Killer Robots

Evan Williams, 18015029, Applied Computer Science BSc

1 Abstract

This document describes how robots were created for the competitive programming game “Robocode” using genetic algorithms.

Contents

2 Introduction

Robocode is a programming game in which people create algorithms for controlling robots and then fight them against each other [1]. It is a fiercely competitive field, with players submitting robots to an online ranking system [2].

A genetic algorithm is a system designed to find good solutions to a problem through techniques similar to how organisms breed and evolve in the natural world [3]. They are a heuristic system, which means they are good at finding a good solution, but are unlikely to find an optimal one [4].

Robocode is an ideal candidate for a GA because it has a large search space, but the problem can be easily broken down in to a simple representation. Furthermore several representations can all be used to solve the problem making it a very good challenge to compare genetic algorithms on.

3 Framework

A framework was created that allows a developer to create arbitrary genetic algorithms for Robocode. Once this coursework has been submitted it will be open-sourced and released on http://github.com/malacandrian. The framework allows any number of genetic algorithms to be coevolved against each other over a number of generations. Once all the generations have been executed, it saves the best few, and displays their battle for the user to observe.

The framework is divided in to three main classes, each representing one area of concern.

3.1 AbstractRobotEvolution

This abstract class is extended to create the robot itself. Its main concerns are converting the genome in to a phenome, and deciding what happens when it is bred with a set of other robots. Because the abstract class performs operations based on the genome (such as choosing the robot's name) it needs to be defined at this stage. It was decided to use a byte array, as everything else can be converted in to one.

The developer needs to define the string of C# that is inserted into each function the resultant robot has (Run, OnBulletHit, OnBulletHitBullet, OnBulletMissed, OnHitByBullet, OnHitWall, and OnScannedRobot) as well as how to

breed with a set of other robots. The abstract class then handles the processes of calling the C# compiler as well as correctly structuring the C# file.

3.2 AbstractPopulation

This abstract class is extended to create an evolutionary strategy. It owns a set of robots and decides which of them get to breed (and which ones breed with each other), which ones survive to the next generation, and which ones die.

The developer must define a fitness function, a manner of selecting robots to breed, a manner of pairing robots from the breeding pool, and a manner of selecting robots to be carried forwards to the next generation. The abstract class will then chain these actions together so that the evolution happens at the correct time, as required by the environment. In theory all genetic strategies should be reproducible in this system.

3.3 EvolutionEnvironment

This class handles all of the interactions with the Robocode control engine, meaning all the developer has to do is instantiate it, add all of their populations, tell it how many generations to go for and what training set to use, and kick it all off. It then pools all of the robots, ensuring each of them are in exactly the requisite number of battles each generation and reports the scores back to the populations so they can judge fitness.

4 Robots

No particular genetic strategy was chosen, rather bits were taken from each one as they were seen as useful.

4.1 Trivial

Trivial robot is a very simple case of a parse tree. It created as an early step in the development of the ParseTree robot (as such might be named ParseTree in some tests) as the minimum example of a robot that acts based on genes and can breed, it was preserved because it showed interesting results.

4.1.1 Genotype

One instruction is represented as a pair of bytes: the first always has a value between zero and five, each mapping to a single instruction, the other is valued 0-255 (pending a transformation dependent on the instruction) acts as the argument to the instruction. In the case of Execute() which has no parameters, the second byte is never read.

Each of the six event handlers and the run function own a set of these pairs. When initially generated, each of the functions has random number of instructions (mean 10, standard deviation 3). Because the length of each function varies between robots there is a short header at the beginning of the genome is the set of lengths of the functions.

See Figure 11-1 for an example of how the genotype is constructed.

4.1.2 Phenotype conversion

There is a one-one match between the phenotype and genotype the conversion is very simple: the genome is split in to its function parts. When compiling each function it looks at that part, considering each pair in turn. The first byte of the pair is mapped to a function (0 to SetAhead, 1 to SetFire, etc), the second is transformed from $\mathbb{Z}[0,255]$ to fit the valid range for the argument ($\mathbb{Z}[-128,128]$ for SetAhead, $\mathbb{R}[0,3]$ for SetFire, $\mathbb{R}[-\pi,\pi]$ for SetTurn/Gun/RadarLeft). This is repeated for all pairs within all functions.

4.1.3 Breeding

Breeding Trivial is only possible using two parents, attempting to use any more will result in additional parents being ignored. The breeding function strips both genomes of their headers, and divides them in to their function parts. This ensures that instructions do not bleed between functions. A random chunk from the beginning of one of the parent's functions is prepended to a random chunk from the end of the other's. A new header is then derived for the child. There is no mutation.

The chunking of functions means that even if a TrivialRobot is selected to breed with itself it is unlikely to produce a clone, encouraging diversity in the population. It also allows for the deletion of gene sequences that are not useful, rather than only allowing the reordering of a set number of parts.

See Figure 11-2 for a graphic representation of the breeding process.

4.2 ParseTree

ParseTree builds on Trivial by reducing the number of things it can do to remove invalid options from the search space, and adding branching logic.

4.2.1 Genotype

Instructions are now represented by a quartet of bytes. If and while were added as valid instructions, allowing branching logic. The argument to each of them is scaled to the range $\mathbb{Z}$ [1,3] and indicates how many of the following lines are owned by the block. The condition is always in the form if(a > b), where a and b are decided by the two bytes following the conditional statement. They are either a number in the range $\mathbb{Z}[0,127]$, or one of the properties of the robot (Heading, TurnRemaining, GunHeading, GunTurnRemaining, DistanceRemaining) scaled to the same range. If a non-branching statement is chosen, those two bytes are ignored. This creates a lot of junk DNA, but makes the math easier.

TurnLeftRadians and TurnGunLeftRadians were replaced with TurnRightRadians and TurnGunRightRadians because, by convention, a positive turn value should indicate a right turn. TurnRadarLeftRadians was removed to ensure the radar and gun are always aligned, making it easier for the robot to target.

4.2.2 Phenotype conversion

Phenotype conversion is essentially the same, with the addition of adding branching statements as described in 4.2.1. When a branching statement is hit, a "{" is added following it, and the argument of the statement is added to the top of a stack of ints. Each statement 1 is subtracted from the top number of the stack, if it equals 0 a corresponding "}" is added to the code. If it reaches the end of a function with blocks still open, it adds "}"s until all the blocks are closed.

4.2.3 Breeding

Breeding is the same as in Trivial

5 Populations

5.1 TrivialPop

TrivialPop maintains a constant number of TrivialRobots. Like TrivialRobot it is a fairly simple system designed primarily because it was quick to get running. Again, it has been saved because TrivialRobot produced interesting results in a test run.

5.1.1 Fitness function

The fitness function chosen was the total score the robot achieved across all battles it performed in that generation. This was chosen both because it is what needs to be optimised for, and already combines several factors from the battle.

5.1.2 Breeder selection

The bottom half of the population do not get to breed. The third quarter breed one, the seventh eighth breed twice, and the eighth eighth breed four times. This produces an effective breeding pool the same size as the initial population.

5.1.3 Breeding

Pairs of robots are randomly selected and removed from the breeding pool. They are bred to produce one offspring, as such the total number of new robots produced every generation is half the population size.

