|  |
| --- |
| Carleton University |
| GEPSharp |
| Honours Project Final Report |
|  |
| **Devin Denis** |
| **4/18/2014** |

|  |
| --- |
| Gene Expression Programming (GEP) is a variety of genetic programming that differs significantly from John Koza’s original version (GP). For this project, a library entitled GEPsharp was written in C#, implementing both GEP and GP. Then the library was tested on four traditional genetic programming sample problems. Results for GEP and GP were compared, and GEP was found to be faster, but no more effective. Additional features for the library which are not part of the original GP or GEP description were added and tested on the same sample problems. |

# Acknowledgements

This project is inspired by two books. The first is John Koza’s Genetic Programming, published in 1992. The book describes genetic programming (GP) as a method to evolve computer programs by means of natural selection. The second is Gene Expression Programming (GEP), by Cândida Ferreira, which describes an alternate method to achieve the same results as Koza’s GP. Without the ideas of these two authors, this project could not have been created.

My supervisor, Franz Oppacher, gave me the idea to implement GEP and lent me his copy of Ferreira’s book. Additionally, his class on Programming Paradigms sparked my interest in the theoretical side of computer science. Prior to taking that class, I was generally only interested in software engineering. I enjoyed working on this project a great deal, and I am grateful to Professor Oppacher for giving me the opportunity to do so.

Table of Contents

[Acknowledgements 2](#_Toc385620221)

[Introduction 5](#_Toc385620222)

[Motivation 5](#_Toc385620223)

[Background 5](#_Toc385620224)

[Genetic Programming 5](#_Toc385620225)

[Koza’s Genetic Programming 6](#_Toc385620226)

[Ferreira’s Gene Expression Programming 6](#_Toc385620227)

[Goals 7](#_Toc385620228)

[Methodology 8](#_Toc385620229)

[Language Choice 8](#_Toc385620230)

[Library 8](#_Toc385620231)

[Design 8](#_Toc385620232)

[Implementation 10](#_Toc385620233)

[Test Cases 11](#_Toc385620234)

[Artificial Ant 11](#_Toc385620235)

[Boolean 11 Multiplexer 13](#_Toc385620236)

[Optimal Control 13](#_Toc385620237)

[Symbolic Regression 14](#_Toc385620238)

[Results 15](#_Toc385620239)

[Artificial Ant 15](#_Toc385620240)

[Boolean 11 Multiplexer 16](#_Toc385620241)

[Optimal Control 19](#_Toc385620242)

[Symbolic Regression 19](#_Toc385620243)

[Conclusions 21](#_Toc385620244)

[Bibliography 22](#_Toc385620245)

[Figure 1 - Class Diagram for GEPsharp 9](#_Toc385620246)

[Figure 2 - Santa Fe Trail. 12](#_Toc385620247)

[Figure 3 - Optimal Artificial Ant solution 17](#_Toc385620248)

[Figure 4 - Best Boolean 11 Multiplexer solution 18](#_Toc385620249)

[Table 1 - Artificial Ant 15](#_Toc385620250)

[Table 2 - Artificial Ant original results 16](#_Toc385620251)

[Table 3 - Artificial Ant 1000 population 16](#_Toc385620252)

[Table 4 - Boolean 11 Multiplexer 16](#_Toc385620253)

[Table 5 - Optimal Control 19](#_Toc385620254)

[Table 6 - Optimal Control with node value caching 19](#_Toc385620255)

[Table 7 - Symbolic Regression 20](#_Toc385620256)

[Table 8 – Symbolic Regression 1000 population 20](#_Toc385620257)

[Table 9 - Symbolic Regression with node value caching 20](#_Toc385620258)

# Introduction

GEPsharp is a library for genetic programming in C#. It implements two different styles of genetic programming, which are described in John Koza’s Genetic Programming (GP) and Cândida Ferreira’s Gene Expression Programming (GEP) respectively. The library is designed to aid a programmer in setting up and using these genetic programming methods. It also provides some additional functionality beyond what the authors describe in their books.

This project also includes four sample problems which are solved using the GEPsharp library. The success and speed with which GP and GEP can solve these problems are compared, as are the effects of the additional functionality included in the library.

The first sample problem is the Artificial Ant on the Santa Fe Trail. In this problem, we must move a virtual ant around a 2D board. The goal is for the ant to eat as much food as possible by following the trail laid out. However, the trail has holes in it which the ant must be able to surpass. Using genetic programming we evolve strategies to control the ant’s movement.

The second sample problem is Boolean 11 Multiplexer. A multiplexer is a logical structure which takes a number of inputs and produces a single output. Some of the inputs form an address which points to one of the other inputs. The input which the address points to is output by the multiplexer. In the Boolean 11 multiplexer, there are 3 address inputs and 8 other inputs. All inputs and outputs receive a value of zero or one. Using genetic programming we try to evolve a function that selects the correct data to output based on the address.

The third sample problem is Optimal Control. In this problem we have a cart which is some distance from zero on a line. We must apply a force to the cart in such a way that it comes to rest as close to zero and in as little time as possible. Using genetic programming we evolve a function which takes the cart’s position and decides whether to apply the force at that moment.

