GP-Robocode: Using Genetic Programming to Evolve Robocode Players

Yehonatan Shichel, Eran Ziserman, and Moshe Sipper

Department of Computer Science, Ben-Gurion University, Israel
{shichel,eranz,sipper}@cs.bgu.ac.il, www.moshesipper.com

Abstract. This paper describes the first attempt to introduce evolutionarily designed players into the international Robocode league, a simulation-based game wherein robotic tanks fight to destruction in a closed arena. Using genetic programming to evolve tank strategies for this highly active forum, we were able to rank third out of twenty-seven players in the category of HaikuBots. Our GPBot was the only entry not written by a human.

"I wonder how long handcoded algorithms will remain on top."
Developer's comment at a Robocode discussion group,
robowiki.net/cgi-bin/robowiki?GeneticProgramming

1 Introduction

The strife between humans and machines in the arena of intelligence has fertilized the imagination of many an artificial-intelligence (AI) researcher, as well as numerous science fiction novelists. Since the very early days of AI, the domain of games has been considered as epitomic and prototypical of this struggle. Designing a machine capable of defeating human players is a prime goal in this area: From board games, such as chess and checkers, through card games, to computer adventure games and 3D shooters, AI plays a central role in the attempt to see machine beat man at his own game—literally.

Program-based games are a subset of the domain of games in which the human player has no direct influence on the course of the game; rather, the actions during the game are controlled by programs that were written by the (usually human) programmer. The program responds to the current game environment, as captured by its percepts, in order to act within the simulated game world. The winner of such a game is the programmer who has provided the best program; hence, the programming of game strategies is often used to measure the performance of AI algorithms and methodologies. Some famous examples of program-based games are *RoboCup* (www.robocup.org), the robotic soccer world championship, and *CoreWars* (corewars.sourceforge.net), in which assembly-like programs struggle for limited computer resources.

While the majority of the programmers actually write the code for their players, some of them choose to use machine-learning methods instead. These

methods involve a process of constant code modifications, according to the nature of the problem, in order to achieve as best a program as possible. If the traditional programming methods focus on the ways to solve the problem (the 'how'), machine-learning methods focus on the problem itself (the 'what')—to evaluate the program and constantly improve the solution.

We have chosen the game of *Robocode* (`robocode.alphaworks.ibm.com`), a simulation-based game in which robotic tanks fight to destruction in a closed arena. The programmers implement their robots in the Java programming language, and can test their creations either by using a graphical environment in which battles are held, or by submitting them to a central web site where online tournaments regularly take place; this latter enables the assignment of a relative ranking by an absolute yardstick, as is done, e.g., by the Chess Federation. The game has attracted hundreds of human programmers and their submitted strategies show much originality, diversity, and ingenuity.

One of our major objectives is to attain what Koza and his colleagues have recently termed *human-competitive machine intelligence* [1]. According to Koza *et al.* [1] an automatically created result is human-competitive if it satisfies one or more of eight criteria (p. 4; ibid), the one of interest to us here being:

H. The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs).

Since the vast majority of Robocode strategies submitted to the league were coded by hand, this game is ideally suited to attain the goal of human-competitiveness.

The machine-learning method we have chosen to use is *Genetic Programming* (GP), in which the code for the player is created through evolution [2]. The code produced by GP consists of a tree-like structure (similar to a LISP program), which is highly flexible, as opposed to other machine-learning techniques (e.g., neural networks).

This paper is organized as follows: Section 2 describes previous work. Section 3 delineates the Robocode rules and Section 4 presents our GP-based method for evolving Robocode strategies, followed by results in Section 5. Finally, we present concluding remarks and future work in Section 6.

2 Previous Work

In a paper published in 2003, Eisenstein described the evolution of Robocode players using a fixed-length genome to represent networks of interconnected computational units, which perform simple arithmetic operations [3]. Each element takes its input either from the robot's sensors or from another computational unit. Eisenstein was able to evolve Robocode players, each able to defeat a single opponent, but was not able to generalize his method to create players that could beat numerous adversaries and thus hold their own in the international league.

