UCL DEPARTMENT OF
COMPUTER SCIENCE

Research Note

RN/12/09

Genetically Improving 50000 Lines of C++

19 September 2012

William B. Langdon and Mark Harman

Abstract

There has been much recent interest in genetic improvement of programs. However, genetic improvement has yet to scale beyond toy laboratory programs. We seek to overcome this scalability barrier. We evolved a widely-used and highly complex 50 000 line system, seeking improved versions that are faster than the original, yet at least as good semantically. Our approach found a version that is 70 times faster (on average) and is also a small semantic improvement on the original.

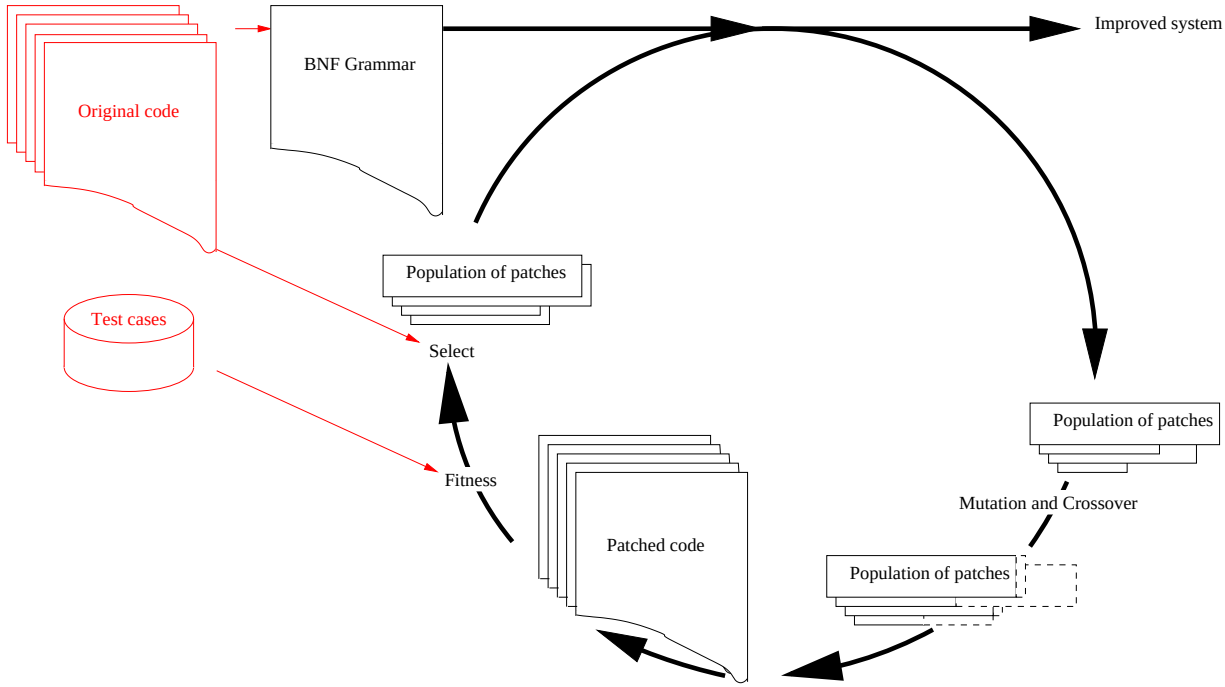

Figure 1: Major components of GISMOE approach. Left system to be improved and its test suite. Right genetic programming optimises patches which originate from a grammar that describes the original system line by line. Each generation mutation and crossover create new patches. Each patch's fitness is evaluated by applying it to the grammar and then reversing the grammar to get a patched version of the system. Each patched system is tested on a randomised subset of the test suite and its answers and resource consumption compared to that of the original system. Patches responsible for better systems procreate into the next generation.

Keywords: automatic software re-engineering, SBSE, genetic programming

1 Introduction

Genetic improvement [1; 2; 3; 4; 5] is the process of automatically improving a system's behaviour with respect to some desired criteria using genetic programming [6]. Starting from an original system, typically written by a human software developer or team, genetic improvement seeks to evolve the system into a new version that is improved with respect to the given criteria.

The criteria for improvement have usually been non-functional properties of the system, such as execution time, power consumption, though many others are possible [1; 2; 5; 7]. The functional properties of the evolved system are typically required to mimic as faithfully as possible those of the original system from which the evolution process started.

In order to check that the original's semantics are not disturbed by the genetic improvement process the genetic improvement process relies on a set of test cases, obtained from running the original system. Where the system has an automated oracle able to check an output's validity, this can also be used to test the functional behaviour of the genetically improved program. (See Figure 1.)

Genetic improvement has many potential applications. An existing program can be ported from one platform and language to another [3], thereby helping to manage software multiplicity [8]. Genetic improvement also allows programs to be automatically sped up [5] or to consume less power [1], while still performing the useful functions offered by the original.

The goal of genetic improvement research is to automate as much of the improvement process as possible,

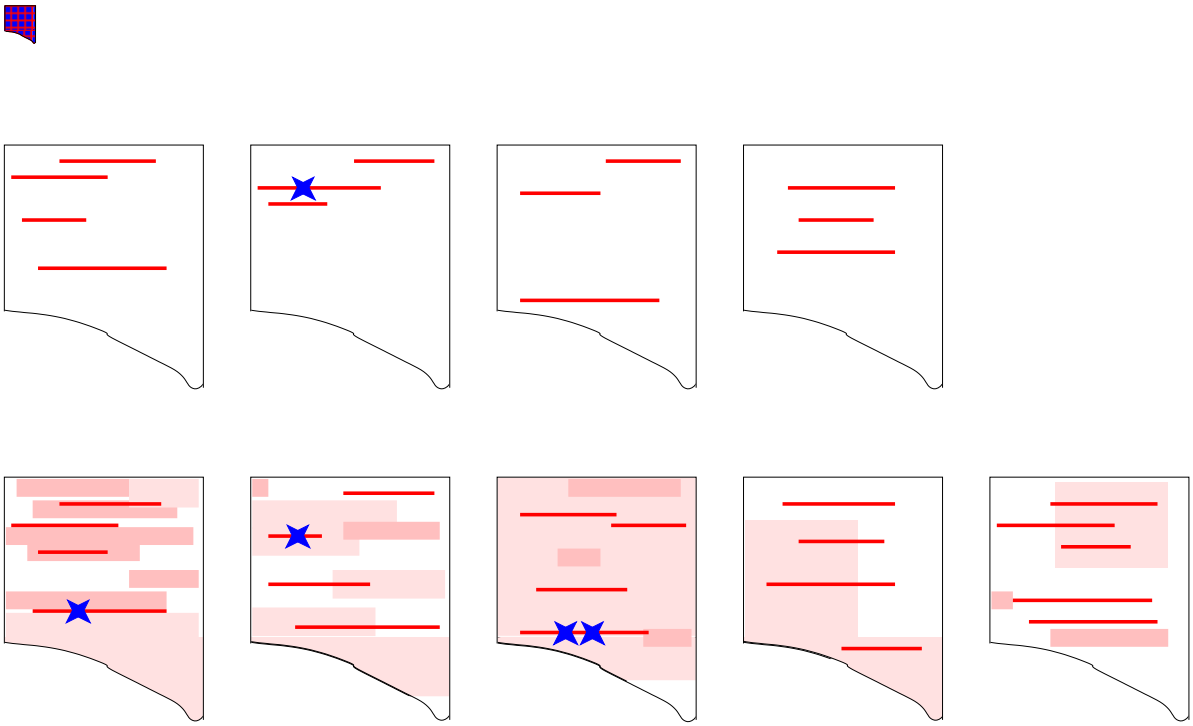

Figure 2: a) Initial whole program (shaded) approaches update code (dark shading) which may be throughout single source file. b) Bug fixing. Genetic programming is directed to parts of code needing fixing (shaded) and the bugfix (star) is small. c) Evolution is directed to used and heavily used code (light shaded, shaded, heavily shaded) several lines of code may be updated (stars).

so that new implementations can be discovered by an evolutionary process, rather than being hand-crafted by human programmers, in the currently familiar (yet time-consuming, tedious and expensive) method of hand coding.

Ultimately, genetic improvement looks forward to a world in which our successors regard human *programmers* as a 'quaint anachronism of the past' in much the same way that we now regard the human *computers* of our nineteenth century forbearers. More details can be found in the recent 'survey and manifesto' of this approach to genetic programming for software engineering [2].

Genetic improvement is one example of Search Based Software Engineering (SBSE) [9; 10; 11]. SBSE seeks to apply techniques from computational search to software engineering problems. There has been much recent progress in SBSE research [12; 13; 15; 14], with a great deal of interest in evolutionary approaches to software engineering in general [16] and genetic programming approaches in particular [11]. Genetic programming has proved useful in a range of software engineering problems including predictive modelling [17], testing [18; 19], software diversity [20] and bug fixing [21; 22; 23].

Like many of these applications of genetic programming to software engineering, genetic improvement has great potential for impact on the research and practitioner communities. It provides a way to automate one of the most expensive and time-consuming aspects of the software engineering process: the production of the code itself. However, achieving genetic improvement for real world programs presents many challenges. The size and complexity of the programs to be evolved are considerably more demanding than those previously attempted using genetic programming.

We report the results of applying genetic improvement to a real-world system. Our approach reduces the search space for genetic improvement and manages the scalability of testing for functional and non-functional properties. We report the results of applying it to Bowtie2, a widely-used DNA sequencing

system, consisting of 50 000 lines of C++ code for which we evolved 20 000 LoC (excluding headers and conditionally compiled debug code). In fact by also excluding code that is not executed we focus on the search on 2744 LoC. We used test cases from the 1000 genome project, backed up by the Smith-Waterman score (as an automated test oracle).

Our primary finding is that genetic improvement can find new evolved versions of Bowtie2 that are, on average, 70 times faster than the original (and semantically slightly improved) when applied to DNA sequences from the 1000 genome project. This is an important finding because genetic improvement has previously only been applied to laboratory programs (of up to about 100 lines of code). This previous work demonstrated proof of concept, but not the practical scalability required for realistic program improvements on real-world systems. Bridging this divide entails catering for all of the complexity and scale of real world systems.