The final sample problem is Symbolic Regression. In this problem, we generate a number of points on a line and attempt to derive the equation of the line. For this problem, we used the line:

We then use genetic programming to evolve functions which are as close as possible to this line, using the generated points as a guide.

# Motivation

## Background

### Genetic Programming

In nature, organisms evolve over time to adapt to their environment. An individual’s ability to survive and reproduce (its ‘fitness’) determines the chance it has to pass on its genes to the next generation. Genetic programming is an attempt to use this principle to inform the automatic evolution of programs.

Genetic programming is a form of program induction- the determination of a program to solve a problem from the problem itself. Genetic programs are not written by humans, but evolve through generations of programs that are evaluated in their ability to solve a given problem.

The ability to induce programs from problems successfully is hugely useful. If all you need to solve a problem is the problem itself, then you can theoretically solve anything. Of course, in practice things are not that simple. However, even with many limitations in place, program induction techniques such as genetic programming have massive potential.

The solution space (the set of all possible solutions) for most problems is infinite or near infinite. Even within a limited environment with a maximum solution size and a finite set of functions and terminals, the number of possible programs is incredibly massive. Exploring all the possibilities in any reasonable amount of time is simply out of the question.

Genetic programming attempts to solve this issue. Instead of evaluating all possible solutions, we create a pool of individuals and modify them over time. The individuals who are best at solving the problem (they have the highest fitness) are more likely to be kept for the next generation. Over a number of generations, we explore the solution space in a somewhat directed manner, hopefully coming closer to finding an optimal solution.

### Koza’s Genetic Programming

John Koza is the inventor of genetic programming (GP). He is a consulting professor at Stanford University and received his Ph.D. in computer science from the University of Michigan in 1972. Genetic algorithms, of which genetic programming is a subset, had been around for many years, however, Koza described the generation of full programs through genetic means for the first time in his 1992 book Genetic Programming.

Koza’s GP represents individual programs as trees of functions and terminals. These functions and terminals are selected by the operator during setup. When the initial population of programs is created, trees are populated randomly using the functions and terminals. After each generation, individuals are selected to survive to the next generation based on fitness. Then pairs of individuals are selected randomly, and branches of their trees are swapped (a process called crossover). This process creates entirely new individuals, which allows the exploration of new parts of the solution space.

### Ferreira’s Gene Expression Programming

Cândida Ferreira is the inventor of gene expression programming. She studied biochemistry at Kharkov University in the former USSR, and graduated with a Ph.D. in Biology from Lisbon University in 1995. Ferreira’s book Gene Expression Programming described a variation on the idea of genetic programming more closely inspired by biological systems.

Gene expression programming (GEP) includes a genotype/phenotype divide, unlike Koza’s GP. This means that the structure which is modified after each generation is not directly evaluable, but instead is used to create an equivalent evaluable structure. The advantage to this approach is the ability to use more powerful and varied genetic operators than crossover without fear of creating programs which are syntactically incorrect. A potential downside to this approach is that there is no longer a strong correlation between the magnitude and nature of the change made to the modifiable structure and the resulting change to the evaluable structure.

## Goals

My goals for this project were as follows:

* Implement GEP and GP in a flexible, easy-to-use library for C#.
* Add parallel fitness evaluation and node value caching features.
* Set up and run the four sample problems.
* Compare results between GEP and GP with and without the additional features turned on.

Having an implementation of GEP and GP as a C# library allows reuse of the algorithms in a commonly used language. Although the ideas behind these algorithms are not too complicated, creating an actual implementation can be somewhat challenging, and most definitely time consuming. With this library in place, the work only needs to be done once.

My focuses in writing the libraries were flexibility and ease of use. The library is intended to be used for any problem which can be expressed as a genetic programming problem, and it should be simple to set up once the user decides how they want to represent it. The standard parameters for genetic programming (population, generations, functions and terminals) are all easily configurable.

Parallel fitness evaluation can greatly reduce the time to run a genetic programming generation under certain conditions. First, the computer which is being used should be multi-core. This is standard in modern computers. Second, the fitness evaluation must be at least somewhat computationally expensive. If the cost of computation is low, the overhead of running in parallel may overshadow the gains from running multiple fitness evaluations concurrently. When these conditions are met, the time to run a generation can be slashed to a fraction of what it would have been sequentially.

Node value caching refers to the practice of considering each node of a genetic program tree as a potential solution, and caching the result of evaluating each node so that it does not need to be recalculated when nodes above it in the tree are evaluated. Ideally, this increases the pool of potential solutions immensely without increasing the computation cost much. For some problems however, it does not really make sense to cache the value of a node for reuse, such as problems where the node structure describes a strategy for actions rather than a specific value.

The four sample problems (described in detail later) used in this project are the first four example problems used in John Koza’s book on genetic programming. They were selected to represent various common types of problems that GP could be used to solve. It seemed appropriate to reuse them for this project to make a comparison between GP and GEP.

