Robocode and Genetic Programming

Ted Hunter

Introduction:

This project was an attempt to develop and implement a genetic programming algorithm for Robocode. The main goal was to gain a general understanding of genetic programming in both practice and theory. Because the focus was on learning rather than results, no outside genetic programming implementations or libraries were used.

The results of my project have been much more successful than originally anticipated, and my initial objective of generating programs that defeat novice robots proved to be too easy. This made setting a specific target in terms of performance difficult, and so I simply focused on maximizing the success of the generated programs in general. As my genetic programs became able to defeat each opponent, I continued to set my sights on more and more challenging opponents, refining my program along the way. The included figures focus on the final results of the project. The opponents used for generating the data were the most challenging ones that still provided useful feedback.

Robocode:

Robocode is a programming game in which players design virtual tanks (Robots) that compete with one-another, rather than the direct player-vs-player interaction of traditional games. Robocode was initially released in 2001, and has developed a steady following. Similar games such as Crobots and Core Wars date back to mid-to-late 1980s. The game itself consists of a central Java program that runs battles under conditions specified by the host. Robocode has a fully developed API in both Java and .NET which is used in conjunction with standard libraries to create Robot class files.

There are multiple leagues within the Robocode community, most of which take part in regular online competitions. Popular leagues include 1-vs-1, Melee (multiple opponent free-for-all), Team, and TwinDuel (2-vs-2 survivalist). The size of a robot program determines which leagues it can compete in. There are six major divisions based on code size: HaikuBots, NanoBots, MicroBots, MiniBots, MegaBots, and Unrestricted. This factors heavily on implementing a genetic programming approach as the resulting program is usually bloated with inaccessible or redundant (“junk”) code.

In each round, the Robocode battle program instantiates a Robot object from each competitor's Robot-inheriting class file. Turns follow an unchanging order, and for each turn, the robot is allotted one tic of the virtual machine for computation and action. If a robot is unable to finish its computation within the tic, it's turn is skipped and the computation will continue on its next turn.

Previous work in this topic (Shinchel, et al) focused on the HaikuBot division as the size limit was based on the number of semicolons used rather than the number or bytes. This approach made sense as genetic programs tend to produce long lines of code. Since then, the HaikuBot division has disappeared in the current Robocode community. As I was more interested in the evolution of advanced strategies than the ability to compete, my programs' size limits were determined by computation-time rather than overall size. This allowed my base Robot design to implement variables at different scopes – an important factor for evolving more advanced combat techniques. For this reason, any competition robots designed by the GP will likely be restricted to the larger code-size divisions.

Genetic Programming (in general):

Genetic programming (GP) is a form of evolutionary computation in which compilable and executable programs are generated through a process of natural selection. Unlike normal genetic search algorithms, the goal of the search is to find an optimal program rather than specific variable values.

As stated in *Clever Algorithms*, genetic programming and evolutionary programming are both specialized forms of evolutionary computation that generate entire programs. Whereas evolutionary programming takes place at the level of machine-instructions, genetic programming is usually restricted to higher-level programming languages (commonly LISP). A unique feature of genetic programming is that it takes on a tree structure as opposed to a linear structure such as a bit-string. The fundamental concepts of evolutionary computing remain true (Brownlee, 100, 120).

An important detail about the structure of the GP tree is that it consists of two specific types of nodes: function nodes and terminal nodes. Function nodes are functions with an arity of one or more. Terminal nodes are both zero-arity functions are numerical constants. The arity of a node determines the number of child nodes it must have. This distinction also serves in setting the upper and lower bounds of the GP tree's size. The lower bound determines the minimum depth under which no node can be a terminal. The upper bound determines the maximum depth after which all nodes *must* be terminals.

The genetic operators in a GP are modified from traditional genetic algorithms to suit a tree structure. Crossover is done by selecting one or more subtrees from one parent and inserting them into randomly selected subtrees of another parent. Mutation is done similarly, simply generating a random subtree to insert instead of selecting one from a second parent. Other reproduction operators include replication and architecture changes. Replication simply passes a Robot from one generation to the next. Architecture changes modify the rules the GP must conform to (such as changing the minimum/maximum depths of the tree).

