Genetic Programming of Autonomous Agents

Senior Project Proposal

Scott O'Dell

Advisors: Dr. Joel Schipper and Dr. Arnold Patton

December 9, 2010

Introduction to Genetic Programming

Genetic programming (GP) is a machine learning technique based on the theory of evolution. Solutions to a problem are discovered through combining small blocks of code and evaluating the result. The goal is to produce a program that performs a task specified by the designer.

GP begins by producing a **generation** of random programs (**genomes**) composed of designer specified primitives. The primitives must be chosen to give the program sufficient perceptual, computational, and locomotive ability to effectively perform the task. Primitives that require arguments comprise the **function set**, while primitives without arguments comprise the **terminal set**. Each program can be represented by a tree made from the primitives in the function and terminal set, where each primitive is represented by a node. Program trees are the digital equivalent of genetic material. Figure 1 shows an example of a genome that was generated from primitives that are designed to evolve a wall-following vehicle. The 'if-wall-ahead' primitive evaluates the first subtree if there is currently a wall directly in front of the robot, or evaluates the second subtree otherwise. The primitives 'left', 'right', and 'forward' cause the robot to turn left, turn right, or move forward respectively. A tree is evaluated by starting at the top node and branching downward. The program tree in Figure 1 represents an ideal control program for a wall following robot that has a single sensor on its front face.

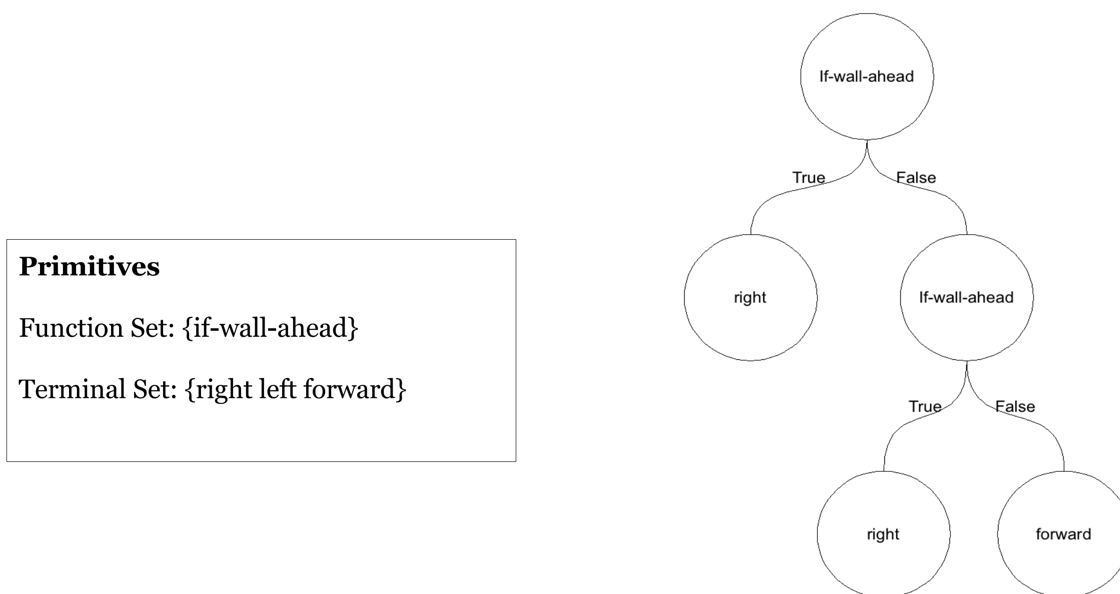

Figure 1: Decision Tree for a wall-following Robot

GP continues by using a fitness function to evaluate how well each randomly generated program performs a designer specified task. A **fitness function** returns each program's **fitness score**. Programs with higher fitness scores are more likely to contribute their genetic material to the next generation. Fitness functions must correctly separate more fit from less fit individuals, even if all individuals are relatively unfit. The fitness for a wall-following example might be the number of unique, wall-adjacent cells visited during a simulation.

