Virtual Prairie Dog Simulation Using Genetic Programming

Jason R. Nelson
Computer Science Major
Doane College
jrnelson@doane.edu

Abstract

This student research project involved designing and implementing a simulation of the behavior of black-tailed prairie dogs using genetic programming techniques and tools. This simulation program will be used as a tool in the ongoing research at Doane College by a biology professor interested in studying black-tailed prairie dogs.

For this project, specific behaviors of actual prairie dogs were observed and recorded. The behaviors modeled for this simulation were the head bob and standing alert. The simulation model was developed using genetic programming from initial data obtained during observation of prairie dogs in a colony at a research site owned by my institution. The simulation model will be verified by comparing the actual behavior data to simulated behavior data. If the prairie dog is effectively modeled, one could not tell whether data was from a real prairie dog or a simulated prairie dog. At that time, the program will be then wrapped in a user interface.

The purpose of this paper is to describe the process of creating a virtual prairie dog using genetic programming. Specifically, this paper will include a discussion of:

1) the black-tailed prairie dog and the behaviors modeled,
2) the initial data collected and used to develop the simulation,
3) the genetic programming technique used to develop the simulated model,
4) the testing of the simulated model to ensure it accurately reflected the behavior of a real prairie dog, and
5) the planned user interface that will be created to allow for user interaction.

The presentation will include a demonstration of the virtual prairie dog. It is the goal of the professor to eventually build upon this project and create an entire virtual prairie dog colony. This virtual colony would accurately simulate the real colony at the research site.

Introduction

Doane College acquired land, near Grafton, Nebraska, that contained a black-tailed prairie dog town. Dr. Russ Souchek, a biology professor at Doane, desired a computer program to simulate the prairie dog town. He and his students will be using this program to learn the nature of these creatures. Eventually a user interface will be developed that will allow students to use and interact with the model. Students will be able to study the prairie dog habits without traveling to Grafton. This model will be able to be manipulated unlike the real town near Grafton. This project begins the modeling of a prairie dog town by simulating the action of a single prairie dog.

Recognition & Analysis

This project involved designing and implementing a simulation of black-tailed prairie dog behavior using genetic programming.

Doane College's prairie dog colony is Nebraska's farthest eastern known colony. Doane College received possession of this colony and the land surrounding it in 2000. The biology department was immediately interested in studying the behavior of prairie dogs, specifically head bobs and standing alerts. So much of students' experience is lab based; this gives biology students the possibility of working in an outdoor laboratory to gain valuable field experience.

A prairie dog lifts its head for a short period of time (0.2 to 5.0 seconds) and appears to examine its surroundings while foraging. Each such lift is a head bob (See Figure 1).

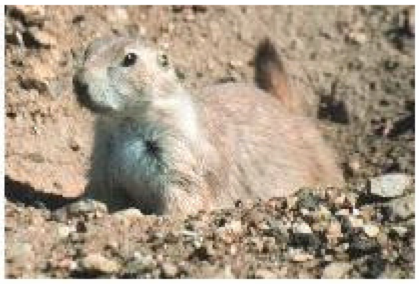

Figure 1: Prarie Dog Performing a Head bob

A standing alert occurs when a prairie dog suddenly stops all previous activity, stands on its two hind legs and searches its surroundings for danger (See Figure 2).

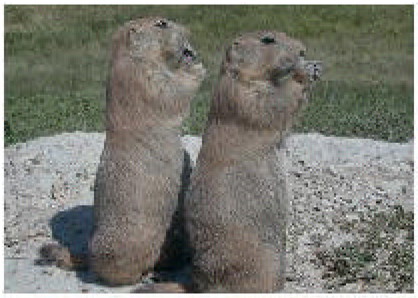
Figure 2: Prarie Dogs Performing a Standing Alert

In the summer of 2002, a student performed a project that collected behavioral data on the prairie dogs. That project was an animal behavior study on the predator alertness of the prairie dogs at Grafton. A laptop computer was used to record two types of qualitative data, head bob and standing alert, which helped measure the prairie dog's alertness. That study will be compared to one done on another prairie dog colony.

