Evolving Robot Tank Controllers

Jacob Eisenstein

May 15, 2003

Abstract

In this paper, I describe the application of genetic programming to evolve a controller for a robotic tank in a simulated environment. The purpose is to explore how genetic techniques can best be applied to produce controllers based on subsumption and behavior oriented languages such as REX. As part of my implementation, I developed TableRex, a modification of REX that can be expressed on a fixed-length genome. Using a fixed subsumption architecture of TableRex modules, I evolved robots that beat some of the most competitive hand-coded adversaries.

1 Introduction

RoboCode is a tank-combat simulator developed by IBM Alphaworks [12]. The tank must navigate the environment to avoid being shot by its opponent and also avoid running into walls. In addition, to succeed, the tank must locate its adversary and shoot it. Like most simulations, RoboCode is a mix between realism and fantasy.

REALISM

- The actuators take time; while you are rotating your turret you might get shot.
- Bullets take time to arrive. If you shoot at a moving target, you must account for where it will be when the bullet gets there.
- The unidirectional radar sensor must be pointed at adversary to see it.
- The gun heats up when fired and cannot fire again until it cools.
- Bumping into things hurts.

FANTASY

- Sensors and actuators are noiseless.
- Radar sensor detects velocity, bearing, heading, and energy remaining.

- Combat takes place in flat, walled rectangular area.

Although RoboCode is not the ideal simulator, it has two key advantages. First, it has a built-in visualization for its simulations, which I used to produce the movies shown during my presentation. Second, and more important, it is tremendously popular. This means that it is easy to get a fairly unbiased evaluation of my work; I can compare my evolved robots with any of the 4000 different human-coded adversaries that can be found online [14].

On a more general level, I claim that tank fighting is a reasonable evaluation platform for robot evolution. There is a clear and objective fitness metric: survive in a dangerous environment. This task poses a number of interesting requirements:

- Tank fighting requires intelligent navigation to achieve optimal position relative to the opponent.
- Robots must display careful management of time and resources. A robot that wastes either commodity will not last long.
- To be truly successful, a robot must model its adversary, while preventing the adversary from doing the same. Such a model need not be explicit, but an effective targeting and tracking system requires some way of predicting where the adversary will be in the future. Similarly, evasion requires unpredictable movements that defeat the opponent's tracking system.

Of course, real creatures do not come equipped with guns and radar. But more generally, evolution has produced solutions to all three of the requirements listed above. Predators and prey both try to navigate into favorable positions and outmaneuver their adversaries. Animals typically don't waste time or energy, particularly when threatened. Evolution has produced very sophisticated tracking and evasion behavior, relying on predictions of the adversary's next move. The RoboCode platform can be viewed as an extremely impoverished simulation of the challenges that real animals face in their quest to survive another day.

1.1 Rules of the Game

RoboCode tanks are written as Java programs. They are event-driven; there is a main loop, and it can be interrupted by a number of event handlers. Events include: seeing an adversary, getting hit by a bullet, getting rammed by an adversary, among others.

Tanks start with a fixed amount of energy. There are several ways to spend energy: getting shot, bumping into things, and shooting bullets. Also, after a fight has proceeded for while, all robots will have energy deducted until one of them dies. This keeps the battle from going on forever. When two robots collide, both robots take equal damage; ramming can be a good strategy for a robot that is already winning.

There is only one way to gain energy: shooting an adversary. If a robot runs out of energy due to shooting bullets, it is disabled. If a bullet later hits its target then some of the energy is returned. But if a robot runs out of energy for any other reason – such as getting shot or banging into a wall – then it dies. The full rules and physics of the RoboCode simulator are described in the Appendix, Section 8.3.

1.1.1 Actuators

A robot may rotate its entire body, the gun/radar turret, or the radar alone. All rotation takes time, but rotating the entire body takes more time, and rotating the radar alone takes the least time. In addition, robots may move forward or backward, at a fixed rate of acceleration, with a fixed cap of velocity.

The robot may fire its gun, with a variable amount of power. Using more power requires more energy, but does more damage. Below is a short program fragment that rotates the robot ten degrees, then moves forward 45 units, then rotates the gun ten degrees in the other direction, and then fires at maximum power:

```
left(10);
forward(45);
gunRight(10);
fire(3);
```

Each robot is implemented as a thread, and the program may be interrupted at several points so that the enemy can move. Very long sequences of actions are usually a mistake, since an event will typically interrupt the flow sooner or later. But robots cannot be purely reactive either. A short sequence of actions is usually required to respond to an event, such as rotating the gun to face the enemy, and then firing.

1.1.2 Sensors

All robots are equipped with a single radar sensor. The sensor has infinite range, but can only sense opponents within a one degree scan width. The radar sensor is the only way that the robot can get information about its adversary. The sensor returns the enemy position, orientation, bearing, gun angle, and energy. The robot is also always aware of its own position, orientation, energy, and the orientation of its gun and radar. The complete set of input is listed in Section 8.2.