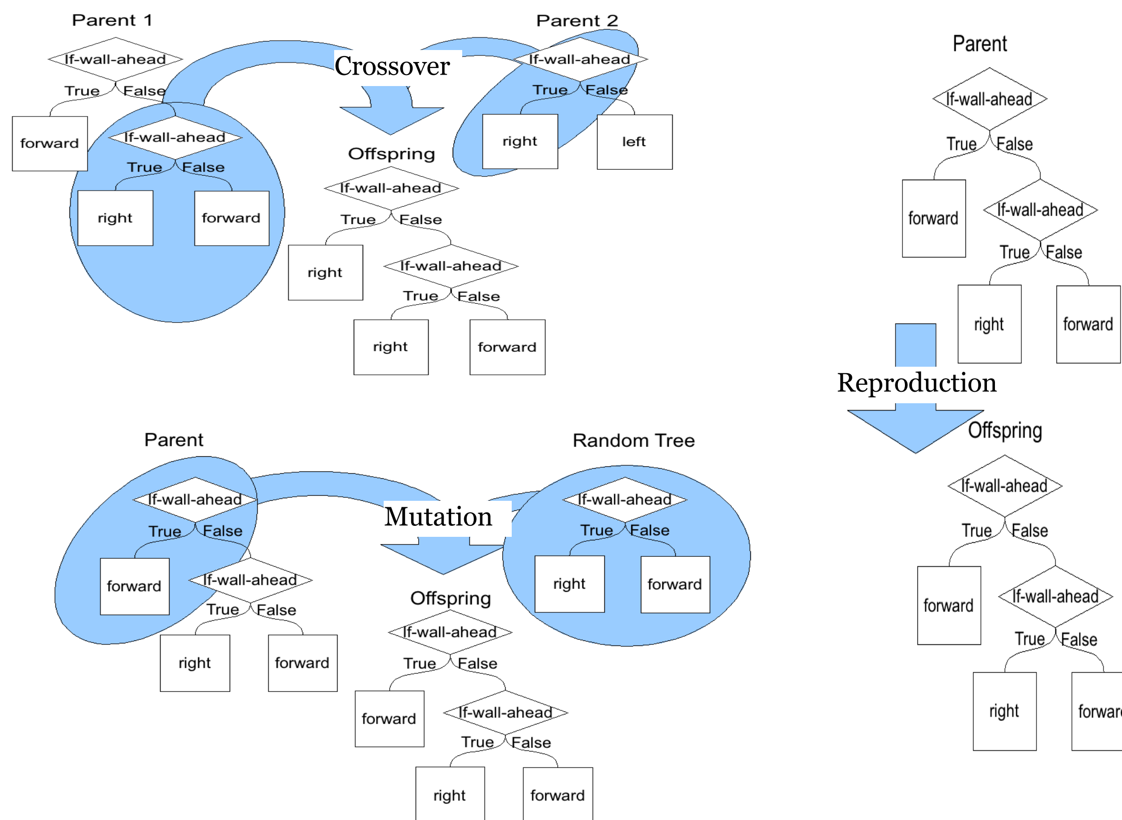

Figure 2: Illustration of Genetic Operators

GP produces the next generation of programs using **genetic operators**. First, members of the current generation are selected in proportion to their fitness. Genetic operators then simulate sexual reproduction (**crossover**) by combining sections of two program trees into a new program tree, asexual reproduction (**reproduction**) by making an exact copy of a program tree, and biological mutation (**mutation**) by randomly changing sections of a subtree. These genetic operators produce a new generation that behaves similarly to most fit individuals from the previous generation. Figure 2 shows an example of each genetic operator taking parent program trees and creating new offspring from the “genetic material”. The next generation of programs is evaluated by the fitness function and is used to produce another

generation. This process of evaluation and reproduction repeats until a specified number of generations is produced or a specified fitness level is met. The output of a genetic programming sequence is the most fit individual produced during any generation.

Project Summary

The purpose of this project is to use genetic programming to develop autonomous vehicle control programs that display perimeter maintenance behavior. Perimeter maintenance has the potential military application of using autonomous agents to protect an area or escort a convoy. The initial focus will be to implement a genetic programming framework. The framework will then be used to evolve **guard** agents that maintain a secure perimeter around a **base** that is being approached by **enemy** agents.

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, more complex simulators will be implemented 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.

Project Description

Genetic programming involves evaluating the fitness of thousands to millions of programs. Therefore, it is impractical to calculate each program's fitness by running it on a physical system because evolving a solution would take far too much time. By using a software simulator to evaluate fitness, a solution can evolve in a reasonable amount of time (as little as 45 minutes on personal computer in preliminary runs).