In Ferreira’s book, she claims that GEP is vastly superior to Koza’s GP. She stresses that the genotype/phenotype divide is a prerequisite for truly powerful and versatile evolution, and that operating on strings rather than trees decreases computation time by a large factor. These claims being quite sensational, I decided to compare the speed and effectiveness of GEP and GP in solving the four sample problems. This data could be useful in making decisions on which type of genetic programming to employ in the future.

# Methodology

## Language Choice

As an imperative, object-oriented language, C# is not an obvious choice for implementing GEP and especially GP. However, the classification of the language is somewhat deceiving. C# is extremely flexible and includes a number of features modeled after functional programming. Lambda expressions, dynamic typing, relational-algebra-style list manipulation, pure functions (guaranteed to have no side effects), delegates (treating functions as data), and the yield keyword (used to create functional style streams) are all powerful tools which allow C# to work well for a wide variety of tasks- including GP and GEP.

One of the main goals of the project was to create code that would be reused on a variety of problems outside this project alone. C# is an excellent choice for achieving this goal, as it is a very common and widely used development language. Although it may not be the number one choice for every problem, C# is well rounded enough to be a decent choice for almost anything.

## Library

### Design

The design goals for the GEPsharp library were as follows:

* Wherever possible, treat GEP and GP as interchangeable.
* Expose only a small interface and hide implementation details from the user.
* Make sure that setting up a new problem is simple once the user understands how they want to represent the problem (in terms of functions, terminals, and fitness).

To achieve these goals, I broke the code down into three levels. The problem level contains information about the whole population of solutions. It stores the fitness function, various parameters such as the rate of occurrence for each reproductive operator, the best solution so far, and a list of individuals.

The individual level corresponds to a single member of the population. Each individual can be made to evaluate itself and return an answer. An individual may include multiple domains. In the case of GP there is only ever one domain, but for GEP multi-genic chromosomes are possible, and these are represented through multiple domains.

The domain level contains an actual tree or string structure for the individual of which it is a part. It also stores pointers to the sets of terminals and functions that are allowed for the problem. Domains contain code to generate starting random configurations. While which individual will be selected for genetic operators is determined at the problem level, the actual implementation of these operators takes place at the domain level.

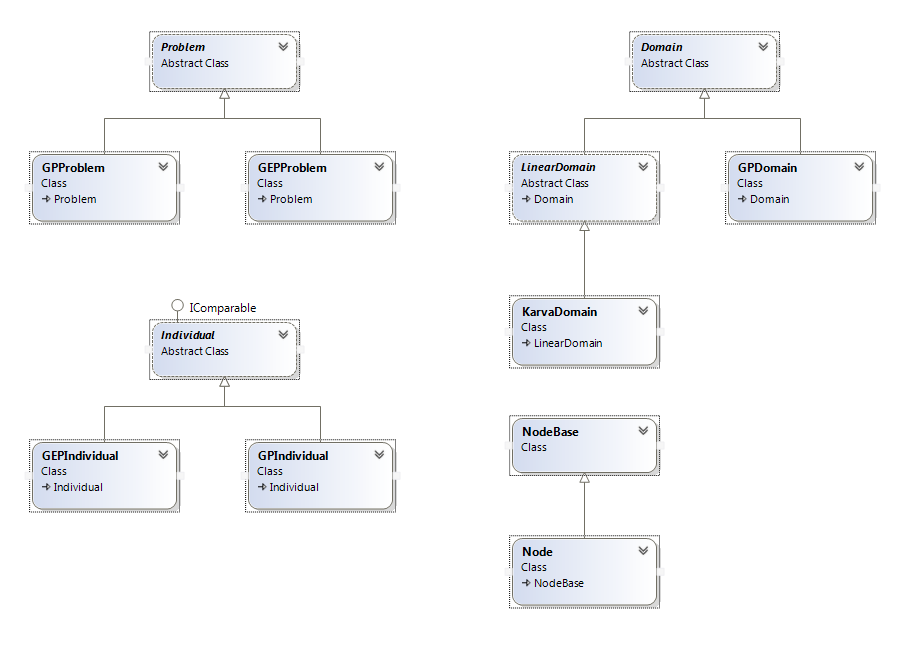


Figure - Class Diagram for GEPsharp

As shown in Figure 1, at each level there is an abstract base class (*Problem, Individual,* and *Domain*), from which classes specific to GEP and GP derive. In the case of *Domain*, there are two GEP related derived classes. *LinearDomain* describes a string based domain, which applies to all GEP domains. *KarvaDomain* is a subclass of *LinearDomain* which describes a domain which contains a Karva expression, the basic type of GEP gene. Other domain types for GEP are possible (for example, a domain encoding numerical constants), but I have not specifically implemented anything for these domains at this time.

The *NodeBase* and *Node* classes are used to describe and organize functions and terminals. Both functions and terminals are represented as a *Node* object. Originally, *NodeBase* was used for GEP and *Node*, which includes child pointers, was used for GP, but later this was changed so that both GEP and GP used *Node*. This was so that the same initial set of functions and pointers could be given to both a GEP and GP problem. For GEP problems the child pointers are simply ignored. Given more time, I would have liked to merge *NodeBase* and *Node*, as there is now no real purpose to having them split.

