Genetic Programming for Robocode Strategy

Danny Wyatt and Dan Klein
CSE573 Autumn 2003

December 18, 2003

Abstract

We present an approach to learning Robocode controller strategy. Robocode strategies are represented as trees of atomic elements corresponding to actions and observations in a Robocode battle. Genetic programming is used to search the space of such representations. Through this approach, we were able to induce stategies capable of defeating many hand-coded tanks.

1 Approach

Our goal for the project was to induce a strategy of action for a Robocode controller. We did not want to learn basic Robocode actions—exactly how many degrees to turn the gun, exactly how far to advance forward—rather we wanted to learn a "higher level" behavior built from sequences of such actions. Our guiding principal was that the tank controller would only have to learn to make logical decisions over abstract, boolean features of its environment, and that it would have access to a menu of predefined actions. While the actions themselves might need to make use of continuos features of the environment or have access to trigonometric functions all of these underlying complexities would be hidden from the learned part of the agent. (This is in contrast to Jacob Eisenstein's [2] attempt at learning all of the math needed for each turn of the tank.)

2 Tank Representation

2.1 Atoms

The three fundamental pieces of our representation are actions, tests, and conditionals. Actions (as one would expect) cause the tank to perform some action in the game. Tests are boolean features of the battle environment. Conditionals perform actions based on the results of a test. We refer to the collection of all actions, tests, and conditionals as "atoms". Our tank is a collection of these atoms.

Example action atoms are `TurnParallelToNearestWall`, `AheadDistanceToEnemy`, `TurnGunToEnemy`, `Fire2`, and `TurnRadarRightLeft60`. Example test atoms are `Testenergybelow10`, `TestEnergyLessThanEnemys`, and `TestEnemyWithin10TicksOfFire1`. A complete listing of atoms appears in appendix A.

2.2 Collections of Atoms

At first, we considered collecting atoms in an ordered list similar to a sequential program. Conditionals (corresponding to `if` statements) would have an associated integer n and would execute the next n actions based on the result of the test. We would generate conditionals with different values n for each test and the negation of each test ahead of time. The conjunction of tests could be achieved by nesting conditionals. Given conjunction and negation, disjunction could also be achieved—but only if the program independently learned the application of DeMorgan's law. Since we wanted our tanks to learn strategy from precomposed actions, it made sense to also provide precomposed logic. They did not have to learn the fundamentals of Robocode, so they should not have to learn the fundamentals of logic.

Instead, we chose the traditional representation for genetic programming: trees [5]. The above representation is equivalent to trees that branch at each conditional and are rooted at an implicit conditional that always evaluates to true. Similarly, Eisenstein's TableREX controllers can be represented as trees with duplicate subtrees for rows whose outputs are reused (which actually makes them branching programs, but they are expressible as trees nonetheless). The fundamental property that all of these share is that they are acyclic: execution never loops.

Our trees branch at one of three provided conditionals: `And`, `Or`, and `If`. `And` and `Or` each have two children. They have short-circuit execution, so `And` only executes its second child if its first child evaluates to true, and `Or` only executes its second child if its first child evaluates to false. `If` has three children: it either executes its second child if its first child evaluates to true, or it executes its third child if its first child evaluates to false. When a test is executed it returns its value. When an action is executed it returns whether or not it completed successfully—our current action atoms always complete successfully. To fit the event-driven architecture of Robocode tanks, we use five such trees for a tank controller. There is one tree for the main execution loop, and one tree for each the following event handlers: `onScannedRobot`, `onHitByBullet`, `onHitRobot`, and `onHitWall`. The trees are serialized into a traditional LISP-like representation corresponding to a preorder traversal which can be loaded and interpreted by the core tank program.

The observed state of the battle environment is saved in a global table that is automatically updated when any information arrives via an event. Information that is inferable from the event (such as the angle to the enemy given the x and y location of the bullet that just hit him) is also filled in automatically. The learned tank controller, as defined by our trees, only has access to this state information through the test atoms. That is, the only features of the battle environment that can be learned to be used or not used explicitly are those exposed through tests. Action atoms make use of other data in the table, but this data is not fully exposed to the decision making of the conditionals. Since the table is global, and all event trees have access to the same state information. Data in the table can go out of date, and actions that rely on state information will continue using the old data until new data arrives.