Top Level Description

The code written to connect software subcomponents will be written in Ruby to take advantage of Ruby's ability to easily interface with other languages. 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 3 shows the relation of software subcomponents. To begin the simulation, a function set, a terminal set, and reproduction parameters are provided to the genetic programming evolutionary sequence (GPES) block, which organizes the progression of generations. The randomly produced genomes of the first generation are 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 determines the fitness score of the genome. These fitness scores are passed back to the GPES, where a new generation initializes genomes by performing genetic operations on genomes of the previous generation. This process continues until generation N is evaluated for fitness. The final result is the program with the highest fitness from any generation.

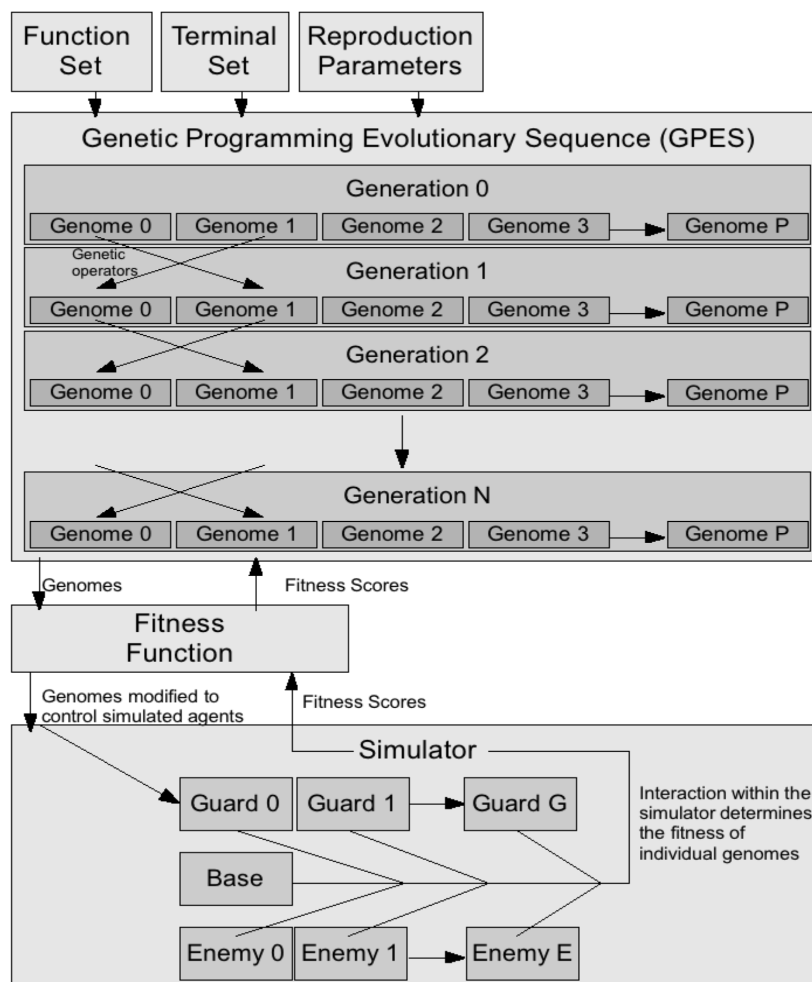

Figure 3: Top Level Software Architecture of Genetic Programming System

Subcomponent Descriptions

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.

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 3rd or 4th subtree based on the value of the 1st and 2nd subtrees,
  - pseudo code: if(1st > 2nd) then 3rd else 4th,
  - returns the last evaluated subtree,
  - allows agent to perform different actions based on sensor inputs.
- +, -, *, /, and %
  - accept 2 subtrees,
  - perform standard arithmetic calculation of evaluated subtree values,
  - division by zero results in value '1' [1],
  - allow 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
    - cause agent to move forward (f), turn left (l), or turn right (r),
    - return same value as perim.
- I
    - returns random integer (0-6),
    - generated during creation of genome, not during execution of program.

Reproduction Parameters

The reproduction parameters specify values used during the progression of the GPES. Optimizing the parameters can improve the efficiency of the system by modifying the way it creates and evaluates genomes. The reproduction parameters include the following values:

- number of generations produced,
- number of genomes in each generation,
- maximum depth of each program tree,
- proportion of each genetic operator used to create a new generation.

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.

