Monte-Carlo Tree Search for Robocode

Jacob Grooss (jcgr@itu.dk), Jakob Melnyk (jmel@itu.dk)

***Abstract*— Monte-Carlo Tree Search (MCTS) generally performs well in games where it is able to correctly simulate game states and adversarial behaviour. The goal of this paper is to explore whether MCTS is capable of performing well in partially observable games where there is no single best strategy for the adversary to use.**

We compare four different versions of an MCTS controller for the game Robocode by matching them up against simple controllers included with the game. Our best controller performed quite poorly against the included controllers. Our results suggest that regular MCTS is unlikely to perform well in these kinds of environments, however a number of changes to the algorithm would likely result in a vast improvement of performance

I. Introduction

This paper was written as part of a project for the "Modern AI in Games" course on the Games Technology track at the IT-University of Copenhagen. The source code and test data are available on our repository[1] on Github.

A. Problem Statement

The Monte-Carlo Tree Search (MCTS) algorithm is commonly used in games to search for the best path in a game tree. The algorithm is generalizable to most games as it, in its base form, "does not require any strategic or tactical knowledge about the given domain to make reasonable decisions"[1]. This can change drastically, however, when MCTS is introduced to a game where states of the game are only partially observable and where there is no single best strategy that the algorithm can assume that an adversary is using.

In this paper, we will test our hypothesis that partially observable games against complex adversaries will make MCTS perform poorly. In order to do so, we will discuss how MCTS works, what makes a game partially observable, and how the mechanics of our test game, Robocode, may influence the performance of a MCTS game-playing algorithm. We will then present the algorithm we have used to test against sample robots in Robocode, followed by a reporting of the results and a discussion of the potential changes that could be made to the algorithm.

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).

[1]https://github.com/jcgr/MAIG-RoboCode-Project

A. Monte-Carlo Tree Search

Monte-Carlo Tree Search[2] (MCTS) 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[3][4].

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 a reward system and its value 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 perform. 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 is able to perform well. Considering how important information is to MCTS, 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 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 game state based on that, but such an evaluation will be inaccurate to some degree.

That a game is only partially observable does not mean that MCTS will not work, however. It simply means that the algorithm will have to take into account that there are things it does not know, which means that it must either figure these things out or make assumptions about what has happened.

Research indicates that is is possible to write MCTS implementations that works in a partially observable environments.

The research has been done on very simple games (phantom tic-tac-toe)[5] and on systems where it is possible to build a history of previously encountered states and run statistics on that[6][7]. 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"[8]. It is a programming game where the players program robots to battle other players' robots.

Fights between robots take place on a battlefield (see figure 1), 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.

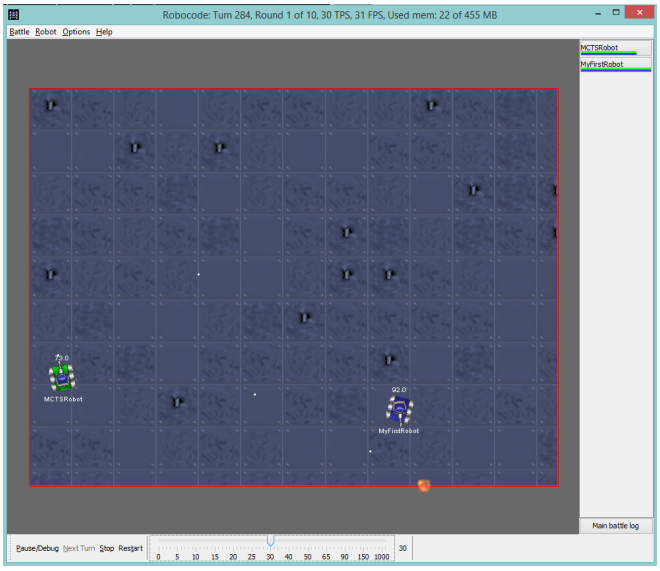

Fig. 1. A battle between two robots..

A robot starts with 100 energy, no idea of where the enemies are and can either move (ahead or back), turn its base, gun or 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 decreases its energy. When a robot is out of energy and it is hit by a bullet, it dies. There are limits to how much a robot can move or turn during a single turn[9] in order to mirror real-life tanks properly.