3 Search Space

Given this representation, our search space is the space of all tanks representable as trees of our atoms. Since we have two binary branching nodes and one tertiary branching node, the average branching factor for our trees is 7/3. The trees also have some fixed maximum depth d, where d is between 4 and 10 (see Section 4.3.4). This means that our search space contains $O((7/3)^{d+1})$ tanks. It is a discrete space, and each point in the space (each tank) does not have a well-defined set of neighboring points. The relief of our search space is determined by each tanks competitiveness in Robocode, which is explained more in Section 4.2.

4 Genetic Programming

To search this space we use genetic programming. Genetic programming (GP) is an optimization technique based on biological evolution[5]. Its goal is to produce a computer program that maximizes a given fitness function. This section outlines the basics of GP, explains how GP was used to optimize our tank, and examines GP parameter selection. If you are an expert at GP, please feel free to skip to section 4.2

4.1 The Basics of Genetic Programming

There are three main components of GP: the fitness function, the selection process, and the genetic operators. The fitness function determines how close to optimal a program is a returns an associated fitness value for that program. The selection chooses among programs based on the fitness functions evaluation of them. The genetic operators transform one or more programs into new programs. There are the three genetic operators: cross-over, copy, and mutation.

The entire GP process is stochastic search through the space of all possible programs. A state in the search space is a set (or population) of individual programs. The transition between states is the transformation of one set of individuals into a new set through the three genetic operators, guided by the selection process based on individuals' fitness values. The initial state is a set of randomly generated programs. The search ends for any number of reasons: a certain number of generations have been examined, a fitness goal is attained, or a time limit is met.

The process of selecting individuals from a population based on their fitness is well studied [3, 4, 5]. A common approach is to weight each individual's chance of selection according to its fitness and then choose uniformly over the weighted individuals. This "weighted roulette wheel" approach has many drawbacks. The biggest is that individuals' fitness values converge as the search proceeds, and this selection algorithm then has difficulty choosing accurately between close fitness values. This is the selection method Eisenstein employed.

A better selection method is n-Tournament selection [1]. This method works by randomly choosing n individuals from the population and returning the one with the highest fitness value. Note that this is based on relative fitness and thus continues selecting effectively even as the population's fitness values converge.

Once individuals have been selected from the population, one of the three genetic operators is applied to the selected individuals. Since our programs are represented as trees, these operations transform the program trees. Crossover, the primary state change operation for GP, takes two individuals and swaps subtrees between them at randomly chosen positions. Copy is as simple as it sounds: an individual is copied unchanged into the next generation. Mutation works on a single individual by substituting a newly generated random subtree for a randomly chosen node in an individual. Mutation maintains diversity of tree contents against the converging effects of selection and helps to push the search out of local maxima.

4.2 Our GP Implementation

Since each of our tanks is actually represented as five separate trees, the transition operations are performed within each tree category only. That is, an `onScannedRobot` tree is only crossed over with another `onScannedRobot` tree, an `onHitWall` with an `onHitWall` and so on. But, the entire individual—all five trees—is assigned a single fitness value.

To evaluate the fitness of each individual, we ran multiple Robocode rounds against an opponent. As detailed in the results section, the choice of an opponent has a large influence on the learned strategy. It is necessary to run multiple rounds per individual because the random initial conditions have a strong affect on score. Following Eisenstein and the default number of rounds per battle in Robocode, we ran ten rounds per individual.

The Fitness Function

We experimented with three different fitness functions. Initially, we used the raw Robocode score of our tank as the fitness function. As expected, it proved ineffective for producing winning tanks since it does not consider the enemy's score. It was often the case that the behavior that maximized our Robocode score also maximized the enemy's score, and our tank still lost the battle.