5.1.4 Population carried forward

The top 50% of the robots in a generation are carried forward to the next generation (that is, all the robots who bred). In addition, due to integer division rounding errors fewer than 50% of population size may be produced through breeding. In cases where this occurs a small number of additional robots may be carried forward to keep the population size constant.

5.2 ParseTreePop

ParseTreePop is exactly the same as TrivialPop, except it generates ParseTree robots instead of Trivial robots.

6 Runs

Robots were evolved from scratch using different properties assigned to the EvolutionEnvironment object. At the end of each run the best members from each population are fought against the training set for 5 rounds. The victors from that battle are tested first against five SittingDucks, then against Corners, Crazy, Fire, RamFire, SittingDuck, SpinBot, Target, Tracker, TrackFire, VelociRobot, and Walls to assess their performance.

6.1 Run #1

The system was run with the below parameters

Table 6-1 – Parameters of the first run

Training set	Populations	Generations	Battles per generation	Rounds per battle	Select best
None	TrivialPop(50)	50	1	1	10

The fitness over all generations is graphed below.

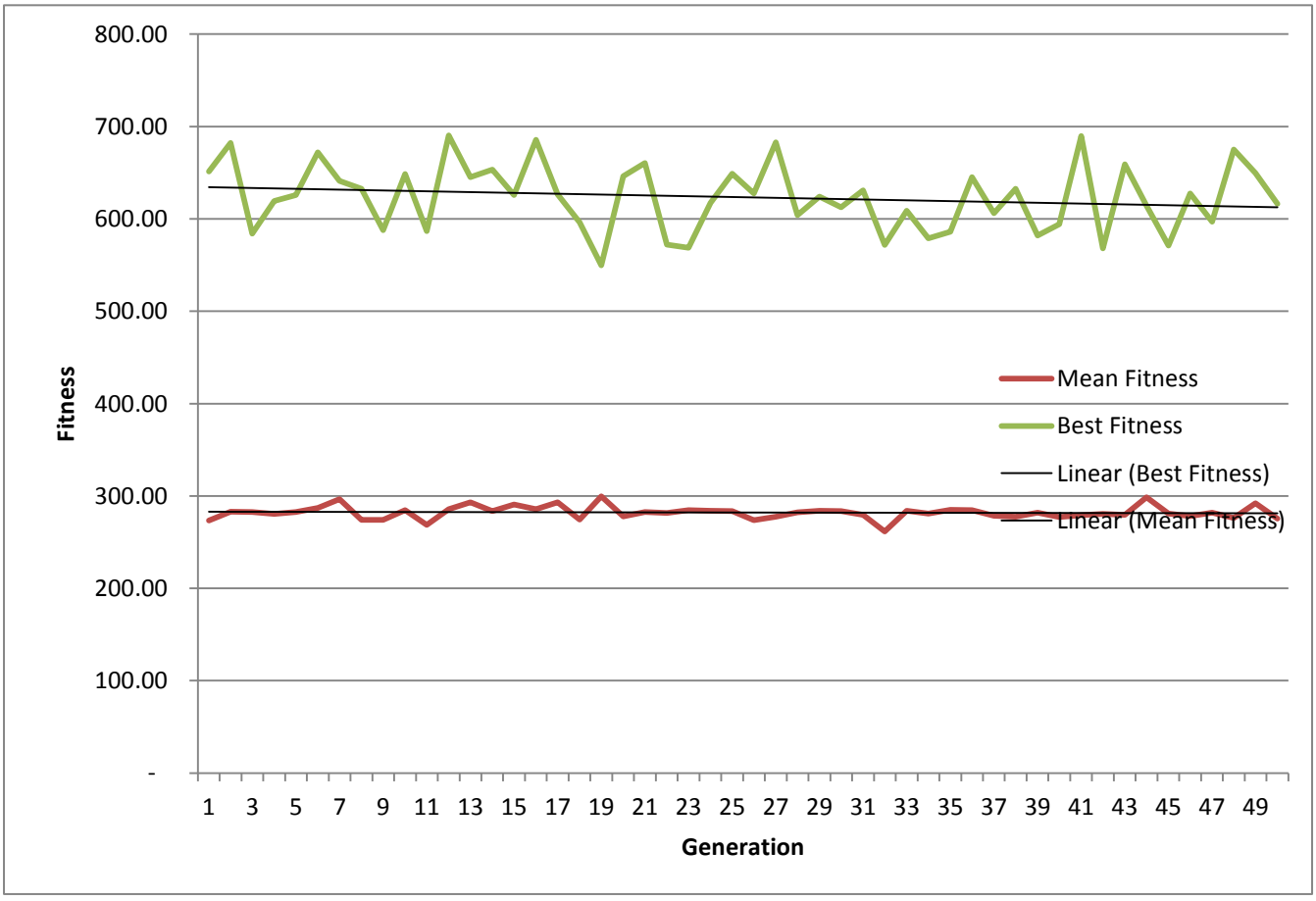

Figure 6-1 - Fitness for Run 1

The fitness (both mean and peak) appear to vary randomly around a set point, and their trendlines show no significant pattern. In retrospect this was the obvious result, as the fitness is the score that the robots had in battles against each other. The score of any given battle is effectively a constant sum game, meaning that so long as the distribution of abilities between any two battles stays constant, the top score will not change, and the average score will remain roughly constant regardless.

In the final battle between the top ten robots of the population, all robots showed very similar behaviour: spinning on the spot until they were hit by a bullet at which point they would move to a new location. This behaviour was clearly selected for, presumably at one point there was a robot that would target the same spot repeatedly and this allowed the robot not to be in it multiple times in a row.

Table 6-2 - Scores from the final battle of Run 1

Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	EvoRobots.ParseTree.B34E6802E43A343CCF33650268E58953	1885 (14%)	1700	180	0	0	5	0	2	0	0
2nd	EvoRobots.ParseTree.bA17E4466F2AC716202758A315015E88	1660 (12%)	1650	0	0	0	10	0	0	2	1
3rd	EvoRobots.ParseTree.aB75BF5DF4717E37C6B908CF2674C27D	1454 (11%)	350	0	1031	31	31	10	0	0	0
4th	EvoRobots.ParseTree.a93315235DE04149967F066A81F62EEC	1391 (10%)	250	0	1087	26	29	0	0	0	0
5th	EvoRobots.ParseTree.iF68143657B3328E52EE21B31387D146	1352 (10%)	1350	0	0	0	2	0	0	2	0
6th	EvoRobots.ParseTree.D1CF6B97368525927938791D6F9D7263	1290 (9%)	1200	90	0	0	0	0	1	1	0
7th	EvoRobots.ParseTree.j0B5B551706CDFFE8B045BCA51084C5A	1250 (9%)	1250	0	0	0	0	0	0	1	1
8th	EvoRobots.ParseTree.hAF10F42A2BF7B13E500A4DD142D5...	1223 (9%)	1100	90	0	0	30	3	1	0	0
9th	EvoRobots.ParseTree.aB773457F519533CB94F047F207EC8F5	1192 (9%)	1100	90	0	0	2	0	1	0	0
10th	EvoRobots.ParseTree.d3595EDDB05E45E9C5927EC1DB7E9...	1025 (7%)	1000	0	0	0	25	0	0	1	1

The scores for this battle are in the above table. The winning robot never successfully hit another robot, instead relying entirely on survival points. Looking at the source code, this behaviour is obvious: setFire is only called inside event handlers that depend on a bullet having been fired, therefore there is no way this robot can fire. Most of the remaining event handlers and the main function contain no meaningful actions (just a lot of spinning of various things) but OnHitByBullet causes the robot to perform a major retreat.