This latter failure may be due either to problems with the methodology or to lack of computational resources—no conclusions were provided.

Eisenstein's work is the only recorded attempt to create Robocode players using GP-like evolution. The number of works that have applied machine-learning techniques to design Robocode players is meager, mostly ANN-based (Artificial Neural Network), and produced non-top-ranked strategies. In most cases the ANN controls only part of the robot's functionality, mainly the targeting systems. We found no reports of substantive success of ANNs over hand-coded robots. Applications of GP in robotics have been studied by several researchers, dating back to one of Koza's original experiments—evolving wall-following robots [4] (a full review of GP works in the field of robotics is beyond the scope of this paper).

3 Robocode Rules

A Robocode player is written as an event-driven Java program. A main loop controls the tank activities, which can be interrupted on various occasions, called *events*. Whenever an event takes place, a special code segment is activated, according to the given event. For example, when a tank bumps into a wall, the *HitWallEvent* will be handled, activating a function named $onHitWall()$. Other events include: hitting another tank, spotting another tank, and getting hit by an enemy shell.

There are five actuators controlling the tank: movement actuator (forward and backward), tank-body rotation actuator, gun-rotation actuator, radar-rotation actuator, and fire actuator (which acts as both trigger and firepower controller).

As the round begins, each tank of the several placed in the arena is assigned a fixed value of energy. When the energy meter drops to zero, the tank is disabled, and—if hit—is immediately destroyed. During the course of the match, energy levels may increase or decrease: a tank gains energy by firing shells and hitting other tanks, and loses energy by getting hit by shells, other tanks, or walls. Firing shells costs energy. The energy lost when firing a shell, or gained, in case of a successful hit, is proportional to the firepower of the fired shell.

The round ends when only one tank remains in the battlefield (or no tanks at all), whereupon the participants are assigned scores that reflect their performance during the round. A battle lasts a fixed number of rounds.

In order to test our evolved Robocode players and compare them to human-written strategies, we had to submit them to the international league. The league comprises a number of divisions, classified mainly according to allowed code size. Specifically, we aimed for the *one-on-one HaikuBot challenge* (`robocode.yajags.com`), in which the players play in duels, and their code is limited to four instances of a semicolon (four lines), with no further restriction on code size. Since GP naturally produces long lines of code, this league seemed most appropriate for our research. Moreover, a code size-limited league places

GP at a disadvantage, since, ceteris paribus, GP produces longer programs due to much junk "DNA" (which a human programmer does not produce—usually).

4 Evolving Robocode Strategies using Genetic Programming

We used Koza-style GP [2], in which a population of individuals evolves. An individual is represented by an ensemble of LISP expressions, each composed of functions and terminals. The functions we used are mainly arithmetic and logical ones, which receive several arguments and produce a numeric result. Terminals are zero-argument functions, which produce a numerical value without receiving any input. The terminal set is composed of zero-argument mathematical functions, robot perceptual data, and numeric constants. The list of functions and terminals is given in Table 1, and will be described below.

As part of our research we examined many different configurations for the various GP characteristics and parameters. We have tried, for instance, to use Strongly Typed Genetic Programming (STGP) [5], in which functions and terminals differ in types and are restricted to the use of specific types of inputs; another technique that we inspected was the use of Automatically Define Functions (ADFs) [6], which enables the evolution of subroutines. These techniques and a number of others proved not to be useful for the game of Robocode, and we concentrate below on a description of our winning strategy.

Program architecture. We decided to use GP to evolve numerical expressions that will be given as arguments to the player's actuators. As mentioned above, our players consist of only four lines of code (each ending with a semicolon). However, there is much variability in the layout of the code: we had to decide which events we wished to implement, and which actuators would be used for these events.

To obtain the strict code-line limit, we had to make the following adjustments:

- Omit the radar rotation command. The radar, mounted on the gun, was instructed to turn using the gun-rotation actuator.
- Implement the fire actuator as a numerical constant which can appear at any point within the evolved code sequence (see Table 1).