1.2 Strategies

Among human-coded robots, there are several classes of strategies. Many robots attempt to follow the walls and get into corners, so that they can restrict the range that their radar needs to scan to find their opponent. Other robots try to move back and forth in an erratic pattern to confuse their opponent's targeting system. Still others focus on pattern recognition techniques, trying to learn their opponent's movement patterns so that they can predict where to shoot next. Finally, tracker robots try to lock onto to their opponent and follow them around the arena, possibly even ramming into them to do additional damage.

2 Controller Design

As mentioned above, RoboCode controllers are written in Java. However, evolving Java code directly seems infeasible. From a strictly implementational point of view, the time cost of compiling Java to bytecode every time an individual is to be evaluated would be prohibitive. In addition, the space of all Java programs is comprised mainly of programs that do not parse, or fail to compile for some other reason. Instead of evolving Java code, genetic programming is more frequently used to evolve Lisp-like programs [11], which are weakly typed and are organized in a tree structure that is well-suited to crossover and mutation [2].

I decided to evolve programs written in TableRex. TableRex is a language that I designed, using Leslie Kaelbling's REX language [9] as a starting point. To evaluate a genome, I print out the corresponding TableRex program. My Java robot controller then interprets the TableRex program at runtime. Communicating with the robot controller via files was necessary due to the security constraints built into the simulator. I was initially concerned about the time cost of performing file i/o, but evaluation of my TableRex-controlled robots did not take noticably more time than the evaluation of hand-coded robots.

2.1 TableRex

TableRex is based on the same ideas behind REX: actuators are controlled through a network on interconnected computation elements, which perform simple operations such as arithmetic and basic logic. Each computation element can take inputs either from the robot's sensors or from the output of another computation element. Some computation elements output to the actuators of the robot. Feedback loops are allowed, but a delay of one time-step is imposed.

The main distinction between TableRex and the original language is that TableRex organizes the computation elements in a table, and control flows down the table. Each row in the table consists of a function (e.g. addition, greater than), the addresses of two inputs, and the most recently recorded output. When the control pointer reaches a given row, it looks up the values of the inputs in the table, applies the function, and updates the output. If one or both of the inputs is later in the table than the current row, then its old value is used; this is how feedback loops are handled. All output values in the table start off initialized to zero; unlike in REX, feedback elements cannot be initialized to a nonzero value.

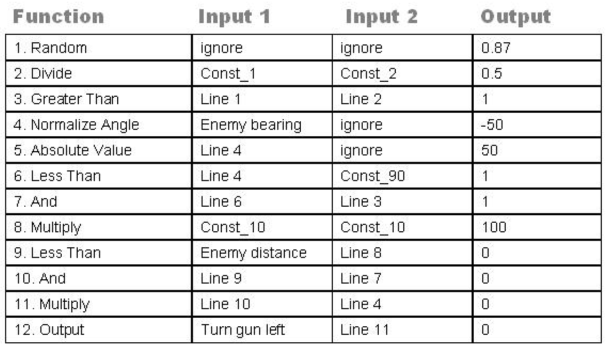
Figure 1 shows a simple TableRex program. This program will turn the gun to the bearing at which an enemy is seen, if three conditions are met. Line 3 ensures that the random number from Line 1 must be greater than 0.5. Line 6 ensures that the absolute value of the relative bearing of the opponent is less than ninety degrees. Line 9 requires that the distance to the enemy be less than 100. These conditions are combined with a series of “and” functions, and the result is outputted to the “TurnGunLeft” actuator.

The modification of REX into TableRex was based on three design requirements. First, TableRex had to be easily encodable as a fixed-length genome. The genetic encoding of TableRex will be described in Section 2.3. Second, TableRex had to support the design of an efficient interpreter. The Java robot controller that I have designed can interpret a TableRex program in linear time with respect to the number of computation elements. The interpreted program is stored in a set of arrays, and can be evaluated in linear time. In contrast, I was unable to construct a linear-time compiler for arbitrary the original REX language. [1] Finally, TableRex is able to easily output ordered sequences of actuator control commands, given a single set of inputs. This is critical for this domain, since even elementary robots require sequences of actions, e.g. “turn left 10 degrees and shoot.” REX is also capable of producing sequences of commands, but this requires feedback loops to keep track of state variables.

2.2 Subsumption Architecture

As mentioned above, RoboCode tanks are meant to be written in an event-driven architecture. There is a base program, and a set of handlers for events such as being hit a bullet, sensing an adversary, or bumping into a wall. This event-driven structure was well-suited for a subsumption-inspired architecture [1]. Each event

[1] The problem stemmed from the ordering of the computational elements in the original representation. For any program, there was always *some* initial ordering for which I could produce a linear-time compilation. But I was unable to develop an algorithm that could compile from any initial ordering in linear time.