Genetic Programming (GP) has been used to fix bugs in real world programs of a similar scale to the one considered here. However, ours is the first work to introduce an approach to genetic improvement that has been applied to a real-world system. Though both genetic improvement and bug fixing have used GP as an underlying technique, the two applications of GP are different and pose different technical challenges as a result.

The difference in previous approaches to GP for software engineering is illustrated in Figure 2. The figure consists of three lines. In each line, the icons denote the files that comprise a program or system. The first line depicts a program consisting of a single file containing a single procedure. In the second and third lines depict entire systems (comprised of several files, each of which may have many procedures and functions).

Previous work on genetic improvement [1; 3; 4; 5] is depicted in the first line. This applies genetic operators to the entire program to improve it with respect to non-functional properties while maintaining [3] (or gracefully reducing [5]) functional properties. Initial, foundational, proof-of-concept work on GP for bug fixing [24; 25; 26] also applied genetic operators to small laboratory programs and so this initial work is also depicted in the first line of Figure 2.

Subsequent work on bug fixing [21; 22; 23] extended this initial work to whole systems, using fault localisation techniques to identify the parts of the system that might required changing. This demonstration of scalability of bug fixing is depicted in line two of Figure 2; though the whole system is executed, the GP search is concentrated on only that small part to which the bug is localised. This localisation is depicted by the shaded lines, of which a specific location (depicted by the star) is actually modified by the genetic operators to fix the bug.

The problem addressed in this paper is to extend genetic improvement from proof-of-concept to real world applicability. In order to do this we apply GP to multiple points in a system (of multiple files), guided by a sensitivity analysis that identifies parts of the system that are most relevant to the non-functional property of interest (in our case, the most frequently executed code). This modus of operation is depicted in line three of Figure 2; the whole system is executed and the GP search is directed to multiple parts of the system (shaded). Although multiple parts of the system may be modified by GP to improve the performance of the overall system the final number of lines changed may be modest (stars).

The primary contribution of this paper is that it demonstrates that genetic improvement, previously only applied to laboratory programs, can scale to real world systems of the order of tens of thousands of lines of code. We show that genetic improvement can produce dramatically faster versions of the program, for well defined and useful subsets of the input domain and without loss of semantics (indeed, even with some modest improvement in semantics). In order to achieve this overall goal the paper introduces a number of techniques and approaches that may prove to be useful contributions for the future development of genetic improvement:

1. **Semantic Improvement**: We show that the presence of an automated test oracle opens up the possibility that genetic improvement might improve, not only non-functional properties but also a system's

semantics (rather than merely seeking to maintain faithful semantics (using testing).

2. **Sensitivity Analysis**: We introduce a pre-analysis phase that tests the sensitivity of the program to the non-functional property we seek to improve (in this case execution time). We show how our grammar-based GP approach suits this sensitivity analysis, because it can identify the parts of the system to be evolved and those that are to remain untouched. This reduces the search space that genetic improvement has to consider.

3. **Output Bins**: We introduce an approach that caters for test case difficulty by binning test cases according to the amount of output they produce. This allows us to sample uniformly over the quantity of output produced, rather than merely sampling offer the happenstance of test data availability. Our Output–Bin approach is also used to assess the algorithmic complexity of the non-functional properties as they are empirically observed at each line of the program.

4. **Operator Choice**: We demonstrate that the simple genetic operators like those used in automated bug fixing work can also apply to genetic improvement, thereby further reducing the search space required by genetic improvement.

5. **Grammatical Representation**: We introduce an adapted form of Langdon's Grammar-based representation to help to guide the GP search.

6. **Local Search**: We show how a local search post processing phase can be used to address the potential (observed widely in many GP applications) for the solutions to become bloated. The local optimisation is sufficient to reduce the modification required to a surprisingly small (and thus manageable) set of cut–and–paste operations.

Section 2 presents an overview of the real-world system, Bowtie2, to which we apply genetic improvement, motivating our selection of this system. Section 3 outlines the GISMOE approach we use for genetic improvement, which is applied to Bowtie2 in Section 4 using DNA data from the 1000 genome project as test cases and the Smith-Waterman algorithm as a test oracle. The improvements we find using our approach are described (and investigated on held-out data sets) in Section 5. Section 6 describes related work and the relationship of our contributions to this context of previous work, Section 7 briefly discusses limitations, while Section 8 concludes.

2 The Real World System Genetically Improved: Bowtie2

Since the primary difference between our work and previous work lies in the scale of the system, we sought to genetically improve, it is important to explain the size and complexity of the task we set out to accomplish. Also, any attempt to apply genetic improvement techniques to real world programs will have to address a number of issues that are pertinent to the application in hand, such as its domain-specific characteristics and the means of determining semantic correctness of outputs. For the tiny laboratory programs that have previously been genetically improved in the literature, this is not necessary because these laboratory programs operate on simple vectors of integers. For a real world program such simple input domains are seldom sufficient. Therefore, this section presents background information about the application, Bowtie2, that we set out to genetically improve.

The exponential growth in DNA data and the ever-changing analysis requirements for computer systems that operate on this data have lead to many systems being created for DNA matching and analysis. Naturally, since genetic improvement techniques are in their infancy, these systems have been entirely hand-coded. Each implements techniques for searching and processing DNA data. For example, in August 2012, Wikipedia alone listed more than 140 Bioinformatics tools that perform some aspect of sequence analysis either on protein databases or DNA sequences. The production of so many tools requires a large amount of human effort, making this a natural target application domain to evaluate an automated genetic improvement approach such as that advocated in this paper.

One of the most popular tools for querying next generation DNA sequences is the Bowtie system [27]. Bowtie can process more than 25 million short DNA sequences per hour per CPU core [27]. It is written in C and is open source. All of these properties have contributed to its popularity. However its speed comes at the cost of some lost of functionality. In particular, it cannot deal with mutations which either insert or remove DNA bases (known as indels). Other tools (such as BLAST [28]) can handle indels.

Although derived from the Bowtie system, the Bowtie2 system [29] was written from scratch. It consists of 50 000 lines of C++ spread over 50 main system modules and 67 header files (plus documentation, scripts and support modules). It is also open source (available from sourceforge). The Bowtie2 development effort is an attempt to recover the functionality of tools like BLAST which is missing from Bowtie, while retaining the speed of the original Bowtie system. However, though Bowtie2 is much faster than BLAST, it is, nevertheless, still much slower than Bowtie.

Due to its speed, Bowtie is one of the tools used in processing DNA sequences generated by next-generation DNA sequencing machines, such as Illumina's Genome Analyzer II Solexa scanners. The 1000 genome project uses Solexa and other scanners to generate vast numbers of DNA sequences. These data are publicly available and can be obtained via FTP[1]. For experimenting with genetic improvement applied to real world programs it is important to have a realistic pool of test data.

The properties of Bowtie2 and the test data we use make this an ideal target for the application and evaluation of our approach. More specifically:

1. The code is available, supporting full replication by subsequent authors.
2. There are realistic and widely-used sets of test cases available.
3. Test cases come from a non-trivial application (the on going analysis of human genetic variation, particularly with regard to disease factors and medical applications) that generate much interest. They are therefore more likely to involve real-world challenges than the artificially constructed code examples used so far.
4. Bowtie2 is much larger (being at least two orders of magnitude larger) than previously studied systems in work on genetic improvement.
5. Bowtie2 it is not merely larger, but also more complex than any previously studied systems for which results are reported for genetic improvement. Its scale crosses complexity boundaries not previously encountered, because it includes many software engineering and programming features that any practical genetic improvement approach would need to address, yet which have been left unaddressed in previous work. Such features include modularisation (functions and procedures), distinctions between main and support code (libraries, test harnesses etc.), separation into files, use of complex data structures, file access, pre-processing and macro calls.

Previous work on genetic improvement has demonstrated the theoretical possibility of using genetic programming to improve a program's non-functional properties and this has been very important. However, it is insufficient on its own. The development of techniques that can apply to programs like Bowtie2 can provide evidence that genetic improvement can be applied to programs used in demanding, complex, real-world applications.

3 The GISMOE Approach

This section outlines our GISMOE (Genetic Improvement of Software for Multiple Objective Exploration) framework [30; 2] and how its principle components are instantiated to achieve genetic improvement for the Bowtie2 System.

[1] ftp.1000genomes.ebi.ac.uk

3.1 O–Bins: Output Bins for Test Cases

We introduce a concept of Output Bins (O–Bins), with which we partition the available test cases. Our motivation underlying this binning process, is that testing practitioners intuitively have a concept of the difficulty of a tests case and this is typically correlated to the amount of output that the test case causes the program to create; tests that cause the generation of a lot of output are, in some sense, *more difficult* than those that cause comparatively little output to be produced.

O–Bins play a role in the assessment of both functional and non-functional properties of the code. For the functional properties, we use F different O–bins. Test cases are sampled uniformly from these F O–bins (rather than uniformly over all available test cases). The binning process ensures that we sample demanding test cases for fitness evaluation as well as less demanding ones, even though we only sample $n \times F$ test cases for fitness at each new generation. For the assessment of the non-functional properties, we also use O–bins to ensure that tests are sampled uniformly over their perceived difficulty (rather than merely over their availability). This use of O–bins is explained in more detail in Sections 3.3 and 3.4.

3.2 Determining Functional Correctness

The functional properties of a system are typically assessed by GP using a test-based approach [31; 32]. However, testing suffers from the oracle problem [33; 34]. That is, we need an automated oracle that will determine whether a given output observed is correct. Fortunately, one of the advantages of genetic improvement is that the original program can serve as an oracle. I.e. it can be used as an automated system that provides a reasonable output for a given input. This has been the basis of previous approaches to both genetic improvement and bug fixing [1; 2; 3; 21; 5].

However, using the original program as an oracle has its drawbacks. The original program may be buggy, in which case the 'improved' program may merely faithfully replicate buggy behaviour. The original may also be either partially defined or non-deterministic, in which case it will not provide a reliable oracle for every possible input.