We soon changed to a fitness function that was the difference between our tank's Robocode score and the opponent's score. As described in section 5.3.2, the immediate result of this fitness function was suicidal tanks. After correcting for suicides, the tanks evolved successful dodging strategies. Dodging is a local maximum on the fitness landscape with a peak fitness of a mere 60 points per round. Dodging is *not* the global maximum. Higher scores can be attained by shooting the enemy. Thus, to guide the search toward shooting tanks, we added (or re-added since the Robocode score does add them once already) the assigned bullet damage points and bullet damage bonus points to the score difference. This fitness function proved successful and resulted in competitive fighting machines.

It should be kept in mind that the raw Robocode score is always in some sense our fundamental fitness measure or utility, inasmuchas tanks that score well also tend to survive their battles. However, the Robocode score did not always make for the best fitness *function.* By changing the fitness function we change the relief of our search space, but always with the hope that the new elevations correspond to genuine elevations in the "underlying" utility.

4.3 GP Parameters

The performance of GP is largely determined by problem specific parameters including population size, selection intensity, crossover rate, mutation rate, and maximum program size. The problem specific nature of these parameters comes from the relative smoothness of the fitness landscape. This section highlights the challenges of parameter selection and explains the parameters we chose.

4.3.1 Population Size

The size of the population is related to the genetic diversity of the population. Because genetic diversity is associated with the "takeover time"—the time it takes for a population to become mainly copies of a few similarly fit individuals—a population of about 5,000 members is typically used for GP. However, the larger the population the longer it takes to evaluate all individuals between each state transition. For our tanks, this was a nontrivial task requiring approximately 500 milliseconds per evaluation. The question becomes, given a limited amount of time to reach a solution, is it better to have a large population and evaluate only a few generations or have a small population and evaluate many generations? The population size parameter is directly related to the tradeoff between a small, hill-climbing search and a broader, more complete search. In the end, we decided a population size of 150 gave a good tradeoff between genetic diversity and time.

4.3.2 Selection Intensity

Tournament selection was used in our implementation to avoid the problems associated with the roulette wheel method. The tournament size parameter determines the selection intensity, which can be measured by the takeover time [1, 5]. A larger tournament size results in a greater selection intensity, shorter takeover time, and greater loss of genetic diversity. Because our fitness landscape has many local peaks, it was necessary to keep the tournament size to its minimum: 2 individuals per tournament.

4.3.3 Rates for Genetic Operators

The rate of each genetic operator is the percentage of the population (chosen via the selection method) to which each operator is applied in order to transition to the next generation.

Crossover Typical crossover rates range from 75% to 95%. We choose a cross-over rate of 87% based on Koza [5] and Eisenstein [2].

Mutation Mutation rate must be chosen carefully because a value that is too large risks losing fit members. We choose a moderate rate of 3%. As with crossover, this value is in accordance with Koza [5].

Copying Copying's purpose is to carry on a small portion of the population. All remaining individuals, (the last 10%) were copied to the next generation. We learned that elitism—copying the absolute fittest members of a generation into the next generation—should not be used with a small population. Copying even just the single fittest member into the next generation resulted in more of a short, steep hill-climb than a broad search.

4.3.4 Tree Depth

The final parameter for our search is the maximum depth of the program trees in our tanks and the depth of the trees generated for mutations. The larger the program trees, the more tanks there are in the search space. We experimented with values between 4 and 10 and found 10 to provide enough diversity at first to produce competitive tanks. As the search proceeds, the trees can get shorter and they frequently do.

For subtrees produced for mutation, the larger the tree the more of the original program it replaces. Since a random subtree is likely to be less fit that than an evolved subtree, mutations are kept small to keep a mutated individual from losing too much fitness and falling out of selection for the next generation. Our mutation subtree depth was set at 3 when maximum program tree depth was 10.

5 Results

Our experimental setup consisted of a 2.4GHz server with 2GB RAM running the GP and up to six[1] 2.4GHz clients running Robocode battles to obtain fitness values. This parallel setup allowed us to run 15 Robocode battles per second, corresponding to about two individuals per second (each individual's fitness was based on ten rounds). Over the course of a month of evolution, we performed nearly 5 million Robocode battles. This section highlights results of these battles and demonstrates our design evolution by describing the motivation for, results of, and analysis of each trial.