Function	Input 1	Input 2	Output
1. Random	ignore	ignore	0.87
2. Divide	Const_1	Const_2	0.5
3. Greater Than	Line 1	Line 2	1
4. Normalize Angle	Enemy bearing	ignore	-50
5. Absolute Value	Line 4	ignore	50
6. Less Than	Line 4	Const_90	1
7. And	Line 6	Line 3	1
8. Multiply	Const_10	Const_10	100
9. Less Than	Enemy distance	Line 8	0
10. And	Line 9	Line 7	0
11. Multiply	Line 10	Line 4	0
12. Output	Turn gun left	Line 11	0

Figure 1: A simple TableRex program.

handler corresponds to a TableRex program, and they are organized in a hierarchy of importance. If a high-priority event is triggered, its TableRex program gets control of the actuators for one pass through its table. The base program keeps control until another event is received.

The base, onHitByBullet, and onRammed AFSMs all take identical inputs and control the same actuators. These AFSMs do not take input from the radar scanner, and cannot control the gun, since they don't know where the opponent is. The onScan program has additional inputs, from the radar sensor, and it is able to control the gun. Section 8.2 lists the inputs available to each AFSM.

In a slight deviation from the subsumption paradigm, I made the last two rows of each TableRex program readable as system inputs to the other AFSMs. Strict subsumption requires that the modules communicate only through inhibition and suppression. However, by looking at the hand-coded robots that humans produced, I noticed that the event handlers often modified global variables that were accessible to the other event handlers. For example, the onScan AFSM might create a global variable indicating the last location at which the enemy was seen. The base program might use this information in deciding which way to go to evade bullets. Several of the evolved robot controllers did take advantage of this feature.

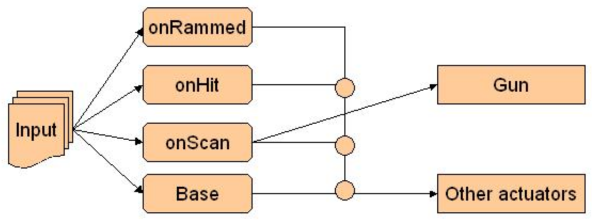


Figure 2: The RoboCode subsumption architecture.

2.3 Genetic Encoding

TableRex was designed to be easily encoded as a fixed-length genome. Each row in the table is encoded as an n-bit word:

```
ffffa...ab...b
```

The first four bits indicate the row's function; there are sixteen possible functions, and they are enumerated in the Appendix, Section 8.1. The remainder of the string identifies the inputs. The first 20 inputs are reserved for system inputs, constants, and visible outputs from other AFSMs. Since the onScan AFSM has access to more system inputs than the other AFSMs, it has access to fewer visible outputs from other AFSMs.

I experimented with robots with tables of length 44 and 108. For the 44-row tables, there were a total of 64 inputs, requiring six bits. The total encoding length for a row in the table was therefore 16. For the 108-row tables, there were a total of 128 inputs, requiring seven bits per input and 18 bits to encode a row. The overall encoding length for an individual is given by:

$$L = ER(\lg F + 2\lg(I + R))$$

Where E is the number of events, R is the number of rows, F is the number of functions, and I is the number of reserved inputs. For 108-row tables, the encoding length was 7776 bits. For 44-row tables, the encoding length was 2816 bits.

3 Training

I trained my robots using the `GeneticAlgorithm` Java class that I developed for research assignment 4. It contains methods for initializing a population, performing crossover, mutation, and elitism, and selecting the population of each subsequent generation. The GeneticAlgorithm class is abstract; it must be subclassed to be instantiated. The subclass must override the `evaluate()` method, which is the one thing that is unique to each problem domain.

The top 2% of each generation was copied on without mutation. I did this to avoid accidentally losing the best individuals due to bad luck. Additionally, 10% of the population is copied with mutation. This group is selected using proportional selection over scaled fitness, which will be described momentarily. The remaining 88% of the population is composed using crossover; parents are again chosen using proportional selection over scaled fitness.

The mutation rate was pegged to the fitness diversity of the population. The idea here is that as the population homogenizes, we run the risk of settling in a local maxima, and additional mutation can help get us out. The maximum mutation rate was set at eight divided by the genome length. This maximum rate was then multiplied by the square of the ratio of the average fitness to the best fitness. Thus, if all individuals have the same fitness, then on average eight bits will be flipped in each individual. If the average fitness is half the best fitness, indicating a fairly diverse population, then the typical individual will experience only two bit flips in mutation. Since I used elitism whereby the top members of the population are passed on without mutation, I felt it was safe to use such a high mutation rate on the rest of the individuals.