It is therefore always advisable to supplement the original program with an automated oracle (or partial oracle) if one is available. The use of a partially automated oracles (other than the original program) also brings with it additional advantages: the genetically improved program may improve the *functional* properties of the system as well as its *non-functional* properties. Our approach to functional faithfulness is therefore to use the original program as one source of oracle information, but to additionally seek other partial oracles in order to check the output produced by the genetically improved system.

3.3 Sensitivity Analysis for Non-Functional Properties

In order to evolve systems to better meet non-functional requirements, we first apply a form of sensitivity analysis to determine the parts of the system that have the greatest effect on the non-functional property of interest. The parts of the system with greatest impact will have the highest priority during GISMOE's evolutionary phase.

Depending upon the requirements, there are a number of techniques that can be used to measure non-functional properties of software. Some of these can be fairly direct. On a server there is usually accounting information (e.g. number of page faults or number of pages of RAM in use), which can be harnessed as part of a fitness function. Similarly the operating system might keep track of bytes sent/received via a wireless port. In other cases, the accounting information may not be available or may be too inaccurate and so the experimenter may have to devise their own measures. It is not common to keep track of the power consumed by individual software components. However, White *et al.* [5] demonstrated how simulators can do this, and that they can be incorporated into a GP fitness measure.

In this paper, the non-functional property of concern is the execution time of the system. As might be expected, typically, which lines are used and how many times they are executed varies a great deal. We use

execution frequency as an indication of those lines of code that are likely to have the strongest influence on our non-functional property of interest. We weight lines of code both in proportion to how much they are used and also how this use scales with the difficulty of the problem.

We use a non-linear weighting in order to try to ensure that GP samples critical parts of the system to be improved more heavily. The determination of what makes parts of systems to be genetically improved 'critical' is both a domain-specific consideration and also depends upon the non-functional properties to be improved and must be defined for each kind of system to be improved. If such domain knowledge is unavailable or there is no meaningful characterisation of 'difficulty' of the non-functional properties then this aspect of our weighting scheme can simply be ignored. However, where there is a domain information, it makes sense to ensure that it plays a role in the determination of weights.

We are interested to assess the way in which the non-functional values observed vary with test case difficulty. The test cases are therefore partitioned into N O–bins. Conceptually, we plot the variation of the non-functional property (on the vertical axis) against the test case bin-number (ordered by output size on the horizontal axis). Using this plot we determine the 'algorithmic complexity' of the non-functional property for each line of code.

We assess the algorithmic complexity of this conceptual plot allocating a score of 10 for any complexity up to linear, 100 for quadratic and 1000 for cubic and higher complexity. We combine this algorithmic complexity measurement with a scalar measurement that is simply a measure of the number of times that the line is executed on average by a set of test cases, sampled uniformly from the O–Bins. The overall measurement of the sensitivity of a line of code is the maximum of the scalar and algorithmic complexity measurement obtained for the line. To prevent search concentrating overmuch, during a GP run each line of code that is mutated has its weight is reset to 1. Thus encouraging search to move on to also consider other lines.

3.4 Testing for Fitness According to both Functional and Non-Functional Properties

3.4.1 Compilation

To reduce compilation time, an instrumented version of the system is compiled without optimisation, with the gcc `-Wfatal-errors` option and using precompiled header files. Initially compilation time will be light because there will be few changes but this will increase as more files are touched and re-compilations requires a larger 'build'. For example, in our experiments with Bowtie2, compilation time was observed to grow by an order of magnitude during the genetic improvement process (from below a second at the start, up to about ten seconds by the end).

3.4.2 Randomised Test Suite Sub-sampling

We give each mutant several tests. To make them independent and so prevent an error on an earlier test affected later tests, each mutant is run on each test case separately. (See also Table 2.)

There are many test cases available for most programs we might wish to genetically improve. We therefore we adapt our sampling approach [35; 3], which incorporates the concept of O–bins. That is, at each generation, we select a single test case from each of F O–bins to form a test suite for that generation. At each generation the set of test cases to be used is re-selected to ensure diversity of testing. However, since we sample our tests from the O–bins rather than the un-binned set of available test cases, we are uniformly sampling over test case difficulty rather than over all available test cases.

3.5 Handling Infinite Loops

Since we allow `for` and `while` loops to be changed by the genetic improvement process it is quite possible that the modified code could enter an indefinite loop. We do not want non-termination of a genetically improved variant to lead to non-termination of our whole genetic improvement system. Several approaches

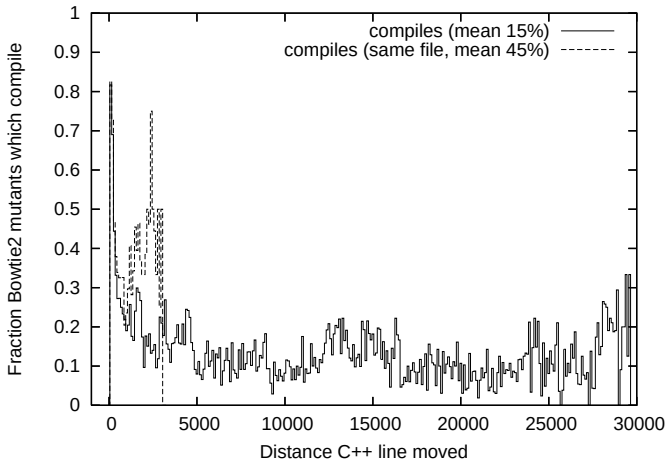

Figure 3: Fraction of Bowtie2 mutants which compile by distance replacement C++ line is moved. 82% of cases where code is moved $\leq$ 100 lines compile. When lines are only moved within the same source file (dashed line) on average three times as many mutants compile since there are fewer out of scope errors. Data binned in units of 100 lines.

have been suggested in the GP literature to handle this problem. For example, Maxwell [36] suggests a way to allow fitness comparison as programs run, while Teller [37] suggests using an "anytime" algorithm whereby answers, and hence fitness, can be extracted from an executing program, rather than waiting for it to terminate.

Especially since we know how many lines of code were executed by the original unimproved version of the system, we could instrument the code to implement a limit on the number of lines executed on a given DNA sequence. However this would increase the instrumentation overhead. Therefore, we take the simpler and more robust approach of using the operating system to time out and abort any mutant that takes more than a pre-defined cut-off execution schedule. This schedule defines a different cut off time for each test cases, based on its expected maximum executions time. As the Bowtie2 documentation says "Bowtie2 is not particularly designed with –all mode in mind, and when aligning reads to long, repetitive genomes this mode can be very, very slow" [38], hence some test cases need longer time outs than others.

3.6 Representation of the System to be Evolved

The existing program is used as the template for its own upgrade. The template is created automatically from the program's source code. Whilst evolution has great freedom to change the code, it is constrained by the template. For example the template ensures classes, types, functions and data structures are retained. Similarly evolution cannot change the program's block structure. So for example, in C++, opening and closing `{` and `}` brackets have to remain in the same place but lines between them can be changed. Thus, for example, each function's name and arguments cannot be changed but their contents can be re-written and indeed so too can the code that calls them. Similarly variables retain their names and types but evolution can use them, change their use or indeed ignore them totally.

We use a specialised BNF grammar to ensure that the evolved code is legal. However GP can generate syntactically valid code which contains semantic errors that are trapped by the compiler causing the modified code to fail at the compilation stage. Our experience is that almost all such compilation errors involve variables being of out scope. Earlier experimentation confirmed this to be the case.

Such scoping issues might be tackled by a detailed type analysis. However, increasing the fraction of shorter distance moves by restricting moves to be within the same source file has proved to be a simple and effective way of increasing the fraction of mutants which compile. See Figure 3.

Timeouts can be easily used to enforce termination. Although we have not found them to be needed in

```
<bowtie_main_42>::= "int main(int argc, const char **argv) {\n"
<bowtie_main_43>::= "{Log_count64++;/*29823*/} if" <IF_bowtie_main_43> "  {\n"
#"if
<IF_bowtie_main_43>::= "(argc > 2 && strcmp(argv[1], \"-A\") == 0)"
<bowtie_main_44>::= "const char *file = argv[2];\n"
<bowtie_main_45>::= "ifstream in;\n"
<bowtie_main_46>::= "" <_bowtie_main_46> "{Log_count64++;/*29826*/}\n"
#other
<_bowtie_main_46>::= "in.open(file);"
<bowtie_main_47>::= "char buf[4096];\n"
<bowtie_main_48>::= "int lastret = -1;\n"
<bowtie_main_49>::= "while" <WHILE_bowtie_main_49> "  {\n"
#WHILE
<WHILE_bowtie_main_49>::= "(in.getline(buf, 4095))"
<bowtie_main_50>::= "EList<string> args;\n"
<bowtie_main_51>::= "" <_bowtie_main_51> "{Log_count64++;/*29831*/}\n"
#other
<_bowtie_main_51>::= "args.push_back(string(argv[0]));"
<bowtie_main_52>::= "" <_bowtie_main_52> "{Log_count64++;/*29832*/}\n"
#other
<_bowtie_main_52>::= "tokenize(buf, \" \t\", args);"
```

Figure 4: Fragment of BNF grammar used by GP. Most rules are fixed but <IF_, <_, WHILE_ etc. can be manipulated using rules of the same type to produce mutants of Bowtie2. Log_count64++ etc. are automatically added to instrument Bowtie2. Lines beginning with # are comments.

our work, there are a number of "Sand boxing" [18] and "virtualisation" techniques to ensure even rough C programs do not cause damage.

BNF rules that correspond to single lines of source code are modified so that they now invoke another rule with the same name but with a leading underscore inserted. (For example, for the Bowtie2 program, the original rule <bowtie_main_46> in Figure 4 was modified so that it invokes new rule <_bowtie_main_46>.) GP can replace this (the underscore rule) with another rule also starting with an underscore and the resulting program will be syntactically valid.

For example <_bowtie_main_46> could be be copied to replace <_bowtie_main_51>. This gives gives two calls to open `file` but now `args.push_back()` is never called. The second call of `in.open()` finds that stream `in` is already open and does nothing. The resultant code is syntactically valid and, in this case, compiles.