The main loop contains one line of code that directs the robot to start turning the gun (and the mounted radar) to the right. This insures that within the first gun cycle, an enemy tank will be spotted by the radar, triggering a *ScannedRobotEvent*. Within the code for this event, three additional lines of code were added, each controlling a single actuator, and using a single numerical input that was evolved using GP. The first line instructs the tank to move to a distance specified by the first evolved argument. The second line instructs the tank to turn to an azimuth specified by the second evolved argument. The third line instructs the gun (and radar) to turn to an azimuth specified by the third evolved argument (Figure 1).

Robocode Player's Code Layout

```
while (true)
    TurnGunRight(INFINITY); //main code loop
...
OnScannedRobot() {
    MoveTank(<GP#1>);
    TurnTankRight(<GP#2>);
    TurnGunRight(<GP#3>);
}
```

Fig. 1. Robocode player's code layout.

Functions and terminals. Since terminals are actually zero-argument functions, we found the difference between functions and terminals to be of little importance. Instead, we divided the terminals and functions into four groups according to their functionality:

1. *Game-status indicators:* A set of terminals that provide real-time information on the game status, such as last enemy azimuth, current tank position, and energy levels.
2. *Numerical constants:* Two terminals, one providing the constant 0, the other being an ERC (Ephemeral Random Constant), as described by Koza [2]. This latter terminal is initialized to a random real numerical value in the range [-1, 1], and does not change during evolution.
3. *Arithmetic and logical functions:* A set of zero- to four-argument functions, as specified in Table 1.
4. *Fire command:* This special function is used to preserve one line of code by not implementing the fire actuator in a dedicated line. The exact functionality of this function is described in Table 1.

Fitness measure. The fitness measure should reflect the individual's quality according to the problem at hand. When choosing a fitness measure for our Robocode players, we had two main considerations in mind: the opponents and the calculation of the fitness value itself.

Selection of opponents and number of battle rounds: A good Robocode player should be able to beat many different adversaries. Since the players in the online league differ in behavior, it is generally unwise to assign a fitness value according to a single-adversary match. On the other hand, it is unrealistic to do battle with the entire player set—not only is this a time-consuming task, but new adversaries enter the tournaments regularly. We tested several opponent set sizes, including from one to five adversaries. Some of the tested evolutionary configurations involved random selection of adversaries per individual or per generation, while other configurations consisted of a fixed group of adversaries. The configuration

Table 1. GP Robocode system: Functions and terminals.

Game-status indicators	
$Energy()$	Returns the remaining energy of the player
$Heading()$	Returns the current heading of the player
$X()$	Returns the current horizontal position of the player
$Y()$	Returns the current vertical position of the player
$MaxX()$	Returns the horizontal battlefield dimension
$MaxY()$	Returns the vertical battlefield dimension
$EnemyBearing()$	Returns the current enemy bearing, relative to the current player's heading
$EnemyDistance()$	Returns the current distance to the enemy
$EnemyVelocity()$	Returns the current enemy's velocity
$EnemyHeading()$	Returns the current enemy heading, relative to the current player's heading
$EnemyEnergy()$	Returns the remaining energy of the enemy
Numerical constants	
$Constant()$	An ERC in the range [-1, 1]
$Random()$	Returns a random real number in the range [-1, 1]
$Zero()$	Returns the constant 0
Arithmetic and logical functions	
$Add(x, y)$	Adds x and y
$Sub(x, y)$	Subtracts y from x
$Mul(x, y)$	Multiplies x by y
$Div(x, y)$	Divides x by y, if y is nonzero; otherwise returns 0
$Abs(x)$	Returns the absolute value of x
$Neg(x)$	Returns the negative value of x
$Sin(x)$	Returns the function $\sin(x)$
$Cos(x)$	Returns the function $\cos(x)$
$ArcSin(x)$	Returns the function $\arcsin(x)$
$ArcCos(x)$	Returns the function $\arccos(x)$
$IfGreater(x, y, exp_1, exp_2)$	If x is greater than y returns the expression exp_1, otherwise returns the expression exp_2
$IfPositive(x, exp_1, exp_2)$	If x is positive, returns the expression exp_1, otherwise returns the expression exp_2
Fire command	
$Fire(x)$	If x is positive, executes a fire command with x being the firepower, and returns 1; otherwise, does nothing and returns 0

