12-7-2015

GBGP-Exploring Algorithm Discovery by Genetic Programming in imperative Language

Louis Harold Getz IV

GBGP- Exploring Algorithm Discovery by Genetic Programming in an Imperative Language

by

Louis Harold Getz IV

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

of

The University of Alabama in Huntsville

12/7/2015

Honors Capstone Director: Dr. Dan Rochowiak

Associate Professor of Computer Science

[illegible] 12-10-2015

Student Date

[illegible] 12-10-2015

Director Date

[illegible] 12-10-2015

Department Chair Date

[illegible] 12-10-15

Honors College Dean Date

HONORS COLLEGE
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

Louis Getz
Student Name (printed)

[illegible]
Student Signature

12-10-2015
Date

Abstract

Evolutionary algorithms, based on observations of biology, have experienced success in modern computing for many specific problem sets. Each time requiring an advanced programmer to create a specific solution for each case. In this paper I present software designed to give full control over a grammar-based genetic programming application for the purpose of testing and research. I describe main concepts of how the software functions, and show how a researcher can quickly create or integrate any code into the genes of a genetically built algorithm. This simplifies and shortens the time required and learning curve involved in building and evaluating cases using a Grammar-Based Genetic Programing (GBGP).

Keywords: Genetic Programming, Java, Evolutionary Algorithms, Fitness Landscape

1 Introduction

One of the great difficulties in conducting research is the time and resource gap between an idea needing research and its execution and testing. Currently, if a researcher wants to apply genetic programming to problem they have to understand how to create the underlying software then invest time and resources into creating the software to test the specific hypothesis. There are a couple of software package options available (EpochX, JAGA, JGAP) but the process of learning to use the software or API generally nulls the benefit of its use in the first place. The purpose of this paper and the software it presents is to reduce that resource gap and provide researchers, with or without a strong software engineering background, an easy to use tool for creating and manipulating evolutionarily built applications, namely with Genetic Programming (GP). This tool, named GBGP (Grammar-Based Genetic Programming) gives quick GUI (Graphical User Interface) access to many common parameters involved in creating genetic programs as well as providing a generic interface for programmers to quickly create or integrate existing Java code as the genes in a GP environment.

To facilitate the software, this paper provides: a brief background in the field of evolutionary algorithms in relation to genetic programming (Section 2), a brief description of the purpose of GBGP, a comprehensible breakdown of how to interact with the software (Section 4), a description of its design and algorithms (Section 5), results of a test run an integer array sorting algorithm (Section 6), and finally close with conclusions and further improvements (Section 7). The problem I will attempt to solve using GBGP is to generate an algorithm for sorting an array of random integers of an arbitrary yet limited size.

Readers from any background interested in genetic programming will find this paper and its accompanied software useful since it is designed to be as flexible, simple, and quick to use as possible.

2 Background

Genetic programming is a subclass of evolutionary algorithms inspired by biological evolution where instead of making species of organisms it designs programs based on a fitness or survivability [2]. First invented in the 1960's by a man named John Holland and sometimes referred to as a hill climbing technique, due to its slow goal approaching nature, it optimizes a population of computer programs across a fitness landscape . This landscape is referred to as the search space where the computer program once tested gives access to one point in the search space; and the search space is all possible programs that can be created given the constraints, usually infinite in size [1]. By applying evolutionary techniques observed in nature such as DNA mutation, crossover, and reproduction on program structures, we can change the output of a program without human intervention and hopefully move towards a more correct program for the given problem and its fitness. Generally speaking, for most programs where random code is mutated or taken from other programs, the outcome is noise. A fitness function evaluates the efficacy of each program to separate useful programs from the noise. Programs that score higher on the fitness function are rated as more valuable and have a higher probability of being selected for the genetic operations in the next generation. By repeating this process continually, the aim is that some extent of evolution will occur on computer programs where the final result is a program that correctly solves the problem described by the fitness.