3.1 Raw Fitness

RoboCode assigns a score to each robot based on how much damage it did to its opponent, plus a substantial bonus for being the last robot standing. Being the last robot standing does not assure you of winning the fight; if your opponent shoots you repeatedly but kills himself by ramming into a wall, it may still get a better score than you. However, in practice the last robot standing almost always has the higher score.

My original raw fitness function corresponded to the total number of points awarded to the individual across all of its battles. This led to some pathological behavior. If a robot was evaluated on its performance in ten fights, it would often learn to win one or two fights by a huge margin, and then lose the others.

Next, I scaled the robot's score by taking its square root, thus valuing the first several points scored much more than the "piling on" that happened at the end. I also awarded an additional bonus for winning the fight. This technique produced a series of robots that learned to dodge all of their opponent's bullets until the adversary ran out of energy and died: a successful approach, but somewhat boring. Finally, I settled on rewarding the difference between the score of the robot being evaluated and the score of its opponent. This proved an adequate compromise. The robots produced under this fitness metric didn't concentrate quite so much on winning one or two blowouts, and actually learned to shoot back some of the time (Section 5, discusses the difficulties in targeting an adversary).

3.2 Scaled Fitness

Regardless of the raw fitness function, I applied *linear fitness scaling* [5]. A linear transformation is applied so that the scaled fitness of the best individual differs from the average by a constant factor throughout the run. Fitness scaling serves two purposes. At early stages of evolution, when the differences in fitness between individuals are great, scaling prevents the best fit individuals from dominating selection and homogenizing the population prematurely. At later stages of evolution, when the differences in fitness are likely to be small, scaling confers a significant selection advantage on individuals that are slightly more fit, which keeps evolution moving. Linear fitness scaling on proportional selection has been shown to be practically equivalent to tournament or ranking selection schemes [6].

3.3 Adversaries

The RoboCode distribution ships with ten initial robots, which are intended to provide useful examples of coding techniques and serve as competent adversaries. The code size of these robots ranges from 1k to 5k; anecdotally, code size does appear to correlate reasonably well with performance. Most of my evaluation focused on these "starter" robots. However, I also wanted to test my system against the upper limit of what human programmers could develop. The RoboCode Repository website [14] hosts roughly 4000 robots, and four of these were identified as "exemplary." Of these four, three performed some kind of adaptation, where they learned the opponent's techniques over multiple rounds. For the purposes of training, it was impractical to hold enough rounds of combat to make these adaptive systems relevant. Consequently, I chose the only exemplary robot that did not perform adaptation. This robot – "SquigBot" – contained roughly 29k of code, and easily outperformed all of the robots from the starter set.

3.3.1 Coevolution

In addition of training against hand-coded robots, I also experimented with coevolution. The technique here was simply to choose two robots at random from the population, make them fight, and have the winner continue on as part of the selection pool for the next generation (subject to crossover and mutation). This idea seemed attractive, since it would allow me to observe the "arms race" as robots developed more and more sophisticated strategies and counter-strategies over time. It also seemed more likely to produce more general and novel solutions than training robots against a small collection of hand-coded adversaries.

Unfortunately, I was not able to get any success with this approach. In the initial generation, the most successful robots were those that stood still and didn't do anything. Robots that moved typically ran into a wall and lost energy; robots that fired the gun never hit their targets and again, lost energy. Thus, after several generations, I found the population rife with catatonics.

I believe that coevolution could be successful, but it would probably require a more nuanced selection method than simply picking the winners of each fight and passing them on to the next generation. Some kind of incremental training that rewarded more basic actions such as rotating the radar and firing the gun would probably help. Another approach would be to start off by seeding the population with a collection of hand-coded robots. This approach might somewhat constrain the space explored by evolution, but it would at least get past the initial hump of getting the robot to do *something*.

3.4 Time

Most of my experiments were conducted on my office computer, a 1GHz Pentium III with 256 MB of RAM. The RoboCode simulator is written in Java, and executed using the Sun Java Virtual Machine. To increase the speed of the simulator, I decompiled it and removed the graphics routines. The average time to running a single battle was roughly 0.6 seconds. My robots did not require noticably more time to evaluate than any of the human-coded robots, despite the fact that they had to read in and interpret TableRex programs.

Still, the time cost of running a single battle limited my ability to perform more sophisticated evaluations. For example, I wanted to train a robot against multiple enemies, using multiple randomized starting positions. Since the outcome of a battle is actually quite sensitive to initial starting position, roughly 25 starting positions are required to form a reasonable sample. Training against four different adversaries with 25 starting positions each requires 60 seconds per individual. With a population of 500, more than eight hours is required to evaluate a single generation. This made it nearly impossible to run evaluations of this kind. I did perform evaluations with multiple randomized starting positions, and other evaluations with multiple adversaries, but I was unable to successfully combine the two.

4 Results

I explored four different conditions for training the my robots. They are presented here in order of difficulty, beginning with the easiest condition.