### Implementation

The initial implementation of the library went fairly smoothly. Although I encountered some problems, none were serious enough to stop progress for more than a few hours. Evaluating Karva expressions proved to be very simple. The implementation of the wide variety of genetic operators used in GEP was time consuming but not particularly difficult. Tree generation in GP took some time to get right, especially the ramped half-and-half method. Crossover in GP initially seemed trivial, but I later discovered a bug with how node depths were updated, which was causing the maximum tree depth to be exceeded. Once this was fixed, I had no further problems with GP.

After implementing the test cases, I realized that my implementation did not support fitness cases well. The initial implementation did not give the user access to an evaluable tree- instead it simply evaluated the tree itself, *one time*, and handed off the answer. There was also no way to pass problem parameters to the evaluation of the tree. Obviously, this presented a serious problem for the implementation of several of the test cases.

I later solved this problem by creating a class called *Evaluable*, which has three members- an array of child *Evaluable*s, a pointer to a function representing the logic for a function or terminal, and a function which takes problem parameters and calculates a result based on the delegate and children. The purpose of this class is to allow re-evaluation and problem parameters to be used on a solution provided by the library.

To use the *Evaluable* class, the list of functions and terminals given to the library are written such that instead of returning an answer, they return a structure of *Evaluables* which is identical to the GEP or GP tree. Then that *Evaluable* structure can be evaluated as many times and with as many parameters as needed to calculate the fitness of the individual.

Because the *Evaluable*s process is somewhat complicated, I will walk through an example here. Consider the Artificial Ant problem. In this problem we have three functions and three terminals. We will look at the ‘If-Food-Ahead’ function. For this function, we need to ask the Trail object, which contains data about the position of the Ant and the food, whether there is food directly ahead of the ant. If there is, we evaluate the first child of the ‘If-Food-Ahead’ node. If not, we evaluate the second child instead. Note that some detail is removed from the actual function IfFoodAhead, for the sake of readability.

public static object IfFoodAhead(Evaluable[] Children, object[] outsideParams)

{

Trail trail = outsideParams[0] as Trail;

return trail.FoodAhead() ?

Children[0].Evaluate(outsideParams) :

Children[1].Evaluate(outsideParams);

}

This function takes an array of *Evaluable* objects representing the children of the given instance of IfFoodAhead that is being evaluated, as well as an array of outside parameters. Because the function must meet a general template for all functions and terminals used to create *Evaluables*, the outside parameters are passed as a general object array instead of being typed. Similarly, the return is stated as object. In this case, the function returns an integer, the number of food pieces eaten during the evaluation of its selected child.

To create a *Node* object which can be given to the GEP or GP problem and which returns an *Evaluable* structure correctly for ‘If-Food-Ahead’, we use the following code.

Node node = new Node("If-Food-Ahead", 2, (children =>

{

new Evaluable(AntFunctions.IfFoodAhead, children.Cast<Evaluable>().ToArray());

}));

There are three parameters for the *Node* constructor. The first is the name of the *Node,* as it will appear in printed trees. The second is the number of children the node must have, its ‘arity’. The third is a function which describes what the node should return when evaluated. We use a lambda expression for this function. The syntax “children =>” begins a lambda expression which has one parameter, named children. When the node is evaluated by the GEP or GP problem, the results of evaluating the children will be in this array.

Inside the lambda function, we need to return the result of evaluating the *Node*. We want the *Node* to return an *Evaluable,* so we construct one. The *Evaluable* constructor takes two parameters- the function it will use each time it is evaluated, and a list of its children. The function is IfFoodAhead, from above, and the children are given in the parameters. To make the constructor accept the children parameter as an array of *Evaluable*s, we need to cast it. The cast method used is a generic one which boxes the array as an *IEnumerable*, so we need to unbox it by calling ToArray().

Because this process of converting functions and terminals to *Nodes* which in turn create *Evaluables* is verbose and somewhat involved, I provide a static function which takes a list of the functions and terminals as function pointers, such as IfFoodAhead from above, and produces a list of *Nodes* that will return equivalent *Evaluables*, so that the user does not have to perform this process themselves.

## Test Cases

### Artificial Ant

In the Artificial Ant problem, the goal is to guide an ant along a food trail to the end, while navigating the gaps and turns in the trail. The trail is represented as a series of marked locations on a 2D board. The ant is represented with a position and facing direction. Each turn the ant either moves forward or turns, and it has four hundred turns to eat as much food as possible.

The terminals used for this problem are the ant’s possible actions- turn-left, turn-right, and move. One of the functions is if-food-ahead, which runs the first of its two children if there is food directly ahead of the ant, and the second child otherwise. The other functions are Do-2 and Do-3, which have two and three children respectively, and simply run each child in turn.