3 What is GBGP

GBGP is a tool I developed as a solution to the genetic programming paradigm. It has been designed with three goals in mind: First, it's purpose is to abstract all variables that are not exactly knowns in genetic programming and give the user access to those variables. This is important because the outcomes of changes on these values are not known and therefore are subject to change and consequently research. The specific variables and each of their impacts is described thoroughly in section 4. Second, its purpose is to allow advanced or amateur programmers to create structures that they may find useful in the construction of species (programs) in a population. It accomplishes this by keeping the software completely generic and separate from any implementation. To include your own code comes down to adding one line for each class, method, and variable you wish to give access as genetic code. Lastly, it gives live information and graphs summarizing the population as it goes through each generation of genetic operations. By combining these three traits, you have an easy to use tool that allows useful quick modifications both in parameter changes and in code changes that provides important information of your GP run.

4 Interaction with GBGP

GBGP has been designed with simple interaction on several levels. The front end or user interface, provides access to many of the parameters of interest in genetic programming, as well as displaying live infographics (line graphs) of important summary information of the population over time, such as mean . and standard deviation. The back end or code interaction, has no API and requires almost no knowledge of how the code operates, but is designed to be flexible with literally any Java code. With only a basic understanding of Java code, anyone from any background can interact and work with GBGP on any level.

4.1 User Interface. The user interface serves two main purposes, first to give access to values that affect the operation of GP. These values include program structures such as the number of statements allowed to be generated in each method or the number of variables allowed to be generated in each method or class. These parameters can be edited on the fly and tests can be immediately run to see the changes that the different values effect. The results are displayed in a few different formats: the left panel displays a short summary of the current iteration of the population which includes the best individual score, the mean score, and standard deviation of scores in the population. To gain more information or to view these numbers as they change over time, navigate to the InfoGraphics tab where line graphs are drawn as each iteration is completed. This information is displayed live as each iteration of genetic operations is completed. Summaries over time are also provided where the results of each iteration are stored and the mean and standard deviation values are displayed in a line graph. The interface is equipped with popups on hover that explain each field. Interacting with the Graphical side of the program is simple and self-explanatory with the info bubbles given. For that reason no more details are necessary about interacting with the UI.

4.2 Parameter Manipulation. Key to the purpose of GBGP is the ability to change parameters during runtime. The simulation can be paused, parameters changed, and then resumed again with the new parameters. For instance, if you wanted to allow more statements for the programs to use, you could pause the simulation, change the parameter, and resume operation. This could have very interesting results and requires further investigation, especially on mutation and crossover depth. Since those appear to be more beneficial as low value early on and larger value later, equating to large changes in code in infancy stages and perhaps more fine tuned changes with already somewhat evolved programs.

It is not possible to pause the simulation, make code changes, and resume. In the future adding new code might be possible, but currently changes won't go through until a new population is generated.

4.3 Code Interaction. Creating code as the genes in a genetic programming environment GBGP is a very simple process. Only a single Java class must be created. This class is the Input/Output class, best named IO. Each program in the population creates and manipulates its own IO object. It is in this class that all variables used in determining fitness must be defined. The IO class requires only one static method: "rate", with a return type of double and parameter of the IO object. Beyond this the programmer is allowed to create any other classes, methods, and variables that is or is not used by the genetic code. As long as you create a static "rate" function with IO parameter that returns a decimal value fitness, you can run GBGP. You are also allowed to create any further Java classes you desire. These must be categorized as Library classes.

- IO - The Class responsible for holding Input and Output as well as defining a "rate" function which returns a fitness value of type double.
- Library - A support Class which can have any desirable purpose. It can be a static method Library class with methods that have helpful logic or it can be an Object defining class that a program is allowed to instantiate. If you want something visible to GBGP to be inserted as genetic code it must be defined as a Library class (other than the IO class).

Now that you have a clear distinction between the two class types how do you create them? GBGP uses comments as commands that are parsed before your code is compiled.