While a robot is limited to how much it can move or turn during a single turn, its API[10] makes the process easier. For example, if a robot wants to move 100 pixels, it calls the method Ahead(100)[2]. This method does not return until the robot 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[3], the robot's OnScannedRobot() method is called, even if the robot is moving or turning already. When the OnScannedRobot() method returns, the robot continues doing what it was doing prior to scanning an enemy.

In the OnScannedRobot() method, a robot can take actions as normal. A robot can for example decide to shoot and then move a bit before continuing doing what it was doing before. During this method, a robot has access to some information about the scanned robot, but 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. Considering this, a robot will find it difficult to keep track of what exactly an enemy robot is doing.

Firing the gun is 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), and also determines the damage and velocity of the bullet. The more power, the more damage it deals if it hits and the slower the bullet moves. Firing the gun also produces gun heat, and a robot is unable to fire as long as its gun is hot. Gun heat is lowered every turn.

As the gameplay of Robocode consists of developing an AI controller for a robot that does battle with other AI controlled robots, AI is an instrumental tool in making Robocode into what it is.

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 completely 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 performs.

[2]The Ahead() method takes a double value as parameter, in this case 100.

[3]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

Our current implementation assumes that there is only one opponent in the game. The more opponents that are added to the game, the less precise the simulations of the game state will become.

A. Monte-Carlo implementation

Our implementation of the MCTS algorithm differs from the original MCTS algorithms in a few of ways. The two most interesting differences are related to our child selection and our search depth.

1) Child Selection: When a node selects which child it should explore, normal MCTS chooses the child based on the UCT[11] value of the children. It is possible, however, that a child will score lower than the average of its siblings when playout happens, thus leaving the child with a low UCT-value. As higher UCT values are prioritized by MCTS, the parent of the bad child will not be explored for quite some time (if ever again). This will generate an extremely asymmetric tree, which we would like to prevent. In order to do so, we only use the UCT value for exploration if all children of a the node we are exploring has been visited at least three times.

Through trial and error, we concluded that an exploration constant of 1 was what worked best in our implementation of Monte-Carlo Tree Search. In general, a higher exploration constant will lead to a more breadth-first search than a depth-first search, which is desirable for our current implementation as we do not have the luxury to simulate too far down the tree (both due to time constraints and due to inaccuracy in simulation).

2) Max Search Depth: Due to the nature of Robocode, we are unable to simulate our opponents properly (see section III). This means that our simulations become more inaccurate the further down the tree we go. We chose to incorporate a maximum search depth to prevent simulations that were too inaccurate from affecting the tree too much. We chose a maximum search depth of 25, as it lets us reach a simulation that is relevant, without risking that the simulation becomes highly inaccurate.

3) Branching Factor: The branching factor of the algorithm is equal to the number of different moves that can be made by our robot in a single turn. This is a staggeringly high number, as the game uses double precision for degrees, distances, and firing power. Because of this, we have discretized the different options down to 32 (see equation 1).

$$\begin{aligned} 32 = 5(speedintervals) + 11(robotturnintervals) \\ +9(gunturnintervals) + 5(radarturnintervals) \\ +2(shoot) \end{aligned} \quad (1)$$

B. Game Simulation

Because Robocode is somewhat competitive - the inner workings of the game is not accessible from a robot - we had to do an emulation of the game in order to simulate future game states in the MCTS tree. Each node contains a game state which consists of information about our robot, the enemy robot, the projectiles that are currently flying through the air, and the instructions given to our robot in order to get from the previous game state to this one.

When simulating the next tick of the game, we follow steps 4 & 5 of the Robocode Processing Loop[9]. Currently we do not simulate our robot and the enemy robot simultaneously, which is a difference between our simulation and the Robocode Processing Loop.

1) Bullets: Our bullet simulation involves getting the next position of every currently active bullet and using those new positions to check for collisions with the two robots.

2) Our Robot: The Monte-Carlo Tree Search selects a robot instruction for the game state simulation to use when getting the next state of our robot. Currently we do not simulate collisions between robots (ramming) or radar scans.

3) Enemy Robot: When we simulate the enemy robot, we assume that it knows the location of all projectiles that are currently active. If the next position of the enemy robot is hit by a bullet, we assume that it will prioritize dodging this projectile. If it attempts to do so, it will search for another possible location that it can get to this turn that will not be hit by a projectile. In addition, we assume that the enemy knows our location and will attempt to turn its gun towards our robot. If the angle to our robot is within five degrees of the enemy gun, we assume that he will fire at our robot.

