Genetic Programming of Autonomous Agents

Functional Requirements List and Performance Specifi cations

Scott O'Dell

Advisors: Dr. Joel Schipper and Dr. Arnold Patton

November 23, 2010

Project Goals

The purpose of this project is to use genetic programming to develop control programs for autonomous vehicles. The initial focus shall be to implement a genetic programming framework. This framework will be used to evolve a **guard** agent that maintains a secure perimeter around a **base** which is being approached by **enemy** agents. This goal has the potential military application of using autonomous agents to protect an area or escort a convoy.

During early development, the project shall use crude, grid-based simulations to simplify the problem and to ascertain that the function set, terminal set, and fitness function are sufficient to meet the project's goals. As the project progresses, it shall use more complex simulators until the simulations approximate the continuous navigation and environmental noise of a physical autonomous agent. If time remains, the project shall focus on implementing the evolved programs on a physical system.

Top Level Description

The code written to connect the subcomponents shall be written in Ruby to simplify the process of interfacing subcomponents. If a simulator written in a different language becomes necessary as the project progresses, interface code will be written rather than re-implementing the entire system in the new language. Figure 4 shows the relation of software subcomponents.

To begin the simulation, a function set, a terminal set, and reproduction parameters will be provided to the genetic programming evolutionary sequence (GPES) block, which will organize the progression of generations. The randomly produced genomes of the first generation will be passed to a fitness function block that handles communication between the GPES and simulator subcomponents. The genomes are used to control guard agents in the simulator. The interaction of guard, enemy, and base agents within the simulator will determine the fitness of the genome. These fitness scores are passed back to the GPES where a new generation will initialize genomes by performing genetic operations on genomes of the previous generation. This process will continue until generation N is evaluated for fitness.