Robot Program Layout:

Robocode Robots are fairly limited in structure. There must be a *run*( ) method that defines the Robot's behavior while no other events are occurring. The Robot class can also contain any subset of the Robot abstract methods (Fig. 1). These methods are called by Robocode's event handler when combat-related events happen. The main events my GP focused on were:

```
onScannedRobot(ScannedRobotEvent e)
onHitByBullet(HitByBulletEvent e)
onHitWall(HitWallEvent e)
```

These events are called when one would expect, and contain bundled data within event 'e' that can be used to set variables. As the program developed, emphasis was placed on the *onScannedRobot( )* event, as I found that evolving code for the remaining events slowed the algorithm down far more than they increased fitness.

Within the *onScannedRobot( )* even, the GP evolves a series of instructions for where to move, turn, turn the Robot's gun and radar, and fire. In addition, the GP uses this event to set the values of static variables which could be called on by the other areas of the program (specifically within the *run( )* method). Each action method is its own GP tree with its own root, but the evolution process allows for genes to cross between the different trees or entire trees to be moved to different action methods.

Genetic Program Layout:

The genetic program created for this project consists of four Java classes: *RunGP*, *MetaBot*, *ExpressionNode*, and *BattleRunner*. The *ExpressionNode* class defines the characteristics of the nodes used in the trees, as well as the available expressions and low-level breeding helper methods. Each node contains a string array representing the expression itself, and an *ExpressionNode* array that holds each of its child nodes. Other class fields include the node's arity, depth, and whether or not it is a terminal.

Most of the methods used in this program are recursive, with the recursion itself being a method of the *ExpressionNode* class. The most important method of the class is *compose*( ), which uses a depth-first tree traversal, with each node compiling it's child nodes arguments to a string and adding its own expression before returning. Other methods involve assigning appropriate expressions based on depth, and finding and returning a specific node for breeding methods.

The *MetaBot* class represents the actual output of the program, which are compiled Java class files. Each *MetaBot* object has an array of *ExrpessionNodes* that act as roots of the expression trees. When using static variables there are seven trees, but normally there are five. The main body of the Robot's code is represented as a string. On compiling Robots, the *MetaBot* method *setCode*( ) is used to insert string representations of the expression trees into the Robot's code. This class also handles most of the file management and all compilations.

The *BattleRunner* class serves as the interface for the Robocode program itself. It initializes a Robocode engine and maintains it throughout the run. The engine is used to run the battles and provides feedback in the form of scores and other relevant information. The *BattleRunner* class then takes these scores and calculates the fitness of each robot. It was designed to be passed two string arrays: the file names of the GP robots, and the file names of the robots used in training.

Lastly, the *RunGP* class contains the actual genetic algorithm. It uses an array of *MetaBots* to represent the population, and handles the selection and determination of which genetic operator to use. The overall program architecture involves *MetaBot* building on *ExpressionNode*, and *RunGP* building on *MetaBot* and *BattleRunner*.

The most challenging design aspect of the program was figuring out how to regulate the growth/shrinkage of the expression trees. This was ultimately solved by developing an algorithm that determines which sizes of subtrees are usable in crossovers, finding such a subtree in the second parent, and finding a valid insertion point for it in the first parent. This was made difficult by the need to preserve random selection depths. The resulting algorithm can be seen in the *insert*( ) method in the *ExpressionNode* class, though it carries over to methods *insertAt*( ) and *getSubTree*( ). That growth becomes regulated can be seen in figure 3.1 below.

Determining Fitness:

For each program within the population, the GP must assess their fitness by actual execution. This was achieved through the Robocode Control API, which has built in engine control methods. For each generation, each member of the population is run against each opponent for a specified number of rounds. After each round, the default score (Fig. 2) is collected for both the GP robot and the opponent.