To help collect data, results of another student project were used. That project provided a database in which to record behaviors. Input to the database is done via a game controller. A researcher can be observing the behavior of prairie dogs and simply push the appropriate buttons on the game controller to record the behavior in the database. The database reports contain the start time, end time, and elapsed time of each action. These fields are recorded to the thousandth of a second.

The data from this initial study was converted to zeros, ones, and twos to represent the actions of doing nothing, performing a head bob, and performing a standing alert respectively. Since the data from the database was in thousandths of seconds, this file turned out to be a rather large data set although it was only a ten minute period of time. Once the data was converted it was read into the simulation and stored in an array. This array remains static throughout the simulation so it can be used as a comparison.

This project used the data collected and, applying genetic programming techniques, attempted to develop a simulation of the head bob and standing alert behavior of a prairie dog.

Design & Implementation

I first obtained a genetic programming environment in which I could write and run a Java package, since an interface could easily be developed with Java. I searched the Internet for such an implementation but did not find one to meet my needs. Fortunately, however, Dr. Mark Meysenburg, who had previous knowledge and experience with genetic programming, had an implementation for me to use. This is gpjpp version 1.0, a Java package for genetic programming. This was copyrighted in 1997 by Kim Kokkonen and written by Adam Fraser and Thomas Weinbrenner. This software is GNU general public licensed.

As defined by John Koza, [Koza, 1992], genetic programming is getting computers to solve problems without being clearly programmed to do so. The programs evolve to become solutions to the problem at hand based on the requirements of the problem. Genetic programming must be an intelligent search, verses a random search, as the search space of all possible programs in infinite.

In a typical GP system, programs are represented as Lisp s-expressions, and visualized as program trees. Each program is made up of function and terminal nodes, drawn from sets of nodes thought to be useful for the problem at hand. An initial "population" of programs is created at random. Each program in the population is evaluated, to see how good of a solution it is to the problem at hand; thus each program is assigned a fitness value. Then, a new generation of programs is created through genetic operators that mimic natural processes. Individual programs may *mate* and produce *offspring* via a crossover operation. Individuals may be *mutated* by a mutation operation. Finally, programs are selected, based on fitness, to become members of the next generation. This cycle of operators and selection is repeated over and over, to produce successive generations. In a successful GP run, programs in the population evolve to become better and better solutions to the problem at hand, until at least one program is produced that adequately solves the problem. A typical program tree generated by a GP system is shown in Figure 3. This is the tree developed from the well known “Artificial Ant” problem.