4.1 One Adversary, Fixed Starting Position

The simplest possible experiment condition was training against a single adversary where all battles begin from a fixed starting position. The fixed starting positions were designed to be fair: each robot is placed close to a corner, and oriented ninety degrees away from its opponent. Under this condition, I was able to reliably evolve robots that could beat each of the hand-coded "starter" opponents, as well as SquigBot, the "showcase" adversary. For some of the starter opponents, only a few generations were required. For example, one of the starter opponents is a wall-following robot with 2Kb of code. Using a population of 500 individuals, I evolved a robot that could beat the wall-follower in only 13 generations.

There are at two reasons that this condition was so favorable for my approach. First, evaluation was very fast, since only one battle was required to evaluate an individual. In practical terms, that meant that even with a population of 500 individuals, an entire run could be completed in a few hours. More importantly, however, this condition allowed robots to develop very brittle strategies that only worked under the precise training conditions. In general, these robots were unable to beat adversaries other than one that they trained against. The ability of these robots to generalize beyond their initial starting positions was mixed. Some runs produced robots that could win fairly reliably even at novel starting positions, but other runs produced extremely narrow solutions that only won at a single starting point.

4.2 One Adversary, Multiple Starting Positions

As mentioned above, combat outcomes are very sensitive to the initial starting position. Even an inferior robot can win easily if it starts the battle with its radar and gun fixed on the back of an opponent who is trapped in a corner. Thus, to produce a fair evaluation at multiple starting positions, a reasonable sample size is necessary. My sample sizes ranged from 10 to 25 starting positions; this had the obvious effect of increasing the training time by a large factor. To fairly evaluate all individuals in the same generation, I used the same set of random starting position for each individual within a generation. The starting positions were updated every generation to prevent overfitting.

I chose several adversaries from the “starter” set, and in all cases I was able to evolve a robot that could win more than half the time. For example, it took 60 generation to develop a robot capable of regularly beating the starter robot SpinBot 80% of the time. Against SquigBot, the 29 Kb showcase bot, another 60 generation run produced a robot that could win 50% of the time, but it appeared to plateau at this result. In most cases, when the evolved robots won, the margin was small, whereas when the hand-coded robots won, the margin was very large. In part this was due to the fact that most of these experiments were conducted when I was using a raw fitness function that actually encouraged this outcome by offering a declining rate of return for high scores (see Section 3.1). Changing the fitness function to more greatly reward the margin of victory improved the situation somewhat. However, to a large extent this problem was due to the fact most of my robots never even attempted to shoot their adversary. Instead, they won by dodging all fire until the opponent ran out of energy. The relative difficulty of dodging versus shooting will be discussed in Section 5.

4.3 Multiple Adversaries, Single Starting Position

The robots evolved in the previous condition typically learned to move in a pattern that evaded their adversary’s firing pattern. Predictably, these robots had no success against other adversaries. In this condition, I attempted to evolve robots that learned general programs that could defeat multiple adversaries.

After 80 generations, a robot evolved that could beat four out of five of the starter robots. The only robot that it couldn’t beat employed the “tracker” strategy: it would follow its adversary around and try to corner it. My robot had evolved a complex movement pattern that avoided the fire of the other four adversaries, who would shoot from a distance. But the tracker inevitably chased my robot down and killed it. In another run, I trained my robot against 10 adversaries, including SquigBot. In 51 generations, a robot evolved that could beat four of the ten adversaries, through a combination of dodging and ramming.

4.4 Multiple Adversaries, Multiple Starting Positions

This condition was designed to produce the most general solutions: Robots that could beat multiple adversaries from a variety of different starting positions. Unfortunately, as described in Section 3.4, this condition was extremely time-consuming. I recently began an experiment with two adveraries, twenty random starting positions, and a population of 200. Under these conditions, each generation took roughly 130 minutes. After 14 generations, very little progress has been made. In the previous two conditions, 60 to 80 generations were required to produce winning robots. Since this condition is more general than either of the previous two, it would seem that 60 generations – 130 hours – would be soonest that any success could be expected.

5 Discussion

One thing that proved particularly difficult to learn was targeting. Most of the evolved robots found it much easier to dodge bullets and occasionally ram their adversaries, rather than try to shoot at them. The only robots that used the gun much were those evolved in the simplest condition, with only one adversary and a single fixed starting position.

Targeting is a difficult problem, even for the hand-coded robots. Bullets move at a top speed of 20 units per tick, and tanks can move as fast as 8 units per tick. Thus, the velocity of the target tank must be taken into account. In addition, the time required to rotate the gun turret must also be factored in. Even if the movement of the target is totally predictable, accurate targeting requires some complex mathematics. To make matters worse, the simulator will tell a robot whether its shot hit or missed, but it provides no information about how close the shot came. This prevents the design of a targeting system based on a feedback loop.