In some test cases (e.g. where the first command line argument is not `"-A"`, cf. line 43) the mutant runs despite the missing call to `args.push_back()` and generates identical output to the released code. Such test cases do not reach the mutant site and so it cannot propagate its effects and so, in such cases, the mutant is equivalent to the original code [39].

The conditional parts of `if`, `else` and `while` as well as the initial, test and increment parts of `for(;;)` loops are extracted into new rules. (With rule names beginning <IF_, <ELSE_, <WHILE_, <for1_, <for2_ and <for3_, see examples in Figure 4.) GP is free to exchange these with other rules of the same type, to generate a syntactically valid program.

We limit GP to evolving code in the main modules. I.e. evolution cannot modify the include files. As in our previous work [18], we used the gcc compiler's `-E` option to strip comments, to ensure compile time configuration (using the release configuration) and to perform macro expansion on the source code.

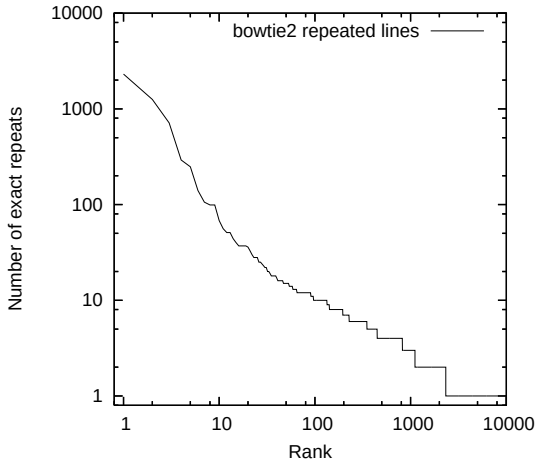

Figure 5: Distribution of repeated Bowtie2 C++ code, after macro expansion, follows approximately Zipf's law [42], which predicts a straight line with slope of -1.

Human written code is highly repetitive; whole source lines of code occur more than once in the source. For example, Gabel and Su recently found [40] that almost all small code fragments have been written before. Previous studies, e.g. [41], have reported Zipf's law [42; 43] in programmers' use of language tokens, e.g. `()` and `if` in Java, which are enforced by the compiler.

These results are also observed for the program considered in this paper: Excluding white space, Figure 5 plots the number of times lines of C++ code in Bowtie2 that are exactly repeated. It is no surprise to discover lines composed of a single `}` or a single `;` occur many times. (Actually 2310 and 1255 times.) But many more interesting lines are also repeated. E.g. the eighth most commonly repeated line is a non-trivial line of 56 characters including, branches, variables and constants. This is longer than most of the 5 848 (29%) lines that are unique. (Their median length is 28 characters).

Whilst Gabel and Su investigated code repetition across an entire suite of programs and systems, we present, in Figure 5, results for code repetition within one single C++ system, Bowtie2. We suspect that the results we observed and those reported for much larger corpuses (e.g. the results reported recently by Gabel and Su [40]) reflect a wider trend.

It is also well known that crossover can produce large amounts of repeats, both in natural DNA and in linear and tree genetic programming [44; 45].

Genetic improvement should take these observations about code repetition into account. Therefore, instead of allowing GP complete freedom to invent any syntactically valid code, we insist it reuse code that has already been written by the creator of the program to be genetically improved. Evolution thus proceeds by "cut and paste". "Cutting" i.e. removing lines of code, and "pasting" means to make a copy of a line of code in another place.

We were surprised that such a simple approach to modification could yield dramatic genetic improvements. We believe that this is a potentially important finding of our work. Perhaps this finding is less surprising when consider that it has been known for some time that product code written by humans contains a non-trivial amount of cloned code [46; 47]. In the nomenclature of the clone detection community, our approach to genetic improvement operators essentially uses only the so-called 'type 1' clones (clones which are verbatim copies with no substitutions or re-writes). However, unlike work on clone detection, we allow clones to be small fragments (no bigger than a single line of code). By contrast, for the clone detection community, a default of 5 lines is often used as a minimum [48; 49], which, interestingly, corresponds closely to the amount of code found by Gabel and Su to be needed for uniqueness [40].

Our approach is similar to the 'plastic surgery' [50] approach of Weimer *et al.* [21] in which code is 'scavenged' from other parts of the program under evolution. However, whilst Weimer *et al.* consider code

at the statement level, we will deal with lines of C++ code. We are able to perform syntactically correct substitutions at this lexical level, since we use a grammatical representation of the program to be genetically improved.

3.7 Representation of a Genetically Improved Variant

In earlier work on genetic improvement [1; 3; 4; 5], the entire program was evolved. This was feasible because there was only a (small) program [1; 4; 5] or part of a program [3] to be evolved. However, in order for Genetic Improvement to scale it must cater for programs of several orders of magnitude larger than have previously been considered. We therefore adapt an approach recently used to scale up bug fixing [22]. We represent a GP individual as an ordered list of *changes* that are to be made to the BNF grammar. To delete a line of code, the individual gives its name (i.e. the name of its BNF rule). To replace a line, the name of the corresponding BNF rule is given together with the name of the line of code which is to replace it. An insert operation is essentially the same, except we add + to the text, so we know to add a copy of the line of code and not to remove the original line of code.

Here are some examples of the application of this approach to the Bowtie2 system:

```
<for3_sa_rescomb_111><for3_sa_rescomb_69>
```

This GP individual causes the increment part of the `for` loop on line 111 of source file `sa_rescomb.cpp` to be replaced by the increment part from the for loop on line 69.

```
<_aligner_swsse_ee_u8_804>
```

This individual causes line 804 of `aligner_swsse_ee_u8.cpp` to be deleted.

```
<_aligner_result_47>+<_aligner_result_114>
```

This individual inserts a copy of line 114 in front of line 47 in file `aligner_result.cpp`.

In the second generation we start to see individuals that make two or more changes. These individuals are simply one line of text with a space between each of their constituent mutations. Mutations are applied in order. However we can readily spot mutations which replace the same line of code.

In which case only the last one need be applied. In fact we use genetic "repair" [51] so where conflict arises an individual's genome only contains the relevant, i.e. the last, mutation. Notice we can easily keep track of which source files have been changed and use a unix "make" file to ensure only the modified files are recompiled.

All this manipulation is done in plain text, unlike other work, based on CIL, which operates on abstract syntax trees (AST) [21]. Our grammatical representation of the program to be improved makes this practical, even though it operates at a lexical level. Even for the largest set of genetic changes and even prior to the final local search bloat-removal phase, the time required to make the changes is typically less than that required to compile the resulting modified system.

3.8 Selection

Up to half the current population can be selected. Those below the cut point, as well as mutants which failed to compile or which never exceeded the released code in any way are not transmitted to be parents of the next generation. As mentioned in Section 3.10, if fewer than half the population are selected, two new children per missing member are created from scratch. I.e. they are effectively reinitialised. The new individuals are created in the same way as the initial population was created in generation zero.

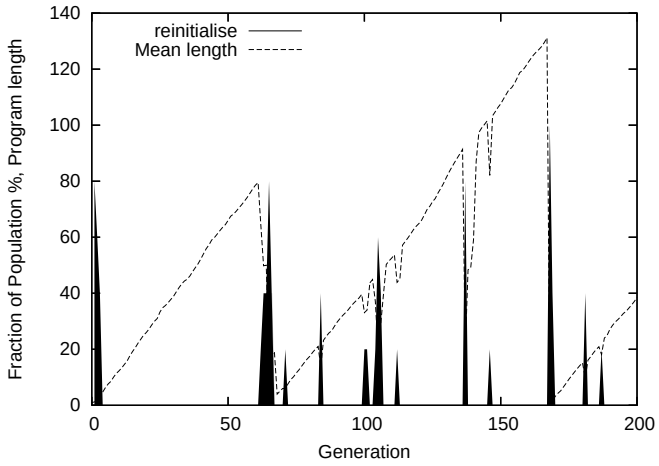

Figure 6: Increase in mean number of mutations as evolution improves Bowtie2. Note many members of population (10) reinitialised near generations 65 and 167, causing long mutation lists to be replaced by new (much short) individuals.

3.9 Mutation

An individual is mutated by appending a new grammar modification to the list that denotes an individual (see Section 3.7). The additional line to mutate is chosen from all lines executed at least once by the test cases selected from the O–Bins for sensitivity analysis. The line to be mutated is chosen with a probability that is defined by its sensitivity analysis weight.

One of the three types of mutation (deletion, replacement and insertion) is chosen (with equal probability). However it makes no sense to delete one of the trivial lines (i.e. lines just containing a single disabled `assert` or `;`). Trivial lines can, with equal probability, be used either as the point to insert new code or be replaced with a non-trivial source line. The new code is chosen uniformly at random from non-trivial lines of the same type (captured by our grammatical representation approach) in the same source file that were executed at least once. (The types were described in Section 3.6 above.)

We avoid the generation of 'no operation' and duplicate code. That is, a child is rejected if either it makes no change (i.e. replacing self with self) or where the corresponding sequence of changes already exists. (Both can be spotted efficiently since changes are essentially cut-and-paste operations). If a child is rejected in this way, then the parent is mutated again until a non-duplicate is created.

3.10 Crossover

We anticipate that many changes are somewhat independent; a genetically improved program can be found by combining multiple changes. This is the role of crossover. In our case crossover simply concatenates two individuals. The first parent is selected from the current population according to its fitness. The second is drawn uniformly from the members of the current population which compiled (i.e. have a fitness value). Naturally, such a crossover leads to rapid growth in chromosome length (bloat [52]). See Figure 6 for an example of the increase in genotype length which we observed in our experiments with Bowtie2.

As with mutation (previous section) each child's genotype is reduced to canonical form and a crossover will be rejected if it is already present in either the new or the previous generation. If, after a small number of retries, crossover cannot find a unique individual, the new member of the population is created by mutating a fit member of the population.

Normally half the new population is created by mutating the fittest parents and half by crossing over the fittest parents with other fit members of the population. (To ease reproduction by others all the key parameters of our evolutionary system are given collected together in Table 1.) However if the number of

Table 1: Genetic Programming Parameters(including, for replication purposes, the specific parameters used for Improving Bowtie2 on 1000 Genome Project Solexa short DNA sequences).

