Abstract—Genetic programming can optimise software, including: evolving test benchmarks, generating hyper-heuristics by searching meta-heuristics, generating communication protocols, composing telephony systems and web services, generating improved hashing and C++ heap managers, redundant programming and even automatic bug fixing. Particularly in embedded real-time or mobile systems, there may be many ways to trade off expenses (such as time, memory, energy, power consumption) vs. functionality. Human programmers cannot try them all. Also the best multi-objective Pareto trade off may change with time, underlying hardware and network connection or user behaviour. It may be GP can automatically suggest different trade offs for each new market. Recent results include substantial speed up by evolving a new version of a program customised for a special case.

Index Terms—GP, genetic programming (GP), Automatic software re-engineering, Bowtie2 [21], multiple objective exploration, search based software engineering (SBSE), GPGPU.

I. INTRODUCTION

Genetic programming [Koza, 1992; Poli et al., 2008] has been very widely applied. For example in

- modelling [Kordon, 2010],
- prediction [Langdon and Barrett, 2004; Podgornik et al., 2011; Kovacic and Sarler, 2014],
- classification [Freitas, 1997],
- design [Lohn and Hornby, 2006],
- creating art [Reynolds, 2011; Jacob, 2001; Langdon, 2004; Romero et al., 2013],
- the generation of hyper-heuristics [Burke et al., 2013],
- configuring intelligent telephony networks [Martin, 2000] and
- Web mashups [Rodriguez-Mier et al., 2010],
- Hashing [Hussain and Malliaris, 2000],
- Heap managers [Risco-Martin et al., 2014],
- multiplicity computing [Feldt, 1998; Cadar et al., 2010] and
even to create benchmarks which demonstrate the relative strengths and weaknesses of optimisers [Langdon and Poli, 2005].

Recently genetic programming has been applied to the production of programs itself, however so far relatively small programs have been evolved. Nonetheless GP has had some great successes when applied to existing programs. Perhaps the best known work is that on automatic bug fixing [Arcuri and Yao, 2008]. Particularly the [Humie] award winning work of Westley Weimer and Stephanie Forrest [Forrest et al., 2009]. This has received multiple awards and best paper prizes [Weimer et al., 2009; Weimer et al., 2010]. GP has been used repeatedly to automatically correct most (but not all) real bugs in real programs [Le Goues et al., 2012]. Weimer and Le Goues have now shown GP based automatic software correction to be effective on several millions of lines of C++ programs. Their GenProg [Le Goues et al., 2012b] approach is based on re-using existing human written code to patch the source code defect. A recent study [Barr et al., 2014] showed many updates to Java code made by people are not totally novel but could have been made by re-using existing code. Indeed, baring layout and identifier names, most human written code of up to five lines has already been written somewhere by someone else [Gabel and Su, 2010].

Once GP has been used to do the impossible (i.e. automatic bug fixing) it was improved [Kessentini et al., 2011] and people felt brave enough to try other techniques, e.g. [Nguyen et al., 2013].

Andrea Arcuri was again in at the start of inspirational work on showing GP can create real code from scratch. Although the programs remain small, David White, he and John Clark [White et al., 2011] evolved programs to accomplish real tasks such as creating pseudo random numbers for ultra tiny computers where they showed a trade off between “randomness” and energy consumption. Such tradeoffs are vital if RFID based nano-computing devices are to be programmed.

Fig. 1. The original code is instrumented to record the inputs and outputs (blue arrows) of the target function (red) every time it is called. These become the fitness function and test suite for the automatically evolved replacement module running on novel hardware (actually GPUs). The CUDA code generated by GP is functionally identical to the C code inside gzip [Langdon and Harman, 2010].
II. AUTO PORTING FUNCTIONALITY