C. Evaluation Heuristic

When the Monte-Carlo Tree Search algorithm calls for the score of a node (game state), we use a heuristic (equation 2) to evaluate the state of the game at that node.

$$score = OurRobotScore + \left(\frac{1}{2}\frac{1}{100}EnemyEnergy\right) \quad (2)$$

Equation 3 describes how we calculate the score of our own robot.

Where *EnergyScore* is $\left(\frac{1}{100}OurEnergy\right)$, *BulletDamageScore* is the amount of damage dealt by bullets until this point, *ShootScore* is 0.1 if we shot this round; 0.0 if not. *MovementScore*[4] is the change in velocity that the robot made. If the robot did not change velocity or move, *MovementScore* is instead given a -1 score. Finally the movement score is multiplied by 1.5. *RobotHeadingScore* is the amount of degrees that the robot turned this round multiplied by 0.4.

$$\begin{aligned} ourRobotScore = EnergyScore + ShootScore \\ BulletDamageScore + MovementScore \\ +RobotHeadingScore \end{aligned} \quad (3)$$

D. Interfacing it to the Game

Because the main "gameplay" of Robocode is actually coding a robot, we simply coded the robot inside the framework provided to all Robocode players.

[4]MovementScore is also penalized if the robot moves too close to the walls of the battlefield.

TABLE I

ROBOT SETTINGS FOR SCORE REWARDS

Robot	Movement	Turning	Shooting
Default	1.5	0.4	0.1
MCTS-1	**3.0**	0.4	0.1
MCTS-2	1.5	**0.8**	0.1
MCTS-3	1.5	0.4	**0.2**

TABLE II

SCORE FOR THE DEFAULT-MCTS ROBOT

Enemy	Mean	Median	Mode	Std. Dev	Std. Err.
MyFirstRobot	25.6%	25.4%	None	1.609	0.509
Spinbot	15.7%	16.3%	None	2.145	0.678
Tracker	10.8%	10.5%	None	0.779	0.246

V. RESULTS

In this section, we discuss how well different versions of our robot played against some sample robots provided with the Robocode installation. We also present data gathered while testing as well as evaluate the significant differences between each robot (if any).

A. Test Set-Up

We tested four different set-ups of the robot versus four different sample robots. Each test consisted of 10 tests of a 10-round battle with otherwise default Robocode rules. In order to test the effect of each setting, we only changed a single setting in each variation of our robot from our default robot. Because of the large number of different possible combination of settings, it is likely that we have not found the best possible one. The settings for our robots can be seen in table I.

1) Enemy Robots: We tested our robot against the *MyFirstRobot*, *SittingDuck*, *Spinbot*, and *Tracker* sample robots. *MyFirstRobot* moves in a see-saw motion, rotates its gun and fires whenever it scans another robot. *Spinbot* moves in a circular motion and fires a powerful shot whenever it scans an enemy. The *Tracker* robot scans for a target, locks on to it, moves close and then fires at the target. As the name suggests, the *SittingDuck* takes no action, does not move and does not react. It is often used to test how well the guns of a robot works.

We have not provided tables and data sets for the results of our tests versus the *SittingDuck* robot, as we obtained a 100% score against it with all four versions of the MCTS robot on every trial.

2) Default-MCTS Robot: We established the baseline "Default" robot settings by trying out different settings and deciding on a combination that seemed to be somewhat successful. The settings used for this robot offer a good compromise between shooting, moving, and turning. This default robot is somewhat trigger happy, however, and at times attempts to shoot every time it is possible.

3) MCTS-1: This robot should prioritize movement more than it ended up doing. It is, however, difficult to truly evaluate how much more it moves without tracking it specifically, which we did not.

4) MCTS-2: This robot is highly rewarded for turning around, which causes it to spin wildly while it shoots seemingly at random. It is possible that the doubling of the turning reward was too much and it would have been more useful to test with less of an increase (or perhaps even a decrease).

5) MCTS-3: We had anticipated that this robot would shoot more than the default robot, but we did not notice any apparent difference in the amount of shots fired. This is likely due to the trigger happiness of the default robot; if it already shoots as often as it can, then increasing the shooting reward is unlikely to make much of a difference.

