Genetic Programming of Autonomous Agents

Functional Description and Complete System Block Diagram

Scott O'Dell

Advisors: Dr. Joel Schipper and Dr. Arnold Patton

October 19, 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.

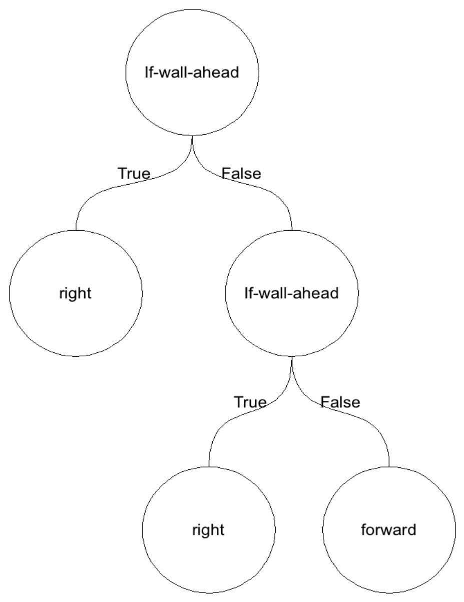
GP begins by producing a **generation** of random programs (**genomes**) composed of designer specified primitives. The primitives must be chosen to give the program sufficient perceptual, computational, and locomotive ability to effectively perform the task. Primitives that require arguments comprise the **function set**, while nodes 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 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. 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