4.3.1 GBGP Commands. Telling GBGP to make a class, method, and variable visible all use the same syntax. When I say visible I am referring to a program in the population being allowed to create an object of (non-static) or use (static) a visible class, call a visible method, and write to or read from visible variables. First the "//#" tells GBGP that this line is a command, second the command type must be written directly after the '#'. The only available command types are as follows:

- IO - Declares the *only* IO class
- Library - Declares a new visible Library class
- Var - Declares a new visible variable
- Method - Declares a new visible method

After the type is written the order of the parameters is no longer relevant. You can reorder them however you like. If you do not include a command when it is required a default value is used. In the case of the "name" command you must always include it. All possible parameters after the type are as follows:

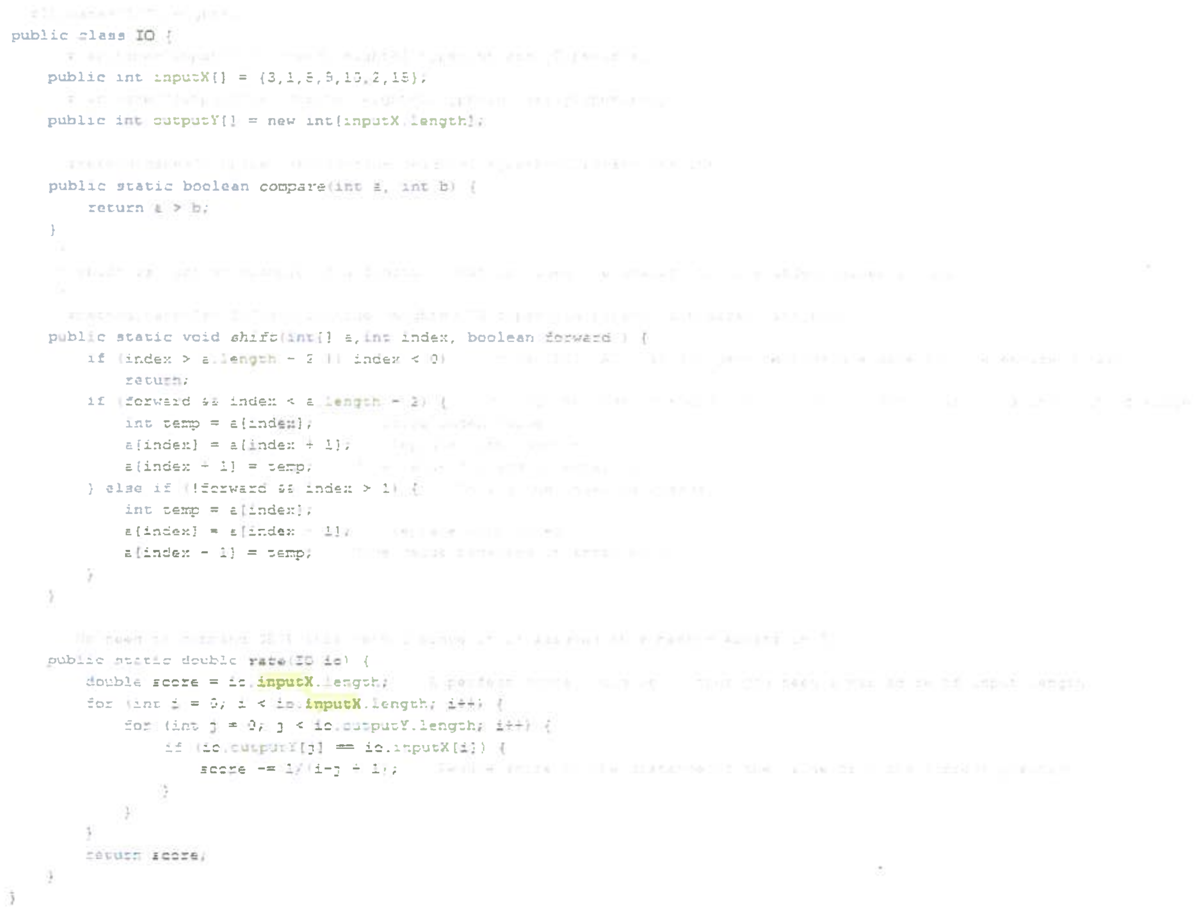
Command	Description	Example	Allowed on	Value 1	Value 2	Value 3	Value 4	Default Value
name	defines the name of the class, variable, or method	name=var1	Library, Method, Var					""
r/w	describes a variable as read only, write only or both	r/w=read	Var	read	write	both		both
weight	defines the likelihood this structure is selected during gene selection	weight=0.5	IO, Library, Method,Var	0.0-1.0				0.0
type*	describes the variable type or method return type, it also describes the array type	type=myClass,array type =int,single type =dbl,arrayList	Method, Var	int	bool	dbl	void & Custom classes	null
arrayType	defines the array type of a variable or return array type of a method	arrayType=single	Method, Var	single	array	arrayList		single
static	describes a variable, method, or class as static	static=true	Library, Method, Var	true	false			false