Representation:	List of replacements, deletions and insertions into BNF grammar
Fitness:	Based on compiling modified code and testing it. See Sections 3.4 and 3.8 and Table 2
Selection:	A mutant cannot be selected to be a parent unless it does better on at least one test case than the original (instrumented) code. It is said to better if the unmodified code failed on the test case and it doesn't or its mean Smith-Waterman score is higher than that of Bowtie2 or if it used fewer statements. However normally it must also report at least one match. Scores are sorted by number of test cases where the mutant returned an answer, mean Smith-Waterman, and finally by number of statements executed. The first five are selected to have 2 children in the next generation. One child is a mutant, the second is a crossover between a selected parent and another member of the current population. Children must be different from each other and from the current population. If crossover can not create a different child, the child is created by mutating the selected parent.
Population:	Panmictic, generational. 10 members. New training sample each generation.
Parameters:	Initial population of random single mutants weighted towards heavily used statements. 50% append crossover. The 3 types mutation (delete, replace, insert) are equally likely. No size limit. Stop after 100 generations.

fit parents in the current population falls below half, mutation and crossover will not create sufficient new children to fill the new population. In this case, to restore diversity to the population, the missing children are created at random (in the same way as the initial random population).

3.11 Post Processing Solution Cleanup

It is common for solution programs evolved using genetic programming to be 'bloated' [52]. That is, some parts of the evolved changes make little or no difference. From the point of view of software engineering, maintenance, ease of integration of genetic changes into human written code, etc., it is easier to work with a minimal number of changes. It is possible for genetic programming to minimise evolved code, but in [32, page 152] we had used larger populations. Therefore we decided to use a simple hill climbing strategy to minimise the size of the ordered list of changes after the end of the GP run.

Starting from the beginning of the best individual in the last generation, each of the changes are disabled one at a time. If removing the change makes the mutant worse, then the evolved change is kept. Otherwise it is removed. The hill climber then goes on to test the effect of removing the next evolved change and so on, until the whole evolved mutant has been so-processed.

4 Applying the GISMOE Genetic Improvement Approach to Bowtie2

In this section we explain how the GISMOE genetic improvement approach we introduced in the previous section is applied to the Bowtie2 program.

4.1 Determining Functional Correctness for Bowtie2

For the automated oracle we use the Smith-Waterman algorithm [53] to compare the answer given by Bowtie2 with the human genome. Unlike Bowtie2 itself (and related tools), Smith-Waterman performs a complete comparison, rather than using heuristics. However Smith-Waterman can only allow us to check the reported sequence matches for correctness, it does not allow us to check for missing answers. Smith-Waterman thus provides a partial oracle, that can evaluate the answers given.

In order to use the Smith-Waterman score, we need to allow for partial matches (indels). To do this, the reference string against which matches are checked is extended by nine characters at either end. The

genetically improved system's Smith-Waterman score for a test case is the mean of the Smith-Waterman scores over all matches it suggests for that test case. However, if the output from the modified system is a match that fails to lie exactly where Smith-Waterman locates the optimum match, then the match's score is reduced by 1.0 for each DNA string position by which its output disagrees (subject to the match's score not going below zero).

Bowtie2 reports many accountancy details about the matches it finds between the test Solexa DNA sequence and the human genome. We give credit only for the matches themselves. Potentially evolution can make minor saving by mutating Bowtie2 so that it no longer generates this unwanted output.

4.1.1 Training Data – Human Genome, Bowtie2 and the 1000 Genome Project

The complete official release of the reference human genome (release 37 patch 5) was downloaded from the National Center for Biotechnology Information (NCBI). The NCBI also maintains BLAST. The 64 bit Linux version of Blast was downloaded from its FTP site (version 2.2.25+) and this version was used in the experimental comparisons reported below. The C implementation of the Smith-Waterman local alignment algorithm [53] was down loaded from Cologne University Biological Physics dept. The Smith-Waterman algorithm does a complete search to find the optimum match between two strings.

The C++ sources for the 64 bit Linux version of Bowtie2 (version 2.0.0-beta2) were down loaded from sourceforge (50 745 lines in 50 modules). This version of Bowtie2 was used to create an ASM format database holding the reference human genome from the NCBI DNA sequences. We have evolved all new versions of Bowtie2 by fitness testing against the complete human genome (3.9 GBytes). Fitness testing might be speed up by using only part of the database or indeed a smaller genome from a non-human source (e.g. yeast or mycoplasma bacteria). However our goal was to tailor Bowtie2 to the task of looking up human DNA sequences. Indeed, to create a version specific to real DNA sequences generated by a particular sequencing technique, rather than synthetic data.

The 1000 genome project [54] has sequenced, wholly or in part, DNA from more than one thousand individuals using a variety of next generation sequencers. Our goal is to show the automatic generation of improved software for a particular task, so we use data from a popular scanner make used in a laboratory for which we have copious training data. The data can be obtained via FTP from `ftp.1000genomes.ebi.ac.uk`. For the time being in order to minimise true biological variation we chose homogeneous data (i.e. DNA sequences with exactly 36 bases) from one well studied CEU family and the Solexa data provided by the Broad Institute, Cambridge, MA.

In common with other next generation DNA scanners, the Solexa scanner includes both an estimate of the quality of each base in the sequence and uses "N" to indicate any DNA base which it cannot decide which of the four bases (A, C, G, T) it really is. The data quality is highly variable. In one dataset less than 1 in a thousand sequences has an N. In the worst training dataset every record had at least one (typically two or three).

4.1.2 Preparing the Training Data

The performance of Bowtie2 depends strongly upon the number of matches it finds between the query DNA sequence and the reference human genome. To get a good spread of Solexa DNA sequences for training we started by using the released version of Bowtie2 to annotate a sample of DNA sequences with the number of times they occur in the human genome. We randomly selected 500 of the ≈ 8 million DNA sequences in each of 11 Solexa runs for a CEU female (NA12878). Then we ran the released version of Bowtie2 against the human genome on groups of 50 randomly chosen sequences and for each counted the number of matches it reported. (Total $11 \times 10 \times 50 = 5500$.) Even on a 32GB 8-core server, in five cases Bowtie2 was aborted (after failing to respond), leaving us with 5250 DNA Solexa sequences. The distribution of the number of matches is plotted in Figure 7.

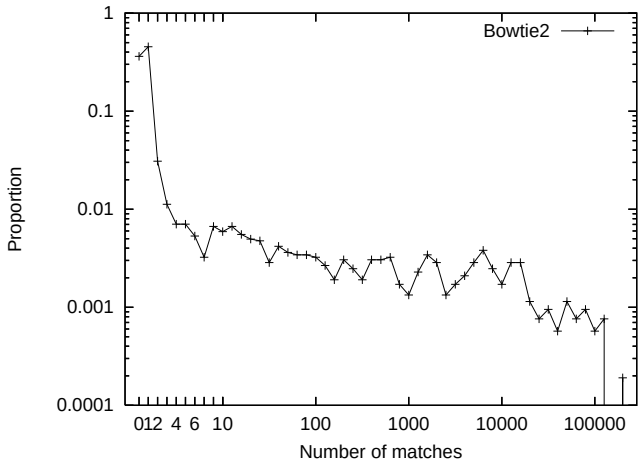

Figure 7: Distribution of number of matches in human genome for Solexa DNA sequences found by Bowtie2. (Note non-linear scales. With more than ten matches decile bins are used.)

4.2 O–Bin Sampling as Applied to Bowtie2

We implemented the O-bin sampling described in Section 3.1 as follows: Each generation five DNA sequences are chosen uniformly at random from the 5250 Solexa sequences described in the previous section:

1. a sequence which Bowtie2 cannot find in the human genome
2. a sequence which it matches exactly once
3. a sequence which matches between twice and ten times
4. a sequence which matches between 11–99 times
5. a sequence which matches between 100–200 times

The first two cases can be viewed as positive tests to ensure the modified Bowtie2s still retain their essential ability to both report the absence of matches and find them. Whilst with the later ones we hope to detect modified Bowtie2 that are faster. Cases 3 and 4 are intermediate. They seek to guard against chance playing too great a role in parenthood selection. Bowtie2 run time grows cubically with number of matches. Figure 7 shows the number of matches within the human genome that Bowtie2 can find is essentially unlimited but given O(n^3) run time we cannot possibly use them in fitness testing. Instead we imposed an upper limit of 200 matches. Even so the fifth test case (which is drawn form the O–Bin containing the most demanding test cases) frequently takes the most computational effort.

4.3 Representing the Bowtie2 Source Code to be Evolved as Grammar

Following the general GISMOE approach outlined in Section 3.6 (page 8) we limit GP to evolving code in the main modules of Bowtie2. This yields 39 modules containing about twenty thousand lines of code. These were automatically translated line for line into a BNF grammar of 19 949 rules. (See Figure 4.)

4.4 Sensitivity Analysis for Non-Functional Properties Applied to Bowtie2

To illustrate the importance of our sensitivity analysis, consider Figure 8, which shows the statements execution frequency for an example input for which Bowtie2 produced a lot of output. Of the 13 498 executable lines that were instrumented, 9 760 (72%) were never used, 1518 (11%) were used exactly once, 846 (6%) more than once but less than the number of matches (9934), 309 (2%) were used exactly

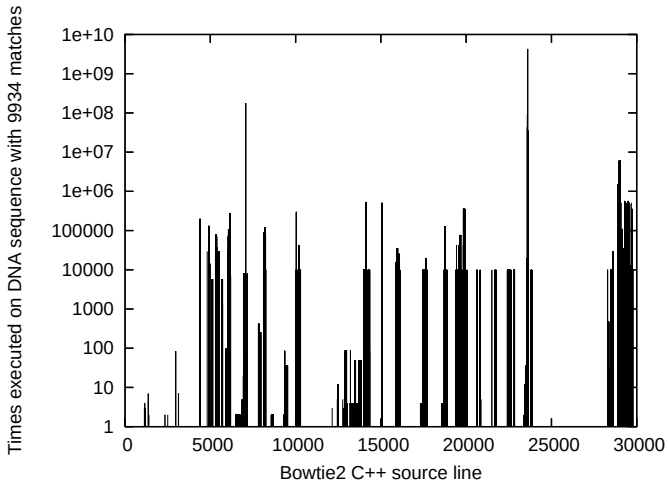

Figure 8: Example when Bowtie2 finds many matches of the distribution of the number of times each Bowtie2 C++ source line is used. (All 39 source files but excluding all 67 header files.) 72% of lines are not used but 80 lines are run more than a million times. (Note log scale.)