we ultimately chose to use involved a set of three adversaries—fixed throughout the evolutionary run—with unique behavioral patterns, which we downloaded from the top of the HaikuBot league. Since the game is nondeterministic, a total of three rounds were played versus each adversary to reduce the randomness factor of the results.

Calculation of the fitness value: Since fitness is crucial in determining the trajectory of the evolutionary process, it is essential to find a way to translate battle results into an appropriate fitness value. Our goal was to excel in the online tournaments; hence, we adopted the scoring algorithms used in these leagues. The basic scoring measure is the fractional score F, which is computed using the score gained by the player S_P and the score gained by its adversary S_A:

$$F = \frac{S_P}{S_P + S_A}$$

This method reflects the player's skills in relation to its opponent. It encourages the player not only to maximize its own score, but to do so at the expense of its adversary's. We observed that in early stages of evolution, most players attained a fitness of zero, because they could not gain a single point in the course of the battle. To boost population variance at early stages, we then devised a modified fitness measure $\widetilde{F}$:

$$\widetilde{F} = \frac{\epsilon + S_P}{\epsilon + S_P + S_A},$$

where ϵ is a fixed small real constant.

This measure is similar to the fractional-score measure, with one exception: when two evolved players obtain no points at all (most common during the first few generations), a higher fitness value will be assigned to the one which avoided its adversary best (i.e., lower S_A). This proved sufficient in enhancing population diversity during the initial phase of evolution.

When facing multiple adversaries, we simply used the *average* modified fractional score, over the battles against each adversary.

Evolutionary parameters:

- *Population size:* 256 individuals. Though some GP researchers, such as Koza, use much larger populations (up 10,000,000 individuals [1]), we had limited computational resources. Through experimentation we arrived at 256.
- *Termination criterion and generation count:* We did not set a limit for the generation count in our evolutionary runs. Instead, we simply stopped the run manually when the fitness value stopped improving for several generations.
- *Creation of initial population:* We used Koza's ramped-half-and-half method [2], in which a number d, between *mindepth* (set to 4) and *maxdepth* (set to 6) is chosen randomly for each individual. The genome trees of half of the

individuals are then grown using the Grow method, which generates trees of any depth between 1 and d, and the other half is grown using the Full method, which generates trees of depth d exactly. All trees are generated randomly, by selection of appropriate functions and terminals in accordance with the growth method.

- *Breeding operators:* Creating a new generation from the current one involves the application of "genetic" operators (namely, crossover and mutation) on the individuals of the extant generation. We used two such operators:
  - *Mutation (unary):* randomly selects one tree node (with probability 0.9) or leaf (with probability 0.1), deletes the subtree rooted at that node and grows a new subtree instead, using the Grow method. Bloat control is achieved by setting a maxdepth parameter (set to 10), and invoking the growth method with this limit.
  - *Crossover (binary):* randomly selects a node (with probability 0.9) or a leaf (with probability 0.1) from each tree, and switches the subtrees rooted at these nodes. Bloat control is achieved using Langdon's method [7], which ensures that the resulting trees do not exceed the maxdepth parameter (set to 10).