5.1 A Typical Training Result

All of the training trials presented later in this section resulted in a common trend of fitness over generations, as shown in Figure 1. This figure displays several important features that will be used in the analysis of our data. As one would expect, the first generation has a very low fitness corresponding to its completely random make-up. Fitness then improves rapidly over the first few generations as improvements are made to the trees. Physically, this rapid increase in fitness corresponds to changes in strategy. Once a local maximum fitness is attained, the slope rapidly decreases. This corresponds to one strategy taking over the population. The generation at this slope reduction point is commonly referred to as the takeover time.

After the takeover time, the GP algorithm continues to improve the best and average fitness values, just at a slower rate. This period corresponds to enhancement of the converged behavior. Typical enhancements include pruning unused atoms and exchanging other atoms.

[1]We wish to thank Professor Kristi Morgansen for the use of the computers in her Autonomous Underwater Vehicles Lab.

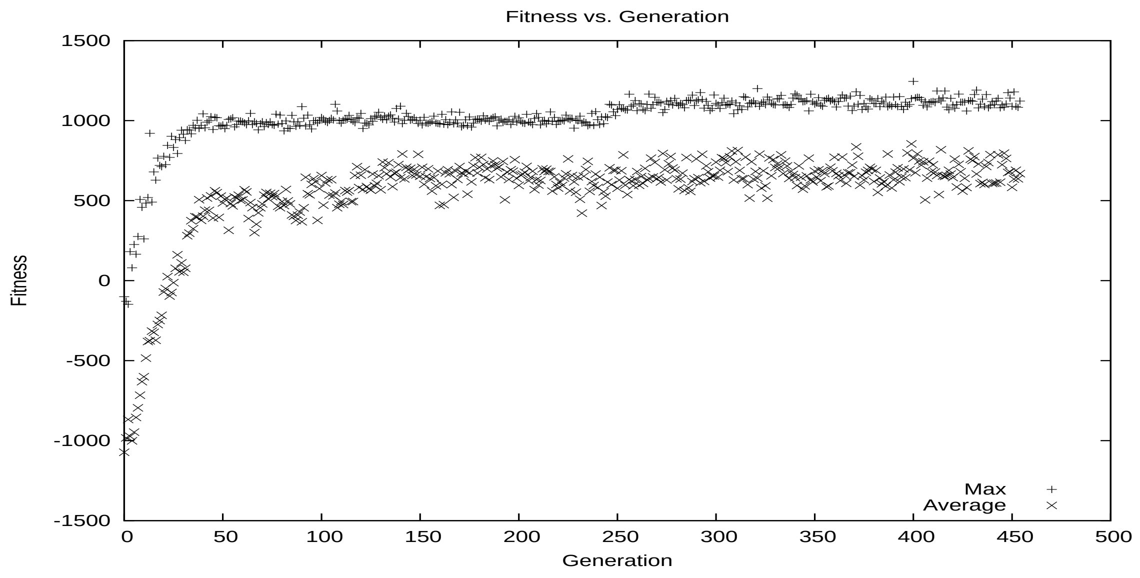

Figure 1: Maximum and average fitness values vs. generation for a typical training example.

5.2 Training Trials

Trial	Opponent	Pop. Size	Fitness	Rounds	Tree Depth	Elitism	X-Over	Mutation	Copy	Gens	Avg. Score Diff.
1	Crazy	10	A	1	4	Yes	70%	4%	10%	57	11.4
2	Crazy	50	A	2	5	Yes	70%	4%	10%	250	16.9
3	Crazy	50	A	3	10	Yes	70%	4%	10%	109	47.7
4	TrackFire	50	B	5	5	Yes	70%	4%	10%	3	N/A
5	TrackFire	50	B	5	5	Yes	70%	4%	10%	7	N/A
6	Crazy	100	B	10	10	No	87%	3%	10%	456	78.7
7	SquigBot2.8	150	C	10	10	No	87%	3%	10%	304	-21.9
8	SandBoxDT	150	C	10	10	No	87%	3%	10%	1007	-118.3
9	SquigBot2.8	150	C	10	10	No	87%	3%	10%	1357	-3.0

Table 1: Fitness functions are (A) raw Robocode score, (B) score difference, (C) score difference + bullet damage + bullet damage bonus.