The Unix compression utility gzip was written in C in the days of Digital Equipment Corp.’s mini-computers. It is largely unchanged. It contains a procedure (of about two pages of code) which is so computationally intensive that it has been re-written in assembler for the Intel 86X architecture (i.e. Linux). The original C version of gzip has been retained and is distributed as part of Software-artifact Infrastructure Repository [sir.unl.edu [Hutchins et al., 1994]. SIR also contains a test suite for gzip. In Genetic Improvement, as with Le Goues’ bug-fixing work, we start with an existing program and a small number of test cases. In the case of the gzip function, we showed genetic programming could evolve a parallel implementation for an architecture not even dreamt of when the original program was written [Langdon and Harman, 2010]. Whereas Le Goues uses the original program’s AST (Abstract Syntax Tree) to ensure that many of the mutated programs produced by GP compile, we have used a BNF grammar. In the case of [Langdon and Harman, 2010] the grammar was derived from generic code written by the manufacture of the parallel hardware. Note that it had nothing special to do with gzip. The original function in gzip was instrumented to record its inputs and its outputs each time it was called (see Figure 1). When gzip was run on the SIR test suite, this generated more than a million test cases, however only a few thousand were used by the GP. Essentially GP was told to create parallel code from the BNF grammar which when given a small number of example inputs returned the same answers. The resulting parallel code is functionally the same as the old gzip code.

III. BOWTIE2$^2$GP IMPROVING 50 000 LINES OF C++

As Figure 2 shows, genetic programming produces populations of programs which may have different abilities on different scales. While Figure 2 shows speed versus quality, other tradeoffs have been investigated. For example it may be impossible to simultaneously minimise execution time, memory footprint and energy consumption. Yet, conventionally human written programs choose one trade-off between multiple objectives and it becomes infeasible to operate the program with another trade-off. For example, consider approximate string matching.

Finding the best match between (noisy) strings is the life blood of Bioinformatics. Huge amounts of people’s time and computing resources are devoted every day to matching protein amino acid sequences against databases of known proteins from all forms of life. The acknowledge gold standard is the BLAST program [Altschul et al., 1997] which incorporate heuristics of known evolutionary rates of change. It is available via the web and can lookup a protein in every species which has been sequenced in a few minutes. Even before the sequencing of the human genome, the volume of DNA sequences was exploding exploding at a rate like Moore’s Law [Moore, 1965]. With modern NextGen sequencing machines throwing out 100s of millions (even billions) of (albeit very noisy) DNA base-pair sequences, there is no way that BLAST can be used to process this volume of data. This has lead to human written look up tools for matching NextGen sequences against the human genome. Wikipedia lists more than 140 programs (written by some of the brightest people on the planet) which do some form of Bioinformatics string matching.

The authors of all this software are in a quandary. For their code to be useful the authors have to chose a point in the space of tradeoffs between speed, machine resources, quality of solution and functionality, which will: 1) be important to the Bioinformatics community and 2) not be immediately dominated by other programs. In practise they have to choose a target point when they start as once basic design choices (e.g. target data sources and computer resources) have been made, few people or even research teams have the resources to discard what they have written and start totally from scratch. Potentially genetic programming offers them a way of exploring this space of tradeoffs [Harman et al., 2012]. GP can produce many programs across the trade-off space and so can potentially say “look here is a trade-off which you had not considered”. This could be very useful to the human, even if they refuse to accept machine generated code and insist on coding the solution themselves.

We have made a start by showing GP can transform human written DNA sequence matching code, moving it from one tradeoff point to another. The overall frame work is shown in Figure 3 In our example, the new program is specialised to a particular data source and sequence problem for which it is on average more than 70 times faster. Indeed on this particular problem, we were fortunate that not only is the variant faster but indeed it gives a slight quality improvement on average [Langdon and Harman, 1].

![Fig. 2. Example of automatically generated Pareto tradeoff front. Genetic programming used to improve 2D Stereo Camera code [Stam, 2008] for modern nVidia GPU [Langdon and Harman, 2014]. Left (above 0) many programs are faster than the original code written by nVidia’s image processing expert (human) and give exactly the same answers. Many other automatically generated programs are also faster but give different answers. Some (cf. dotted blue line) are faster than the best zero error program.](image-url)
IV. IMPROVING PARALLEL PROCESSING CUDA CODE WRITTEN BY EXPERTS

In other examples we returned to computer graphics hardware. In the first GP was able to automatically update for today’s GPUs software written specifically by nVidia’s image processing expert to show off the early generations of their graphics cards [Stam, 2008]. Genetic improvement lead (on the most powerful modern Tesla GPU, see Figure 4) to almost a seven fold speed up relative to the original code on the same GPU [Langdon and Harman, 2014]. In another example a combination of manual and automated changes to production 3D medical image processing code lead to the creation of a version of a performance critical kernel which (on a Tesla K20c) is more than 2000 times faster than the production code running on an 2.67GHz CPU [Langdon et al., 2014].