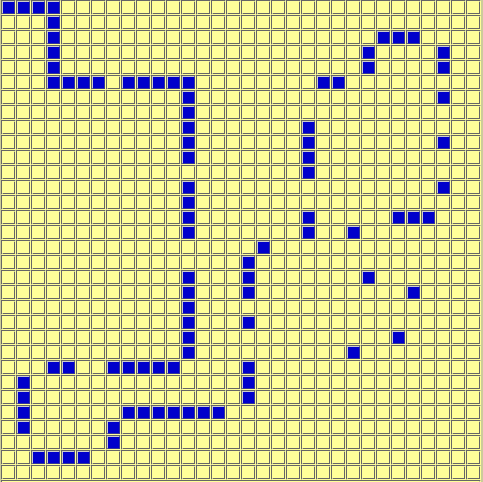


Figure - Santa Fe Trail.

Figure 2 shows the trail which the ant must follow in blue. The ant begins in the top left corner of the board facing to the right. In my version of the trail, the starting location does not count as a food location. As the trail goes on, more and more difficult gaps are placed for the ant to navigate.

I began my implementation of Artificial Ant by writing the *Trail* class. This class represented the Santa Fe trail as a 2D array. The food locations are read in from a text file. The ant was represented by a point and a facing direction (up, down, left, or right). The class has functions for checking if food is ahead of the ant, moving the ant, turning the ant, and checking how much food has been eaten and how many turns have passed.

When I originally implemented Artificial Ant, my library did not contain support for *Evaluable*s. At the time it was only designed to return single value solutions, such as a string or integer. I came up with the solution to create an *AntNode*, which basically worked identically to an *Evaluable*, except in that the functions for the Artificial Ant problem were hard-coded in, and instead of taking an object array, it took a *Trail* object and a turn counter. For more detail on *Evaluable*s, see the section on implementation of the library.

With this solution in place, the fitness function for the problem receives an *AntNode* object which can be evaluated multiple times. It then creates a *Trail* object and passes that to the *AntNode* to run repeatedly until either four-hundred turns have passed (turns are counted for each movement and change of direction, not each evaluation of the *AntNode*), or until all the food is eaten. The fitness is scored as the number of food pieces eaten. The max score is eighty-nine.

Later I realized that the *AntNode* solution was both very useful and required for any problem for which fitness calculations used multiple fitness cases. This eventually led to the development of the general *Evaluable* class, which replaced the *AntNode*.

My first implementation of the *Trail* class had a bug in it which caused the If-Food-Ahead check to look down when it should be looking up and vice versa. This was simply a case of putting a +1 where there should be a -1. This bug was present when the Artificial Ant problem was first tested, but I have re-run my tests since fixing it.

### Boolean 11 Multiplexer

A multiplexer is a logical function which takes a large number of inputs and produces a single output. The inputs can be split into two groups. The first is a collection of data inputs, each of which has a value of zero or one. The second group of inputs form a binary address to one of the data inputs. The output is whichever data input is addressed. A Boolean 11 multiplexer has 3 address inputs and 8 data inputs. In general, a Boolean multiplexer has *n* address inputs and 2*n* data inputs.

My goal was to evolve the Boolean 11 multiplexer function using genetic programming. The terminals for this problem were the eleven inputs, labeled A0 – A2 (address) and D0 – D7 (data). The functions were the logical AND, OR, NOT, and IF.

Fitness was determined for this problem by checking how many of the 211 different possible configurations of inputs resulted in the correct output after applying the evolved function. Essentially, there are 211 different fitness cases, each of which recieves a score of one if the output matches the correct value and a score of zero otherwise.

Because this problem has multiple outside inputs and multiple fitness cases, it required an *Evaluable* – like structure. As I had already solved this problem for Artifical Ant when I began working on Boolean 11 Multiplexer, I used the same strategy. Once the *Evaluable* class was created, I replaced the specialized nodes for Boolean 11 Multiplexer with *Evaluables*. For an explanation of this strategy, see the Artificial Ant and library implemenation sections.

### Optimal Control

Optimal Control actually represents a whole branch of problems. Specifically, the problem I worked with was cart centring. In the cart centring problem, the goal is to move a cart along a line from its random starting location until it comes to rest at the goal location. This is represented on a number line, with zero as the goal location. The cart can only be moved by a ‘bang-bang’ force, which is a fixed magnitude force that can be applied in the positive or negative direction at any time step.

The goal in my case was to evolve by means of genetic programming a strategy for when to apply the bang-bang force to the cart such that it will always come to rest at the goal location as quickly as possible. The input for this strategy is the cart’s position and velocity, and the output is a decision to either apply the bang-bang force or not in that situation.

The terminals for this problem are negative one, the cart’s position, and the cart’s velocity. The functions are the mathematical operators plus, minus, multiply, divide, and absolute value. Additionally, the ‘greater than’ function is available. This function simply returns one if the first input is larger and negative one otherwise. When the strategy is evaluated, it returns a numerical value. If the value is positive the bang-bang force is applied in the positive direction, and vice versa for negative. If the value is zero, the bang-bang force is not applied that time step.

Fitness for this problem is determined by applying the evolved strategy to a number of fitness cases. In each fitness case, the cart is placed on the number line at a random location with a random velocity. Then, the cart’s position and velocity are given to the evolved strategy, which decides whether or not to apply the bang-bang force that time step. Then, the cart’s position and velocity are updated and the process begins again. This continues until either five hundred time steps have passed, or the square root of the squares of the cart’s position and velocity is less than the capture radius, which for this problem was set at 0.02.