The specific trials in Table 1 were chosen to show our design evolution and illustrate the results of our approach. Many other trials were run, but their results were not unique or otherwise substantial. The tanks we submitted for all three class tournaments came from trial 7.

5.3 Trial Motivation, Results, and Analysis

5.3.1 Trials 1-3: Initial Tests

Motivation Our first trials were run simply to test the GP, battle manager, and tank. The trials were run with extremely constrained population size, maximum tree depths, and number of rounds.

Results To our surprise, within 20 generations our algorithm evolved a tank similar to sample.TrackFire and sample.SpinBot that was able to beat sample.Crazy consistently despite never moving. Specifically, it turns the gun to the left constantly during the main loop. When it scans the opponent, it fires and then jerks the gun back 10 degrees to the right. The gun then begins moving to left again, and the behavior repeats. In this way, the gun is kept aimed at the opponent. When the opponent runs into the tank—as sample.Crazy is wont to do—the tank shoots. Since the gun is already aimed at the opponent and the opponent is always next to the gun these shots always hit.

Analysis These results were very promising in that they proved our concept. The co-evolution of the main loop and the `onScannedRobot` events allowed target tracking. The fact that the algorithm converged very quickly shows that elitism on a small population results in a near hill-climb.

5.3.2 Trials 4-5: Suicide Elimination

Motivation At this point, we wanted to evolve a tank that moved. To accomplish this, we trained against a new opponent, sample.TrackFire, who remains stationary and fires at his opponent. Any tank that does not move when fighting against sample.TrackFire will be quickly defeated. In the same trials, we switched to the score difference fitness function. We felt comfortable changing two parameters at once because others have used this same fitness evaluation function [5].

Results The GP surprised us again. We evolved suicidal tanks.

Analysis If a tank does not perform any action within 600 milliseconds of the start of a round, Robocode sets the tanks energy to zero (thus disabling it). A tank with zero energy can be shot at most one time before being destroyed, and the opponent earns no points since no energy is lost. Our tanks learned to allow themselves to be disabled, thus minimizing the opponent's score by not allowing him to score any points. To correct this, we modified our tanks to perform the equivalent of a no-op once in each main loop. After this correction, idle tanks took full damage and were accordingly selected out of the population.

5.3.3 Trial 6: Elimination of Elitism

Motivation With suicidal tanks out of the way, we wanted to see how the modified algorithm would do against sample.Crazy, our baseline opponent. At this same time, we discovered that elitism was causing the GP to act like a hill-climb. The primary motivation of this trial was to evaluate the non-elitist GP parameters.

Results We evolved a left-wall following behavior. When the top corner of the left wall is reached, the tank turns 180 degrees to the right. Similarly, when the bottom corner is reached, the tank turns 180 degrees to the left. In other words, the tank always turns facing into the battlefield. This allows the radar, which is fixed to the gun, to pan across the battlefield. When the enemy is scanned by the radar, a `Fire3` is executed.

Analysis This was a great success. We knew that the tank was capable of wall following so it was exciting to see this behavior in an evolved optimized form. It is important to note that we did not need to fix the starting position like Eisenstein did. This behavior took longer to evolve, because of the lack of elitism, but the tank attained a higher overall fitness. In general, learning slower results in more fit behaviors.

5.3.4 Trial 7: SquigBot

Motivation We wanted to test the algorithm against one of the better (or so we believed) hand-coded tanks in the world. We also increased the population size to 150 to further slow the learning process in hopes of increasing the ultimate fitness.

After restarting this trial several times, it was determined that a firing bonus should be added to the fitness score to prevent evolution of an entirely defensive behavior. This trial was the first with the fitness function labeled C in Table 1.

Results The peak behavior for this trial was a wall following bot much like sample.Walls. Again, the `onScannedRobot` event handler has a fire action. This simple behavior was able to beat SquigBot and all of the sample tanks on nearly every round from random starting spots. This is a great improvement over Eisenstein's accomplishments.