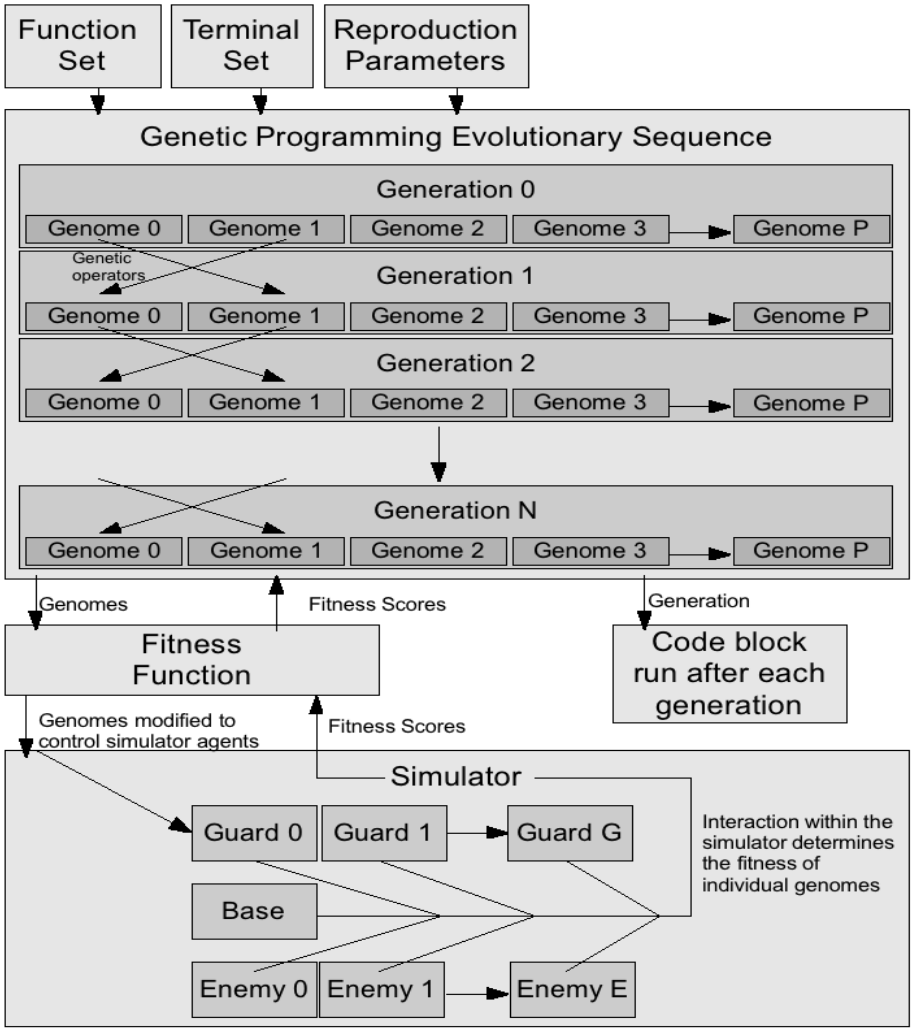

Figure 1: Top Level Software Architecture of Genetic Programming System

Subcomponent Descriptions

The final implementation of each subcomponent will depend heavily on problems discovered during early experiments. The general algorithm, however, shall remain largely unchanged.

Function and Terminal Set

The following set is designed to be as small as possible to avoid redundancy (which adds inefficiency to the evolution process) while still allowing the agent to:

- know its distance from the base,

- move to catch enemies,
- make logical decisions based on sensor inputs.

All these attributes are necessary to create sufficiently complex behavior.

The function set includes:

- prog
  - accepts 2 subtrees,
  - evaluates the subtrees in sequence,
  - returns the value of the second evaluated subtree,
  - allows multiple calculations and movements to be made each program iteration.
- ifGreater
  - accepts 4 subtrees,
  - evaluates the 3^{rd} or 4^{th} subtree based on the value of the 1^{st} and 2^{nd} subtrees,
  - pseudo code: if(1^{st} > 2^{nd}) then 3^{rd} else 4^{th},
  - returns the last evaluated subtree,
  - allows agent to perform different actions based on sensor inputs.
- +, -, *, /, and %
  - accepts 2 subtrees,
  - performs standard arithmetic calculation of evaluated subtree values,
  - division by zero results in value '1' [1],
  - allows agent to develop complex input weighting systems.

The terminal set includes:

- perim
  - returns Manhattan distance from the base,
  - allows agent make decisions according to its distance from the base.

- f, l, and r
  - causes agent to move forward (f), turn left (l), or turn right (r),
  - returns same value as perim.
- I
  - returns random integer (0-6),
  - generated during creation of genome, not during execution of program.

Genome Class

The genome class creates and stores program trees to be analyzed by the fitness function. Objects created from this class must:

- create random program trees from the function and terminal sets,
- create program trees from parents using crossover, mutation, and reproduction,
- store an evaluated fitness value.

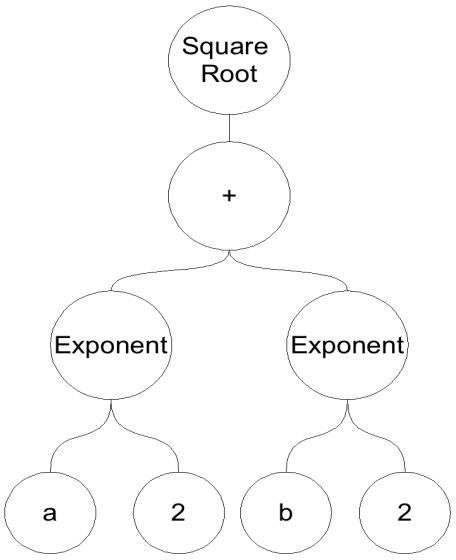

Figure 2: Tree Representation of Pythagorean Theroem

The genetic material produced by this class shall be used on a variety of platforms with different programming languages. It is a common practice in this situation to use **string representation** for the genomes [1]. Because strings are supported by most languages, the genome will not have to be converted into a different data structure. String representation also has the advantage of minimizing the space needed to store a program. For example, the translation of the Pythagorean Theorem - $\sqrt{(a^2+b^2)}$ - into the Lisp programming language is:

```
(sqrt (+ (expt a 2) (expt b 2)))
```

This Lisp program can then be translated into program tree form as seen in figure 2. To convert this program tree into a string representation, each primitive must be given a unique character to represent it.

