Creatively Named Grammar Guided Genetic Programming System

USER GUIDE

Because the hardest part of writing great software is coming up with a creative name

Part 1

Introduction

Creatively Named Grammar Guided Genetic Programming System, or CNGGGPS is as the name implies, a grammar guided genetic programming system. It has been written to be easy to extend and preconfigured to be easy to start using. An emphasis has been placed on speed of execution and careful memory management to avoid frustrating crashes.

It is assumed throughout this documentation that anyone reading it is already familiar with Genetic Programming, and has worked with other Genetic Programming systems before. It is not assumed that the reader is familiar with the concept of grammar guided genetic programming.

Part 1.2

System Requirements

CNGGGPS was written in ANSI compliant C++, and has been demonstrated to compile using gcc with pedantic error checking. That said, as of yet, CNGGGPS was written and has only been tested under Linux 2.6 with GCC and GNU Make. CNGGGPS is not threaded and will not take advantage of a multi-core or multiple CPU system at this time, though because many steps of the system theoretically could run in parallel, this may change in the future if a good multiplatform threading library is found. Experiment statistics are written to a tab delimited file, so if they are needed then a good spreadsheet capable of reading such a file is crucial. One such program is freely available from www.openoffice.org. Memory requirements vary with program size, if you receive a memory related abort message try increasing the swap space available to the program or using more stringent limits on maximum program depth.

Part 1.3

Grammar Guided Genetic Programming

In traditional genetic programming control over the resulting program is more or less limited to the choice of functions and terminals. The arrangement of these functions and terminals has always been more or less up to chance. This often results in programs which make very little sense, particularly given that one of the goals of genetic programming often is to develop an algorithm to be implemented in some language for future use. Languages are often specified in terms of context-free grammars, which for those unfamiliar may be thought of as a series of rules which define how syntax can be put together. One popular method of defining a context-free grammar is through the use of Backus-Naur Form as seen in the figure below:

```
<start> :=      <expr>
<expr> :=       <term> |
                <preop> <expr> |
                <expr> <op> <expr>
<term> :=       const |
                X |
                Y
<op> :=         + |
                - |
                * |
                /
<preop> :=      sin
                cos
```

The grammar contains a <start> which is our entire algorithm. We can see that <start> := <expr> which means that our algorithm contains an <expr>. Looking at <expr> we see three lines separated by pipe symbols ('|') which means that an expression may be either a <term> a <preop> followed by another <expr> or another <expr> followed by an <op> followed by yet another <expr>. Each of those <expr> may be any one of those three options again. A <term> must be a const, an X, or a Y, which are called grammar terminals, and are symbols which actually appear in our algorithm, similarly a preop must be sin or cos, and likewise op must be one of the symbols following it. Generally any string inside angle brackets refers to another rule, and any string which is not inside angle brackets is directly part of the phenotype. Of course there is much more to context-free grammars than is described in this brief introduction, and if you have not worked with them before it is recommended you it up on one of the many web pages or books on the topic.

Part 2

Running CNGGGPS

When you first extract CNGGGPS you'll notice a mess of source files. We will get into each of them later in the manual, what is important now is to know how to compile and run CNGGGPS.

The first step, to make all of those source files into one executable program, enter your terminal emulator of choice, navigate to the CNGGGPS folder, and type 'make', then hit enter. Assuming you are running a computer with GCC and Gnu Make you should see a number of lines resembling "g++ -pedantic_errors –Wall programname.cpp –o programname.o" The process should take less than a minute depending on the speed of your computer, at the end of which you will have an executable file called cngggps.

To run CNGGGPS type "./cngggps [enter]" This will run the default configuration of CNGGGPS, which as of this writing was to find a program which given input x=2.5 attempts to approximate the number 42 over 5 runs of 50 generations with 100 programs per generation.

At the end of this you will have three more files in this directory: expr.par, expr.bst, and expr.csv, storing the runtime parameters used, the best solution found in each run, and runtime statistics respectively.

Part 2.1

Runtime Parameters

Modifying the size of the population, number of runs, number of generations, and so forth can be done at runtime by passing values into CNGGGPS. For example, to use 100 generations instead of 50 you would run the command "./cngggps –g 100", and to also use 10 runs instead of the default 5 you would run "./cngggps –g 100 –r 10" As of this writing the complete list of parameters which could be specified at run time were:

Parameter	Flag	Default Value
Runs	-r	5
Generations	-g	50
Population Size	-p	100
Crossover Rate	-c	0.8
Mutation Rate	-m	0.2
Mutation/Crossover **A**ttempts	-a	5
M**i**nimum Program Depth	-i	1
Ma**x**imum Program Depth	-x	30
Tournament Size	-t	3
Filename Prefix	-f	expr
Random **S**eed	-s	time

Part 2.2

Making Sense of the Results

Probably the most interesting of the results can be found in the file expr.bst, or <user specified prefix>.bst
In this file you will find the best result of each run written as a LISP program. For those unfamiliar with LISP it will look like a mess of brackets, but with some practice it's not hard to read. Like in math, brackets represent order of operations, whatever is in the inner most brackets is evaluated first, followed by those in the next outer set, and so forth. The expressions themselves are written in prefix notation, so instead of 2 + 2, it would be written + 2 2. Along with the program is the fitness value it obtained, which is defined by the Evaluator method. In the case of the 42 test, this is the difference between the result of the program and the number 42. The number of hits are also stored. Hits is another way of keeping track of how close the program was to the solution, typically at each checkpoint if the error between the program and the expected result is less than a certain amount a hit is scored. In the case of the 42 test a hit is rewarded if the difference between the program result and the number 42 is less than 0.5. When you write your own evaluator functions you will determine your own measure for hits and fitness which will be recorded in this file.