Set against five sitting ducks, there is nothing to trigger any of the event handlers, so all robots just sit there. Against a group of the sample robots, it consistently outperforms SittingDuck as it is able to evade being hit multiple times in a row, and even gets comparable survival scores to a good number of other robots. However it still loses to everything but SittingDuck.

It appears that the robot has optimised for battling against other robots that do not know how to battle. This could be solved by using a training set of other robots that are in every battle, forcing the robots to optimise for battling against them.

6.2 Run #2

The system was run with the below parameters

Table 6-3 – Parameters of the second run

Training set	Populations	Generations	Battles per generation	Rounds per battle	Select best
Corners RamFire Tracker TrackFire	TrivialPop(50)	50	1	1	10

The fitness over all generations is graphed below

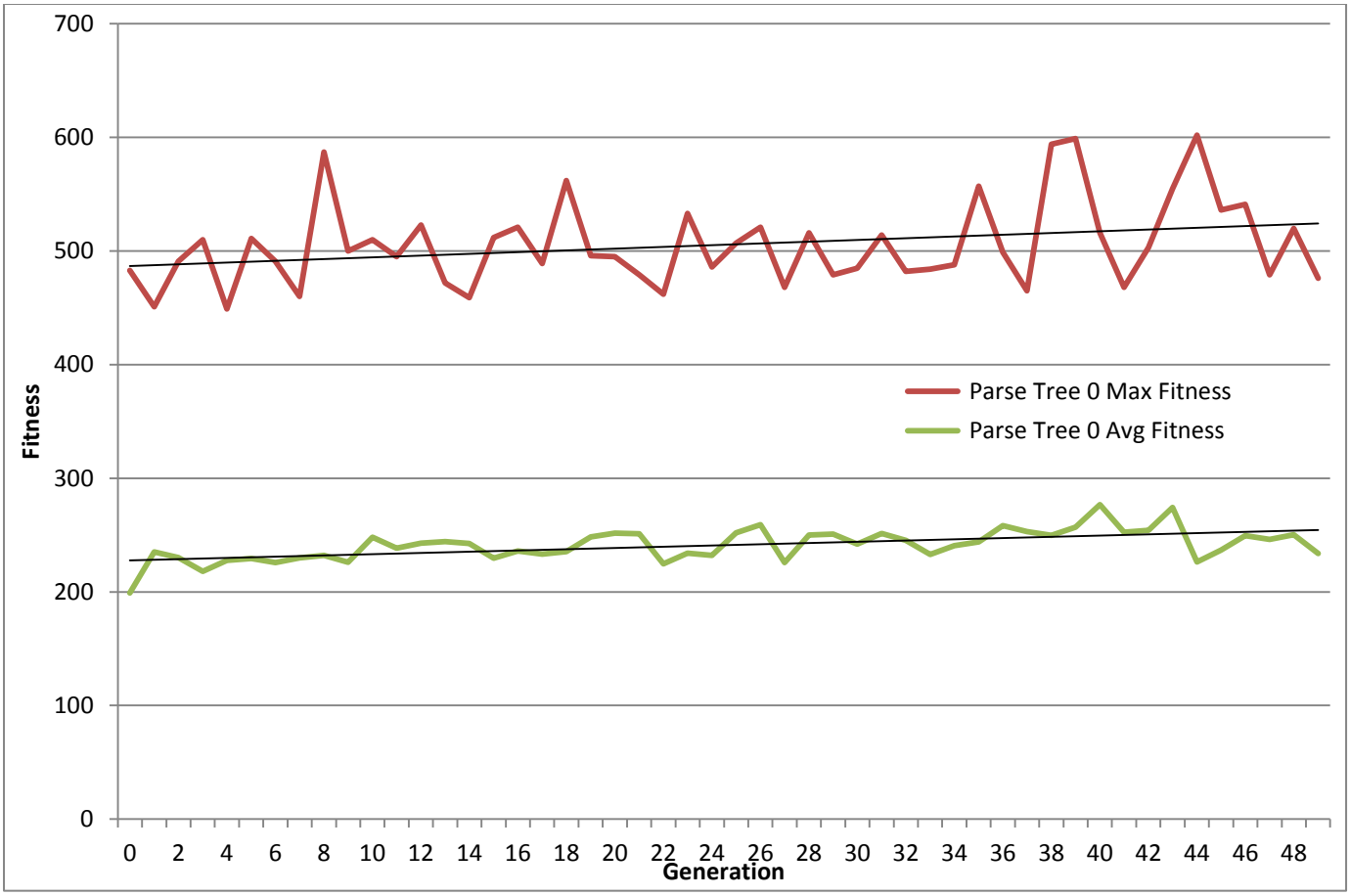

Figure 6-2 Fitness for Run 2

In this case the fitness start significantly lower than in Run #1, which is to be expected as the total score achievable in each battle is the same, but is divided across both the population and the training set. It then makes a slight upward trend, though it is well within the range of random variation so should not be considered significant.

In the final battle the robots were very good at surviving. Now that they were actively hunted during their evolution they no longer still, instead driving around and making some very interesting decisions that allow them to avoid a large number of bullets. Their score from damaging other robots is increased from previous rounds, likely because they are actually able to shoot this round. The scores from this battle are listed below.

Table 6-4 – Scores from the final battle of Run #2

Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	sample.TrackFire	3492 (21%)	1850	270	1185	187	0	0	3	1	0
2nd	sample.Corners	2498 (15%)	1500	90	875	33	0	0	1	0	1
3rd	sample.Tracker	2225 (13%)	1250	90	742	84	48	11	1	0	1
4th	EvoRobots.ParseTree.j8410F160805A3D4F816DF5D5C885297	1638 (10%)	1400	0	207	0	31	0	0	1	0
5th	sample.RamFire	1564 (9%)	750	0	570	21	163	60	0	0	0
6th	EvoRobots.ParseTree.bB07B4F81702BE180DB43F9567734DB2	1364 (8%)	1200	0	147	2	16	0	0	1	1
7th	EvoRobots.ParseTree.C9E11BA25316BE5A0026EFDF0046C66C	1207 (7%)	1000	0	178	0	29	0	0	0	2
8th	EvoRobots.ParseTree.eF07D05359B43BEDD9580D77CEDFE7B0	1071 (6%)	950	0	91	4	26	0	0	1	0
9th	EvoRobots.ParseTree.c9DFD03A9C0C21AA7CDB069DDD2F4...	992 (6%)	650	0	171	0	170	0	0	1	0
10th	EvoRobots.ParseTree.b69E26AC4AAC11C433D5F5B6FC4C64FA	814 (5%)	650	0	116	0	48	0	0	0	0

Set against five sitting ducks it is clear why the robot did not score very well: it cannot aim. It spends most of the match driving around the other robots, shooting in random directions until finally becoming disabled and losing to the sitting ducks. It is likely that in the previous run shooting was bred out due to this inability to aim. The fact that all the robots were driving around this time means that there's every chance the bullets that hit did so because the target drove in to them. Once again, in a battle against most of the sample robots it consistently outperforms SittingDuck, but not much else. It out survives a few others, but is let down by its inability to aim.