params*	defines the parameters of a method	params={int,dbl} params={(int,array),(dbl,arrayList)}	Method	int	dbl	bool	custom class	

Figure 1. Table of commands read by GBGP.

* Parameters when are not of array type 'single' can be given another array type by surrounding the type with parenthesis and including the array type after a comma. Ex: params={(myClass,array)}

By now you may have noticed that only types of integer, double, and boolean are supported. That is correct; any additional object types must be defined as a non-static Library class. To better understand the commands below you in Figure 2 you will see an example code snippet of the IO class used in testing GBGP to find an integer array sorting algorithm. It contains an input variable (integer array "inputX") and an output variable (integer array "outputY"). It is important to note that this is not mandatory. However, it is helpful to keep at least 2 separate variables for input and output. This is because you can declare the input as read only, and the output as write only, as is done in the example.



```
[illegible]
public class IO {
    [illegible]
    public int inputX[] = {3,1,5,9,10,2,15};
    [illegible]
    public int outputY[] = new int[inputX.length];

    [illegible]
    public static boolean compare(int a, int b) {
        return a > b;
    }
    [illegible]

    [illegible]
    public static void shift(int[] a,int index, boolean forward ) {
        if (index > a.length - 2 || index < 0)      [illegible]
            return;
        if (forward && index < a.length - 2) {      [illegible]
            int temp = a[index];        [illegible]
            a[index] = a[index + 1];    [illegible]
            a[index + 1] = temp;        [illegible]
        } else if (!forward && index > 1) {     [illegible]
            int temp = a[index];
            a[index] = a[index - 1];    [illegible]
            a[index - 1] = temp;        [illegible]
        }
    }

    [illegible]
    public static double rate(IO io) {
        double score = io.inputX.length;    [illegible]
        for (int i = 0; i < io.inputX.length; i++) {
            for (int j = 0; j < io.outputY.length; j++) {
                if (io.outputY[j] == io.inputX[i]) {
                    score -= 1/(i-j + 1);   [illegible]
                }
            }
        }
        return score;
    }
}
```

Figure 2. Example code of IO object

From this example you can see that the IO class contains three method that are visible to the programs. The required rate method, and two extra methods: compare and shift. The results of this solution turns out to be essentially useless which is further discussed in Section 6. But here the important take away is the syntax of commands and their ability to allow code to be inserted for use in genetic operations.

5 Design

Programs are built out of Java objects first. Each Java object is representative of some term in the grammar with the exception of definition objects such as variable, method, and class definition. These objects represent the idea of instantiating a variable that can be used in other parts of the program rather than direct translation to the grammar. Program generation is very simple since it derives terms in the grammar from probabilities where all probability parameters are provided in the UI. Program creation at its core is composed of a class, a set of variables and the main method as seen in Figure 3.