It should be no surprise that SquigBot devotes 400 lines of Java code to its targeting algorithm. Perhaps accurate targeting is simply too difficult to construct out of the elementary computation units afforded by TableRex. Providing TableRex with more advanced computation units, such as trigonometric functions, might help address this problem.

On the other hand, many of the robots from the starter set attempt to target and shoot their opponents. These robots use extremely simple targeting algorithms that are not guaranteed to hit a moving target even in the simplest posisble situation. But apparently they do hit enough of the time to be worthwhile. Why can't evolution produce something comparable?

There are at least two possible explanations. First, consider that at early stages in the evolutionary process, most of the robots that use the gun are going to miss. This costs energy, so these robots will achieve lower scores, and this trait will eventually be selected out of the population. It is likely that all of the robots that use the gun are selected out of the population before any robot can evolve that uses the gun well enough to actually improve its chances of survival.

Another explanation is that using the gun is actually a poor strategy. On this view, the easiest way to construct a competitive robot is to ignore the gun and focus on evolving evasive maneuvers. Perhaps the fact that nearly all hand-coded robots use the gun is merely a reflection of the irrationality of human engineers. Evolution has no preconceived notion that tanks win fights by shooting their adversaries, or that winning without shooting is "poor form." Of course, the strategy of evasion only works in a world where most of one's adversaries *do* use the gun. As the experiment with coevolution shows, the "do nothing" robot can actually be a local maxima. When competing with another "catatonic" adversary, nearly all mutations away from "do nothing" result in a decline in fitness.

By modifying the simulator to remove the rule that shooting costs energy, it would probably be possible to produce robots that use the gun more frequently. For a fair evaluation, however, it would eventually be necessary to switch back to the conventional simulator, since that is the world for which the hand-coded robots were designed. Another approach would be to train the onScan AFSM separately from the other modules, in a non-combat situation that rewarded hitting a moving target rather than winning the fight. After the onScan AFSM was trained in isolation, it could then be trained to work in synchrony with the other modules.

6 Related Work

On the general topic of evolving robot controllers, there are two main streams of thought. Some researchers have focused on evolving neural networks. Harvey et al. [7] argue that evolving programs in relatively high-level languages like TableRex constrains the search space, imposes the designer's prejudices on the evolutionary process, and leads to coarser-grained fitness landscape. They propose and demonstrate the evolution of a neural network that performs navgiation and obstacle avoidance. Floreano and Mondada [4] apply similar techniques to a real robot, which learns a similar set of tasks. Miglino et al. [13] show how neural network robot controllers evolved in simulation can be used in real robots.

While a great deal of the research on evolving robotic controllers centers on neural networks, such controllers seem best suited for the design of reactive systems, such as wall-followers and collision-avoiders. It is not clear that neural networks are scalable to more complicated tasks, such as tank control. For this reason, other researchers have worked on evolving robot controllers using relatively high-level languages. Koza shows that wall-following behavior can be learned by evolving controllers in a subset of lisp [10]. Brooks [2] proposes that evolution be performed using GEN, a high level language that is compiled to BL, the Behavior Language. Since BL is a language specifically designed for specifying AFSMs, this approach would appear to be the closest to my own.

6.1 RoboCode Evolution

Several web pages and newsgroup posts indicate that people have tried to apply genetic techniques to the development of RoboCode tanks. However, I was unable to find a single follow-up report describing any success whatsoever. I personally emailed some of the people who claimed to be exploring this topic, and I posted to discussion groups asking for information, but received no replies at all.

One group of students at California Polytechnic tried this idea as a course project in the fall of 2002 [3]. Their goal was to be able to beat a robot from the starter kit called "SittingDuck" eighty percent of the time. "SittingDuck" is pretty much what it sounds like – it sits still and doesn't shoot back. From their writeup, it is unclear whether they met this goal; their group name was "Unrealistic Expectations." Part of the problem for this group was that the evaluation of each robot took forty seconds. They were evolving Java code directly, and may have been compiling to bytecode as part of the evaluation process.

On the RoboCode discussion forum, one person mentioned that he had tried to evolve neural networks to control the robots, but that the resulting robots were "unable to beat a single other bot" that they were tested against. This was one of the factors that led me to pursue a REX-based encoding instead of using a neural network.

7 Conclusion and Future Work

As a class project, the research documented here could be only a first pass. The intent was to investigate the feasibility of this basic framework for evolving robot tank controllers in a challenging and marginally realistic environment. The results are generally positive; successful robot controllers were evolved for three of the four experimental conditions. Progress on the fourth condition – multiple adversaries and multiple starting points – was limited mainly by the high computational cost of evaluation. There is no reason to believe *a priori* that this condition will not eventually yield to faster hardware. However, the fact that the evolved robots generally do not develop targeting systems is somewhat troubling. This problem may have to be overcome before truly general and competitive robots can be evolved.