  The breeding process starts with a random selection of genetic operator: a probability of 0.95 of selecting the crossover operator, and 0.05 of selecting the mutation operator. Then, a selection of individuals is performed (as described in the next paragraph): one individual for mutation, or two for crossover. The resulting individuals are then passed on to the next generation.
- *Selection method:* We used tournament selection, in which a group of individuals of size k (set to 5) is randomly chosen. The individual with the highest fitness value is then selected.
  In addition, we added elitism to the breeding mechanism: The two highest-fitness individuals were passed to the next generation with no modifications.
- *Extraction of best individual:* When an evolutionary run ends, we should determine which of the evolved individuals can be considered the best. Since the game is highly nondeterministic, the fitness measure does not explicitly reflect the quality of the individual: a "lucky" individual might attain a higher fitness value than better overall individuals. In order to obtain a more accurate measure for the players evolved in the last generation, we let each of them do battle for 100 rounds against 12 different adversaries (one at a time). The results were used to extract the optimal player—to be submitted to the league.

On execution time and the environment. Genetic programming is known to be time consuming, mainly due to fitness calculation. We can estimate the time required for one run using this simple equation:

$$ExceutionTime = RoundTime \times NumRounds \times \\ NumAdversaries \times PopulationSize \times NumGenerations$$

A typical run involved 256 individuals, each battle carried out for 3 rounds against 3 different adversaries. A single round lasted about one second, and our best evolutionary run took approximately 400 generations, so the resulting total run time was:

$$ExecutionTime = 1 \times 3 \times 3 \times 256 \times 400 \approx 9.2 \times 10^5 \ _{seconds} = 256 \ _{hours},$$

or about 10 days. In order to overcome the computational obstacle, we distributed the fitness calculation process over up to 20 computers. Needless to say, with additional computational resources run time can be yet further improved upon.

We used Luke's *ECJ11* system, a Java-based evolutionary computation and genetic programming research system (`cs.gmu.edu/~eclab/projects/ecj/`).

5 Results

We performed multiple evolutionary runs against three leading opponents, as described in Section 4. The progression of the best run is shown in Figure 2.

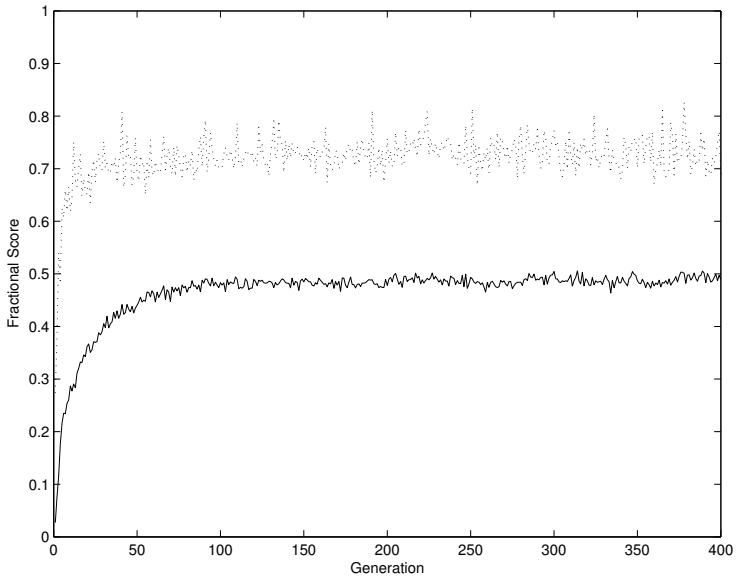

Fig. 2. Modified fractional score (Section 4) averaged over three different adversaries, versus time (generations). Top (dotted) curve: best individual, bottom (solid) curve: population average.

Due to the nondeterministic nature of the Robocode game, and the relatively small number of rounds played by each individual, the average fitness is worthy

of attention, in addition to the best fitness. The first observation to be made is that the average fractional score converged to a value equaling 0.5, meaning that the average Robocode player was able to hold its own against its adversaries. When examining the average fitness, one should consider the variance: A player might defeat one opponent with relatively high score, while losing to the two others.

Though an average fitness of 0.5 might not seem impressive, two comments are in order:

- This value reflects the average fitness of the population; some individuals attain much higher fitness.
- The adversaries used for fitness evaluation were excellent ones, taken from the top of the HaikuBot league. In the "real world," our evolved players faced a greater number of adversaries, most of them inferior to those used in the evolutionary process.