- 's' – square root, accepts 1 argument
- '+' – addition, accepts 2 arguments
- 'p' – power, accepts 2 arguments
- 'a' – input representing the first leg of a right triangle
- 'b' – input representing the second leg of a right triangle
- '2' – integer value 2

Using these character representations, the genome object for this program would store the string “s+pa2pb2”. To execute the string representation in different languages, a **string representation interpreter** must be written for each language.

Generation Class

The generation class organizes the creation and storage of genome objects. Objects created from this class must:

- store an array of genome objects that represent the generation's members,
- create random genomes for the first generation,
- produce a new generation based on the fitness scores of the previous generation,
- identify the most fit individual in the generation.

After each genome in a generation is evaluated using the fitness function, a new generation object is created. The new generation calls the old generation to provide parent genomes using a combination of **fitness proportional selection** and **tournament selection**. In fitness proportional selection, each genome's chance of being selected for reproduction is equal to its fitness score divided by the sum of all fitness scores in the generation. In tournament selection, a group of genomes is randomly selected (group size is specified in the reproduction parameters) and the genome with the highest fitness score is chosen for reproduction. The new generation will use these methods to create and store genomes until the population size (specified in the reproduction parameters) is met.

Genetic Programming Evolutionary Sequence Class

The genetic programming evolutionary sequence class organizes the creation and storage of generation objects. Objects created from this class must:

- store an array of generation objects that represent a genealogy,
- organize the creation of generations with a specific number of genomes,
- organize the creation of a specific number of generations,
- send each genome to the fitness function to evaluate,
- present the most fit individual produced by the sequence.

When writing a script to perform a genetic programming sequence, this class eliminates the need to deal with genomes and generations directly. After each generation is produced, this class passes each genome of the current generation to the fitness function for evaluation.

Guard Class

The guard class is used to represent guard agents during simulation. Objects created from this class must:

- store a genome created from the genetic programming evolutionary sequence,
- use the genome as a means of controlling its movements during the simulation.

The guard class includes a string representation interpreter to execute a genome as a control program. Any time the interpreter evaluates a primitive that changes the state of the guard, that command is placed into a command buffer. The guard then executes one command per simulation time-step until the command buffer is empty. The genome is only evaluated when the command buffer is empty.

Enemy Class

The enemy class represents enemy agents during the simulation. In early experiments, the control program for this class will be hard-coded to start near the edge of the simulation space and move directly toward the nearest base. During later experiments, the enemy class may be updated to include a string representation interpreter so the GPES can simultaneously evolve guards and enemies. Results from GP research suggest co-evolution of opponents leads to

results with less exploitable weaknesses [2].

Base Class

The base class represents the base agent during the simulation. The base will not need a control program because it will remain stationary during early experiments. Later in the project, a control program may be implemented to allow the base to move randomly or in a specified direction in order to evolve more robust control programs for the guard agents that can protect moving convoys.

Simulator

The simulator shall be used by the fitness function to produce a fitness score that represents the genome's effectiveness as a perimeter maintenance control program. Objects created from this class must:

- accept parameters to modify size, simulation time, rules for collision, etc.,
- accept objects created from the guard, enemy, and base classes and signal,
- call agents to execute their control program on each step of the simulation,
- return an accurate fitness measure of the genome.