The genetic material produced by this class may be used with a variety of 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 need not be converted into a different data structure. String representation also has the advantage of minimizing the space required 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)))
```

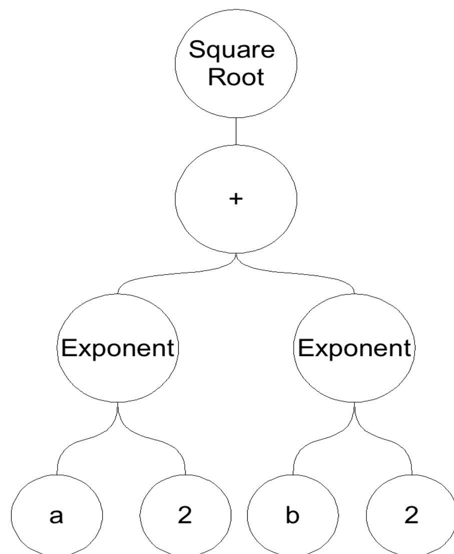

Figure 4: Tree Representation of Pythagorean Theorem

This Lisp program can then be translated into program tree form as shown in Figure 4. 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,
- 'e' – exponent, accepts 2 arguments,
- 'a' – represents length of leg 'a' in a right triangle,
- 'b' – represents length of leg 'b' in a right triangle,
- '2' – integer value 2.

Using these character representations, the genome object for this program would store the string "s+ea2eb2". To execute the string representation in different languages, a **string representation interpreter** must be written for each language.

Generation Class

The generation class creates and organizes genome objects. Generation objects must:

- store an array of genome objects that represent the generation's members,
- create random genomes for the first generation,
- select parents for genetic operators based on the fitness scores,
- 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 uses these methods to create and store genomes until the population size (specified in the reproduction parameters) is met.

Genetic Programming Evolutionary Sequence (GPES) Class

The GPES 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 for evaluation,
- 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 GPES,
- 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 that 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 does 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 to evolve more robust control programs for the guard agents that can protect moving convoys.

Simulator Class

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

- accept parameters to modify size, simulation time, rules for collision, etc.,
- accept objects created from the guard, enemy, and base classes,
- call agents to execute their control program during 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 5 presents a visualization of the grid-based simulator in which each agent occupies a single square. The agents can move forward and turn left or right. Whenever two agents attempt to occupy the same square, a block of code containing collision rules will be called to resolve the collision.

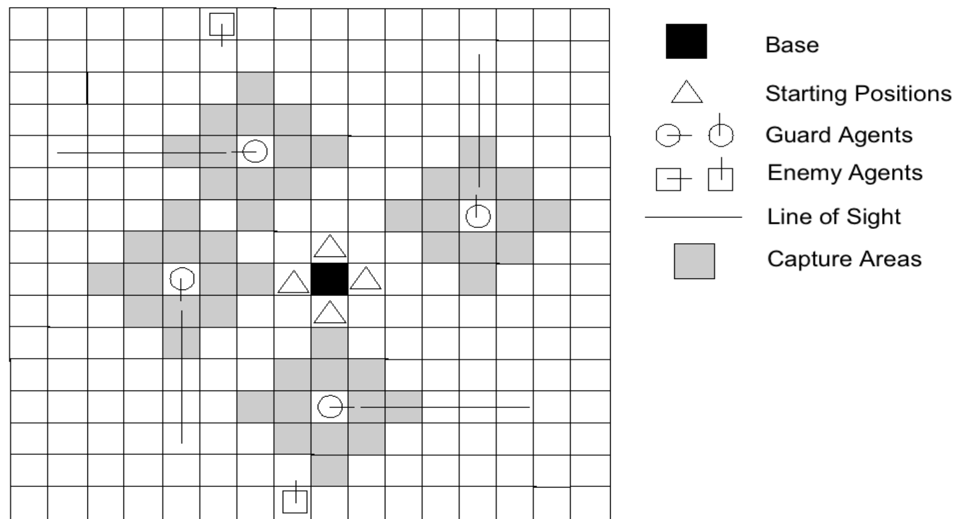

Figure 5: Visualization of Grid-World Simulator Running Perimeter Maintenance Simulation

After the framework produces a controller that functions optimally in the grid-based domain, the simulator shall be rewritten to enable 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.

Fitness Function

The fitness function will be called by the GPES class and return the fitness value of each genome. The fitness function initializes guard agents, enemy agents, and base agents, then begins a simulation. The fitness score is calculated based on the guard agents' behavior during

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 enemies hit the base. Varying the values of these rewards will result in different guard behaviors. If large rewards are given for capturing an enemy far from the base, the guards will evolve to create a large but penetrable perimeter. If rewards are solely based on the number of enemies captured, the guards will cluster around the base. Therefore, it will 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. For the platform to execute a genome created by the GPES, 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 produce the sensor data specified by the function and terminal sets.

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

Literature Review

Perimeter Maintenance

Perimeter maintenance is a military application that uses autonomous guards to detect enemy agents as they approach a military vehicle or base. In [4], engineers use emergent behavior principals to develop perimeter maintenance robots. Information about the positions of nearby objects and the center of the perimeter is sufficient to produce perimeter maintenance behavior. The robots are designed to accept a perimeter size as input and revolve around the base at that distance. The percentage of the perimeter monitored is used to assess the quality of the control program.

Testing was divided into two situations. In one test, enough agents were present to enable complete perimeter coverage. In another test, agents were incapable of covering the entire perimeter. A simple control program was used for the first situation, where each agent revolved around the perimeter in the same direction and navigated to avoid detecting other guards in its sensor range. The second situation demanded a more complex control program that optimizes coverage by navigating around perimeter and exhibiting a repulsion to other guard agents that decays over time. Although the results of these control programs were similar, each situation demanded a unique solution.