The other important result file for anyone analyzing the system in more detail is the .csv file. This file should be opened in a spread sheet capable of reading tab delimited text files. Each run of the experiment will produce two rows of numbers. The first is the fitness of the best individual in the population over each generation. The second is the average fitness of the population over each generation. By graphing these numbers one can determine if their program reached premature convergence, if selection pressure was not great enough, or if Genetic Programming just is not a good approach for their problem the way they described it.

Part 3

Doing more with CNGGGPS

Of course writing programs that approximate the number 42 is not the only thing that CNGGGPS is capable of doing. To get more out of CNGGGPS you must modify the source code.

Part 3.1

Changing the Objective

To modify the objective, all we have to do is specify another Evaluator. CNGGGPS includes with it another Evaluator called SymRegEvaluator. To use it instead open CNGGGPS.cpp in your favorite editor, find the line that reads "Evaluator *peval = new Evaluator()", and change it to read "Evaluator *peval = new SymRegEvaluator()" and run 'make'. To specify your own evaluator copy SymRegEvaluator to a new file, say MyEvaluator, and modify that line in CNGGGPS.cpp to read "Evaluator *peval = new MyEvaluator()." Parameters, such as file names may be added to the constructor and loaded, any new methods may be added, but the evaluate method must keep the same prototype to be recognized by the system. It is strongly recommended that you more or less stay to the format used in the evaluate method, though of course the fitness calculation can be anything you like as long as it returns a value between 0 and 1 where 1 is a perfect solution. To be run MyEvaluator.o will have to be added to the OBJECTS macro in Makefile, then run 'make' and the new objective is ready to be tested.

Part 3.2

Changing the Grammar

Presuming you have already worked out all the details in your new grammar, and you have it written out in Backus-Naur Form, there are three classes you will be particularly interested in: Grammar, Function, and GrammarRules. The first thing you will want to do is implement all the grammar terminals. All of these classes should extend function and implement minimally how they behave when executed. It is convention to use a capitol F, followed by a minimal identifier for the grammar terminal in naming these classes. For further detail on the implementation of grammar terminals see FConst, FArgument, FAdd, FSub, FMul, and FDiv. Next, look for grammar symbols that will have to pass values down to the grammar terminals. Typically these symbols can be recognized because they consist of a grammar terminal which takes more arguments than

the symbol consists of. For example op := add consists of a grammar terminal which takes two arguments, but op itself has nothing to supply it with. These grammar symbols should also be written as classes which extend function, but the execution will be to add the arguments to the grammar terminal, execute the terminal, and pass back the result. These grammar symbols should be prefixed with G and a minimal identifier. For an example, see GOp. The other symbol worth noting is the start symbol. GStart is the expected start symbol used by CNGGGPS and it is already implemented, though it can be modified if necessary. The last kind of grammar symbol is the non-terminating symbol which is comprised of others, and takes on the value of one of its included elements. GExpr, GTerm, GConj, and GDisj are examples of this type. Notice that depending on the option, hereon called rule (separated by the pipe symbol in BNF) the actual execution of these symbols may change. An if or a case statement inside the execute method which depends on the chosen rule is appropriate. This brings us to GrammarRules.cpp. GrammarRules.cpp specifies the relation between the various grammar symbols. The best way to understand GrammarRules is to dive in and read it, but a brief explanation is provided here. For each of your non-terminating grammar rules you will need to define a section in the if statement where strGrammarID is equal to the one you specified when you were creating the class for it. The first line in each block should be the number of rules, that is the number of sections separated by pipes that there are "iRules=#rules". The next line is the number of rules guaranteed to terminate in a finite determined number of steps. This value is not yet used by CNGGGPS but may be in future versions to improve initial population creation and mutation so it should not skipped. The next line randomly picks one of the rules, you can copy this line straight = START as it is always the same. Following that there should be a case block, where dependent on the chosen rule (with terminating rules listed first) the grammars listed after the := in BNF are popped onto the stack with new GName()/new FName(). Finally, remember to add the newly created classes to the #include statements at the top of GrammarRules. Add GName.o / FName.o to the OBJECTS macro in Makefile, run make, and your new grammar is ready to be tested.

Part 3.3

Final Words on Modifying CNGGGPS

For details on how all of the classes interact with each other read the included API documentation, or the source files them selves for even more information. For the most part class names were chosen that make sense for what they do. For example, to modify how an individual is built, or what metainformation it can store, modify the Individual class, for what information is stored in file and how, modify the Log class. If you are going to add your own custom types, they should be added to the Result class, along with any necessary information regarding type conversion. If you make a mistake and cannot get CNGGGPS to run again you can always extract the original classes and start over.