A relative score is calculated by dividing the GP robot's score by the sum of its score and the opponent's score. Because some early Robots do not fire and score a zero, an insignificant constant is added to their scores to ensure fitness reflects defensive abilities as well. The resulting relative score of each round is averaged out to produce the overall fitness of the Robot. A fitness of 0.50 or higher indicates that the Robot is equally matched to or better than its opponent.

Because Robocode is a highly non-deterministic game, any method for determining fitness needs to account for chance. The only way to minimize chance was to use a large number of rounds for each test. However, because Robocode rules dictate that each player has a timed turn based on the tic of the Java virtual machine, program execution speed was limited. This led to a compromise between accuracy and speed, with the determining factor being the number of rounds per test. The ideal number of rounds seemed to be around 25-30 for each GP robot. When multiple opponents were used, the total number of rounds was kept around fifty, divided equally between each opponent. The GP was set to end after 400 rounds, but I usually terminated it well before that, normally when the highest fitness hadn't changed within 20 rounds.

Breeding:

After determining the fitness of each member within the population, the GP can begin the breeding process. After experimenting, I found that competition based (tournament) selection was more effective than roulette selection. While roulette selection allowed for a slightly slower convergence within the population, the end results were insignificantly better. In competition-based selection initial growth was much more rapid, and the earlier convergence was compensated for by having more time to do multiple runs. The competition selection involved selecting six members of the population (four members when using populations under 200), and simply returning the member with the highest fitness.

A pigeon-hole selection was used to determine the breeding operation. The most successful values I was able to find were 0.85 chance of crossover, 0.5 chance of replication, and 0.1 chance of mutation (I did not implement architecture-alteration). While a 1% likelihood of mutation seems high compared to traditional genetic algorithms, Koza and Poli suggest this and it seemed to improve the end results of my program (Koza and Poli, 137).

The crossover method yields one child per call, and allows for a single Robot to be used as both parents. There is a 30% probability that the crossover will occur at the root of a GP tree, resulting in the child Robot handling a specific action exactly like the second parent. In addition, there is a 5% chance that a given root-crossover will place the second parent's GP tree in a different action method. There is a 10% chance that the crossover will occur at a terminal node. The program uses two different randomly selected crossover points from each parent.

Mutation involved selecting a random node within a subtree at a terminal in 15% of cases, at the root in 1% of cases. In the remaining 84% of cases, the node was selected by creating a probability distribution based on the minimum depth and the deepest node in the tree. The likelihood of each depth is uniform, which I found to be very effective as a completely random selection will favor very deep nodes due to an average branching factor greater than one. This method was applied to node selection in crossover as well.

Replication is very straightforward and doesn't allow for much variance. Both the best-so-far Robot and the best in the previous generation is replicated into the next generation in an effort to preserve exceptional but rare genes.

Results:

The overall results of the program have been better than anticipated. In most cases, it was able to generate Robots with finesses well above 0.50, meaning they are winning the majority of matches. When training against a single opponent, results were usually great, with many runs resulting in finesses of 0.85 or greater. However, generalization was not as successful. While more advanced Robot samples (from the developer website) did not prove too difficult, competition-level user's Robots could not be generalized for. This was somewhat remedied by using more variables with broader scope, but most runs resulted in convergence well before reaching finesses of 0.50.

I found that selection of opponents used in training usually evolved consistent strategies when the run was repeated, but were extremely different from strategies developed for other opponents. This was to be expected, but the consistency was more than I had anticipated. I tried using this to my advantage by scheduling different opponents at different points during the run, but the strategies tended to come out less successful.

Often, the Robots would discover very clever strategies. The most common example is that Robots usually develop a targeting system that allows them to track opponents as they both move around the arena. A less common strategy was switching to very low-powered bullets to conserve its own energy once the enemy becomes disabled. (As the robots shoot bullets and take damage, their energy is depleted. If a robot runs out of energy from shooting, it will stop moving and any damage will destroy it.) Similarly, some robots would learn to switch to low-powered bullets when their own energy dropped below a threshold. In terms of defense, it seems that the best strategies evolved included tricking the opponent by pointing in one direction before taking off in another.