Simulations of the robots were run in the MobileSim simulator, which does not simulate the noise that occurs in physical systems. When the control programs were placed on physical robots, the robots did not behave as simulated due to the noise present in sensor measurements and the robot's movement. To create control programs that operate on physical robots, the development simulator must include realistic sensor noise.

Evolution of Cooperative Agents

Cooperative agents in genetic programming can evolve to cooperate explicitly through data exchange or implicitly through interaction in an environment. The effectiveness of cooperation depends on team composition and the task they are attempting to accomplish. In [5], Floreano and Keller outline the ideal conditions for evolving cooperative agents. For tasks that entail high-cost cooperation (an action that benefits the fitness of others at the expense of the individual), the best results occur when using teams of homogeneous agents and basing parent selection on the team's average fitness score. However, for tasks with low-cost cooperation, heterogeneous teams under individual fitness selection produce the best results.

In genetic programming, evolving communication through data exchange is difficult because agents cannot rely on each other to use a common language. Floreano and Keller [5] explain that because communication benefits the group and harms the individual that communicates, homogeneous teams under team selection are more likely to evolve communication. Naeini and Ghaziasgar [6] evolve communication in heterogeneous teams by making communication mandatory. When an agent requests information from another, the exchange benefits the receiving agent, and the transmitting agent cannot suppress the information.

Evolving Agents in a Complex Environment

For evolutionary agents to develop useful reactions within an environment, information about the environment must be present in a useful form. Luke [3] uses genetic programming to develop a team of agents to play soccer in a noisy simulator, which is used to simulate soccer matches for the RoboCup software competition. The simulator provides a huge amount of data about the environment that is not directly useful for evolving strategies. Developing a fit solution using native interactions with the simulator would involve the simultaneous evolution of code that converts the simulator's data into a useful form, accounts for the noise present in the data, and executes soccer-playing strategy. Although genetic programming can work in this situation, the time required for evolution makes it impractical.

Luke [3] solves the problem by condensing complex environmental data and agent actions into elements in the primitive set [3]. For example, the 'intercept' primitive calculates the state of the ball and moves the robot to intercept it. The primitive set is large (29 elements), but provides environmental data and complex movements in an immediately useful form. This strategy reduces the time needed for evolution by allowing the evolutionary sequence to focus on developing soccer-playing strategies rather than data processing routines.

Preliminary Results

At the time of this proposal, basic forms of each subcomponent shown in Figure 3 have been implemented in Ruby. The existing GPES block has the ability to evolve a single population by sending each genome to the fitness function individually. The simulator is grid-based, implements collision detection, and supports the evaluation of a single genome at a time.

Evolved Solutions

Early tests focused on developing a fitness function to evolve guard agents that perform intelligent perimeter maintenance in a grid-based domain. During these experiments, many factors contributed to the fitness score:

- distance from the base when enemy is captured,
- number of enemies captured,
- whether or not the base was eliminated by enemies,
- whether or not guards collided with other agents.

One fitness function that produces optimal perimeter maintenance agents is based solely on the distance from the base that an enemy is captured. Figure 6 shows a visualization of the perimeter created by the most fit guard agent produced in the GPES using this fitness function. At the beginning of each simulation, the agent moves to the location shown in Figure 6 and spins in place for the remainder of the simulation. The positioning shown allows the guards to maintain a large, perfect perimeter when their **capture radius** is 4 units. The result proves that the software framework is effective at evolving guards with perimeter maintenance behavior.