While I believe that the approach described here is on the right track, it suggests a number of a number of possible improvements that could not be investigated in the time allotted. In the remained of this section, I will document a few such refinements that I feel have particular research significance.

7.1 Neural Network Targeting

As discussed in Section 5, targeting appears to be one of the most difficult tasks for which to evolve robot controllers. Accurate targeting requires some complicated mathematics, as well as the ability to predict the opponent's next move. It is possible that targeting would best be handled by a neural network, rather than by a TableRex AFSM. Such a network could be trained separately on a variety of moving targets, and could thereby learn to output the appropriate gun turret rotation once an enemy is spotted.

Evaluating the robot by its overall success in combat is a fairly indirect way to learn a targeting system. It might be more appropriate to train the targeting system separately on specific targeting tasks. One way this could work is through a biologically-inspired two-step learning process. Evolution could supply the basic framework for the targeting network. Then each individual could go through a short "childhood" phase, where its targeting system was trained, before being evaluated. This comports well with biology, where the framework for skill acquisition is supplied by evolution, but the acquisition of specific skills is performed by the organism itself during childhood.

7.2 Incremental Training

Another possible extension of the learning system would be to perform an incremental evolution, where each controller is learned separately. The base program could be learned in a world where the fitness function simply evaluates how long the organism can go before being shot. Next, the onHitByBullet and onCollision AFSMs could be learned in a world where the goal is to survive for as long as possible – recovering from being shot or rammed and avoiding future damage is the key here. In fact, many of the more advanced hand-coded tank controllers actually switch to an entirely new movement pattern after being shot. This is based on the assumption that if you have been shot, then your opponent must have learned a good model of what you are doing, so you should try something else. As described in Section 5, the onScan component could be trained on specific targeting tasks. Finally, the AFSMs could be brought together and trained as a single robot. Such incremental training would provide far more focused evaluation of each AFSM, whereas under the present approach, some modules could become essentially irrelevant to selection.

7.3 Genome Reorganization

Holland's Schema Theorem [8] provides a lower bound on the expected proportion of a given schema in the next generation. A schema's chance of surviving mutation is given by its contribution to fitness and the number of bits in the schema. This suggests that holding fitness constant, the most successful schemas will be concise. More importantly, a schema's chance of surviving crossover is affected by its length; the schema `101***00` is much less likely to survive crossover than the schema `10100`. This suggests that successful schemas should be compact.

An example shows how this can apply to TableRex. Consider the program:

```
1: random     ignore         ignore
2: less_than  enemyDistance  const_90
3: multiply   enemyBearing   line2
4: output     gunRight       line3
```

In this program, line 1 can be ignored; the key effect is to rotate the gun to meet the enemy only if the enemy distance is less than 90. Lines 2, 3, and 4 constitute a potentially successful schema. In addition to contributing to fitness, this schema is also compact; the chance of it being broken by crossover is 2/3. But suppose the program looked like this:

```
1: less_than  enemyDistance  const_90
2: multiply   enemyBearing   line1
3: random     ignore         ignore
4: output     gunRight       line2
```

This program produces identical behavior, but the schema is no longer compact. In fact, it is now guaranteed to be broken by any crossover point. This is a weakness of TableRex with respect to tree representations for genetic programming. Such representations ensure that subtrees form crossover schemas that are minimally likely to be disrupted by crossover.

TableRex evolution might benefit if programs could be reorganized so as to keep together program lines that depend on each other. I envision an algorithm that would automatically transform the second program into the first, by examining the dependencies between each line of code. Such an algorithm would preserve behavior, but would increase the chance that the valuable pieces of the program would be passed on to the next generation.

7.4 Team Cooperation

One recent twist on RoboCode has been the introduction of team combat. Teams are comprised of two types of robots: leaders and droids. Leaders are equipped with a scanner, and can send and receive messages. Droids have no scanner and can only receive messages. Successful teams protect the leader, who communicates with the droids and coordinates their behavior.

With a few modifications, I believe that this could provide an interesting platform with which to study cooperative behavior. I propose to modify the simulator to dramatically limit the size of the messages. This could be accomplished by a hard rule, or by adding noise that could flip each bit of the message with some probability. In addition, I would dramatically limit the size of the droid programs, making it difficult for them to keep more than a few messages in memory at any time.

The hope is that these constraints would force the robot teams to develop a concise, efficient system of communication and cooperation. Unlike some other approaches to evolving collaboration, this proposal comes with an objective, predefined goal that the team must meet: team members must cooperate in order to stay alive.