It also appears to have selected for a longer genome than it did with no training set, this is presumably because the task of avoiding the other robots is more complicated now it is being hunted.

This is likely caused by the combined factors of being able to independently move its radar (and therefore had no guarantee that it was aiming at what it was looking at) and no way of gaining information about the outside world aside from through events. It was decided to allow the radar to be moved because a valid tactic could be generated by having the radar offset from the gun (why else would the Robocode designers have put the option in there), but this made the search space too large, and added too many invalid configurations. As such, this will be changed. There will also be some manner of branching logic used, allowing the robot to make decisions based on its current state.

6.3 Run #3

The system was run with the below configuration

Table 6-5 – Parameters of the third run

Training set	Populations	Generations	Battles per generation	Rounds per battle	Select best
Corners RamFire Tracker TrackFire	TrivialPop(50) ParseTreePop(50)	50	1	1	10

The fitness across all generations is graphed below

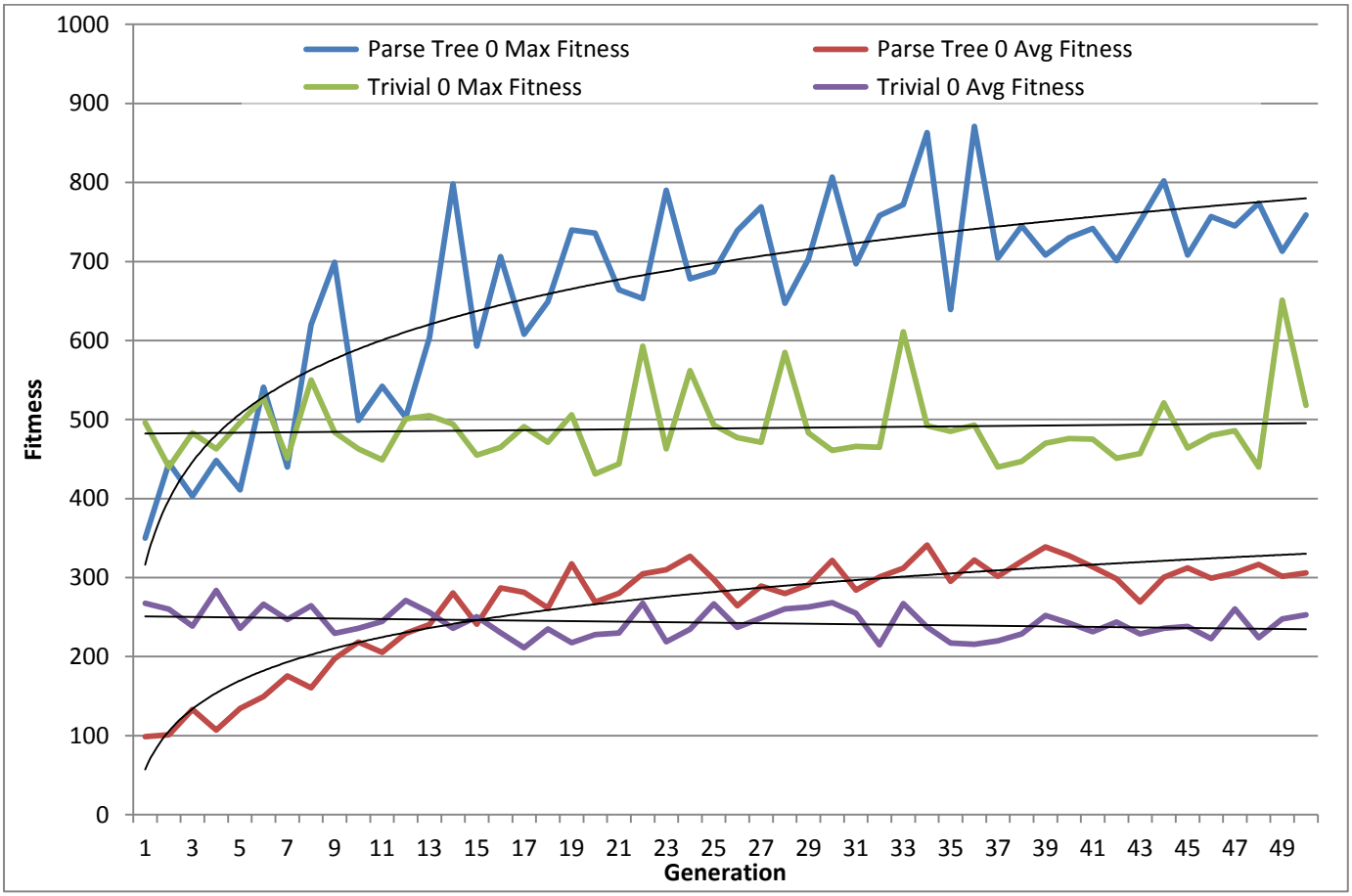

Figure 6-3 - Fitness for Run #3

Trivial once again shows no trend in fitness, though its average does drop slightly around generation 10 as ParseTree surpasses it. This lack of trend across all generations is interesting, because in spite of it there are properties and behaviours that make it better at surviving which are selected for, and vary based on the environment (for example it selected for never moving or shooting when it was on its own, but added them back in when it was being hunted).

It is unclear whether this is because the fitness is calculated as a result of a constant sum game, or the robot design is so bad that randomness is a greater factor than construction in it.

ParseTree starts much lower than trivial, but then steadily increases until it stabilises around generation 30. There are many possible obviously invalid configurations ParseTree can make involving infinite loops that do not contain an Execute() command, this will cause the robot to become disabled almost immediately. It is likely the elimination of such configurations leads to the sudden sharp rise at the beginning.

Table 6-6 - Scores from the final battle of Run #3

Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	EvoRobots.ParseTree.C7FA42ED5B30E799B2CF7E3F3A0B03...	2589 (15%)	1700	180	533	58	110	7	2	0	2
2nd	sample.Tracker	2452 (14%)	1500	90	756	66	40	0	1	1	1
3rd	sample.TrackFire	2434 (14%)	1450	90	780	115	0	0	1	0	1
4th	sample.RamFire	2246 (13%)	1100	0	759	14	239	134	0	1	0
5th	EvoRobots.ParseTree.f2CC23F79542CC8DE1FE2B11FA3DC...	2078 (12%)	1300	90	511	25	132	20	1	1	0
6th	EvoRobots.ParseTree.B8091CF1243841765C72720324A37CA4	1368 (8%)	950	0	388	6	17	8	0	1	0
7th	EvoRobots.Trivial.F9563721342D5AD4085B7A630F1288BD	1292 (7%)	1000	0	217	0	76	0	0	1	0
8th	sample.Corners	1202 (7%)	750	0	451	1	0	0	0	0	0
9th	EvoRobots.Trivial.b1205F5B8DFE5A4C965F4D859E5F738A	992 (6%)	850	0	74	0	67	0	0	0	1
10th	EvoRobots.Trivial.A1562CDACC3C9829B800411C4EADF3D4	855 (5%)	600	0	166	4	85	0	0	0	0

The scores of the final run are listed above. The overall victor was an instance of the ParseTree robot, with all instances of ParseTree coming above all instances of Trivial. While ParseTree is still not amazing at shooting, it managed to achieve fairly good scores at it. Trivial did not even manage to compete in the same ballpark as ParseTree or the training set when it comes to survival scores, the one thing it was able to do in previous iterations.