Analysis This was also a success. One interesting aspect of the evolved behavior is that, once the bot makes a 90° right turn at a corner, it learned to rotate the gun back left 90°. This co-evolved action always keeps the gun pointed into the arena resulting in a better chance of scanning and shooting the enemy. Without the added bonus for shooting, the bot would have evolved a defensive strategy that would have worked well against SquigBot, but poorly against other tanks. This is the tank we submitted for all three CSE573 Robocode tournaments.

5.3.5 Trial 8: Better Opponents

Motivation Because we were able to easily beat SquigBot, we wanted a real challenge. SandBoxDT the bot of the year, consistently placing in the top three in all major competitions.

Results SandboxDT is very, very good. Our tank had no chance against his near-perfect aim and dodging skills. After more than 1000 generations, we converged on an oscillating behavior similar to sample.MyFirstRobot. This behavior was not very successful against the sample set and did not come close to beating SandboxDT.

Analysis We had no chance against SandboxDT because no controller existed in our search space capable of beating him. This is discussed in detail in Section 6.2

5.3.6 Trial 9: New Atoms

Motivation We knew what we needed new actions. This trial was conducted with many new action atoms including `AheadArcRight100x90`, `BackArcLeft200x45`, `MoveToUpperLeft` and other curvilinear and simultaneous turning and moving actions.

Results This result was incredibly surprising: we evolved a *teleporting* wall follower. The tank learned to use the new `MoveToCorner` atoms to create a new type of wall follower. In doing so it found and exploited a bug in Robocode that allowed it to jump from the lower left corner to the upper left corner. Obviously, this was a superb dodging strategy. It left the opponent spinning his radar with no idea where our tank was.

Analysis GP finds the oddest peaks. The tank used the new atoms to enhance dodging, but still fire power was weak. Additional atoms for radar and firing will be necessary to make a more competitive tank. We determined that this tank was no better than the wall following tank we evolved against SquigBot on Trial 7, so we did not submit this tank.

6 Conclusions

6.1 Successes

We were able to evolve a general tank capable of beating all of the sample tanks (including sample.Walls and sample.Tracker) and SquigBot using random starting positions. This is more than Eisenstein's evolved tanks could do, and attribute that success to our higher level representation of actions. Searching over a space of strategies rather than over a space of minute, low level actions seems to provide more success at evolving Robocode tanks.

6.2 Extending the Limits of our Representation

As the last two experiments show, there are opponents against which we cannot learn winning behavior. This is attributable not to our search or overall representation scheme, but to the specific atoms that we have included in our representation. It is easy to extend our representation with new atoms, and we believe new atoms could significantly improve our tank's competitiveness.

Movement

The rectilinear movements available to our tanks in the first 8 trials is very unsuited for battle against opponents with predictive targeting ability. In trial 9 we attempted to fix this. We added curvilinear motion atoms and atoms that turn and move to fixed points on the battlefield. The move-to-point atoms were favored in subsequent learned behaviors, but only the ones for moving to the corners—so the tanks are still wall-followers and do not fare well.

For a larger impact we plan to add "wiggle" atoms corresponding to each motion atom. These would move the tank the same distance and in the same general direction but would randomly change headings along the way. More generally, we need more random motion atoms altogether. A sufficiently random tank should be completely unpredictable, but it would also be completely unstrategic. We would like to use our search to learn a balance between the two.

Targeting

Given the targeting and firing atoms available to our tank, there is not much hope it can learn advanced shooting strategies. However, we still believe that there are atoms that can be dropped in to the representation to correct for this. Many hand-coded tanks use variations of a few targeting schemes (linear, circular, virtual bullet/wave) and corresponding atoms such as `TurnGunToCircularTargetingAngle` could be added.

6.3 Comparison to our Classmates' Tanks

Adding atoms for existing targeting strategies will pull our tank in the direction of Lincoln Ritter and Lucas Kreger-Stickles' Chimera: we would assemble behavior from actions taken whole from other tanks. Unlike Chimera however, our more granular approach to behavior would allow for a strategy comprising actions mixed from many different tanks simultaneously—not a strategy that must adopt all of one tank's actions at one time.