The score for each fitness case is five hundred minus the number of time steps passed- the faster the cart is centred, the better the score. Fitness for the whole problem is the sum of the score for each of the twenty fitness cases. The maximum fitness then is technically ten-thousand, but that high a score can only occur if all fitness cases begin with the cart in the capture radius. In practice, high fitness individuals score in the range of nine thousand.

Like all the problems which involve fitness cases, I originally needed to use a strategy similar to the one used for Artificial Ant for this problem. Once the *Evaluable* structure was in place, I switched to that. See the Artificial Ant and library implementation sections for more details.

### Symbolic Regression

In the Symbolic Regression problem, the goal is to find the function for a line given a set of points on that line. In my case, the line used was:

We try to find this function, or a very good approximation of it, by using genetic programming to evolve functions which are close to this line. The only terminal used in this problem is . The functions are all mathematical, and include plus, minus, the exponential function, multiply, divide, the natural logarithm, sine, and cosine.

We generate twenty points on the line by randomly choosing values of and calculating the corresponding values of We then pass those same values of into the evolved functions and find the difference between the they calculate and the actual value of . If the difference is larger than a certain value (called the accept radius), we calculate fitness as:

If the difference is smaller than the accept radius (for example, if the difference is zero), we instead calculate fitness as:

In this case, the accept radius was given as 0.01. This also gives us a maximum fitness value for a given point, in this case one-hundred. As there are twenty points, we have a total maximum fitness of two-thousand.

Like all the problems which involve fitness cases, I originally needed to use a strategy similar to the one used for Artificial Ant for this problem. Once the *Evaluable* structure was in place, I switched to that. See the Artificial Ant and library implementation sections for more details.

# Results

## Artificial Ant

The table below describes the results of running Artificial Ant. Artificial Ant was not run with node value caching, as it makes little sense to cache the value of a strategy for an Ant’s movements.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter | GEP | GP | GEP-Parallel | GP-Parallel |
| Population | 500 | 500 | 500 | 500 |
| Generations | 50 | 50 | 50 | 50 |
| Number of Runs | 20 | 20 | 20 | 20 |
| Maximum Fitness | 69 / 89 | 69 / 89 | 75 / 89 | 79 / 89 |
| Average Fitness | 48.85 / 89 | 45.25 / 89 | 49.30 / 89 | 47.35 / 89 |
| Average Time (s) | 2.68 | 7.54 | 1.45 | 7.10 |

Table - Artificial Ant

I was still somewhat disappointed with the fitness results from this test. Although a solution scoring 79 was a definite improvement, the average fitness levels were still very low, despite fixing the bug that was previously present in the *Trail* class. The original results are reprinted in table 2 below.

Because of my dissatisfaction with the improvement in average fitness, I decided to try another test with a higher population value. The results are shown in table 3 below. There was a significant improvement in the average fitness, and a perfect solution was found. With parallel fitness evaluation, the time to run each test was still very short.

|  |  |  |
| --- | --- | --- |
| Parameter | GEP | GP |
| Population | 500 | 500 |
| Generations | 50 | 50 |
| Number of Runs | 20 | 20 |
| Maximum Fitness | 65 / 89 | 61 / 89 |
| Average Fitness | 42.50 / 89 | 42.75 / 89 |
| Average Time (s) | 2.14 | 6.70 |

Table - Artificial Ant original results

|  |  |  |
| --- | --- | --- |
| Parameter | GEP-Parallel | GP-Parallel |
| Population | 1000 | 1000 |
| Generations | 50 | 50 |
| Number of Runs | 20 | 20 |
| Maximum Fitness | 89 / 89 | 60 / 89 |
| Average Fitness | 61.6 / 89 | 51.45 / 89 |
| Average Time (s) | 3.51 | 12.89 |

Table - Artificial Ant 1000 population

Figure 3 shows the optimal solution to the Artificial Ant problem found by GEP-Parallel in one of the one-thousand population runs. The numbers at the end of each node name are simply so that the tree-drawing code can identify the nodes separately.

## Boolean 11 Multiplexer

The table below describes the results of running Boolean 11 Multiplexer with default settings, as well as a GP-Parallel run. Because Boolean 11 Multiplexer has very low cost per fitness evaluation, running with parallel fitness evaluation on actually slowed down the test significantly. The overhead cost of running in parallel outweighed the benefits. Because of that, I chose not to do a GEP-Parallel run.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | GEP | GP | GP-Parallel |
| Population | 4000 | 4000 | 4000 |
| Generations | 50 | 50 | 50 |
| Number of Runs | 20 | 20 | 20 |
| Maximum Fitness | 1923 / 2048 | 1911 / 2048 | 1926 / 2048 |
| Average Fitness | 1898 / 2048 | 1894 / 2048 | 1905 / 2048 |
| Average Time (s) | 5.98 | 8.03 | 35.58 |

Table - Boolean 11 Multiplexer

Figure 4 below shows the best solution from the GP-Parallel run.