```
. . . . . . . . 4 . . . . . . . . . .
. . . . . . . 4 . 4 . . . . . . . . .
. . . . . . 4 . . . 4 . . . . . . . .
. . . . . 4 . . . . . 4 . . . . . . .
. . . . 4 . . . < . . . 4 . 4 . . . .
. . . . . 4 . . . . . 4 . 4 . 4 . . .
. . . . 4 . 4 . . . 4 . 4 . . . 4 . .
. . . 4 . 4 . 4 . 4 . 4 . . . . . 4 .
. . 4 . . . 4 . 4 . 4 . . . ^ . . . 4
. 4 . . . . . 4 . x . 4 . . . . . 4 .
4 . . . v . . . 4 . 4 . 4 . . . 4 . .
. 4 . . . . . 4 . 4 . 4 . 4 . 4 . . .
. . 4 . . . 4 . 4 . . . 4 . 4 . . . .
. . . 4 . 4 . 4 . . . . . 4 . . . . .
. . . . 4 . 4 . . . > . . . 4 . . . .
. . . . . . . 4 . . . . . 4 . . . . .
. . . . . . . . 4 . . . 4 . . . . . .
. . . . . . . . . 4 . 4 . . . . . . .
. . . . . . . . . . 4 . . . . . . . .
```

Figure 6: Perimeter Visualization of Guard with Fitness Function Based on Distance from the Base that Enemy was Captured

In real-life applications, it may be desirable for a group of autonomous agents to protect a perimeter where the radius is specified as parameter. To achieve this result, the fitness function is based solely on the number of enemies caught during a simulation, and the simulation must end if an enemy comes within the specified distance of the base. This fitness function leads to the solution found in Figure 7, which maximizes the guards' coverage of the base's perimeter by deploying to positions near the corner of the perimeter and remaining stationary for the remainder of the simulation.

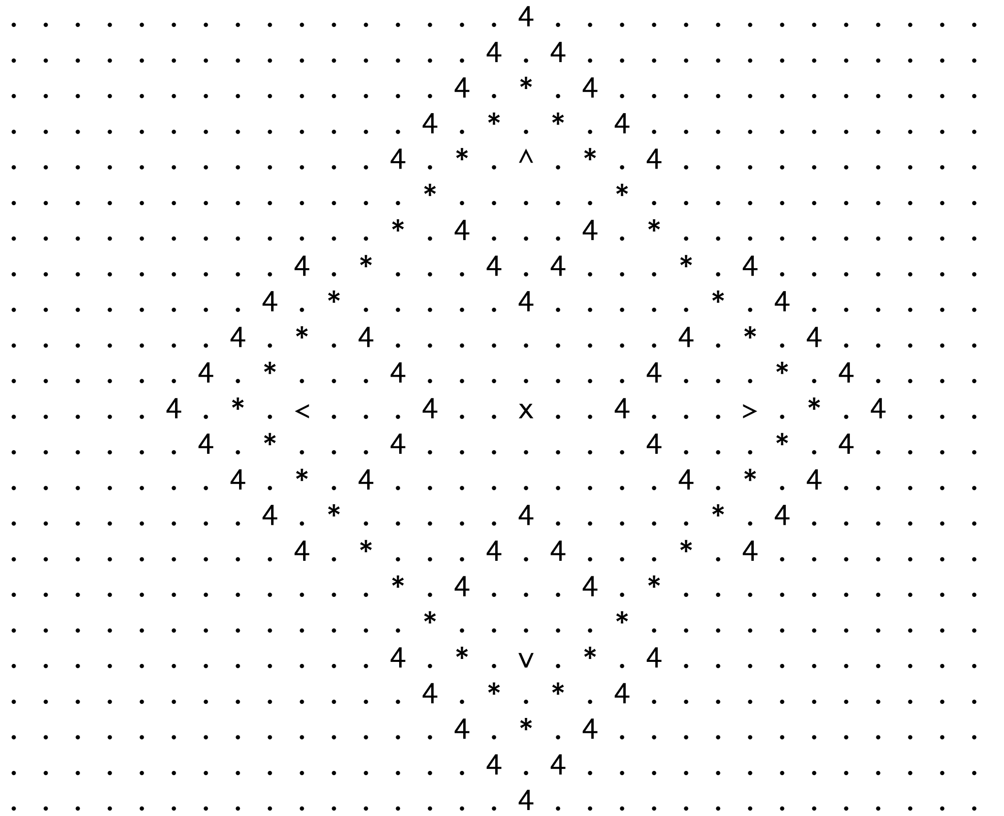

Figure 7: Perimeter Visualization of Guard with Fitness Function Based on Number of Enemies Captured Beyond 9 Units from the Base

Limitations

Although the solutions in Figures 6 and 7 optimize the fitness function, common sense suggests that if the guard agents revolve around the base they would be more effective. For the solution in Figure 6, moving around the base in the grid domain would force the guards to vacate the positions that maximize the average distance from the base at which enemies are captured. Similarly, for the solution in Figure 7, rotating around the base would lead to periods of time when only one side of a guard's capture area is defending the base's perimeter, rather than two sides when the guard is defending the corner. The marginal benefit gained by revolving around the base is outweighed by the disadvantage of suboptimal coverage while moving.

The source of these problems is the asymmetries caused by the grid-based domain. In a continuous domain, the guards would be able to revolve around the base while maintaining a radius that optimizes perimeter coverage. Additionally, the solution in Figure 7 is not ideal due to its predictability. In a practical situation, an enemy would learn to attack from a diagonal with 100% success. Co-evolution is a potential solution to the problem of predictable