B. Data

The scoring data in this section is represented as percentages of the total score in a single battle, e.g. our robot got 25% of the total points and the enemy robot got the remaining 75%. We represent it as such because of the stochastic nature of spawning positions and robot heading at the start of a round. By comparing percentage of the scores instead of absolute values, we are letting these random factors influence the results in the least possible way.

1) Default-MCTS Robot: As the default-MCTS robot was the baseline for our tests, we have done all significance comparisons using the test data for this robot.

Table II indicates large differences in performance against the three different enemy robots. While the default robot never gets below 23% score against *MyFirstRobot*, it does poorly against the *Spinbot* and even worse against the *Tracker* robot. Interestingly, the robot is consistently poor against the *Tracker* robot, as evidenced by the lower standard deviation (and seen on figure 2[5]).

The distribution of scores against the enemy robots are shown in figures 3, 4, and 5. The values are the decimal representations of the percentage scores achieved by the default robot. While the scores mostly fit somewhat nicely around the bell curve, the results against both *MyFirstRobot* and *Spinbot* have one rather extreme outlier.

2) Other versions: Tables III, IV, and V show the scores of the three variations of our robot. As shown by tables VI[6], VII, and VIII, two of the robot variations have significant differences in score percentage. MCTS-1 performs significantly better against the *Spinbot* and the *Tracker* robot, but not so against *MyFirstRobot*. MCTS-2 performs significantly worse against all three enemy robots.

C. Evaluation

As noted above, MCTS-1 performs significantly better than our default-MTCS against two of the three enemy robots. It also averages a better score against the last of the three robots, although this difference is not significant.

[5]The variance bars on the graph represent standard deviation.

[6]The variances were not equal between Default and MCTS-3, so a different critical t-value was used.

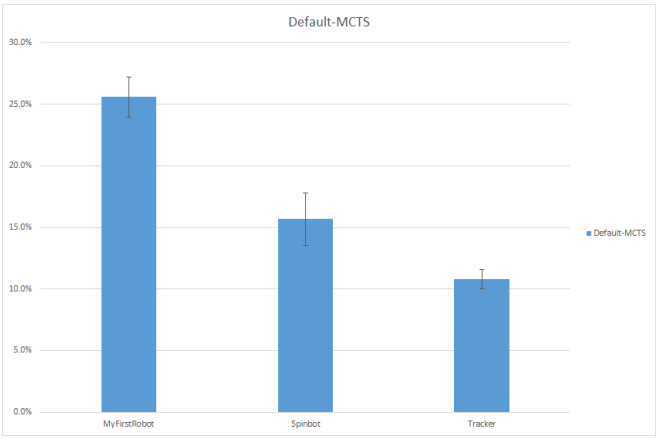

Fig. 2. Mean score percentages for the Default MCTS controller.

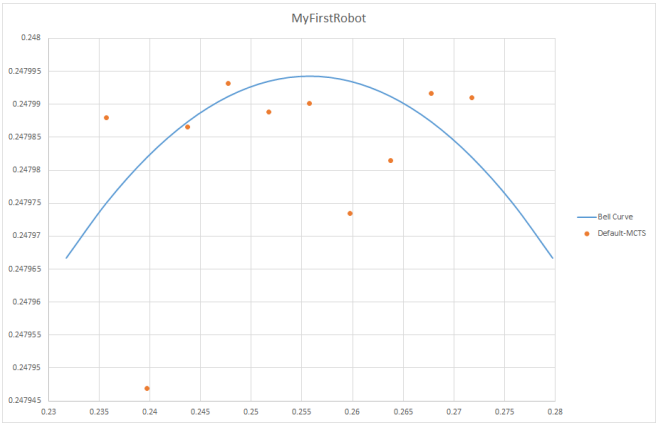

Fig. 3. Distribution of results by default-MCTS against the "MyFirstRobot".

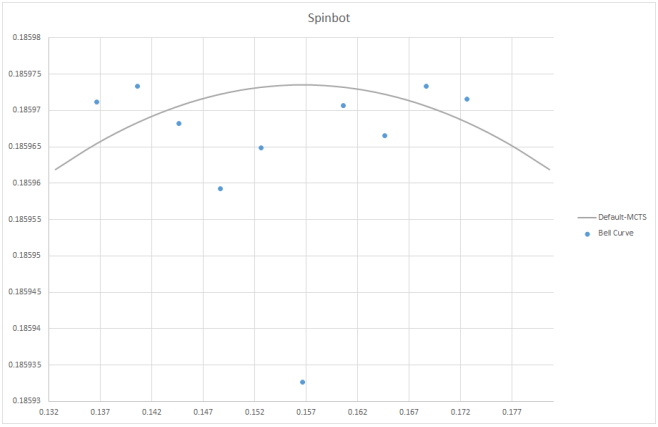

Fig. 4. Distribution of results by default-MCTS against the "Spinbot" robot.

