2871B: the Wold's smallest tree-based symbolic-regression GP system?

Riccardo Poli
Department of Computer Science, University of Essex

Introduction

2871B is a highly optimised GP system, which compiles and runs under Linux, that fully meets the specifications set out in the TinyGP competition of the Genetic and Evolutionary Computation Conference (GECCO) 2004 [1]. The name of the system derives from the fact that the *self-extracting executable occupies 2,871 bytes* (while the source code, in C, is 5,906 bytes and the actual size of the executable after self-extraction is 4540 bytes).

All optimisations in this code have aimed at bringing the executable size (as opposed to the source code size) down, the main purpose being to show that, against popular belief, it is possible to have really tiny and efficient GP systems. It is hoped that this will contribute to convince people in academia and industry that GP is ready for electronic toys, cell phones, microwave ovens, intelligent email attachments, credit card chips, and all sorts of other new and exciting applications.

How does it work?

The system is based on the standard flattened (linear) representation for trees, which effectively corresponds to listing the primitives like in prefix notation but without any brackets. Each primitive occupies one byte. A whole program is simply a vector of characters.

The parameters of the system are as specified in [1]. They are fixed at compile time (which is allowed by the competition's rules) through a series of `#define` statements. The operators used are sub-tree crossover and point mutation as required in [1]. The selection of the crossover points is performed at random with uniform probability. The primitive set and fitness function are as in [1].

The code uses recursion for the creation of the initial population (`grow`), for the identification of the sub-tree rooted at a particular crossover point (`traverse`), for program interpretation (`run`), and for printing programs (`print_indiv`).

A small number of global variables have been used. For example, the variable program is a program counter used during the recursive interpretation of programs, which is automatically incremented every time a primitive is evaluated. Although using global variables is normally considered bad programming practice, this was done purposely, after extensive experimentation, to reduce the executable's size.

Another peculiarity of the code is that it *does not* read command line arguments using the standard `argc` and `argv` parameters. Instead it uses `scanf` to do so. This was done to further reduce the executable size, but, as discussed below, it is totally transparent to the user.

Generally the code is quite standard and should be self-explanatory for anyone who can program in C and has implemented a GP system before. So, very few comments have been provided in the source code.

How come 2871B is so small?

One of the reasons the executable code is so small is that the source code has been streamlined is many ways to make is short, and also to compile into something as short as possible. However, a Linux expert will immediately notice that the size of 2871B is smaller than that of the smallest possible Linux executable that can be produced by `gcc`.[1] How could one pack an entire GP system into such a small space? Two approaches were used.

The first approach was to follow advice available on the Web [2] on how to reduce the size of C executables with a variety of tricks. Some of these tricks required turning to assembler (rather than C) programming and were discarded. Other tricks, however, could easily be implemented by automatically filtering out or replacing some instructions in the assembly code produced by `gcc` when invoked with the flag "`-S`". These include, for example, removing all lines starting with `ident` and changing the last `ret` instruction in the program into a call to the kernel `exit` routine (thereby making it possible to avoid linking the `main`'s initialisation library). The resulting assembly code is then assembled and linked to produce a 4540 byte executable. All of these optimisations are performed automatically in the first part of the script `cc_tiny` which the user should run to compile 2871B.

The second approach was to compress the executable. Direct compression with "`gzip -9`" produces a file of 2620 bytes. This is very good, but, in order to run the program, the user would then have to decompress this file. Also, command line parameters would have to be passed to the system via standard input. To avoid all of this, the script `cc_tiny` makes the compressed file self-extracting and ensures that command line parameters are appropriately passed to the executable. The approach taken to achieve this is similar to that used by the Unix `shar` command to create shell archives. That is, the self-extracting executable for the GP system is actually a shell script containing an executable. The final part of this script is simply the compressed executable mentioned above. The first part of the script, instead, includes instructions to decompress the executable and to run it appropriately feeding into its standard input any command line parameters passed to the script. The shell commands to achieve

[1] When compiling the one-line program `int main(void) { return 1; }` with the flags "`-Os -s`" (to optimise for size and to strip symbols from the executable), `gcc` produces an executable of 2932 bytes!

all of this increase the size of the program by 251 bytes, but this overhead seems fully justified by the improved ease of use of the program.

Running the code

The program comes in a tar file including this document, the source code, the script needed for compiling it, and three datasets of different complexity that can be used for testing it. The code has been developed and tested under Linux with the GNU gcc C compiler (version 2.95.2) on Pentium CPUs. To compile the code, run the script "cc_tiny". This creates a compressed executable called "tiny". The executable must be invoked as follows:

```
tiny SEED DATAFILE
```

where SEED is a integer representing the seed for the random number generator and DATAFILE is a file containing the fitness cases as indicated in the TinyGP competition web page [1].

Alternatively, one can invoke the system with

```
tiny SEED
```

in which case the system assumes that a file "problem.dat" containing the training set is present in the directory from where "tiny" is executed. Finally, one can just use

```
tiny
```

In this case the random number generator is seeded using the time from the system's clock and "problem.dat" is loaded.

Conclusions

2871B has been tested on a number of different Linux PCs showing to be very fast, reliable, portable and memory efficient. The source code, although highly optimised for size and speed, is portable across platforms and compilers. However, some of the more advanced optimisation processes performed in cc_tiny are architecture dependent and would need to be adapted if the code was ported to a different computer architecture. Also, the self-extracting version of 2871B and the cc_tiny script rely on some standard Unix utility programs. A different self-extraction mechanism might have to be used if 2871B was ported to a different operating system.

Bibliography

[1] Genetic and Evolutionary Computation Conference (GECCO) 2004 competitions page, http://www.isgec.org/gecco-2004/competitions.html

[2] A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux, http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html