Primitives

Function Set: {if-wall-ahead}

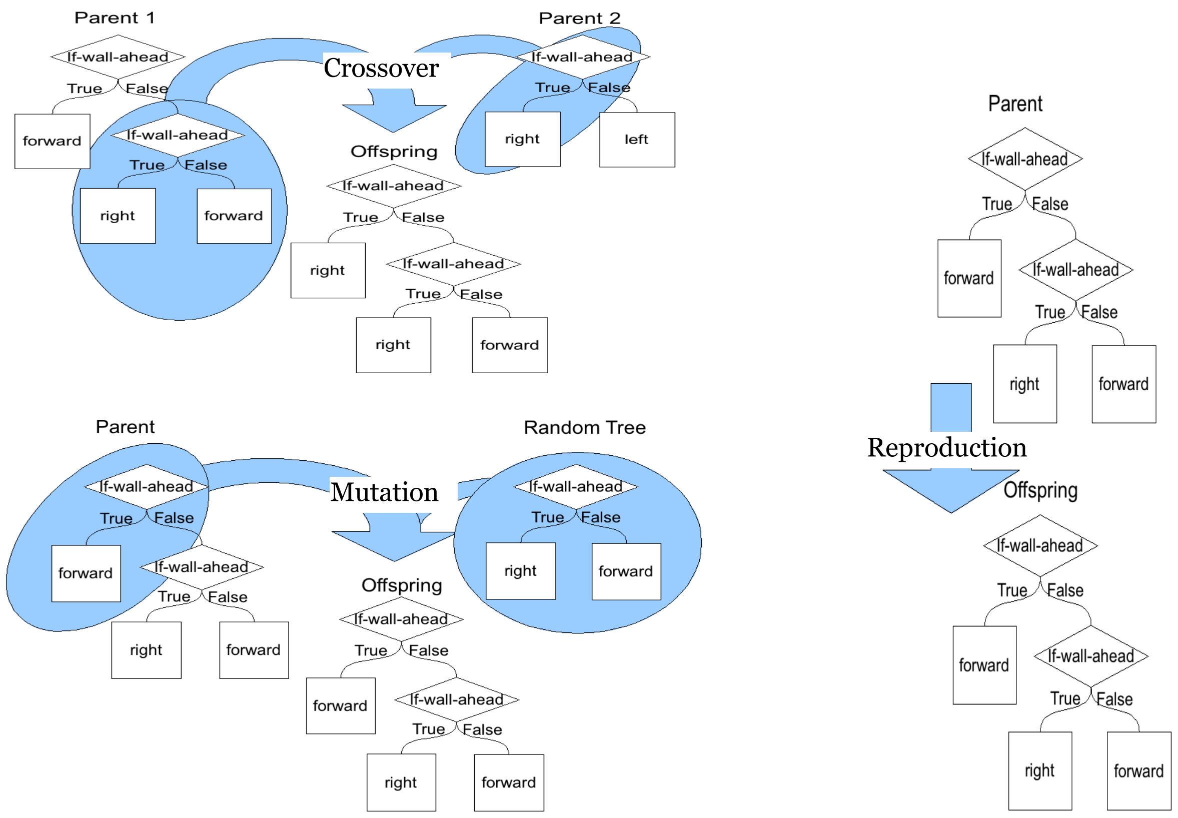
Terminal Set: {right left forward}



GP continues by using a fitness function to evaluate how well each randomly generated program performs the task. A **fitness function** returns each program's **fitness score**. Programs with a higher fitness score 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.

GP then produces the next generation of programs using **genetic operators**. Members of the current generation are selected in proportion to their fitness. Genetic operators simulate sexual reproduction (**crossover**) by combining sections of 2 program trees into a new program tree, asexual reproduction (**reproduction**) by making an exact copy of a program tree, and biological mutation (**mutation**) 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. The output of a genetic programming sequence is the most fit individual produced during any generation of the run.

Figure 2: Illustration of Genetic Operators



Project Goals

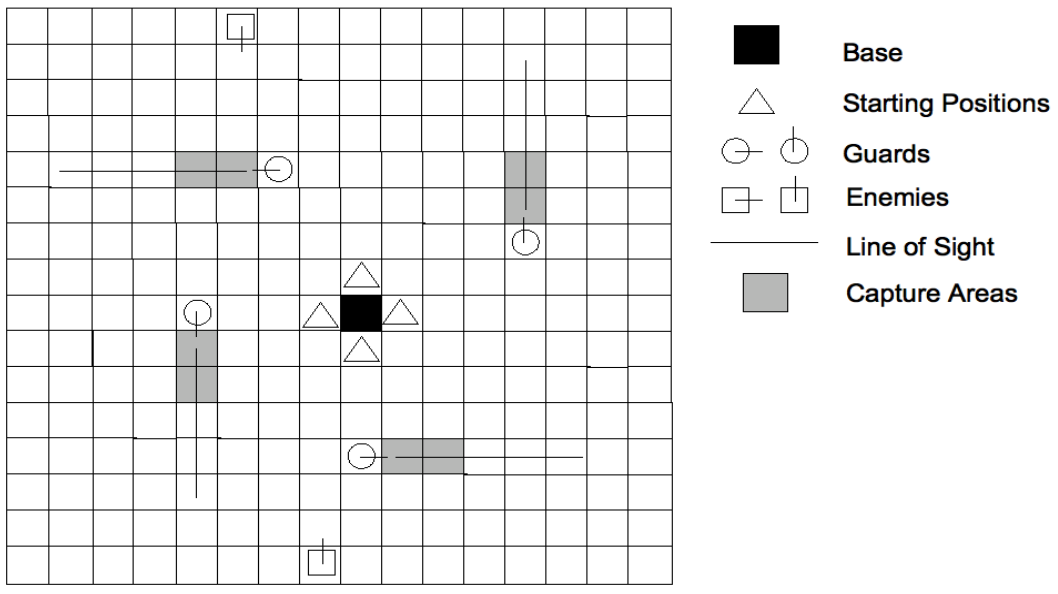
The purpose of this project is to use genetic programming to develop control programs for autonomous vehicles. The initial focus shall be to implement a genetic programming framework. This framework will be used to evolve a **guard** agent that maintains a secure perimeter around a **base** which is 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.

Functional Description

Initial simulations will take place in a grid-world environment similar to the one shown in figure 3. Enemy and guard agents each occupy one cell at a time and are allowed to turn left, turn right and move forward. As the simulations increase in complexity and more continuous movement is possible, the function set, terminal set, and fitness function shall be modified to allow the guard to evolve more complex behaviors.

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



Top Level

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.