V. MINISAT: IMPROVING BOOLEAN SATISFIABILITY CODE WRITTEN BY EXPERTS

The basic GI technique has also been used to create an improved version of C++ code from multiple versions of a program written by different authors. Boolean Satisfiability (SAT) is a problem which appears often. MiniSAT is a popular SAT solver. The satisfiability community has advanced rapidly since the turn of the century. This has been due in part to a series of competitions. These include the “MiniSAT hack track”, which is specifically designed to encourage humans to make small changes to the MiniSAT code. The new code is available after each competition. MiniSAT and a number of human variants were given to GI and it was asked to evolve a new variant specifically designed to work better on a software engineering problem (interaction testing) [Petke et al., 2014b]. At GECCO 2014 it received a Human Competitive award (HUMIE) [Petke et al., 2014a].

Fig. 3. Genetic Improvement cycle extends traditional GA/GP evolutionary cycle. GI starts with human written code (left, colour). It is automatically converting into a BNF grammar, which is used to create the initial generation. The GP evolves small patches, which are converted back into C++ code by effectively reversing the grammar. Finally the fitness of the mutated code is found by compiling and running it and then comparing its answers with those given by the original code on the same test cases. A small number of test cases (e.g. 5) are randomly chosen each generation from a much large stock, which may have been supplied with original code as its regression test suite.

Fig. 4. Tesla K20c contains 13 SMX multiprocessors, a PCI interface to the host PC, thread handling logic and 4800 MBytes of on board memory. Each SMX contains 192 stream processors (only one SMX shown).
VI. BABEL PIDGIN: CREATING AND INCORPORATING NEW FUNCTIONALITY

Another prize winning genetic programming based technique has recently been demonstrated to be able to extend the functionality of existing code [Harman et al., 2014]. GP, including human hints, was able to evolve new functionality externally and then search based techniques [Harman, 2011] were used to graft the new code into an existing program (pidgin) of more than 200,000 lines of C++.

VII. CONCLUSION: SOFTWARE IS NOT FRAGILE

There has been a tremendous fear of making random changes to programs. It was felt that any unthinking change must damage the software. Indeed a single random change may do so. However software engineers have long been familiar with mutation testing [DeMillo and Offutt, 1991; Langdon et al., 2010], in which bugs are deliberately seeded into programs in order to gauge the effectiveness of test methods at finding bugs. One of the lessons of mutation testing has been that there are some “stubborn” mutants which are very hard to detect by testing [Yao et al., 2014]. In other words some mechanically introduced changes to the code have little effect on its operation. That is, not all changes damage the code. Figure 5 shows an experiment (from Section 3) of the range of changes made by genetic programming in the initial generation (i.e. before selection) were done thousands of times. For each mutation the program (Bowtie2) was run and the difference made by the mutation was recorded. Figure 5 plots both the change in solution quality and speed for each run. Notice (left hand side) some changes do indeed cause Bowtie2 to fail or generate junk results. However Figure 5 is dominated by a large spike at the origin corresponding to mutations which changes neither speed nor solution quality. There are even some mutants which produce slightly better answers.

[Schulte et al., 2014] recently investigated the software mutational robustness of twenty two diverse programs and found consistently about a third of mutations do not cause the program to fail under testing. Whilst most investigations have mutated source code, similar robustness has been reported at assembler code end even binaries [Schulte et al., 2013]. So yes, a single random change may break code, but if you are prepared to create a population of mutated programs, some programs in it may be broken but others may run ok. Evolutionary techniques select the better ones, the fitter ones, and create further changes to them. Using survival of the fitness [Darwin, 1859] over time the population can evolve to contain highly fit programs.

Genetic programming aims to tackle, what John Koza called the “S word” in AI, the Scaling problem. Recently there has been considerable progress not so much by evolving complete system from scratch but either by evolving modest code to glue large systems together from existing components or by evolving small changes to existing programs which make large improvements to them.

REFERENCES

[Altschul et al., 1997] Stephen F. Altschul, Thomas L. Madden, Alejan-

[Arcuri and Yao, 2008] Andrea Arcuri and Xin Yao. A novel co-