It is possible that in this case coevolution did not favour Trivial, as ParseTree so quickly outclassed it, it is likely that ParseTree evolved to destroy Trivial and gain easy points.

Against five SittingDucks, Trivial had a similar performance to the previous run, expending all of its energy without taking down a single opponent, while ParseTree managed to consistently kill every SittingDuck. It would traverse the battlefield until it found a target, at which point it circles, continually shooting at it.

The AdjustForTurn property of the robot (which automatically changes the heading of the gun to compensate for the turn of the robot, meaning a gun pointing north will always point north no matter what the robot does) was not activated, but the robot emulated it by constantly moving its gun against the direction of turn, allowing it to track a target while staying in motion to avoid being hit. That a behaviour was generated and selected for that the Robocode designers knew would be useful shows that this is a good genetic algorithm.

Putting them both in a battle against most of the sample robots, and once again Trivial outperforms only the sitting duck, whereas ParseTree manages to place 6th out of 13, even managing to come first on one of the rounds. It is clearly not yet a perfect robot, but is well on its way to being a competitive one.

7 Comparison

In addition to the champions from all of the runs, the best robots produced by Jonathon Welford-Costelloe will also be assed. The full details of how his robots were evolved are available in his report. To test these robots, first they were all put in to a single large battle after five rounds, and then they were each in turn put up against five SittingDucks to see how quickly they could defeat them all. The results of both challenges are recorded below.

Table 7-1 – Results from the large battle

Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	EvoRobots.ParseTree.C7FA42ED5B30E799B2CF7E3F3A0B03...	2071 (33%)	1050	200	708	78	35	0	4	0	0
2nd	EvoRobots.JonathanWelfordCostelloe	1434 (23%)	800	0	576	48	10	0	0	4	0
3rd	EvoRobots.MyRobot_MIX_f9ef5_20_10	887 (14%)	550	0	311	11	14	0	0	0	3
4th	EvoRobots.ParseTree.j8410F160805A3D4F816DF5D5C885297	737 (12%)	500	0	209	0	28	0	0	1	0
5th	EvoRobots.ParseTree.B34E6802E43A343CCF33650268E58953	703 (11%)	600	50	0	0	53	0	1	0	1
6th	EvoRobots.Trivial.F9563721342D5AD4085B7A630F1288BD	507 (8%)	250	0	215	3	38	0	0	0	1

Table 7-2 – Results from the SittingDuck challenge

Robot	Recorded Name	Rounds to complete
Jon1	JonathanWelfordCostelloe	3899
Jon2	MyRobot_MIX_f9ef5_20_10	3274
Run 1 Champion	ParseTree.B34…	DNF
Run 2 Champion	ParseTree.j8410F…	DNF
Run 3 Champion (Trivial)	Trivial.F956…	DNF
Run 3 Champion (Parse Tree)	ParseTree.C7FA…	2346

On both challenges the ParseTree from run 3 clearly outclassed all of its rivals, both hunting down the SittingDucks significantly faster, and consistently winning in the large battle, while all the Trivial robots utterly failed. This gives confidence that the algorithm used to produce ParseTree will produce robots on a sufficiently good local maximum.

Comparing Jon1 to ParseTree is interesting because Jon1 independently evolved the shooting and moving mechanics, whereas they were evolved together in ParseTree. One would expect Jon1 to be significantly better at both of these, yet ParseTree got 250 more survival points and 132 more Bullet damage. This is likely either because the mechanics developed one that could shoot well while staying still, or because ParseTree was evolved over a greater number of generations.

8 Future work

Work that was going to be performed, but due to time restrictions was not is listed in this section

8.1 Adding mutation

Currently the robots have no mutation. This means that if a pattern does not exist in the genes of one of the robots that perform in the top half of the first generation it will never appear in any robots. This is particularly evident in the arguments (size of turn/power of bullet etc) and the branch conditions. Adding mutation in will fix that and allow more of the search space to be explored.

8.2 Neural Net robot

Work began on creating a robot that uses a neural net to control it, as it is a good way of allowing the robot to act based on input from the environment, but it was not finished in time for Run #3 so had to be excluded.

8.3 Further Coevolution

It was hoped to perform more coevolution, both adjusting other people's genetic algorithms to fit the framework used in this piece to see how they coevolve against more advanced systems, and creating multicultural populations (i.e. populations operating under a single genetic strategy with one fitness function, but owning multiple types of robots) . A multicultural population could produce interesting results, as it would favour robots that optimise early and essentially give them a greater share of the resources. It would be interesting to see how Run #3 would have been different in this case, and whether Trivial would have fared better in the end.

8.4 Differing population management strategies

As one of the types of robot developed is essentially an early stage of the other, they both use the same population management strategy. It was also decided not to vary too many things between any two runs, so one can be sure what changes it was that caused the improvement/decline. It would have been good to use an entirely different population management strategy with the same robots, so one could see how it affects both the fitness, and the rate of change of fitness.

8.5 Measuring diversity

As well as recording the mean and peak fitness at the end of each generation, it would be good to measure the diversity of the population, so that assurances can be given that the reason all the robots display similar traits was

because they were gradually selected for after many generations, rather than just winning the lottery after the first couple and being the only behaviours left in the gene pool.

8.6 Greater training set

Once a genetic algorithm that created robots that could outperform the inbuilt sample robots was created, the training set would have been adjusted to include the higher ranking robots on RoboRumble.

9 Innovation

The most innovative feature of this work is the creation of the framework that allows for the creation and coevolution of arbitrary genetic algorithms described in 3. Also, comparing it against not only systems produced by it, but by other people as well.

10 Works Cited

[1] Robocode, “RoboWiki Main Page,” 19 March 2013. [Online]. Available: http://robowiki.net/wiki/Main_Page. [Accessed 20 March 2013].

[2] Robocode, “RoboRumble,” 16 May 2012. [Online]. Available: http://robowiki.net/wiki/RoboRumble. [Accessed 20 March 2013].

[3] Imperial College London, Department of Computing, “Genetic Algorithms,” [Online]. Available: http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/tcw2/report.html. [Accessed 20 March 2013].

[4] StackOverflow, “What is the Difference between a Heuristic and an Algorithm?,” 4 May 2012. [Online]. Available: http://stackoverflow.com/questions/2334225/what-is-the-difference-between-a-heuristic-and-an-algorithm. [Accessed 20 March 2013].