Conclusion:

The program that resulted for this project is currently very specific to Robocode, but with some revisions it could be easily generalized to a much broader class of problems. The foundation for running the GP with many non-compatible Java events is already in place (seen as an "event" parameter in some methods). I would like to refine and reorganize the program to act as a library and use it in future endeavors.

Sources:

Brownlee, Jason. *Clever Algorithms: Nature-Inspired Programming Recipes*. 1st Ed. LuLu Enterprises, 2011. eBook.

Koza, John R., and Riccardo Poli. "Ch. 5: Genetic Programming." Trans. Array *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Edmund K Burke and Graham Kendall. 2nd Ed.Springer, 2005. 127-164.

Poli, Riccardo, William B. Langdon, et al. *A Field Guide to Genetic Programming*. Lulu Enterprises, 2008. eBook.

Shinchel, Yehonatan, Eran Ziserman, and Sipper Moshe. "GP-Robocode: Using Genetic Programming to Evolve Robocode Players" (2005): n. page. Web. 5 Dec. 2012. <http://www.cs.bham.ac.uk /~wbl/biblio/cache/http___www.cs.bgu.ac.il__sipper_papabs_eurogprobo-final.pdf. >

Figure 1: Most Used Structure for Generated Robots:

```
void run( ){
        turnGunRight(Double.POSITIVE_INFINITY);
        turnRight(runVar2);
        setAhead(runVar1);
}

void onScannedRobotEvent(Event e){
        setAhead(phenome[0]);
        setTurnRight(phenome[1]);
        setTurnGunRight(phenome[2]);
        setTurnRadarRight(phenome[3]);
        runVar1(phenome[5]);
        runVar2(phenome[6]);
}
```

* "phenome[ ]" represents the array of expression trees.

Figure 2: Robocode Default Scoring Equation:

Total Score = Bullet Damage + Ram Damage + Survival Score

Bullet Damage:	1 point for each point of damage from bullets hitting opponents
Ram Damage:	1 point for each point of damage from ramming into opponents
Survival Score:	60 points for winning + 20% of bullet damage + 30% of ram damage

* This equation is simplified to reflect 1-vs-1 combat. Melee combat scoring is more complicated.

Fig. 3: Data Collected from a Typical Run:

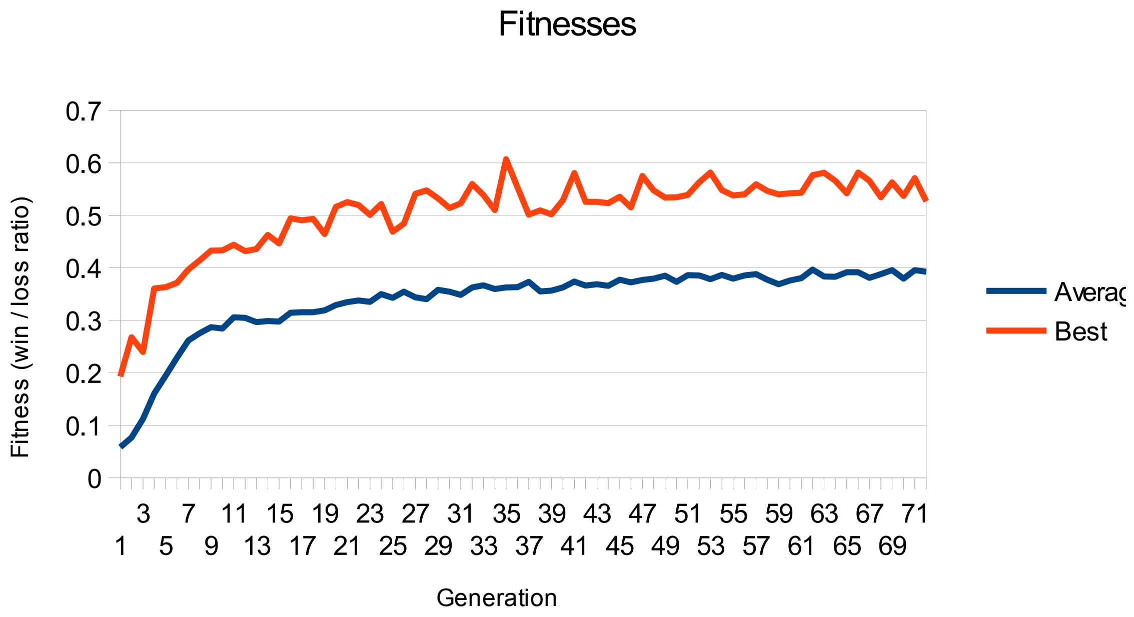


Figure 3.1: This graph illustrates the effect of convergence. As the most successful and prevalent traits spread throughout the population, improvement slows to a halt.

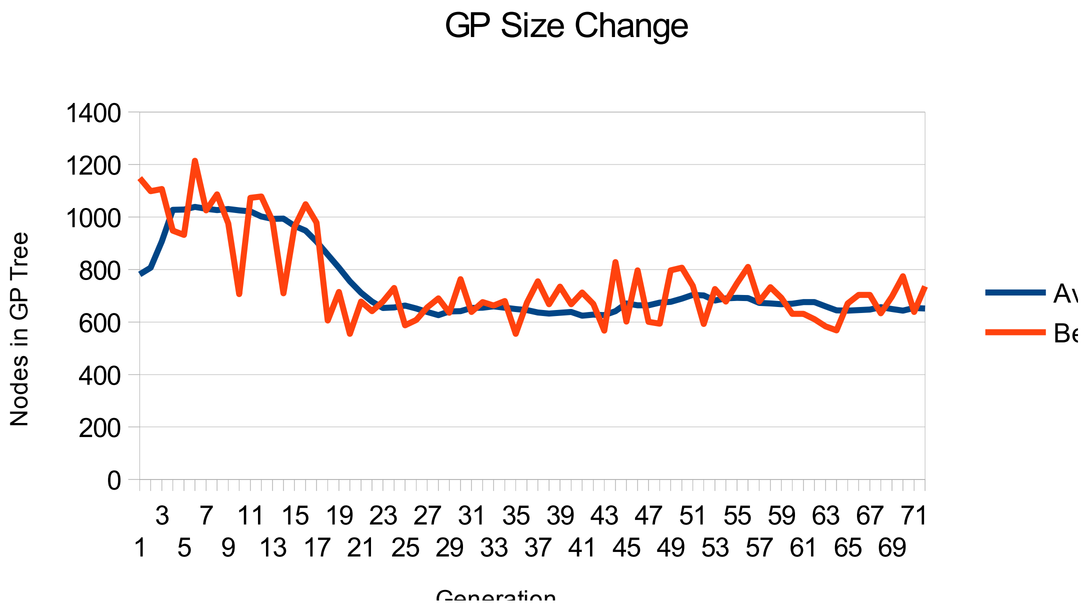
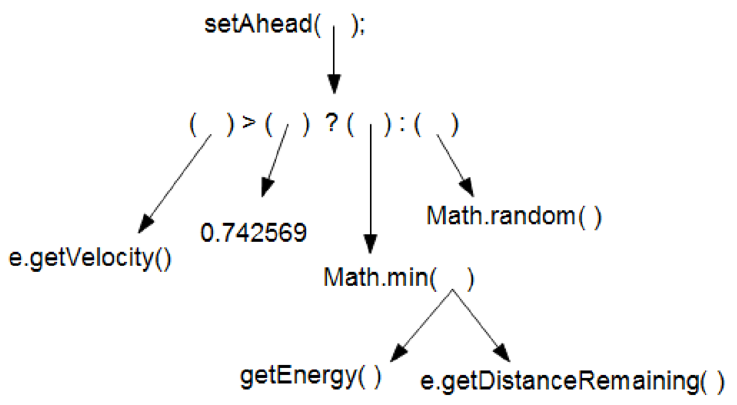


Figure 3.2: This graph illustrates the change in expression tree size over the run. The size can range from under 100 node to over 15,000. Resting around 650 nodes indicates balance between growth and shrinkage.