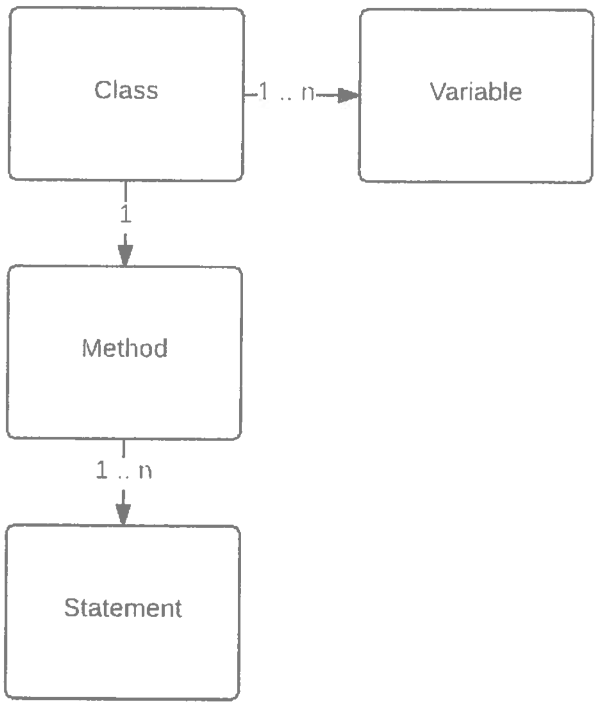


Figure 3. Generated program structure

The method is a set of statements. Since it is grammar based, each statement derives into either a method call, assignment, loop, or conditional as seen in figure 4. The method is simply a method call which searches in the library classes and I/O class defined by the user and picks based on a probability of the weight defined for that class and method. Method selection like this applies to a statement deriving into method as well as an expression deriving into method. The "expression" is the most powerful part of the grammar, as it is used heavily and allows the creation of very complex code. Its derivations are still very simple but provide a lot of capability seen in figure 5.

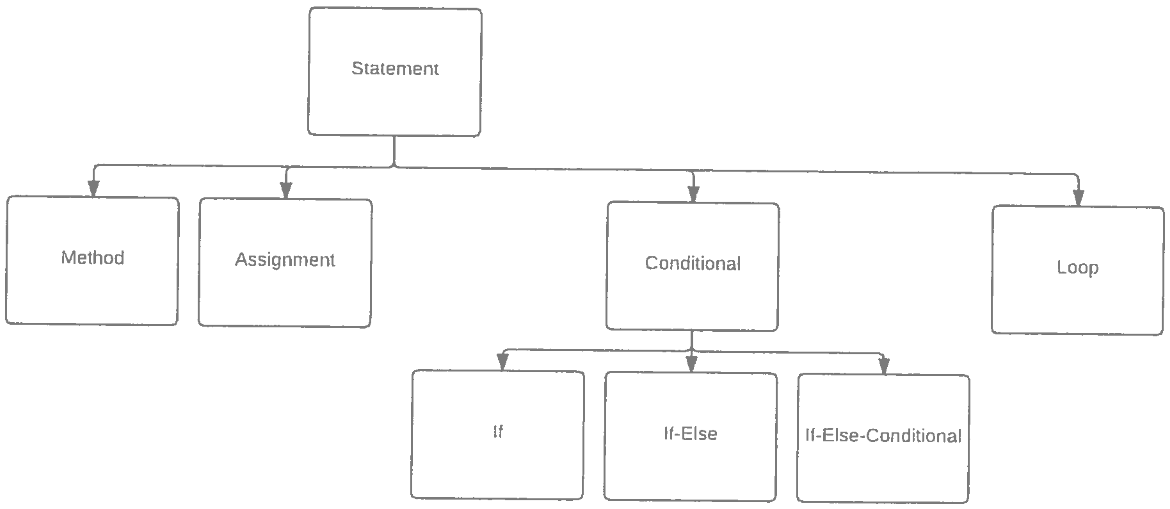


Figure 4. Statement Derivation flow chart

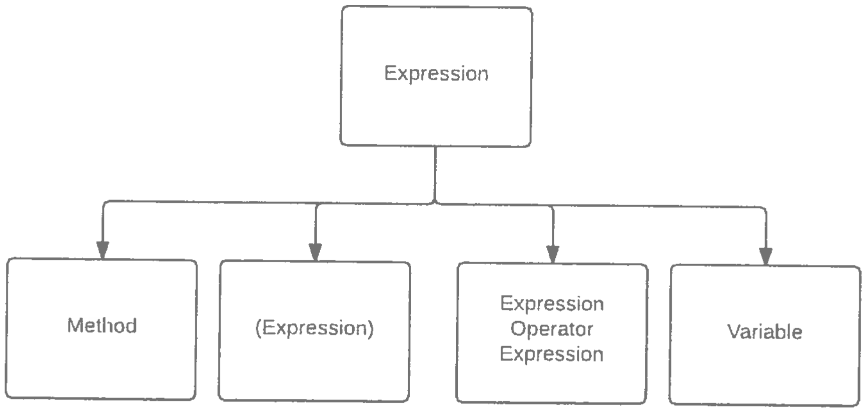


Figure 5. Expression Derivations

With an understanding of the grammar that is used to ensure that a program generated follows the syntax rules of the java language we can move on to what I call *selecting to fit*.