11 Appendix 1: Additional figures

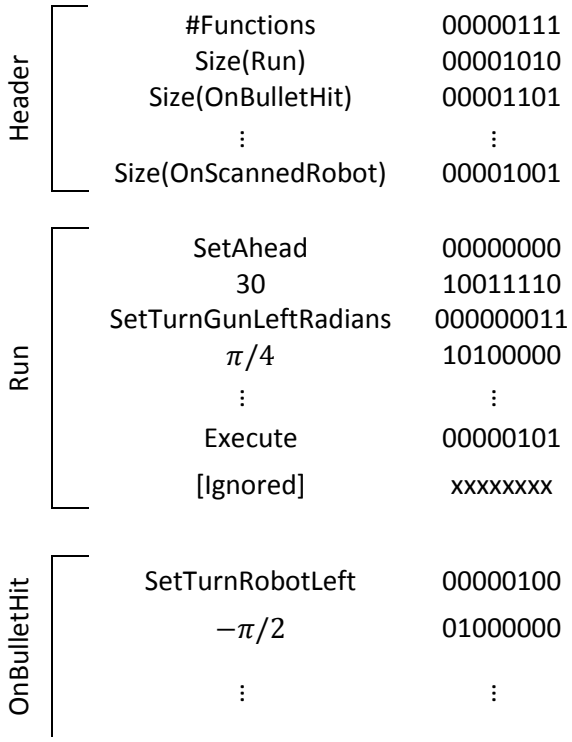

Figure 11-1 - Genotype example of TrivialRobot

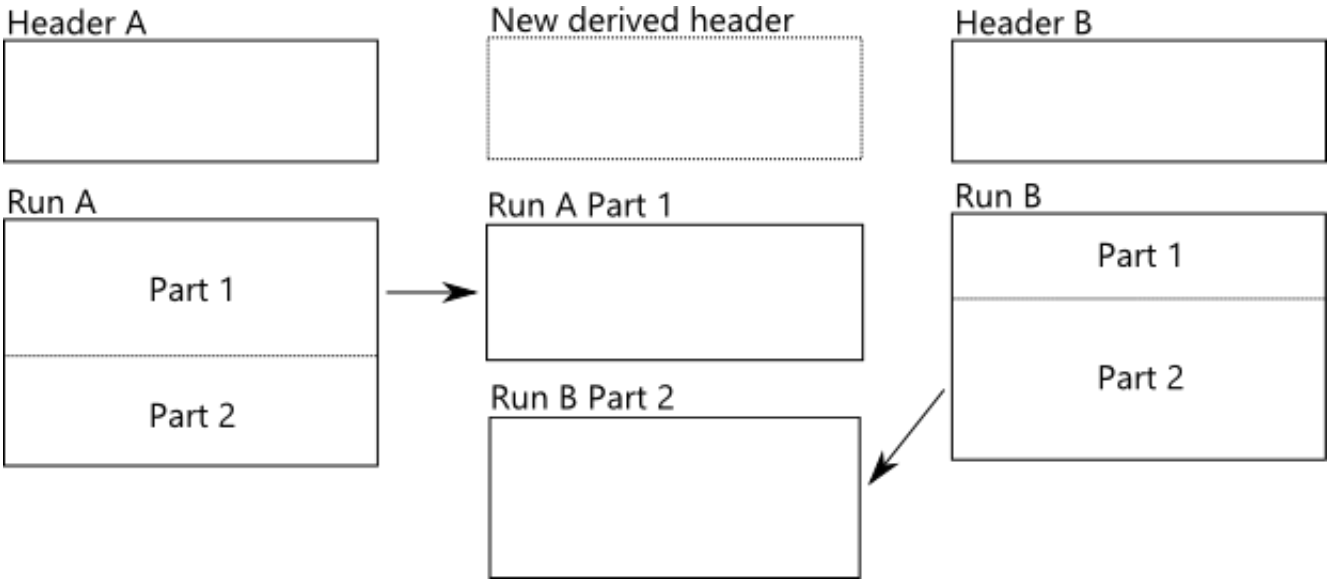

Figure 11-2 Breeding behaviour of TrivialRobot

12 Appendix 2: Source code of champions

12.1 Run 1

```
// Access to standard .NET System
using System;
using System.Collections.Generic;
using System.Text;

// Access to the public Robocode API
using Robocode;
// The namespace with your initials, in this case FNL is the initials
namespace EvoRobots.ParseTree
{
    // The name of your robot is MyRobot, and the robot type is Robot
    class B34E6802E43A343CCF33650268E58953 : AdvancedRobot
    {
        // The main method of your robot containing robot logics
        public override void Run()
        {
            while(true)
{
    SetTurnRadarLeftRadians(1.27627201552085);
Execute();
SetTurnLeftRadians(-1.3989904785517);

}
        }

        public override void OnBulletHit(BulletHitEvent evnt)
        {
            SetFire(0);
SetFire(2.91764705882353);
SetFire(2.02352941176471);
SetTurnRadarLeftRadians(-2.40528187540469);
SetTurnRadarLeftRadians(0.122718463030851);
SetAhead(-9);
SetFire(2.4);
SetAhead(-18);
SetTurnGunLeftRadians(-3.09250526837745);
Execute();
SetFire(0);
SetFire(2.91764705882353);
SetFire(2.02352941176471);
SetTurnRadarLeftRadians(-2.40528187540469);
SetTurnGunLeftRadians(-1.52170894158256);

        }

        public override void OnBulletHitBullet(BulletHitBulletEvent e)
        {
            SetAhead(42);
SetTurnGunLeftRadians(0.417242774304894);
Execute();
Execute();
SetTurnRadarLeftRadians(-1.12900985988383);

        }

        public override void OnBulletMissed(BulletMissedEvent e)
        {
            SetTurnGunLeftRadians(-1.96349540849362);
SetFire(2.34117647058824);
SetTurnGunLeftRadians(-1.96349540849362);
SetTurnLeftRadians(1.17809724509617);
SetFire(2.36470588235294);
```

```
        }

        public override void OnHitByBullet(HitByBulletEvent e)
        {
            Execute();
SetAhead(-101);

        }

        public override void OnHitRobot(HitRobotEvent e)
        {
            SetTurnRadarLeftRadians(1.27627201552085);
SetTurnRadarLeftRadians(1.27627201552085);
SetTurnLeftRadians(1.05537878206532);
SetTurnRadarLeftRadians(1.27627201552085);
SetAhead(75);

        }

        public override void OnHitWall(HitWallEvent e)
        {
            SetTurnRadarLeftRadians(-2.03712648631213);
SetTurnRadarLeftRadians(-2.03712648631213);
SetTurnRadarLeftRadians(-2.03712648631213);
SetTurnGunLeftRadians(0.662679700366597);

        }

        // Robot event handler, when the robot sees another robot
        public override void OnScannedRobot(ScannedRobotEvent e)
        {
            Execute();

        }
    }
}
```