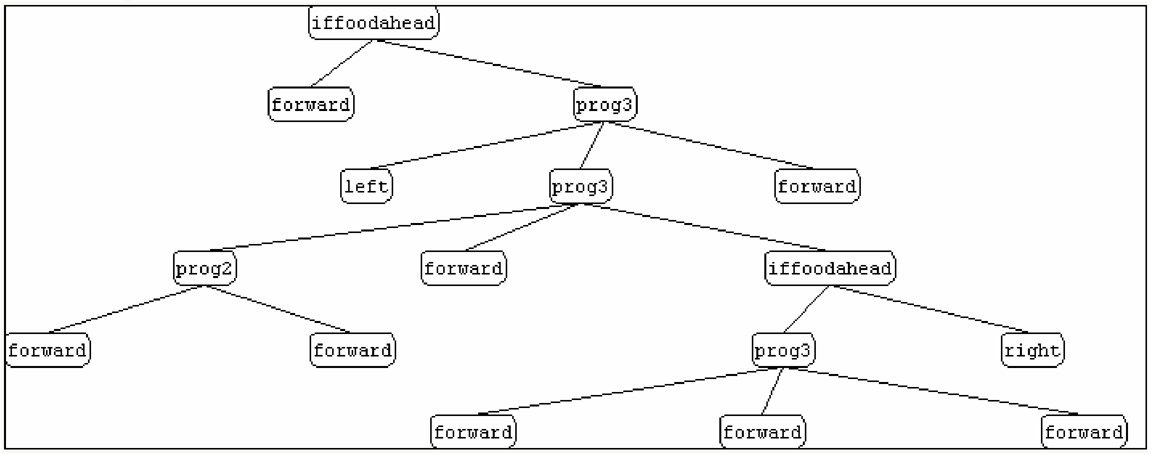

Figure 3: The Parse Tree from a Generation of the Ant Problem

My design and implementation was based on techniques in Koza’s book, [1]. The first step in setting up a GP system includes developing the structures that undergo adaptation, namely the function set and terminal set. The function set contains nodes that perform actions, make decisions, or perform calculations. In a program tree, function nodes are the non-leaf nodes.   Examples of function nodes are: arithmetic operations, mathematical functions, conditional operators, Boolean operators, iteration, or recursion. The terminal set is composed of all the possible variable states, inputs, detectors, or sensors. In a program tree, terminal nodes are the leaf nodes of the tree. Example terminal nodes are constants and variables.

Once the function and terminal sets are determined and implemented, the final step was to determine a fitness measure, also called the fittest. This measures each individual of the population to see how well it performs in the environment. The gpjpp system tries to evolve programs with minimal fitness value, so low fitness scores are better than high fitness scores.

GPJPP Implemented Classes

To run the problem-specific functions, terminals, and fit test, several classes had to be extended from gpjpp, the genetic programming implementation. The first class was GPRun. The *main()* method is in this class, which initializes and runs the test case. The *main()* method creates and evolves populations, writes reports, loads and saves checkpoint files, and does multiple runs until a number of acceptable solutions are found. The GPRun class does the following for the genetic program:

1. Initializes random number generator.
2. Reads the ini file for run properties.
3. Creates output files of the run.
4. Registers all classes.
5. Traps and handles all exceptions.
6. Loads previous checkpoint or creates initial population.
7. Displays status output to console window.
8. Configures the number of generations for long runs.
9. Determines when a fitness target is reached.
10. Writes reports on the best individual at the end of a run.
11. Runs multiple runs until enough acceptable solutions are found.

GPRun returns an initialized instance of *GPAdfNodeSet(),* which contains problem branch and node definitions for the prairie dog. For this the *createNodeSet()* function was replaced with problem-specific functions and terminals.

Another class that is extended is GPGene. The program trees themselves are made of these GPGene objects that represent functions or terminals created in the GPRun class. GP trees are composed of GPGene objects and represent the function or terminal of each element of an s-expression, which is a symbolic expression such as a list in Lisp. It contains references that describe the node’s nature. GPGene doesn’t contain a method to evaluate the fitness of the tree. However it does contain a subclass of GPGene that defines a GPGene fitness function called by *GP.evaluate*(). This enables the evaluation of the program trees. A switch statement is used to determine the functions or terminals to be evaluated and in what order. In this class, I created my own subclass to define a fitness function *GP.evaluate ()*. This is called by my fitness case in the GP class.

The final class used, GPPopulation, is used when new populations are created. It also overwrites *create.GP()* to make prairie dog instances. It contains methods for evolving the population by fitness-based selection, crossover, mutation, and migration.

Prairie Dog Functions and Terminals

As mentioned, functions can be of many types. Those that are used for this problem are conditional operators. Iterative functions are usually avoided because of the possibility of infinite loops. These functions are in the structure of a switch with cases. There are unconditional branches that include: Prog2, Prog3, and Prog4. They at first had two, three, and four branches respectively, and merely added nodes for their result. They were then changed to all have two branches each. Prog2 still used addition for the result, but Prog3 used subtraction and Prog4 used multiplication. This allowed for the use of math other than just addition. With only these unconditional branches the prairie dog never seemed to evolve to be a better individual. The complexity of the generations stayed constant and very simple parse trees resulted. One such tree is below.

Figure 4: The Parse Tree of an early Prairie Dog Generation

The look-back functions were then added and they branched depending on the value chosen at a specified prior time. The look-back functions took three arguments. If the behavior exhibited by the prairie dog during one time increment in the past was nothing (the down position), the function returned the value of the first argument. If the previous behavior was a head bob, the function returned the value of the second argument. Likewise, if the previous behavior was a standing alert, the function returned the value of the third argument. After they were included, a best fitness of 75 was reached. It was learned through testing that the simulation was getting the same result each generation. The trees created were identical. The following is an example of one such tree.