When programs are generated, mutated, or crossed, they are left with the dilemma of finding something to fit that is legal for compilation. The process GBGP uses involves searching recursively up the tree of the virtual program creating a list of all variables or methods that are legal to fit in the slot. Originally programs were allowed to generate their own methods other than the main method. The repercussion was infinite loops. Solving this problem required checking to make sure that no method called another method that called another method and so on. As a fallback method generation is limited to one method per program, the main method. Now when a program needs to find the method it compiles a list of all available methods from all the library classes provided and IO class. The same is true for variables except class variables are also included. Method parameters are checked to make sure that at least one variable of the type exist to fill each argument of different types in the method call. Once a list has been created the weights are added first by class then by method/variable and a random number is selected that is between the range of zero and the sum of all the weights and the spot is now legally filled. In this way all method calls are legitimate and all accesses to variables are also legitimate. By reducing the number of available variables you force those variables to be used more often and the same goes with methods.

6 Results of Test Case

In my experiments I attempted to solve sorting an integer array with unsorted input. In less than a few generations a solution was found in which the programs would fill the output with the correct numbers regardless of the input since the desired output was always the same. In order to prevent the program from just finding the right numbers to put in the right places, the length of the array as well as the numbers generated as input were randomized for each individual program. With randomized input and several days of runs little to no progress was made. When progress was made in terms of score change it was quickly lost within the next few generations as the input changed again. Over and over this same process occurred where a program would experience an advance in scores and in the next generation lose the fitness, being killed off.

In my case the fitness function that defined the problem being solved would not slowly bring in arbitrary program toward success as any success gained would be lost due to the random input changing. Only through sheer luck would a program be able to solve the problem (which never occurred). This is a problem with the way this particular Fitness function impacts program acceptance. By trying to force GBGP to find a generic algorithm to solve any unsorted array, it had no way of improving on past success, since every success was immediately lost in the next generation. A potential solution to this problem includes increasing likelihoods of useful code in Library functions to give early programs a better starting location

7 Conclusions and Further Improvements

The inner workings of the grammar and selection of visible structures such as class, method, and variable were thoroughly explained allowing more research to take place. There was a full discussion of how to interact with the software including the commands it parses in the Java files. In my test case where I tried to generate a generic Integer array sorting algorithm,.no solution was found and based on the nature of the Fitness function, which requires more changes and research. The fitness landscape is paramount to any Genetic Programming problem and remained the main drawback to my tests. It only mattered how well and to what resolution you can "grade" the fitness of programs. Due to the nature of random input with random array size, it was unable to make quantifiable improvements without a better fitness equation.

The project still has more to be done. In the future I would like to give programs the ability to search into non-static sub-objects of library classes. This could add a much larger feature set to the user and allow much more complex data structures for programs to manipulate. Furthermore, data checking could be more robust to make sure the user does not enter incorrect values which can at times cause fatal and unwarned failures. Lastly, more features could be added such as uploading populations to databases as well as saving and loading configuration and settings files.

8 References

1. M. Mitchell, An Introduction to Genetic Algorithms (MIT Press, 1996).
2. M. Mitchell, J. P. Crutchfield, and P. T. Hraber, "Evolving Cellular Automata to Perform Computations: Mechanisms and Impediments", Physica D, 75 (1994), 361-391.