12.2 Run #2

```
// Access to standard .NET System
using System;
using System.Collections.Generic;
using System.Text;

// Access to the public Robocode API
using Robocode;
// The namespace with your initials, in this case FNL is the initials
namespace EvoRobots.ParseTree
{
    // The name of your robot is MyRobot, and the robot type is Robot
    class j8410F160805A3D4F816DF5D5C885297 : AdvancedRobot
    {
        // The main method of your robot containing robot logics
        public override void Run()
        {
            while(true)
{
    Execute();
Execute();
SetAhead(66);
SetTurnGunLeftRadians(2.57708772364788);
SetFire(0.211764705882353);
Execute();
Execute();
SetFire(2.48235294117647);
SetFire(0.682352941176471);
Execute();
SetTurnGunLeftRadians(2.57708772364788);
SetTurnLeftRadians(1.42353417115788);
SetAhead(88);
Execute();
Execute();
SetTurnLeftRadians(-2.7243498792849);
Execute();
Execute();
SetFire(2.63529411764706);
SetTurnGunLeftRadians(-0.171805848243192);
Execute();
SetTurnGunLeftRadians(-1.3989904785517);
Execute();
Execute();
SetAhead(66);
SetTurnRadarLeftRadians(-1.05537878206532);
Execute();
Execute();
Execute();
SetAhead(88);
Execute();
SetTurnRadarLeftRadians(0.171805848243192);

}
        }

        public override void OnBulletHit(BulletHitEvent evnt)
        {
            Execute();
SetFire(0.552941176470588);
SetTurnRadarLeftRadians(1.42353417115788);

        }

        public override void OnBulletHitBullet(BulletHitBulletEvent e)
        {
            SetTurnGunLeftRadians(0.392699081698724);
```

```
SetAhead(-37);
SetFire(2.28235294117647);
SetTurnRadarLeftRadians(-2.69980618667873);
SetAhead(-37);
SetTurnLeftRadians(1.49716524897639);
SetTurnGunLeftRadians(-0.539961237335746);
SetTurnGunLeftRadians(-2.74889357189107);
SetAhead(-6);
SetAhead(-55);
SetFire(2.28235294117647);
SetTurnRadarLeftRadians(-2.69980618667873);
Execute();
SetTurnGunLeftRadians(1.47262155637022);
SetAhead(61);
SetTurnRadarLeftRadians(1.91440802328128);
SetTurnLeftRadians(2.92069942013426);
SetTurnRadarLeftRadians(-2.69980618667873);
Execute();
SetTurnLeftRadians(2.77343726449724);
SetTurnGunLeftRadians(-2.74889357189107);
SetTurnLeftRadians(1.49716524897639);
SetFire(1.71764705882353);

        }

        public override void OnBulletMissed(BulletMissedEvent e)
        {
            Execute();
SetFire(1.21176470588235);
SetFire(0.0235294117647059);
SetTurnGunLeftRadians(-0.0736310778185108);
SetAhead(114);

        }

        public override void OnHitByBullet(HitByBulletEvent e)
        {
            SetTurnGunLeftRadians(-3.01887419055894);
SetTurnGunLeftRadians(-2.79798095710341);
SetTurnLeftRadians(-2.57708772364788);
SetTurnLeftRadians(0.294524311274043);
SetTurnGunLeftRadians(1.17809724509617);
SetTurnGunLeftRadians(2.82252464970958);
Execute();

        }

        public override void OnHitRobot(HitRobotEvent e)
        {
            Execute();
Execute();
SetTurnLeftRadians(2.11075756413064);
SetFire(2.47058823529412);
SetFire(2.35294117647059);
SetTurnRadarLeftRadians(1.44807786376405);
SetFire(2.70588235294118);
SetTurnRadarLeftRadians(1.84077694546277);
Execute();
SetFire(2.97647058823529);
SetTurnRadarLeftRadians(1.44807786376405);
SetFire(2.70588235294118);
SetTurnRadarLeftRadians(1.84077694546277);
Execute();
SetFire(2.97647058823529);
SetTurnRadarLeftRadians(1.44807786376405);
SetFire(2.35294117647059);
SetFire(0.0941176470588235);
```

```
        }

        public override void OnHitWall(HitWallEvent e)
        {
            Execute();
SetTurnGunLeftRadians(-2.38073818279852);
SetFire(2.71764705882353);
SetFire(2.62352941176471);
SetTurnLeftRadians(1.5707963267949);
SetTurnGunLeftRadians(1.32535940073319);
SetFire(2.62352941176471);
SetTurnLeftRadians(1.5707963267949);
SetTurnGunLeftRadians(1.32535940073319);
SetFire(2.62352941176471);
SetTurnLeftRadians(1.5707963267949);
SetFire(2.62352941176471);
SetTurnLeftRadians(1.5707963267949);
SetTurnRadarLeftRadians(-0.417242774304894);

        }

        // Robot event handler, when the robot sees another robot
        public override void OnScannedRobot(ScannedRobotEvent e)
        {
            SetTurnLeftRadians(2.30710710498);
SetTurnGunLeftRadians(0.613592315154257);

        }
    }
}
```

12.3 Run #3

12.3.1 ParseTree

```
// Access to standard .NET System
using System;
using System.Collections.Generic;
using System.Text;

// Access to the public Robocode API
using Robocode;
// The namespace with your initials, in this case FNL is the initials
namespace EvoRobots.ParseTree
{
    // The name of your robot is MyRobot, and the robot type is Robot
    class C7FA42ED5B30E799B2CF7E3F3A0B034C : AdvancedRobot
    {
        // The main method of your robot containing robot logics
        public override void Run()
        {
            while(true)
{
    SetTurnGunRightRadians(-0.171805848243192);
SetTurnGunRightRadians(-2.30710710498);
SetTurnGunRightRadians(-0.171805848243192);
Execute();

}
        }

        public override void OnBulletHit(BulletHitEvent evnt)
        {
            SetFire(1.11764705882353);
SetFire(1.11764705882353);
Execute();
while((((GunTurnRemainingRadians/Math.PI) + 1)*64) > (((GunTurnRemainingRadians/Math.PI) + 1)*64)){
SetTurnGunRightRadians(-0.9081166264283);
}

        }

        public override void OnBulletHitBullet(BulletHitBulletEvent e)
        {
            SetTurnRightRadians(-1.37444678594553);
SetTurnRightRadians(1.54625263418873);
if(((GunHeadingRadians/Math.PI)*64) > (((TurnRemainingRadians/Math.PI) + 1)*64)){
Execute();
}
Execute();
SetAhead(-88);
if(((GunHeadingRadians/Math.PI)*64) > (((TurnRemainingRadians/Math.PI) + 1)*64)){
SetAhead(-88);
}
SetAhead(-88);
if(((GunHeadingRadians/Math.PI)*64) > (((TurnRemainingRadians/Math.PI) + 1)*64)){
if(((GunHeadingRadians/Math.PI)*64) > (((TurnRemainingRadians/Math.PI) + 1)*64)){
SetAhead(-78);
}
SetTurnRightRadians(-1.37444678594553);
}
if(((GunHeadingRadians/Math.PI)*64) > (((TurnRemainingRadians/Math.PI) + 1)*64)){
SetTurnRightRadians(1.54625263418873);
}
Execute();
Execute();
SetTurnRightRadians(-1.37444678594553);
SetAhead(-88);
if(((GunHeadingRadians/Math.PI)*64) > (((TurnRemainingRadians/Math.PI) + 1)*64)){
```

```
SetAhead(-78);
}
Execute();
if(((GunHeadingRadians/Math.PI)*64) > (((TurnRemainingRadians/Math.PI) + 1)*64)){
SetTurnRightRadians(-1.37444678594553);
}
SetAhead(-88);
if(((GunHeadingRadians/Math.PI)*64) > (((TurnRemainingRadians/Math.PI) + 1)*64)){
SetFire(0.705882352941177);
}

        }

        public override void OnBulletMissed(BulletMissedEvent e)
        {
            SetFire(0.2);
SetFire(0.2);
Execute();

        }

        public override void OnHitByBullet(HitByBulletEvent e)
        {
            SetTurnGunRightRadians(1.3989904785517);
SetTurnGunRightRadians(1.3989904785517);
SetTurnGunRightRadians(1.3989904785517);
SetAhead(-54);

        }

        public override void OnHitRobot(HitRobotEvent e)
        {
            if(((HeadingRadians/Math.PI)*64) > ((GunHeadingRadians/Math.PI)*64)){
if(((GunHeadingRadians/Math.PI)*64) > (DistanceRemaining/2 + 64)){
if(((HeadingRadians/Math.PI)*64) > (((GunTurnRemainingRadians/Math.PI) + 1)*64)){
if(((HeadingRadians/Math.PI)*64) > (((GunTurnRemainingRadians/Math.PI) + 1)*64)){
SetTurnGunRightRadians(-1.17809724509617);
if(((GunHeadingRadians/Math.PI)*64) > (DistanceRemaining/2 + 64)){
SetTurnGunRightRadians(-0.9081166264283);
}
}
}
}
}

        }

        public override void OnHitWall(HitWallEvent e)
        {
            Execute();

        }

        // Robot event handler, when the robot sees another robot
        public override void OnScannedRobot(ScannedRobotEvent e)
        {
            SetTurnRightRadians(0.711767085578938);
SetTurnGunRightRadians(2.82252464970958);
SetTurnRightRadians(-1.10446616727766);
SetFire(1.31764705882353);
SetTurnGunRightRadians(2.82252464970958);
SetTurnRightRadians(0.711767085578938);
SetTurnGunRightRadians(2.82252464970958);
SetFire(1.31764705882353);
if(((HeadingRadians/Math.PI)*64) > ((GunHeadingRadians/Math.PI)*64)){
SetFire(1.31764705882353);
}
SetTurnGunRightRadians(-0.368155389092554);
```

```
SetTurnRightRadians(-1.10446616727766);
SetTurnRightRadians(-1.10446616727766);
SetFire(1.31764705882353);
SetFire(1.31764705882353);
SetTurnGunRightRadians(2.82252464970958);
if(((GunHeadingRadians/Math.PI)*64) > (((TurnRemainingRadians/Math.PI) + 1)*64)){
SetFire(1.31764705882353);
SetTurnGunRightRadians(2.82252464970958);
}
SetTurnGunRightRadians(2.82252464970958);
Execute();
SetAhead(41);

        }
    }
}
```

