," in *Advances in Neural Information Processing Systems*, 2010, pp. 2164–2172, [Online; accessed December 11, 2014]. [Online]. Available: http://papers.nips.cc/paper/4031-monte-carlo-planning-in-large-pomdps

[6] S. Thrun, "Monte Carlo POMDPs." in *NIPS*, vol. 12, 1999, pp. 1064–1070, [Online; accessed December 11, 2014]. [Online]. Available: ftp://ftp.irisa.fr/local/as/campillo/micr/bib/thrun1999b.pdf

[7] M. A. Nelson and F. N. Larsen, "Robocode," 2001, [Online; accessed December 11, 2014]. [Online]. Available: http://robocode.sourceforge.net/

[8] R. Wiki, “Game physics,” 2007, [Online; accessed December 11, 2014]. [Online]. Available: http://robowiki.net/wiki/Robocode/Game_Physics
[9] M. A. Nelson and R. contributers, “Robocode api,” 2014, [Online; accessed December 11, 2014]. [Online]. Available: http://robocode.sourceforge.net/docs/robocode.dotnet/Index.html