There seem to be two general axes along which to consider the learning approaches of the tanks made for the course: online vs. offline and holistic vs. modular. Our tank is offline and holistic. Holistic tanks learn their entire behavior at once, where modular tanks separate the learning tasks into subproblems (most commonly targeting and moving). The results of the class tournaments suggest that modular, online learners are the most competitive. However, these tanks have a strategy that is fixed according to their initial modular composition. In other words, they have a fixed behavior of flexible actions, while we have a flexible behavior of fixed actions.

The obvious next step is to try flexible behaviors of flexible actions. If we added, for example, a neural net targeting mechanism and corresponding atoms, could our tanks learn to use that when appropriate? Could the targeting mechanism learn to cooperate inside the larger strategy? Would the two be able to overcome their initial low fitness values together, or would one dominate the learning and simply learn around the shortcomings of the other?

6.4 Alternative Search Techniques

We have previously mentioned the size of our search space in terms of the number of tanks within it. But there are also implicit states (call them battle states) observable by one of these tanks during a battle. Since we have 24 test atoms there are 24 boolean state features, and thus 2^{24} battle states. Our search implicitly tries to factor battle state space according to the most valuable features, where valuable means a feature's appearance in a tree increases the fitness of the tank. Of course, tests and actions are treated indistinguishably in our current representation.

We could separate tests and actions (as our first considered representation did) and use an alternative search technique to map from battle states to actions. Such a representation would be amenable to use with an MDP or reinforcement learning. Indeed, if the tests and actions were kept identical while the larger tank controller structure changed different searches could be performed over a (mostly) equivalent controller space. Evaluating the controllers found by each search would provide a way to comparatively rating different search techniques for this problem.

6.5 Team Member Responsibilities

Overall, we shared the work load 50/50 and worked well as a team. We both discussed all of our projects design and implementation details, and then we split the implementation chores. Danny Wyatt focused on the representation and coded the associated Robocode tank, atoms, and tree parser and interpreter. He also coded the battle manager. He wrote the sections of this paper on our approach, representation, search, and conclusions. Dan Klein focused on the genetic programming algorithm and coded the associated GP server with its population manager, selection method, fitness function and genetic operators. He wrote the GP and results section of this paper.

References

[1] Tobias Blickle and Lothar Thiele. A mathematical analysis of tournament selection. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 9–16, San Francisco, CA, 1995. Morgan Kaufmann.

[2] Jacob Eisenstein. Evolving robocode tank fighters. Technical report, MIT AI Lab, May 2003.

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

[4] Frank Hoffmeister and Thomas Back. Genetic algorithms and evolution strategies: Similarities and differences. 1990.

[5] John R. Koza. Genetic programming: On the programming of computers by means of natural selection. *Statistics and Computing*, 4(2), 1994.

A Atoms

Where no description is given, each atom does exactly what its name suggests.

A.1 Logic

`And` Executes its first argument, if it returns true, it executes the second argument and then returns the conjunction of both results.

`Or` Executes its first argument, if the first argument returns true, it returns true. Otherwise, it executes the second argument and then returns the disjunction of both results.

`If` Executes its first argument, and then the second if the first returned true, the third otherwise.

`Not` Executes its argument and inverts its result.

`False` Simply returns false.

`True` Simply returns true.

A.2 Actions

A.2.1 Moving

`Ahead50`

`AheadDistanceToCenter` Moves forward the distance between the tank and center of the battlefield, regardless of the direction the tank is facing.

`AheadDistanceToEnemy`

`AheadDistanceToWall`

`AheadHalfDistanceToEnemy`

`Back50`

`BackDistanceToCenter`

`BackDistanceToEnemy`

A.2.2 Turning

`TurnToCenter` Turns the tank so it is directly facing the center of the battlefield.

`TurnAwayFromCenter` Turns the tank so it is directly facing 180° away from the center of the battlefield.

`TurnAwayFromEnemy` Turns the tank so it is directly facing 180° away from the last observed position of the opponent.