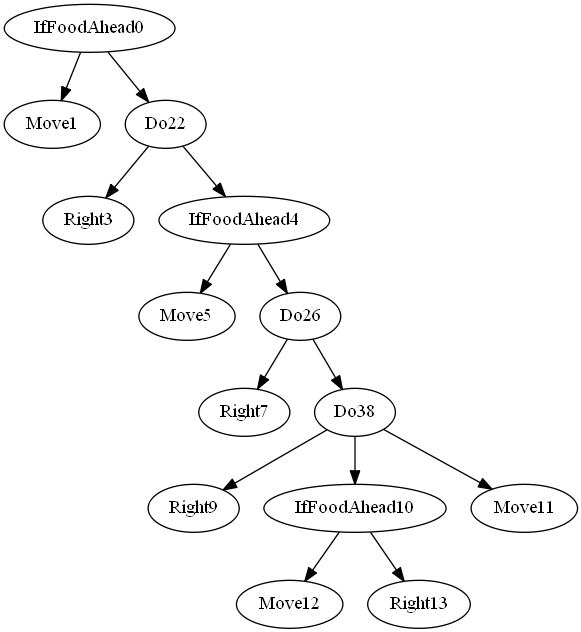


Figure - Optimal Artificial Ant solution

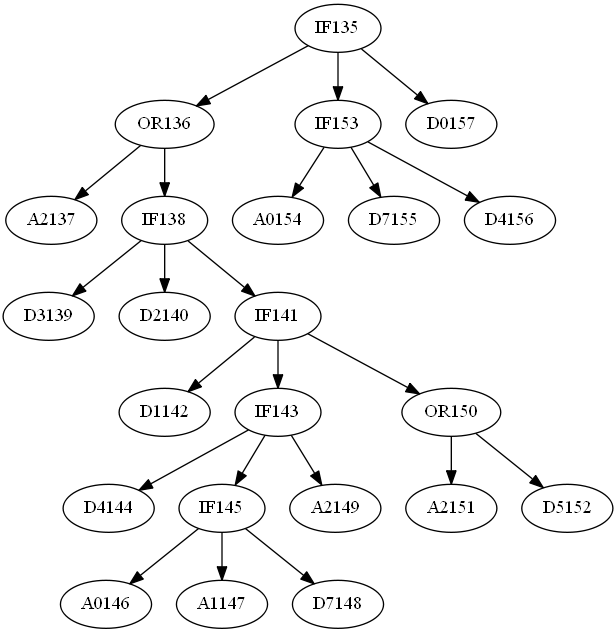


Figure - Best Boolean 11 Multiplexer solution

## Optimal Control

The table below describes the results of running Optimal Control with and without parallel fitness evaluation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter | GEP | GP | GEP-Parallel | GP-Parallel |
| Population | 500 | 500 | 500 | 500 |
| Generations | 50 | 50 | 50 | 50 |
| Number of Runs | 20 | 20 | 20 | 20 |
| Maximum Fitness | 8948 / 10000 | 9069 / 10000 | 8823 / 10000 | 9138 / 10000 |
| Average Fitness | 5767 / 10000 | 6083 / 10000 | 6723 / 10000 | 6714 / 10000 |
| Average Time (s) | 146.87 | 1282.66 | 35.86 | 359.77 |

Table - Optimal Control

It’s clear to see that parallel fitness evaluation made a huge difference in the running time of these tests. Optimal Control was by far the slowest test, with the most computation intensive fitness evaluation. This makes it perfect for parallel fitness evaluation.

I also tried running optimal control with node value caching enabled. Because the values returned are numeric, it does make some sense to try all nodes as their own solution. However, the time cost for this option was very high. I had parallel enabled in hopes of reducing it to a manageable level, but for GP it was simply too long to do more than a single run.

|  |  |  |
| --- | --- | --- |
| Parameter | GEP – Node Value Caching | GP – Node Value Caching |
| Population | 100 | 100 |
| Generations | 50 | 50 |
| Number of Runs | 20 | 1 |
| Maximum Fitness | 8682 / 10000 | 9444 / 10000 |
| Average Fitness | 6331 / 10000 | 9444 / 10000 |
| Average Time (s) | 271.14 | 8 hours 27 minutes |

Table - Optimal Control with node value caching

The results were promising for GP (although with only a single test, it could have simply been luck). However, the eight hour run time is prohibitively long, at least for my purposes.

## Symbolic Regression

Although average fitness levels are quite low for this problem, the maximum fitness is a significant distance from the average. This indicated to me that the problem might benefit from a larger population size, so I tried running it with population one-thousand and parallel fitness evaluation turned on. I included the standard deviation of fitness in these tables for the purpose of comparison.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter | GEP | GP | GEP-Parallel | GP-Parallel |
| Population | 500 | 500 | 500 | 500 |
| Generations | 50 | 50 | 50 | 50 |
| Number of Runs | 20 | 20 | 20 | 20 |
| Maximum Fitness | 2000 / 2000 | 1684 / 2000 | 1651 / 2000 | 1715 / 2000 |
| Average Fitness | 1115 / 2000 | 1168 / 2000 | 1191 / 2000 | 1212 / 2000 |
| Std. Dev. Fitness | 322 | 271 | 208 | 226 |
| Average Time (s) | 0.66 | 3.41 | 0.49 | 1.45 |