TABLE III
SCORE FOR THE MCTS-1 ROBOT

Enemy	Mean	Median	Mode	Std. Dev	Std. Err.
MyFirstRobot	27.2%	26.6%	None	3.292	1.041
Spinbot	19.5%	19.3%	None	2.437	0.771
Tracker	13.7%	13.9%	14.3	1.789	0.567

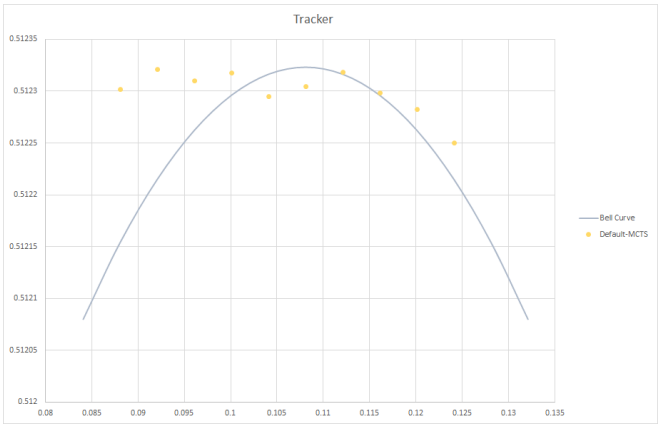

Fig. 5. Distribution of results by default-MCTS against the "Tracker" robot.

TABLE IV
SCORE FOR THE MCTS-2 ROBOT

Enemy	Mean	Median	Mode	Std. Dev	Std. Err.
MyFirstRobot	8.4%	8.3%	None	1.854	0.586
Spinbot	3.4%	3.6%	3.7	0.970	0.307
Tracker	4.4%	4.5%	4.6	0.491	0.155

TABLE V
SCORE FOR THE MCTS-3 ROBOT

Enemy	Mean	Median	Mode	Std. Dev	Std. Err.
MyFirstRobot	26.0%	25.8%	None	3.054	0.965
Spinbot	15.9%	15.1%	None	3.319	1.049
Tracker	11.4%	11.1%	None	1.303	0.413

TABLE VI
SIGNIFICANCE TEST USING THE MYFIRSTROBOT ROBOT

Controller	Mean	Variance	t-value	critical t	Significant
Default	0.256	0.00025	N/A	N/A	N/A
MCTS-1	0.272	0.00108	1.397	2.101	No
MCTS-2	0.084	0.00034	-22.067	2.101	Yes
MCTS-3	0.260	0.0009	0.354	2.145	No

TABLE VII
SIGNIFICANCE TEST USING THE SPINBOT ROBOT

Controller	Mean	Variance	t-value	critical t	Significant
Default	0.157	0.00046	N/A	N/A	N/A
MCTS-1	0.195	0.00059	3.730	2.101	Yes
MCTS-2	0.034	0.00009	-16.502	2.101	Yes
MCTS-3	0.159	0.00110	0.215	2.1001	No

TABLE VIII
SIGNIFICANCE TEST USING THE TRACKER ROBOT

Controller	Mean	Variance	t-value	critical t	Significant
Default	0.108	0.00006	N/A	N/A	N/A
MCTS-1	0.137	0.0003	4.744	2.101	Yes
MCTS-2	0.004	0.00002	-22.067	2.101	Yes
MCTS-3	0.114	0.00017	1.229	2.101	No

Thus we can conclude that MCTS-1 is the best of the four robots tested for this paper, default-MCTS and MCTS-3 are about equal, and by far the worst is MCTS-2, which performs horribly against all three enemies.

1) Speed: The speed of the algorithm is capped at 10 milliseconds per iteration in order to avoid skipping any turns in robot. This does not allow the algorithm to search very deep into the tree, but we believe this to be a somewhat positive behaviour. Because the simulation of the game state and the assumptions we make about the actions taken by the opponent can potentially be wildly inaccurate, we will rarely benefit from simulating too far into the future.