`TurnPerpendicularToEnemy` Turns the tank so it is directly facing 90° away from the last observed position of the opponent. It chooses the shorter direction to turn.

`TurnParallelToNearestWall` Turns the tank so it is parallel to the nearest wall and facing towards the longest stretch of that wall (facing into the battlefield).

`TurnPerpendicularToNearestWall` Turns the tank perpendicular to the nearest wall, facing into the battlefield.

`TurnLeft10`

`TurnLeft90`

`TurnRight10`

`TurnRight90`

`TurnToEnemy` Turns the tank so it is facing the last observed position of the enemy.

A.2.3 Simultaneous Moving and Turning

These atoms were added only for our last experiments against SandboxDT and Marvin.

`MoveToCenter` Turns the tank to either face towards or away from the center—whichever is faster—and then moves either ahead or back until the tank is at the center of the battlefield.
`MoveToLowerLeft` Turns and moves similarly to only to the lower left corner of the battlefield.
`MoveToLowerright`
`MoveToUpperleft`
`MoveToUpperright`
`AheadArcLeft100x45` Moves forward 100 pixels while turning left 45°.
`AheadArcLeft100x90` Moves forward 100 pixels while turning left 90°.
`AheadArcLeft200x45` Forward 200, turning left 45°.
`AheadArcLeft200x90` Forward 200 pixels, turning left 90°.
`AheadArcRight100x45` Forward 100 pixels, turning right 45°.
`AheadArcRight100x90` Forward 100 pixels, turning right 90°.
`AheadArcRight200x45` Forward 200 pixels, turning right 45°.
`AheadArcRight200x90` Forward 200 pixels, turning right 90°.
`BackArcLeft100x45` Back 100 pixels, turning left 45°.
`BackArcLeft100x90` Back 100 pixels, turning left 90°.
`BackArcLeft200x45` Back 200 pixels, turning left 45°.
`BackArcLeft200x90` Back 200 pixels, turning left 90°.
`BackArcRight100x45` Back 100 pixels, turning right 45°.
`BackArcRight100x90` Back 100 pixels, turning right 90°.
`BackArcRight200x45` Back 200 pixels, turning right 45°.
`BackArcRight200x90` Back 200 pixels, turning right 90°.

A.2.4 Gun

`TurnGunToEnemy` Turns the gun to the last observed enemy location.
`TurnGunLeft5`
`TurnGunLeft10`
`TurnGunRight5`
`TurnGunRight10`
`Fire1`
`Fire2`
`Fire3`

A.2.5 Radar

`TurnRadar360`
`TurnRadarLeft20`
`TurnRadarLeft60`
`TurnRadarRight20`
`TurnRadarRight60`
`TurnRadarRightLeft20` Turns the radar right 10°, left 20°, then right 10°.
`TurnRadarRightLeft60` Turns the radar right 30°, left 60°, then right 30°.
`TurnRadarToEnemy` Turns the radar to the last observed enemy location.
`TurnRadarToGun` Realigns the radar with the gun.

A.3 Tests

`TestEnemyEnergy0`
`TestEnemyEnergyBelow10`
`TestEnergyBelow10`
`TestEnergyGreaterThanEnemys`
`TestEnergyLessThanEnemys`
`TestEnemyWithin10Ticks`
`TestEnemyWithin20Ticks`
`TestEnemyWithin50Ticks`

```
TestEnemyWithin10TicksOfFire1
TestEnemyWithin10TicksOfFire2
TestEnemyWithin10TicksOfFire3
TestEnemyWithin20TicksOfFire1
TestEnemyWithin20TicksOfFire2
TestEnemyWithin20TicksOfFire3
TestEnemyWithin50TicksOfFire1
TestEnemyWithin50TicksOfFire2
TestEnemyWithin50TicksOfFire3
TestEnemyWithin5TicksOfFire1
TestEnemyWithin5TicksOfFire2
TestEnemyWithin5TicksOfFire3
TestGunIsHot
TestGunWithin5Ticks
TestTurnToEnemyWithin10Ticks
TestTurnToEnemyWithin5tTcks
```