9934 times and 929 (7%) were used more than 9934 times. In fact 80 lines were used more than a million times, with two being used more than 2 147 483 648 times.

Although Figure 8 describes a single test case, the GISMOE weighting scheme (described in Section 3.3, page 6) is based on a large number of test cases. In more detail, for Bowtie2, the 5250 DNA Solexa sequences described in Section 4.1.2 were sorted by number of matching strings into decile O–bins. I.e. ten O–bins per order of magnitude, so there were ten O–bins for numbers between 10 and 100, ten for 100 to 1000 and so on. That is the O–bins are spread evenly on a logarithmic scale. If there were more than ten DNA sequences in an O–bin, ten were chosen at random from it. This yielded 362 DNA sequences with a wide variety of number of matches in the human genome.

An instrumented version of Bowtie2 was run on them all to give for each of the 362 input test cases which lines of code were used and how many times. Figure 9 shows, for 4 example source lines, the relationship between the number of times the query string matches in the reference database and the number of times the source code is executed. The instrumented version allows us to not only know which lines of code are in use but also to estimate how their usage scales. We find lines which are 1) never used, 2) used once (or a constant number of times), 3) usage varies linearly with output size (n), 4) it varies as n^2 and 5) it varies in proportion to n^3. This gives us a crude assessment of the algorithmic complexity of each the line of code.

For Bowtie2 the weights described in Section 3.3. are calculated as follows: If a C++ line is used in any of the 362 instrumented runs it will be given a weight between 1 and 1000 in proportion to the number of times it is used in that test. If its not used at all, then GP ignores it and will not mutate it. If it is heavily used in any test, it is given the higher weight.

These weights are combined with the algorithmic complexity assessment (which allocates weights of 10, 100 and 1000 for O(n), O(n^2) and O(n^3) complexity, as described in Section 3.3) by selecting the maximum of the two scoring systems. Figure 10 (generation 0) shows the results of this non-functional sensitivity analysis as applied to Bowtie2. Figure 10 shows 2111 lines have initial weight 1, 483 have weight 10, 103 weight 100 and 47 have weights of initial 723 or more.

4.5 Combining Functional and Non-Functional Fitness to Create an Overall Fitness for Bowtie2

Each genetically improved variant that compiles is compared to the instrumented original on the five test cases selected from the O–Bins (to be described in Section 4.6.4, see also Table 2). Two fitness criteria are

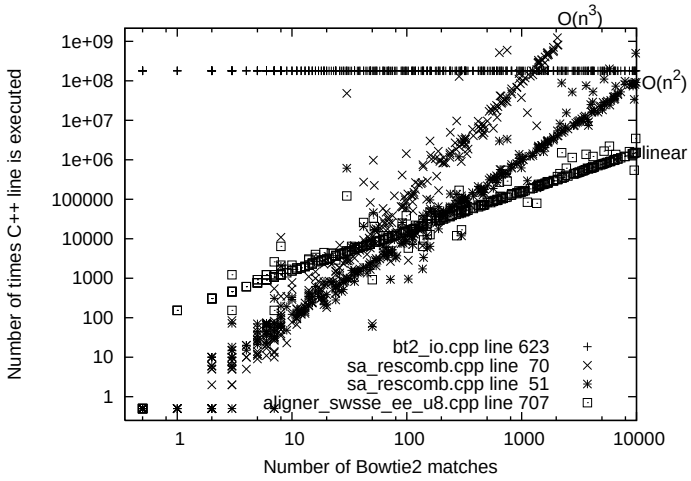

Figure 9: Example heavily used lines in Bowtie2 which scale differently with number of matches found for the input Solexa DNA sequence in the human genome. Constant $+$, linear ($\Box$), quadratic ($*$) and cubic ($\times$). (Note log scales.)

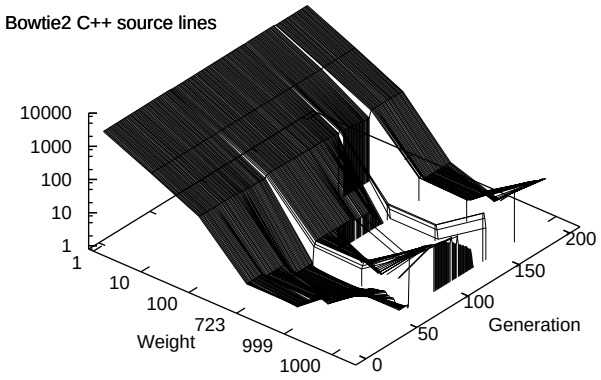

Figure 10: Evolution of distribution of weights. Initial GP population (generation 0) at left. Note fall in proportion of higher weights as population evolves to mutate highly used C++ lines and then recovery as it (partially) resets in generations 68 and 168. (Note non-linear scales.)

used corresponding to the functional and non-functional criteria: Did the mutant run faster (which, in our case, is measured in terms of the execution of fewer lines of code) and did it produce better answers on average (which is measured in a domain-specific manner). It need only do better on either criteria on any of the five test cases to be considered as a parent of the next generation. (However its not considered better if it finds no answers at all, no matter how fast it goes.) Those variants that compiled and were judged better than the original code on the current five test cases are sorted according to three fitness criteria. There are three fitness criteria (in order of precedence):

1. number of test cases completed without run time error
2. mean Smith-Waterman score
3. lines of C++ code executed (minimised)

We order the criteria in this manner to favour functional faithfulness (to the original) highest, then the functional information from the oracle next highest, and finally the non-functional property we see to improve (execution time). This is because we seek solutions that are better according to the non-functional criteria, but at least no worse according to the functional criteria. Other possibilities, that are more 'heretical' in their approach to correctness are discussed elsewhere [2].

4.6 Implementation Details

This section presents implementation details required to make the genetic improvement process practical and which may be required by other researchers for replication purposes.

4.6.1 Aborts, Heap errors, Segmentation Errors, Floating Point Exceptions, `assert` *exceptions*

Almost all mutant programs which compile run all of their test cases and produce an answer on each. Only 6% fail. The most frequent causes of run time failure are segmentation faults (≈3%) and CPU time limit overruns (≈2%). In total the remaining 0.6% of runs either abort (e.g. due to heap corruption), report a floating point error (e.g. divide by zero) or fail one of Bowtie2's own `assert` exception checks.

4.6.2 Zombies

In unix, a process can sometimes fail in such a way that the operating system has difficulty cleaning up after it and instead of terminating it unix places it in a "zombie" state. Since such zombie processes do not terminate or timeout a zombie could potentially hold up our GP indefinitely. We found that zombies only occur infrequently. Indeed, none were created during the runs described in Section 4.6.1. Nevertheless, in order to guard against it we used a background zombie killer.

If the machine is overladen then the failure of a process to respond might be due to its failure to receive any computation time. In our experiments we set this overladen threshold to 16. This means that the machine is considered overladen should there be 16 or more processes awaiting scheduling all of which are able to proceed. 16 seemed to be a suitably conservative value, given that our experiments ran on a machine with 8 cores.

Once a minute the zombie killer background process compares the CPU time taken by each fitness job with that taken by it in the previous minute. If they are identical the job is killed (provided the machine is *not* overladen), freeing the GP to continue to the next fitness test.

4.6.3 Details of compilation failures and aborted runs

Throughout the run (see Figure 11) about 26% of compilations fail. (All but six compilation failures are caused by moving variables out of their scope).

Table 2: Fitness function

- Each generation chose uniformly at random one test case from each O–Bin (Section 3.1, page 6).
- **Fitness test** original (albeit instrumented) system on each of these test cases
- **Fitness test** each member of the population
  - Generate modified source code and then compile it.
  - If fails to compile, GP individual cannot be a parent so skip rest of fitness testing
  - For each of the selected test cases,
    - Run modified system (subject to time out)
    - For each reported match with the human genome
      - Calculate Smith-Waterman score for test case v. match
      - If optimal Smith-Waterman match is not where it was claimed to be, subtract the distance between claimed match and Smith-Waterman match from Smith-Waterman score (subject to score not being made negative).
    - calculate mean Smith-Waterman score for this test case and store number of instructions executed
- At the end of the generation, each member of the population which compiled is compared with the original code on each of the test cases.

  If a modified version of the system does better on any test case (i.e. takes fewer instructions or has higher mean Smith-Waterman score) it is considered to be better. However a modified version cannot be said to be better if it never claims any matches.
- The better variants are sorted in order:
  1. Number of test cases where it did not fail or time out and was not aborted by the zombie killer (Section 4.6.2)
  2. Mean Smith-Waterman score
  3. Number of instrumented lines of C++ run (total on all test cases). To be minimised.
- The top popsize/2 are selected to be parents of the next generation. If fewer than popsize/2 variants are better than the original system, two children per missing variant are created at random (in the same way that the initial population was created.)

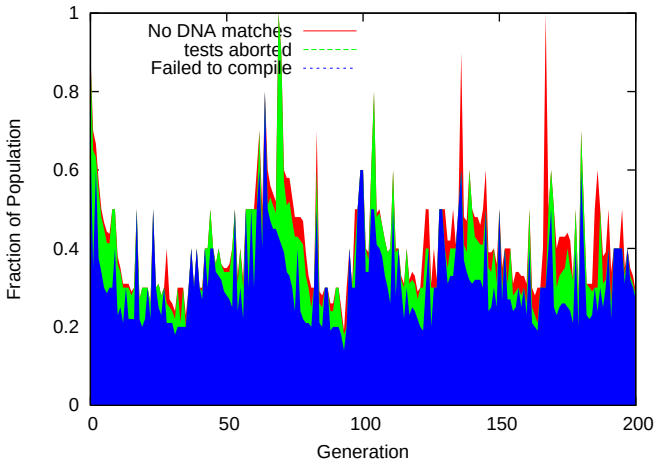

Figure 11: Fraction of population which fails to compile (bottom), aborts on one or more test cases (light) or fails to find any match in the human genome (top). In generation 167 there are no suitable parents and the population is reset. Data smoothed (by averaging to right over ten generations).