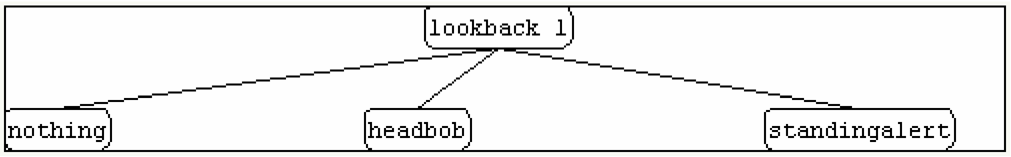

Figure 5: The Parse Tree of a Prairie Dog Generation

This was not a best possible fitness, but as it turned out the simulation was using the test data array to copy what it had done. Thus it was not simulating it on its own. Making a comparison to the current element in the test data array was then attempted. It seemed to be doing the same thing when this function was implemented. The conclusion was then made that the functions could not include a comparison to the array element or it would merely be copied. Two new functions were then added. The first was a conditional statement to compare the previous element in the test data array to the result of the previous generation. The other compares the result to the integers zero, one, and two; that represent doing nothing, a head bob, and a standing alert. The final function set included the three nodes, the three unconditional branches, Prog2, Prog3, and Prog4, with mathematical operators, a comparison to compare branches and evaluate them accordingly, and the two functions where the result is being compared.

The terminals, nodes, I created are nothing, head bob, and alert. These are for when the prairie dog performs nothing, a head bob and a standing alert respectively. This is straight forward with those nodes being the behaviors that we are attempting to simulate.

Prairie Dog Fitness Evaluation

The fitness cases are determined by looping through the original test data. The fitness is determined according to a comparison between the simulated action and the action that came from the real prairie dog in the test case. It receives a minimal raw fitness if they are the same and adds nothing to the previous raw fitness value. If these values are different the raw fitness is incremented by one.

Having a complex fitness function slowed down this implementation which was already quite time consuming. The earlier fitness measure took into account the reasonable ranges for a real prairie dog's head bob and standing alert. If they were out of range, raw fitness was incremented as well as when the result did not equal the test data. This was unsuccessful so the simple comparison to the test data was implemented.

Prairie Dog Simulation Results

In early trials, simulation results included best and worst cases that didn't fluctuate or vary much between generations. The complexity varied and increases with generations. As this complexity grew large parse trees were created. This simulation acted the best when the look-back functions were used. As of right now, the simulation generates individuals with better fitness measures, but it levels off for a number of generations then gets better. This occurs for a few runs but never gets better than 30200 while the worst is 799000, since that is the size of the test data array.

During testing configuration settings were adjusted. Generally runs were tested with smaller population sizes and generations. It was also run where the best individual was guaranteed to be added to the new population as well as not doing so. When a large population number was used the selection type was changed from the default to greedy over selection to adjust for the large population.

As of right now, this project has resulted in the conclusion that the time series data being modeled did not result in an optimal solution to the problem. When plotted this data is close to a straight line. I am working on mapping the behaviors of head bob and standing alert to 500 and 1000 instead of 1 and 2 to increase the range of possible results. I am also testing more functions that might solve this problem.

Prairie Dog GUI

The final stage of this project will be to produce a graphical interface. This will help those in the biology department understand what the prairie dog is really doing. The data from the simulated prairie dog that will be used in this interface are the terminals from the created parse tree. It will switch between three pictures. A head bob, standing alert, and a prairie dog with its head down to represent doing nothing.

In the future, the goal is to have an entire prairie dog colony simulated. From this one will be able to see how prairie dogs react to each other and predators. More behaviors could be included as well as environmental conditions. This simulation is the first of many projects to come at Doane College.

References

1. Koza, J. (1992). *Genetic Programming on the Programming of Computers by Means of Natural Selection*. Massachusetts: The MIT Press.

Acknowledgements

I would like to acknowledge the assistance of Veryl Walstrom for the gathering of test data.

I would like to acknowledge the assistance of Dr. Mark Meysenburg for his assistance in developing the simulation.

I would like to thank Kim Kokkonen for the use of the gpjpp Java interface.