12.3.2 Trivial

```
// Access to standard .NET System
using System;
using System.Collections.Generic;
using System.Text;

// Access to the public Robocode API
using Robocode;
// The namespace with your initials, in this case FNL is the initials
namespace EvoRobots.Trivial
{
    // The name of your robot is MyRobot, and the robot type is Robot
    class F9563721342D5AD4085B7A630F1288BD : AdvancedRobot
    {
        // The main method of your robot containing robot logics
        public override void Run()
        {
            while(true)
{
    SetFire(0.129411764705882);
SetTurnGunLeftRadians(0.466330159517235);
Execute();
SetAhead(-15);
Execute();
SetFire(2.17647058823529);
SetTurnGunLeftRadians(0.245436926061703);
Execute();
SetTurnLeftRadians(0.834485548609789);
SetTurnLeftRadians(0.834485548609789);
SetTurnRadarLeftRadians(-0.294524311274043);
SetAhead(-15);
Execute();
SetTurnLeftRadians(2.92069942013426);
Execute();

}
        }

        public override void OnBulletHit(BulletHitEvent evnt)
        {
            SetTurnRadarLeftRadians(0.0490873852123405);
SetTurnRadarLeftRadians(-1.20264093770234);
SetFire(1.57647058823529);
SetTurnLeftRadians(2.33165079758617);
SetTurnRadarLeftRadians(-1.20264093770234);
SetTurnLeftRadians(0.098174770424681);
SetTurnLeftRadians(2.33165079758617);
SetTurnRadarLeftRadians(2.7243498792849);

        }

        public override void OnBulletHitBullet(BulletHitBulletEvent e)
        {
            SetTurnRadarLeftRadians(-2.11075756413064);
Execute();
SetTurnRadarLeftRadians(-2.11075756413064);
SetTurnRadarLeftRadians(-1.84077694546277);
SetFire(1.85882352941176);
Execute();
SetTurnRadarLeftRadians(-2.11075756413064);
SetTurnRadarLeftRadians(-1.84077694546277);
SetFire(1.85882352941176);
Execute();
SetTurnRadarLeftRadians(-1.86532063806894);
Execute();
SetTurnRadarLeftRadians(-1.86532063806894);
```

```
SetTurnRadarLeftRadians(-1.86532063806894);
SetTurnRadarLeftRadians(-1.86532063806894);
SetFire(0.435294117647059);

        }

        public override void OnBulletMissed(BulletMissedEvent e)
        {
            SetTurnLeftRadians(0.859029241215959);
SetFire(1.25882352941176);
SetTurnLeftRadians(-0.269980618667873);
SetTurnLeftRadians(2.92069942013426);
SetTurnRadarLeftRadians(1.98803910109979);
SetFire(1.52941176470588);
SetTurnLeftRadians(-0.269980618667873);
SetTurnRadarLeftRadians(1.98803910109979);
Execute();

        }

        public override void OnHitByBullet(HitByBulletEvent e)
        {
            SetFire(0.176470588235294);
SetTurnLeftRadians(-1.37444678594553);
SetTurnLeftRadians(-1.37444678594553);
SetTurnRadarLeftRadians(-1.25172832291468);
SetAhead(-6);
SetTurnRadarLeftRadians(-1.25172832291468);
SetTurnLeftRadians(-1.37444678594553);
SetTurnLeftRadians(-1.37444678594553);
SetTurnRadarLeftRadians(-1.25172832291468);
SetAhead(-6);
SetTurnRadarLeftRadians(-1.25172832291468);
SetTurnLeftRadians(-1.37444678594553);
SetFire(2.74117647058824);
SetTurnRadarLeftRadians(2.62617510886022);

        }

        public override void OnHitRobot(HitRobotEvent e)
        {
            Execute();
SetAhead(-5);
SetTurnGunLeftRadians(-0.368155389092554);
SetTurnLeftRadians(3.04341788316511);
SetTurnLeftRadians(0.613592315154257);
SetFire(2.49411764705882);
SetAhead(-5);
Execute();
SetFire(2.49411764705882);
Execute();
Execute();
SetTurnLeftRadians(0.613592315154257);
Execute();
SetAhead(114);
SetAhead(-5);
SetTurnGunLeftRadians(-0.368155389092554);
SetTurnLeftRadians(3.04341788316511);
SetAhead(114);
Execute();
SetTurnGunLeftRadians(2.62617510886022);

        }

        public override void OnHitWall(HitWallEvent e)
        {
            SetTurnGunLeftRadians(-1.5707963267949);
SetTurnGunLeftRadians(1.10446616727766);
```

```
SetTurnRadarLeftRadians(2.77343726449724);
SetTurnRadarLeftRadians(2.77343726449724);
SetAhead(-85);
SetFire(0.282352941176471);
SetTurnGunLeftRadians(-1.5707963267949);
SetAhead(-85);
SetTurnGunLeftRadians(1.10446616727766);
SetTurnRadarLeftRadians(2.77343726449724);
SetTurnGunLeftRadians(-1.5707963267949);
SetFire(1.41176470588235);

        }

        // Robot event handler, when the robot sees another robot
        public override void OnScannedRobot(ScannedRobotEvent e)
        {
            SetAhead(-83);

        }
    }
}
```