VI. Discussion

The goal of this project was to test the Monte-Carlo tree search algorithm in Robocode to learn how well it would perform, considering the partially observable nature of the game. We expected that it would not perform well, as there is a lot of information that a robot (and therefore also MCTS) does not have access to.

We tested four MCTS robots, a default robot and three variations where the reward of actions were changed, against three sample robots. One of our robots was significantly better against two of the three test robots than the default robot, but even that was unable to defeat any of the test robots.

There are a couple of reasons for this: Our reward system is not the best, as there are some things we do not account for, and our simulation of the game is very inaccurate.

Our reward system only considers basic things such as firing, moving, turning and the energy of the robot and its opponent. We do not account for things like ramming the enemy or for scanning the enemy, though both are arguably important. Ramming an opponent deals damage to them and is therefore a good thing to do. Scanning the enemy provides us with up-to-date information about it, which in turn helps the simulation. Adding both to the reward system should make it more accurate.

The reward system could also be changed to include minimizing the opponents score, in the style of the minimax algorithm. This would provide the robot with another point from which to evaluate different game states.

With regards to the game simulation, there is one glaring problem: Our simulation does not simulate the opponent properly. It assumes that the opponent plays in a way that we would deem perfect, but few, if any, robots play that way. This means that our simulation is not consistent with what happens in the game, which again means that the exploration of our search tree (and the values for our reward system) is based on incorrect information.

It is not feasible to improve the simulation by writing robot-specific simulations. It would give very little benefit, especially considering that it would not work against other robots than that specific one. Instead, one could look into learning how the opponent plays. As a battle lasts for ten rounds, it would be possible to use the first round as a ”suicide” run, where the robot focuses only on figuring how the opponent plays. During the remaining rounds the robot should perform well and be able to regain enough points to win.

Such learning would involve looking into machine learning techniques and would still not be entirely accurate (we do not know which direction the opponent fires in, for example), but if done correctly the robot should should be able to fight on even terms with a large variety of robots.

Another way to simulate would be to extend the part of the Robocode API that deals with evolving the game state and use that for simulation. It transform Robocode from a partially observable game into a fully observable game, which means that MCTS would have access to all the information it needs. It should be noted, however, that this would not work for competitions in Robocode, as it is not a functionality available to robots.

When these improvements have been made, looking into how the robot performs against multiple opponents simultaneously would be interesting. While it would likely not be able to perform, or even equally as well, against multiple opponents compared to a single opponent, it would enable the robot to take part in free-for-all battles, which is a facet of the Robocode game that the current MCTS implementation does not support at all.

References

[1] C. Browne, “mcts.ai,” accessed December 12, 2014. [Online]. Available: http://mcts.ai/about/index.html

[2] 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 12, 2014]. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6145622

[3] 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 12, 2014]. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S000437021100052X

[4] G. Chaslot, “Monte-carlo tree search,” Ph.D. dissertation, PhD thesis, Maastricht University, 2010, [Online; accessed December 12, 2014]. [Online]. Available: https://project.dke.maastrichtuniversity.nl/games/files/phd/Chaslot_thesis.pdf

[5] D. Auger, “Multiple tree for partially observable Monte-Carlo tree search,” in *Applications of Evolutionary Computation*. Springer, 2011, pp. 53–62, [Online; accessed December 12, 2014]. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-20525-5_6

[6] 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 12, 2014]. [Online]. Available: http://papers.nips.cc/paper/4031-monte-carlo-planning-in-large-pomdps

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

[8] M. A. Nelson and F. N. Larsen, “Robocode,” 2001, [Online; accessed December 12, 2014]. [Online]. Available: http://robocode.sourceforge.net/

[9] R. Wiki, “Game physics,” 2007, [Online; accessed December 12, 2014]. [Online]. Available: http://robowiki.net/wiki/Robocode/Game_Physics

[10] M. A. Nelson and R. contributers, “Robocode api,” 2014, [Online; accessed December 12, 2014]. [Online]. Available: http://robocode.sourceforge.net/docs/robocode.dotnet/Index.html

[11] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Machine Learning: ECML 2006*. Springer, 2006, pp. 282–293, [Online; accessed December 12, 2014]. [Online]. Available: http://link.springer.com/chapter/10.1007/11871842_29