4.6.4 Comparing with the Original Program

As explained in Section 3.8 the instrumented (but otherwise unchanged) Bowtie2 code is run on the test cases selected as a training set of the generation (See Section 3.4). This took an average of 38 seconds.

4.7 The Role of the Smith Waterman Score in the Post Evolution Clean

To avoid excessive run time, the hill climber used to minimise the evolved solution (cf. Section 3.11) uses a (fixed) subset of all the training data. One hundred different DNA sequences were randomly chosen from each of the five classes described in Section 3.4. (However only 41 of the 5250 Solexa training sequences match in the human genome reference sequence between 100–200 times. So they are all used.) This gives 441 training DNA sequences for the hill climber.

The Smith Waterman score is used to winnow the mutations listed in each individual evolved by generic improvement. A smaller mutation is considered worse if:

1. It does not compile.
2. It uses more than 1% more instructions than the evolved mutant.
3. If fails to find a match.
4. The Smith-Waterman score of all the matches it reports for a DNA sequence is on average more than 1.0 lower.
5. If ten or more of the matches it finds have on average lower Smith-Waterman scores than the evolved mutant.

If, according to this definition of worse the removal of a change from an individual fails to make the resulting version of Bowtie2 worse, the change is considered unimportant and it is permanently removed from the individual.

5 Genetically Improved Bowtie2

Section 5.1 describes the evolution of performance, both on the per generation training sets and on a fixed sample representing all the training data. Whilst Section 5.2 describes out-of-sample performance and then Section 5.3 describes out-of-sample performance of Bowtie2 when modified by the minimised genetic change.

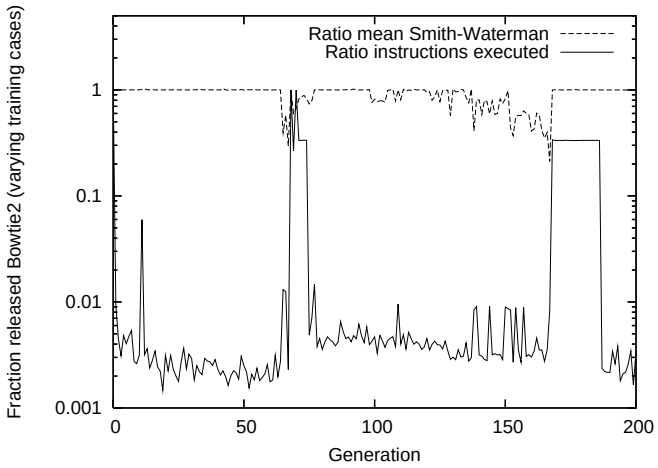

Figure 12: Evolution of performance on randomly changing training cases. In the last generation the best uses 290 times fewer C++ statements than the original code. In most generations there is a small improvement in mean Smith-Waterman score (dashed line) which is obscured by log scale.

5.1 Performance During Evolution

We evolved a population of ten Bowtie2 mutants for two hundred generations. Figure 12 shows the best in the population's fitness at each generation. There is a huge improvement in speed and a very small improvement in mean Smith-Waterman score. Figure 6 plots the evolution of genotype size. Notice size increases (bloat) under the action of our crossover (Section 3.10). However the population is reinitialised near generations 65 and 167. I.e. twice during evolution the population contained very few good individuals and, as described in Section 3.8, the poor ones were replaced by reinitialising them in the same way that the initial population is created.

To show training performance in general, every ten generations, we re-tested the best of generation program on a much bigger and fixed subset of the 5250 training Solexa DNA sequences described in Section 4.1.2. Apart from the random number seeds, we used the same procedure as in Section 4.7 to select 441 DNA sequences. The speed (as a fraction of the number of instructions used by the unmodified instrumented version of Bowtie2) of the best of each generation is plotted in Figure 12.

In the last generation the best individual used only one three hundredths of the instructions used by the instrumented Bowtie2 on the five DNA sequences used for training in generation 200. When both were tested on the 441 DNA sequences one at a time the ratio was five hundred.

Figure 13, like Figure 12, shows the performance of the best in the population every tenth generation. However, unlike Figure 12, the evolved versions of Bowtie2 are run once on the 441 test case. (Rather than 441 times, once on each test case.)

As suggested in Section 3.4, GP has been able to optimise the initialisation code. Therefor when we use the 441 DNA sequences in a single file improvements in it have less proportionate effect. Nevertheless Figure 13 shows in most cases the best mutant of each generation still uses only about a fifth of the instructions used by the unmodified Bowtie2. Also Figure 13 shows in most generations the best of generation mutant finds DNA matches in the human genome which on average are at least as good as the original unmodified version of Bowtie2. Again the improvement in Smith-Waterman score is small (e.g. 0.5% in generation 200.)

5.2 Performance Comparison on Hold Out DNA Sequences

For verification, ten DNA scans were randomly chosen from two different individuals (one male, one female, NA12891 NA12892, being the parents of NA12878) giving 20 complete Solexa scans (a total of

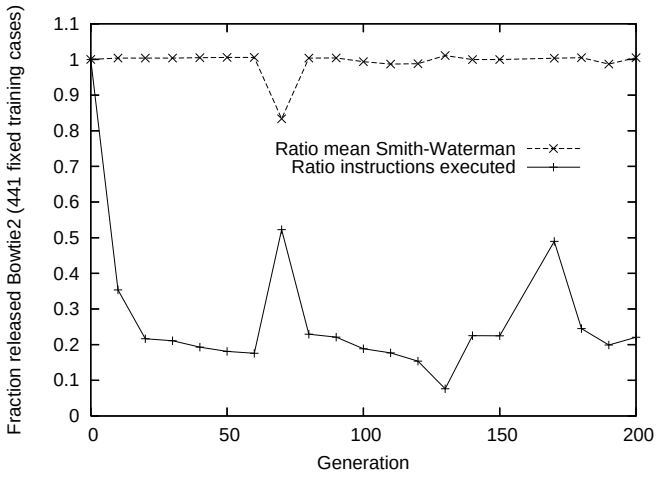

Figure 13: Speed of best in population every ten generations on a fixed training set. (Note Figure 12 gives GP fitness with actual training set, which is changed every generation.) Here we run each genetically improved versions of Bowtie2 on all 441 DNA sequences together.

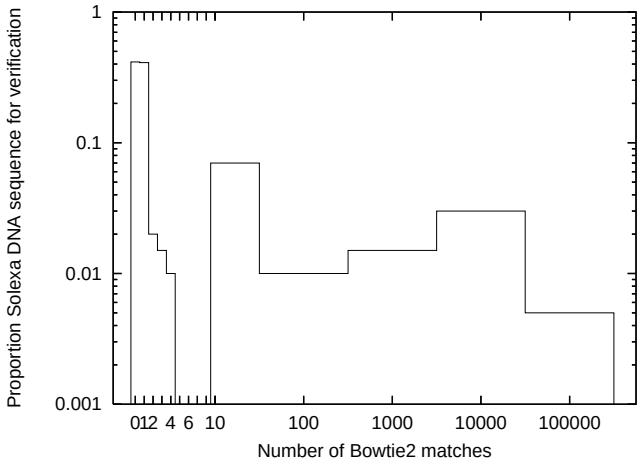

Figure 14: Distribution of number of matches in human genome for verification Solexa DNA sequences. Note non-linear scales. With more than ten matches bins cover an order of magnitude. Cf. Figure 7

176 893 951 DNA sequences). 10 DNA sequences were chosen from each Solexa scan. The distribution of number of matches for these 200 DNA sequences, which is strongly related to run time, is shown in Figure 14. Apart from sampling noise, it should be the same as the whole of the Solexa scans. Notice that it contains a few sequences which match a very large number of times. As we described in Section 4.1.2, such sequences were deliberately excluded from the training data, as they cause the released version of Bowtie2 to become very slow.

The released version of Bowtie2 and the evolved version were both tested on the 200 DNA sequences. Neither was instrumented and both were compiled with the same compiler optimisations as are used in Bowtie2's installation kit (i.e. gcc -O3). The evolved version took 3.9 hours. The released code took 12.2 days. Meaning on average the genetically improved program is seventy four times faster on out of sample data.

In 178 cases (89%) the GP version of Bowtie2 produced identical results to the released code. In 18 cases (9%) the GP version was better (i.e. the matches it reported had a mean Smith-Waterman score better than that of the released code). In one case the Smith-Waterman scores was identical and in three (1.5%) the scores were worse but differed only in the 4^{th} and 6^{th} significant decimal place ($p = 0.001$ sign test). The median improvement was 0.1 (max 6.32). The GP version never reported more matches or reported there were no matches when there were some. In 17 cases it reported slightly fewer matches (median

reduction 1.3%, $p = 2\ 10^{-5}$ sign test).

5.3 Minimised Patch

The best in generation 200 evolved individual (shown in Figure 15) makes 39 changes to the released version of Bowtie2. Using the clean up procedure described in Section 4.7, this was reduced to seven changes (shown in Figure 16). The reduced version was compiled in the same way as the released code (i.e. gcc -O3) and tested on the 200 verification DNA sequences. It produced identical output to the evolved 39 changes version and was 4% faster, giving a speed up compared to the released code on the hold out DNA sequences of 77 times.

6 Related Work

Whilst genetic programming has been used many times in Software Engineering, e.g. in project management [55] and testing [56], we are particularly concerned with evolving code. GP has not demonstrated an ability to write large programs normally associated with "programming". However even modest amounts of evolved code can be useful.