guards. If guards begin to develop a predictable perimeter maintenance technique, the enemies would evolve to exploit the predictability. In response, the guards would have to evolve unpredictable solutions to increase their fitness.

Schedule

As discussed in the "Preliminary Results" section, portions of this project were completed prior to the creation of this proposal. The schedule will therefore be organized by completed work and future work. The primary goal of the project (i.e. simulated evolution of a guard agent in a noisy environment) is scheduled to be completed before spring break. Work on the robotic platform is placed after spring break but will only be pursued if early runs with the continuous simulator suggest that there will be enough time to complete it. If implementation on a robotic platform is pursued, research and part acquisition would occur in parallel with continuous domain simulations.

Week of	**Agenda**
Completed Work	
October 3	Genome Class
October 10	Generation Class
October 17	GPES Class
October 24	Grid-Based Simulator
October 31	Fitness Function, Terminal Set, Function Set, and Initial Simulations
November 7	Code Refactoring
November 14	Capstone Project Deliverables
November 21	*Thanksgiving Break*
Future Work	
January 9	Enemy Co-evolution and Heterogeneous Teams
January 16	Continuous Simulator
January 23	Graphics for Continuous Simulator

January 30	Interface Code for Continuous Simulator and Simulations
February 6	Add Noise to Continuous Simulator
February 13	Code Refactoring
February 20	Simulations with Noise, Modification of Fitness Function
February 27	Simulations with Modified Fitness Function
March 6	Collect Results and Create Presentation
March 13	*Spring Break*
March 20	Research Robotic Platform
March 27	Prepare Robotic Platform
April 3	Write Program Tree Interpreter for Robotic Platform
April 10	Load Evolved Program onto Robotic Platform and Debug
April 17	Evaluate Performance, Modify Simulator, New Simulations
April 24	Load Newly Evolved Program onto Robotic Platform

References

[1] R. Poli, W. Langdon, N. McPhee, *A Field Guide to Genetic Programming*. San Francisco, CA: Creative Commons, 2008.

[2] J. Koza, *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. 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*, Madison, WI, pp. 214-222.

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

[5] D. Floreano and L. Keller, “Methods for Artificial Evolution of Truly Cooperative Robots,” in *Bio-Inspired Systems: Computational and Ambient Intelligence* Heidelberg, Germany, Springer Berlin, 2009

[6] A. Naeini and M. Ghaziasgar, “Improving Coordination via Emergent Communication in Cooperative Multiagent Systems: A Genetic Network Programming Approach,” in *IEEE International Conference of Systems, Man, and Cybernetics* , San Antonio, TX, 2009, pp. 589-594