To join the HaikuBot challenge, we extracted what we deemed to be the best individual of all runs. Its first attempt at the HaikuBot league resulted in third place out of 27 contestants (Figure 3).

6 Concluding Remarks and Future Work

As noted in Section 1, Koza *et al.* [1] delineated eight criteria for an automatically created result to be considered human-competitive, the one of interest to us herein being:

H. The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs).

Currently, all players in the HaikuBot league but GPBot are human-written computer programs. We believe that our attaining third place fulfills the eighth criterion: GPBots are human competitive.

In addition, the complexity of the problem should be taken under consideration: The game of Robocode, being nondeterministic, continuous, and highly diverse (due to the unique nature of each contestant), induces a virtually infinite search space, making it an extremely complex (and thus interesting) challenge for the GPer.

Generalization. When performing an evolutionary run against a single adversary, winning strategies were always evolved. However, these strategies were specialized for the given adversary: When playing against other opponents (even relatively inferior ones), the evolved players were usually beaten. Trying to avoid this obstacle, our evolutionary runs included multiple adversaries, resulting in better generalization, as evidenced by our league results (where our players encountered previously unseen opponents). Nevertheless, there is still room for

Rank	Rating	Robot	Total Score	Survival	Last surv.	Bullet dmg.	Bonus	Ram dmg.	Bonus	1sts	2nds	3rds
1	588.87	ms.AresHaiku 0.4	36006	15400	3080	14346	2801	346	28	308	12	0
2	209.60	kawigi.haiku.HaikuTrogdor 1.1	36355	11950	2390	18692	3047	156	112	241	81	0
3	188.50	geep.haiku.GPBotC 1.0	36477	10750	2150	17901	2966	2275	424	223	99	0
4	179.57	pez.femto.HaikuPoet 0.2	27379	10250	2050	12987	1997	89	0	211	111	0
5	177.87	kawigi.femto.FemtoTrogdor 1.0	37213	11250	2250	17913	2785	2663	340	226	95	0
6	166.37	mz.HaikuGod 1.01	35602	12600	2520	15169	2259	2532	513	293	68	0
7	120.97	kawigi.haiku.HaikuCircleBot 1.0	32786	10700	2140	16942	2836	160	0	217	106	0
8	117.65	pez.femto.WallsPoetHaiku 0.1	29520	8950	1790	14476	2224	1867	202	181	141	0
9	111.24	cx.haiku.Escape 1.0	32086	12300	2460	14353	2509	408	49	248	74	0
10	91.26	cx.haiku.Xaxa 1.1	32396	10800	2160	16555	2614	209	51	219	104	0
11	76.56	ms.ChaosHaiku 0.1	32865	8900	1780	17012	2573	2223	367	186	138	0
12	54.34	kawigi.haiku.HaikuLinearAimer 1.0	18911	4850	970	11765	1239	79	0	99	221	0
13	35.87	soup.haiku.RammerHK 1.0	31684	4300	860	16968	1530	7262	751	92	230	0
14	21.70	cx.haiku.MeleeXaxa 1.0	32843	8950	1790	18100	1958	1329	708	183	141	0
15	−1.07	soup.haiku.MirrorHK 1.0	30132	7150	1430	18959	2312	273	0	143	177	0
16	−5.88	kawigi.haiku.HaikuSillyBot 1.2	18110	6150	1230	9818	854	50	0	140	187	0
17	−49.65	ahf.HaikuAndrew .1	23614	7550	1510	12779	1609	158	0	154	168	0
18	−65.13	kawigi.haiku.HaikuChicken 1.0	20870	5600	1120	12622	1446	77	0	116	204	0
19	−77.20	soup.haiku.CutoffHK 1.0	34705	6650	1330	19957	2277	3818	662	137	183	0
20	−109.83	davidalves.net.PhoenixHaiku 1.0	23609	6300	1260	14341	1669	33	0	126	194	0
21	−145.71	cx.haiku.Smoku 1.1	24434	7450	1490	13535	1587	224	139	150	170	0
22	−174.44	dummy.haiku.Disoriented 1.0	18624	6700	1340	9696	785	98	0	143	181	0
23	−210.68	klo.haikuBounC 1.0	22201	5400	1080	13907	1468	283	58	112	209	0
24	−223.57	soup.haiku.RandomHK 1.0	17140	4000	800	11126	1129	80	0	86	235	0
25	−273.43	tango.haiku.HaikuTango 1.0	15977	4300	860	9508	907	365	28	87	233	0
26	−347.32	soup.haiku.DodgeHK 1.0	13235	2350	470	9579	518	312	0	47	273	0
27	−408.43	soup.haiku.WallDroidHK 1.0	8135	2950	590	3991	135	391	71	65	255	0