9 Additional Bibliography

1. U.-M. O'Reilly and F. Oppacher, "A Comparative Analysis of Genetic Programming", in P. J. Angeline and K. E. Kinnear, eds., Advances in Genetic Programming 2 (MIT Press, 1996).
2. M. J. Keith and M. C. Martin, "Genetic Programming in C++: Implementation Issues", in K. E. Kinnear, ed., Advances in Genetic Programming (MIT Press, 1994).
3. Goldberg, D. (1989) Genetic Algorithms in Search, Optimization, and Machine Learning, Reading MA: Addison Wesley
4. Koza, J.R. (1992) Genetic Programming. The MIT Press.
5. Montana, D.J. (1993) Strongly Typed Genetic Programming, BBN Technical Report #7866, May 1993.

Appendix

Below is a listing of the packages and classes in this project. Code is available upon request.

Interface
Package of files describing the User Interface and handling interface functionality.

```
12/09/2015  10:54 AM          152,299 FXMLDocument.fxml
12/09/2015  10:47 AM            9,318 FXMLDocumentController.java
12/09/2015  10:21 AM            1,163 GP_UI.java
12/09/2015  10:48 AM            6,720 Loop.java
```

Definitions
Classes that represent the declaration of Classes, Methods, and Variables.

```
12/10/2015  09:08 AM           10,373 ClassDefinition.java
12/10/2015  09:00 AM            6,168 DefinitionStack.java
12/10/2015  08:09 AM           14,292 MethodDefinition.java
12/09/2015  11:41 PM           11,950 VariableDefinition.java
```

Grammar
Each Class represents a node in the Grammar tree. Classes also include functionality for genetic operations.

```
12/10/2015  07:53 AM            4,530 Assignment.java
11/02/2015  05:39 PM            5,940 Conditional.java
12/10/2015  08:16 AM           27,728 Derivable.java
12/10/2015  07:09 AM           10,802 Expression.java
12/10/2015  07:21 AM            5,985 Loop.java
12/10/2015  08:56 AM            4,013 Method.java
12/10/2015  08:07 AM            7,005 Statement.java
```

Terminals
Classes that represents the terminals of the Grammar and their functionality.

```
12/04/2015  10:27 PM            2,763 ComparisonOperator.java
12/09/2015  11:29 AM            3,711 Operator.java
11/02/2015  05:34 PM            1,426 Parenthesis.java
12/09/2015  07:34 PM            9,458 Type.java
03/16/2015  12:22 AM              513 UniOperator.java
12/10/2015  08:15 AM           10,145 Variable.java
```

MainPackage
Main package of GBGP that includes main function and connections to the Interface.

```
09/14/2015  06:02 PM              616 ConsoleBuffer.java
12/09/2015  10:20 AM            5,866 DataModel.java
```

```
12/10/2015  08:47 AM               1,809 Main.java
12/04/2015  10:42 PM               7,782 Program.java
11/15/2015  03:04 PM               5,122 Statistics.java


PopFunct
Classes containing algorithms for per program scoring.
12/10/2015  08:40 AM               3,757 Rate.java
12/10/2015  08:39 AM                 496 Result.java


ProgramGeneration
Classes involved in generating and evaluating fitness of programs.
11/15/2015  10:10 PM               7,408 Fitness.java
12/09/2015  10:36 AM              12,339 Generate.java
12/10/2015  08:45 AM              21,002 UserClassBuilder.java


UserDefined
Package for placement of user defines classes.
12/10/2015  10:40 AM                 917 IO.java
12/10/2015  08:40 AM               1,209 Rating.java


Utility
Support classes for common functionality.
12/10/2015  07:24 AM               5,405 Config.java
09/15/2015  12:32 PM               1,289 Log.java
04/07/2015  02:29 PM               2,042 QuickSort.java
11/08/2015  04:12 PM               3,255 RandomNum.java
11/04/2015  12:46 AM               2,203 StringControls.java


Compile
Contains files for in memory Java compilation.
11/15/2015  06:03 PM                 978 CompiledCode.java
11/15/2015  06:03 PM               1,108 DynamicClassLoader.java
11/15/2015  10:07 PM              13,616 DynamicCompile.java
11/15/2015  06:02 PM          1,526 ExtendedStandardJavaFileManager.java
11/15/2015  06:05 PM               1,621 InMemoryJavaCompiler.java
11/15/2015  06:01 PM                 725 SourceCode.java
```