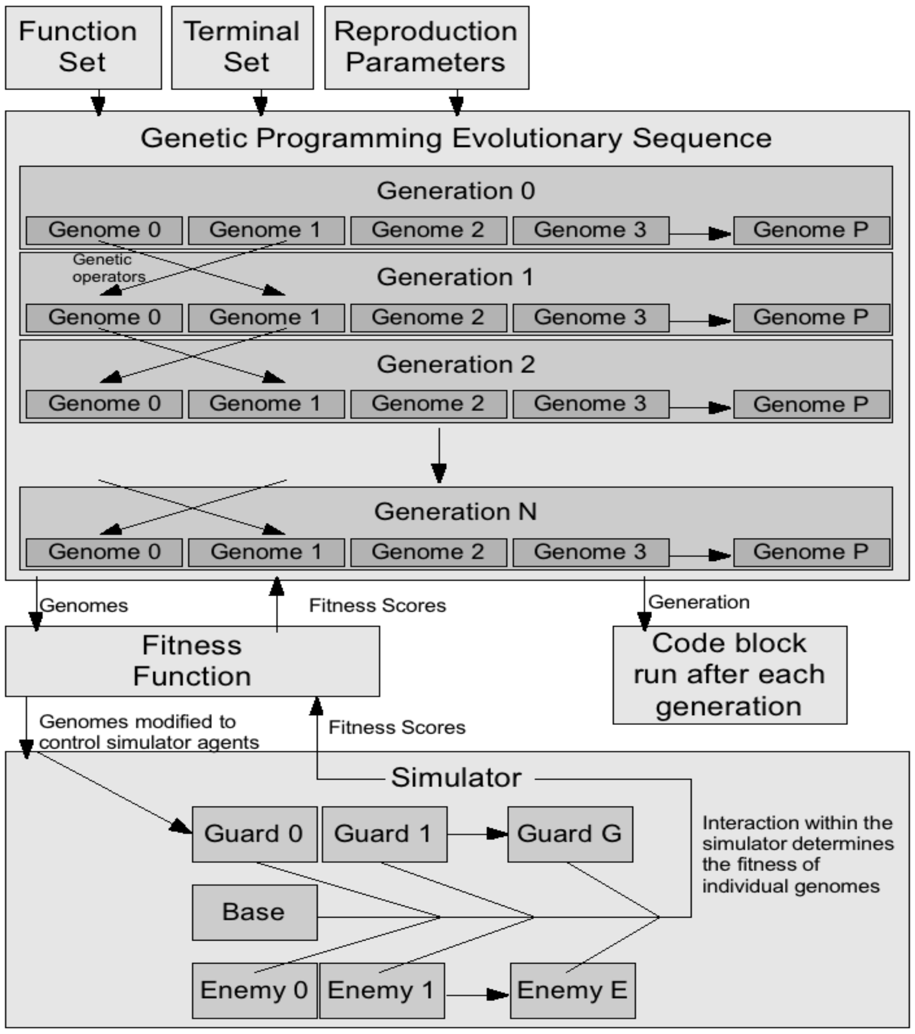


Figure 4: Top Level Software Architecture of Genetic Programming System

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.

Simulator

The simulator shall be used by the fitness function to produce a fitness score that represents a genome's effectiveness as a perimeter maintenance control program. It must be able to complete the following process:

1. Accept parameters to modify its size, simulation time, rules for collision, etc.,
2. Accept genomes produced by the evolutionary sequence,
3. Use the genomes as a means of controlling an agent during the simulation,
4. Return an accurate fitness measure of the genome.

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 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',
    - 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 inherited program trees from parents using crossover, mutation, and reproduction,
- Store an evaluated fitness value.

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 initial generation,
- Produce a new generation based on the previous generation,
- Identify the most fit individual in the generation.

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 for evaluation,
- Present the most fit individual produced by the sequence during any generation.

Modifying the Specifications

Genetic programming is known for its ability to discover ways to exploit the evolutionary framework to optimize fitness scores. These exploitations are often unforeseeable. Throughout development it may be necessary to make modifications to the entire system (particularly the fitness function / simulator) to coerce the system into providing results that are consistent with the project goal.