Fig. 3. Best GPBot takes third place at HaikuBot league on October 9, 2004 (`robocode.yajags.com/20041009/haiku-1v1.html`). The table's columns reflect various aspects of robotic behavior, such as *survival* and *bullet damage* measures. The final rank is determined by the *rating* measure, which reflects the performance of the robot in combat with randomly chosen adversaries.

improvement where generalization is concerned. A simple (yet highly effective, in our experience) enhancement booster would be the increase of computational resources, allowing more adversaries to enter into the fitness function.

Coevolution. One of the evolutionary methods that was evaluated and abandoned is *coevolution*. In this method, the individuals in the population are evaluated against each other, and not by referring to an external opponent. Coevolution has a prima facie better chance of attaining superior generalization, due to the diversity of opponents encountered during evolution. However, we found that the evolved players presented primitive behavior, and were easily defeated by human-written programs. Eisenstein [3] described the same phenomenon, and has suggested that the problem lies with the initial generation: The best strategies that appear early on in evolution involve idleness—i.e., no moving nor firing—since these two actions are more likely to cause loss of energy. Breeding such players usually results in losing the genes responsible for movement and firing, hence the poor performance of the latter generations. We believe that

coevolution can be fruitful if carefully planned, using a two-phase evolutionary process. During the first stage, the initial population will be evolved using one or more human-written adversaries as fitness measure; this phase will last a relatively short period of time, until basic behavioral patterns emerge. The second stage will involve coevolution over the population of individuals that was evolved in the first stage. This two-phase approach we leave for future work.

Exploring other Robocode divisions. There are a number of other divisions apart from HaikuBot in which GP-evolved players might compete in the future. Among these is the *MegaBot challenge*, in which no code-size restrictions hold. Some players in this category employ a unique strategy for each adversary, using a predefined database. Since GP-evolved players are good at specializing, we might try to defeat some of the league's leading players, ultimately creating an overall top player by piecing together a collection of evolved strategies.

Other Robocode battle divisions are yet to be explored: melee games—in which a player faces multiple adversaries simultaneously, and team games—in which a player is composed of several robots that act as a team.

Acknowledgements

We thank Jacob Eisenstein for helpful discussions and for the genuine interest he took in our project. The research was partially supported by the Lynn and William Frankel Fund for Computer Science.

References

1. Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J., Lanza, G.: Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, Norwell, MA (2003)
2. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge, Massachusetts (1992)
3. Eisenstein, J.: Evolving robocode tank fighters. Technical Report AIM-2003-023, AI Lab, Massachusetts Institute Of Technology (2003) `citeseer.ist.psu.edu/647963.html`.
4. Koza, J.R.: Evolution of subsumption using genetic programming. In Varela, F.J., Bourgine, P., eds.: Proceedings of the First European Conference on Artificial Life. Towards a Practice of Autonomous Systems, Paris, France, MIT Press (1992) 110–119
5. Montana, D.J.: Strongly typed genetic programming. Evolutionary Computation **3** (1995) 199–230
6. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge Massachusetts (1994)
7. Langdon, W.B.: Size fair and homologous tree genetic programming crossovers. Genetic Programming and Evolvable Machines **1** (2000) 95–119