These were taken from a 36 hour run, terminated due to 20 generations without improvement. The second graph illustrates an equilibrium between growth and shrinkage. The run consisted of a population of 256, testing against multiple Robots from the "Super Sample Bots" collection found at http://robowiki.net/wiki/Category:Super_Sample_Bots.

Figure 4: Example ExpressionNode Tree



* This example is much smaller than the trees turn out in practice.

Figure 5: The Complete List of Expressions Used.

```
// Zero-Arity Expressions (terminals)
final static String UNIVERSAL_TERMINALS[] =
      {
            // from Robot API
            "getEnergy()",
            //"getGunHeading()",
            "getHeading()",
            "getHeight()",
            "getVelocity()", //???
            "getWidth()",
            "getX()",
            "getY()",
            // from AdvancedRobot API
            "getDistanceRemaining()",
            "getGunHeadingRadians()",
            //"getGunTurnRemaining()",
            "getGunTurnRemainingRadians()",
            "getHeadingRadians()",
            "getRadarHeadingRadians()",
            //"getRadarTurnRemaining()",
            "getRadarTurnRemainingRadians()"
      };

final static String CONSTANT_TERMINALS[] =
      {
            "0.001",                      // Zero, offset to avoid division issues
            "Math.random()",        // random value from [0, 1]
            "Math.random()*2 - 1",  // random value from [-1, 1]
            "Math.floor((Math.random()*10))",   // random integer from [1, 10]
            "Math.PI",                    // 3.14...
            "runVar1",                    // static variables
            "runVar2"                     //    - the GP defines these
            // Ephemeral Random Constants: Double.toString(random.nextDouble());
                  // must be generated during run time
      };

// terminals that can only be called during a ScannedRobotEvent
final static String SCANNED_EVENT_TERMINALS[] =
      {
            //"e.getBearing()",     // difference between enemy and robot heading
            "e.getBearingRadians()",      //in radians
            "e.getDistance()",      // robot's distance to enemy
            "e.getEnergy()",        // Returns energy (life) of enemy
            //"e.getHeading()",     // Returns direction enemy is facing
            "e.getHeadingRadians()",      // in radians
            "e.getVelocity()"       // Returns the velocity of enemy
      };



final static String FUNCTIONS_A1[][] =
      {
            {"Math.abs(", ")"},                 // Absolute Value
            {"Math.acos(", ")"},                // Arc-cosine
            {"Math.asin(", ")"},                // Arc-sine
            {"Math.cos(", ")"},                 // Cosine
            {"Math.sin(", ")"},                 // Sine
            {"Math.toDegrees(", ")"},           // Radians-to-Degrees
            {"Math.toRadians(", ")"},           // Degrees-to-Radians
            {"", " * -1"}                       // Flip sign
      };

final static String FUNCTIONS_A2[][] =
      {
            {"", " - ", ""},                    // add
            {"", " + ", ""},                    // subtract
            {"", " * ", ""},                    // multiply
            {"", " / ", ""},                    // divide (CHECK FOR ZERO!)
            {"Math.min(", ", ", ")"},           // minimum
            {"Math.max(", ", ", ")"},           // maximum
      };

final static String FUNCTIONS_A3[][] =
      {
            {"", " > 0 ? ", " : ", ""}          // X > 0 ? ifYes : ifNo
      };

final static String FUNCTIONS_A4[][] =
      {
            {"", " > ", " ? ", " : ", ""},      // X > Y ? if yes : if no
            {"", " == ", " ? ", " : ", ""},     // X == Y ? if yes : if no
      };

// All expressions available to the GP
final static String[][] TERMINALS =
      {
            UNIVERSAL_TERMINALS,
            CONSTANT_TERMINALS,
            SCANNED_EVENT_TERMINALS
      };

final static String[][][] EXPRESSIONS =
      {
            TERMINALS,
            FUNCTIONS_A1,
            FUNCTIONS_A2,
            FUNCTIONS_A3,
            FUNCTIONS_A4
      };
```

* Sorting all expressions by arity and placing into a single 3-d string array simplified expression assignment