8 Appendix

8.1 TableRex Functions

- `0000: greater than`
- `0001: less than`
- `0010: equal`
- `0011: addition`
- `0100: subtraction`
- `0101: multiplication`
- `0110: division`
- `0111: absolute value`
- `1000: and`
- `1001: or`
- `1010: not`
- `1011: generate constant`
- `1100: modulo`
- `1101: random float`
- `1110: normalize relative angle`
- `1111: control actuator`

Logical operators coerce zero and negative numbers to *false* and positive number to *true*. Similarly, comparators return 1 when true and 0 otherwise. The `absolute value` function ignores the second input. The `random float` function ignores both inputs. `Generate constant` takes the integer values of the two input addresses, and creates a new integer constant by subtracting the second from the first. `Control actuator` sends the value specified by the second input to an actuator specified by the value of the first input address. `Control actuator 4 32` sends the value specified by the 32nd input to the fourth actuator.

8.2 Inputs

- velocity
- energy
- heading
- gunHeading
- gunHeat
- distance to west wall
- distance to north wall
- distance to east wall
- distance to south wall
- constant: 1
- constant: 2
- constant: 10
- constant: 90
- enemyVelocity
- enemyEnergy
- enemyHeading
- enemyBearing
- enemyDistance

Note that only the onScan AFSM has access to the last five inputs. On the other AFSMs, these slots are filled with data from other AFSMs.

8.3 RoboCode Physics

This information is reproduced from the RoboCode website:
`http://robocode.alphaworks.ibm.com/help/physics/physics.html`

8.3.1 Processing Loop

The order that Robocode runs is as follows:

1. All robots execute their code until taking action
2. Time is updated (currentTime++)
3. All bullets move and check for collisions
4. All robots move (heading, accel, velocity, distance, in that order)
5. All robots perform scans (and collect team messages)
6. The battlefield draws

8.3.2 Time and distance measurements in Robocode

Time (t)
Robocode time is measured in "ticks", which are equivalent to frames displayed on the screen. Each robot gets one turn per tick. 1 tick = 1 turn = 1 frame.

Distance Measurement
Robocode's units are basically measured in pixels, with two exceptions. First, all distances are measured with double precision, so you can actually move a fraction of a pixel. Second, Robocode automatically scales down battles to fit on the screen. In this case, the unit of distance is actually smaller than a pixel.

8.3.3 Robot Movement Physics

Acceleration (a)
Robots accelerate at the rate of 1 pixel/tick. Robots decelerate at the rate of 2 pixels/tick. Robocode determines acceleration for you, based on the distance you are trying to move.

Velocity Equation (v)
$v = at$. Velocity can never exceed 8. Note that technically, velocity is a vector, but in Robocode we simply assume the direction of the vector to be the robot's heading.

Distance Equation (d) $d = vt$.

8.3.4 Robot, Gun, and Radar rotation

Max rate of rotation of robot
(10 - .75 * abs(velocity)) degrees / tick. The faster you're moving, the slower you turn.

Max rate of rotation of gun
20 degrees / tick. This is added to the current rate of rotation of the robot.

Max rate of rotation of radar
45 degrees / tick. This is added to the current rate of rotation of the gun.

8.3.5 Bullets

Damage 4 * firepower. If firepower > 1, it does an additional 2 * (power - 1).
Velocity 20 - 3 * firepower
GunHeat generated 1 + firepower / 5. You cannot fire if gunHeat > 0. All guns are hot at the start of each round.

Power returned on hit 3 * firepower.

8.3.6 Collisions

With Another Robot Each robot takes .6 damage. If a robot is moving away from the collision, it will not be stopped.

With a Wall AdvancedRobots take Math.abs(velocity) * .5 - 1; (Never < 0)

References

[1] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, January 1991.

[2] Rodney A. Brooks. Artificial life and real robots. In Francisco J. Varela and Paul Bourgine, editors, *Towards a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, pages 3–10, Cambridge, MA, 1992. MIT Press.

[3] http://counties.csc.calpoly.edu/ team13fk/F02/part1.html.

[4] Dario Floreano and Francesco Mondada. Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot.

[5] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[6] David E. Goldberg. A comparative analysis of selection schemes used in genetic algorithms. In Gregory Rawlins, editor, *Foundations of Genetic Algorithms*. Morgan Kaufman, 1991.

[7] Inman Harvey, Philip Husbands, and Dave Cliff. Issues in evolutionary robotics.

[8] John Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[9] Leslie Kaelbling. Rex: A symbolic language for the design and parallel implementation of embedded systems. In *Proceedings of the AIAA Conference on Computers in Aerospace VI*, pages 255–60, Wakefield, MA, 1987.

[10] John R. Koza. Evolving emergent wall following robotic behavior using the genetic programming paradigm. In *ECAL'91*, Paris, 1991.

[11] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[12] Sing Li. Rock 'em, sock 'em robocode!, 2002. http://www-106.ibm.com/developerworks/java/library/j-robocode/.

[13] Orazio Miglino, Henrik Hautop Lund, and Stefano Nolfi. Evolving mobile robots in simulated and real environments.

[14] http://www.robocoderepository.com.