Table - Symbolic Regression

|  |  |  |
| --- | --- | --- |
| Parameter | GEP - Parallel | GP - Parallel |
| Population | 1000 | 1000 |
| Generations | 50 | 50 |
| Number of Runs | 20 | 20 |
| Maximum Fitness | 1672 / 2000 | 1603 / 2000 |
| Average Fitness | 1222 / 2000 | 1266 / 2000 |
| Std. Dev. Fitness | 189 | 195 |
| Average Time (s) | 1.31 | 9.40 |

Table – Symbolic Regression 1000 population

These one-thousand population runs have slightly higher average fitness than the five-hundred population runs. However, neither found an optimal solution. They did seem to reduce the standard deviation of the fitness values, but not by a huge amount.

Symbolic regression is a problem well suited for node value caching, so I also tried turning on this option. I left the parallel fitness evaluation option on for this test as well, having seen that it sped up the regular run. However, this may have been a mistake. When doing node value caching, many solutions will be smaller than is typical for the problem, as they represent only a small part of a greater solution. Because there are such a large number of fitness evaluations, and each one is so tiny, the overhead cost of running in parallel may have outweighed the benefit.

|  |  |  |
| --- | --- | --- |
| Parameter | GEP – Node Value Caching | GP – Node Value Caching |
| Population | 500 | 500 |
| Generations | 50 | 50 |
| Number of Runs | 20 | 20 |
| Maximum Fitness | 2000 / 2000 | 2000 / 2000 |
| Average Fitness | 1365 / 2000 | 1473 / 2000 |
| Std. Dev. Fitness | 267 | 202 |
| Average Time (s) | 2.79 | 301.84 |

Table - Symbolic Regression with node value caching

Running with node value caching on improved the average fitness significantly, and an optimal solution was found by both GEP and GP. However, the standard deviation of fitness remained high. Additionally, in the case of GP the time to run the test increased by an extremely large amount.

I analyzed and simplified the best tree from the GEP-Parallel one-thousand population run. This run was selected because it was not an optimal solution (which would undoubtedly simplify to the equation of the line we used to generate the points) and because it was quite short and easy to read. The result was the following equation.

Through this genetic programming run it was discovered that the equation above is a reasonable approximation of our original line:

## Conclusions

The most important conclusion I can draw from this project is that GEP runs significantly faster than GP. Consistently, over every problem, with every set of parameters, with and without parallel fitness evaluation or node value caching, GEP takes much less time to complete an equivalent amount of work to GP. Because of the fixed length of GEP strings, and the nature of doing operations on a data structure with constant time random access, the cost for all GEP genetic operators is very small. The GP trees are comparatively speaking quite slow.

Fitness differences between the two styles do not seem to be very significant. I have observed what seems to be a trend of GP solutions having slightly higher average fitness, but the differences and sample sizes are far too small to say anything definitive. Certainly, GEP has been shown to work effectively enough to solve these sample problems, at the very least.

It’s important to remember that all of the problems presented here are ‘toy’ examples. In all cases a known optimal solution exists, and the test was simply to see if genetic programming could be used to find it. To truly test the worth of GEP, it would need to be used to solve a much larger, more significant problem. However, a library now exists with which to begin that work.

As for the reusability of the GEPsharp library, I am quite satisfied with how it turned out. In fact, I used the library to evolve strategies for a simple two player game as part of my COMP 4106 (Artificial Intelligence) final project, and it worked very well. I hope to continue to find use for the GEPsharp library in the future.

# Bibliography

Ferreira, C. (2002). *Gene Expression Programming.* Angra do Heroismo, Portugal.

Koza, J. R. (1992). *Genetic Programming.* Cambridge: MIT Press.

Oppacher, F. (2013). Private Communication.

# Appendix A – Running and Viewing the Code

An exe is provided with this submission for each of the four sample problems. These exes can be found in the “Testing Exes” folder. When the exe is run, it will request the following information:

* Whether to use GEP or GP
* Population
* Generations
* Number of tests
* Whether to use node value caching
* Whether to use parallel fitness evaluation
* A filename

Exercising some care when inputting this information is recommended, as the program will simply crash if you input something unexpected. Error checking the input wasn’t a high priority.

The filename given should be entered without an extension. As each test finishes, the program will write some general data to a file with that name and the .txt extension. It will also write the fitness of the best individual and the time taken for the test to .csv file, and will output the best individual’s structure as a .gv file, which can be displayed graphically using the program graphviz, available for free at <http://www.graphviz.org/>. After all tests have run, three more lines will be added to the .csv. These lines contain the syntax for calculating the max, average, and standard deviation of the fitness and time in excel. The output files will all be contained in the folder “Test Output”. The .gv files will be in a subfolder labeled “Graphs”.

The code can also be run by opening the solution in Visual Studio (2012 or later). The GEPsharp project contains the library itself, and each of the sample problems have their own project. To view the code without Visual Studio, simply open the .cs files in whatever editor you prefer.