During early experiments, the simulated environment shall be grid-based, only allowing agents to move north, south, east, or west. Figure 3 presents a visualization of the grid-based simulator. After the framework produces a controller that functions optimally in the grid-based domain, the simulator shall be rewritten to allow continuous movement. An increase in simulation complexity can often prevent optimal behaviors from evolving. Others [3] have solved this problem by using function and terminal sets that embody complex behaviors consisting of several steps. Eventually the simulator may include noise in the sensor measurements and in agent movement.

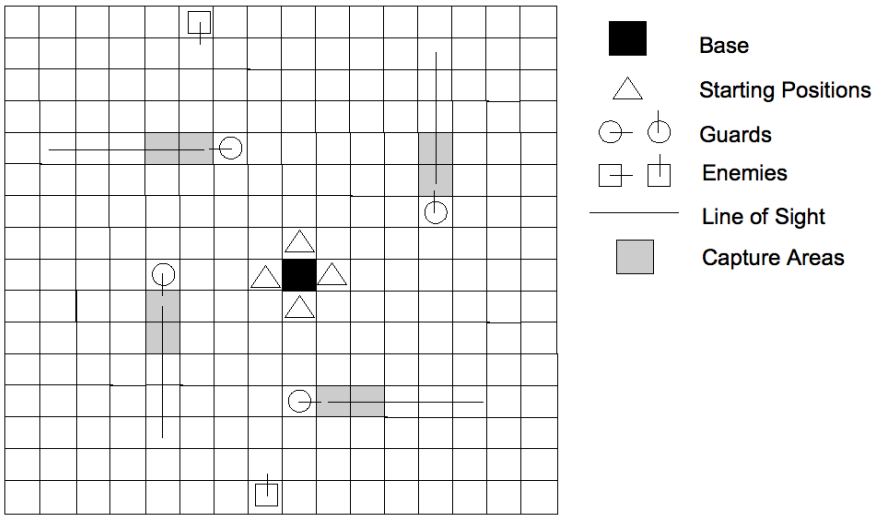

Figure 3: Visualization of Grid-World Simulator Running Perimeter Maintenance Simulation

Fitness Function

The fitness function is called by the genetic programming evolutionary sequence class and returns the fitness value of each genome. The fitness value is determined by the guard agents' behavior during a simulation. The fitness function initializes enemy agents, base agents, and guard agents, then begins a simulation. When a guard captures a enemy its fitness score will increase. More points are awarded for capturing enemies further from the base. Negative points are given if the enemies hit the base. Varying values of these rewards will result in different behaviors in the guards. If large rewards are given for capturing an enemy far from the base, the guards will evolve to create a large but very imperfect perimeter. If rewards are solely based on the number of enemies captured, the guards will cluster around the base. It will therefore be necessary to adjust rewards to yield effective guards.

Robotic Platform

If experimentation shows that a highly fit guard control program is able to evolve in a complex simulator environment, the project will proceed by attempting to implement the control program on a physical autonomous agent. In order for the platform to execute a genome created by the genetic programming evolutionary sequence, the software must:

- contain a string representation program interpreter,
- contain motor control routines that result in movement as specified by the function and terminal sets,
- contain sensor processing routines that result in sensor data as specified by the function and terminal sets.

Even if the simulation environment produces favorable results, robotic platforms generally do not behave as expected [4]. To produce a functional robot, the simulator must be customized to precisely model the robot and target environment.

References

[1] R. Poli, W. B. Langdon, and N. F. McPhee, *A field guide to genetic programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008. (With contributions by J. R. Koza).

[2] J. Koza, *Genetic Programming: on the Programming of Computers by Means of Natural Selction*. Cambridge, MA: MIT Press, 1992.

[3] S. Luke, "Genetic Programming Produced Competitive Soccer Softbot Teams for RoboCup97" in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pp. 214-222.

[4] J. Cohn, J. Weaver, and S. Redfield, "Cooperative Autonomous Robotic Perimeter Maintenance," in *Florida Conference on Recent Advances in Robotics 2009 Proceedings*.