6.1 Current and Existing Research

Martin [57] showed that genetic programming can build a telephone call re-routing service (home/office) from existing telephony components. Although clearly not a like for like comparison, Martin claimed GP elapsed times of about a minute, whereas a commercial "study showed that for a complex service a team of engineers required 4.5 Man years of effort to analyse, design, code and test the service." (Note the commercial study also includes non-coding business activities whereas GP elapsed time covers only coding.) More recently Rodriguez-Mier *et al.* [58] showed that GP can create novel web services by combining human written existing web services using the web ontology language (OWL). Notice the power of web mashups comes from quite small amounts of glue logic. Glue logic might also be usefully evolved to combine other types of software: perhaps on the same server, perhaps written in the same or different languages (e.g. PHP and Cobol). Combinations which are rare or difficult for a human programmer or where skills are difficult to find may turn out to be no more difficult for a machine than a routine combination. Similarly, once objectives can be quantified, it may be as easy for a machine to juggle multiple objectives (perhaps a mixture of functional and non-functional requirements) which a human programmer would be hard pressed to meet simultaneously and would in practise tackle one at a time.

Another approach to side-stepping the scaling problem is Yamamoto and Tschudin's [59] "fraglets" approach. This uses GP in an artificial chemistry like approach whereby small fragments of existing code are combined. Yamamoto and Tschudin [59] considered evolving network communications protocols. More recently Weise and Tang [60] evolved distributed algorithms (election, critical selection for mutual exclusion distributed locking and distributed greatest common divisor) using a GP rule based approach.

One of the earliest attempts to evolve software looked at evolving hashing functions [61]. (See also [62], [63] and [64]). Hashing is often used to speed up search. A hash function takes an object (typically a string) and deterministically converts it to one of the legitimate indexes into a data store. Whilst hashing typically takes constant time, the search in the data store typically grows with the number of items with the same hash index. An efficient hash function will ensure commonly used objects hash to (relatively) unique indexes. There are many good hash function but how good a hash function is in practice depends upon the distribution of objects it has to deal with. Often this is not known in advance by the programmer. So a generic hash function may do poorly with specific examples. Human written hash functions may be tuned for a certain load. Tuning usually means the hash function is coded, tested and recoded in response to the tests, tested again and so on. Notice this is effectively a manual version of the classic evolutionary algorithm: generate and test, then regenerate and retest and so on.

Memory management is another area where traditionally people try to write generic code by making as-

disabled bt2_search.cpp line 1931 IF `(metricsStderr)`

disabled bt2_io.cpp line 612 IF `(r != (ssize_t)(offsLen << 2))`
replaced bt2_io.cpp line 622 for1 `uint32_t i = 0` with `int i = 0`
replaced bt2_io.cpp line 622 for2 `i < offsLenSampled` with `i < this->_nPat`
replaced bt2_io.cpp line 622 for3 `i++` with `i += 3`

inserted `initedRef_ = true;` before aligner_sw.cpp line 145 `(static_cast<void> (0));`

replaced aln_sink.cpp line 758 IF `(readIsPair())` with `(nunpair2 > 0)`
disabled aln_sink.cpp line 919 IF `(nunpair1 > 0)`

replaced sa_rescomb.cpp line 50 for2 `i < satup_->offs.size()` with `0`
disabled sa_rescomb.cpp line 51 IF `(satup_->offs[i] == 0xffffffff)`
deleted sa_rescomb.cpp line 52 `needResolving++;`
disabled sa_rescomb.cpp line 66 for2 `i < refscan_.size()`
deleted sa_rescomb.cpp line 68 `nfound++;`
disabled sa_rescomb.cpp line 69 for2 `j < satup_->offs.size()`
disabled sa_rescomb.cpp line 69 for3 `j++`
replaced sa_rescomb.cpp line 70 IF `(satup_->offs[j] == refscan_[i])` with `(refscan_.empty())`
deleted sa_rescomb.cpp line 72 `found_[i] = false;`
deleted sa_rescomb.cpp line 73 `nfound--;`
disabled sa_rescomb.cpp line 112 IF `(refscan_[i] == off)`

replaced aligner_swsse_ee_u8.cpp line 707 `vh = _mm_max_epu8(vh, vf);` with `vmax = vlo;`
inserted `vlo = _mm_xor_si128(vlo, vlo);` before aligner_swsse_ee_u8.cpp line 711 `pvHStore += 4;`
replaced aligner_swsse_ee_u8.cpp line 746 `ve = _mm_load_si128(pvEStore);` with `met.dpsucc++;`
deleted aligner_swsse_ee_u8.cpp line 766 `pvFStore += 4;`
replaced aligner_swsse_ee_u8.cpp line 772 `_mm_store_si128(pvHStore, vh);` with `vh = _mm_max_epu8(vh, vf);`
deleted aligner_swsse_ee_u8.cpp line 773 `pvHStore += 4;`
deleted aligner_swsse_ee_u8.cpp line 776 `vh = _mm_subs_epu8(vh, rdgapo);`
deleted aligner_swsse_ee_u8.cpp line 777 `vh = _mm_subs_epu8(vh, *pvScore);`
deleted aligner_swsse_ee_u8.cpp line 778 `ve = _mm_max_epu8(ve, vh);`
replaced aligner_swsse_ee_u8.cpp line 779 `_mm_store_si128(pvEStore, ve);` with `d.maxPen_ = d.maxBonus_ = 0;`
deleted aligner_swsse_ee_u8.cpp line 781 `pvScore += 2;`
replaced aligner_swsse_ee_u8.cpp line 785 `pvFStore -= colstride;` with `met.gathcell++;`
inserted `btnstack_.clear();` before aligner_swsse_ee_u8.cpp line 788 `vh = _mm_load_si128(pvHStore);`
deleted aligner_swsse_ee_u8.cpp line 789 `pvEStore -= colstride;`
replaced aligner_swsse_ee_u8.cpp line 796 `vh = _mm_load_si128(pvHStore);` with `d.bias_ = 0;`
replaced aligner_swsse_ee_u8.cpp line 801 `vf = _mm_subs_epu8(vf, rfgape);` with `btnstack_.expand();`
deleted aligner_swsse_ee_u8.cpp line 802 `vf = _mm_subs_epu8(vf, *pvScore);`
inserted `rfgape = ((__m128i)__builtin_ia32_pshuflw ((__v8hi)rfgape, 0));` before aligner_swsse_ee_u8.cpp line 806 `cmp = _mm_movemask_epi8(vtmp);`
deleted aligner_swsse_ee_u8.cpp line 807 `nfixup++;`
disabled aligner_swsse_ee_u8.cpp line 1283 IF `(!btnstack_.empty())`

Figure 15: Evolved Solution. Best of generation 200 mutant changes 39 lines in six Bowtie2 source files.

Weight	Mutation	Source file	line	type	Original Code	New Code
999	replaced	bt2_io.cpp	622	for2	`i < offsLenSampled`	`i < this->_nPat`
1000	replaced	sa_rescomb.cpp	50	for2	`i < satup_->offs.size()`	`0`
1000	disabled	sa_rescomb.cpp	69	for2	`j < satup_->offs.size()`	
100	replaced	aligner_swsse_ee_u8.cpp	707		`vh = _mm_max_epu8(vh, vf);`	`vmax = vlo;`
1000	deleted	aligner_swsse_ee_u8.cpp	766		`pvFStore += 4;`	
1000	replaced	aligner_swsse_ee_u8.cpp	772		`_mm_store_si128(pvHStore, vh);`	`vh = _mm_max_epu8(vh,`
1000	deleted	aligner_swsse_ee_u8.cpp	778		`ve = _mm_max_epu8(ve, vh);`	

Figure 16: Minimised evolved solution (Figure 15). After unneeded changes have been removed, this patch changes 7 lines, in three C++ source files. This patched version of Bowtie2 is 77 times faster on average than the released version on short DNA sequences generated by the Broad Institute's Solexa next generation scanner.

sumptions about typical patterns of use but a specific implementation may turn out to be inefficient when used in unanticipated ways. Risco-Martin *et al.* [65] showed grammatical evolution [66] is able to evolve efficient heap managers for specific circumstances. They also considered non-functional requirements, like power consumption.

Sipper *et al.* used GP to improve existing Java programs (symbolic regression [31], artificial ant on the Santa Fe trail [67], separating intertwined spirals, array sum and tic-tac-toe [4]). Their initial population is seeded [68] with Java byte code [69] from an existing poor program.

Wes Weimer's work on using GP to automatically fix bugs [21] is increasingly well known. Most of this work is at the level of C source code but his group has also shown bugs can be fixed in lower levels [70]. They have also used GP to improve GPU shaders [71]. Rinard's group have also used non-evolutionary ways to make non-semantic preserving transformations to GPU kernels [72]. Other recent work on evolving fixes for bugs includes [73] and [74].

Andrea Acuri and David White (e.g. [5]) considered not only automatic bug fixing [26] but also have shown GP can improve programs. They looked at several well known software engineering benchmarks, including triangle, sort, factorial, remainder, switch 10 and select. (Select is the largest at 94 LOC.) They showed GP can improve the source code and find optimisations which the compiler was unable to find. For example, they showed GP reduced the number of instructions executed by factorial (in Java) by 87.4%. They have also showed GP can optimise non-functional properties. E.g. evolving a pseudo random number generator (PRNG) which trades "randomness" (as measured by information entropy) against reduction in power consumption.

In [3] we showed that non-trivial amounts of code can be automatically created with a test based fitness function and starting from a BNF grammar that constrains the code to be legal, compilable, executable and terminate. We chose the unix gzip compression utility and demonstrated the evolution of the longest_match routine within it. longest_match is responsible for finding duplicated strings within the files that gzip compresses and is not only vital to gzip's operation but consumes the vast bulk of the CPU time. Although not huge, longest_match is definitely not trivial. Its source code is about two pages of C code. GP was asked to evolve a replacement for longest_match which ran on different hardware (an nVidia graphics card) and slightly different programming language (nVidia's CUDA). (See Figure 17.) The BNF grammar was derived from an example supplied by nVidia of how to write CUDA and the fitness test cases were generated by instrumenting gzip and then running it on the SIR test suite [75]. Although not all the resulting one and half million test cases were used during evolution, the final evolved code has been tested on all of them and others. It has been subjected to manual inspection and run back-to-back with the original C code. No discrepancy has been found.

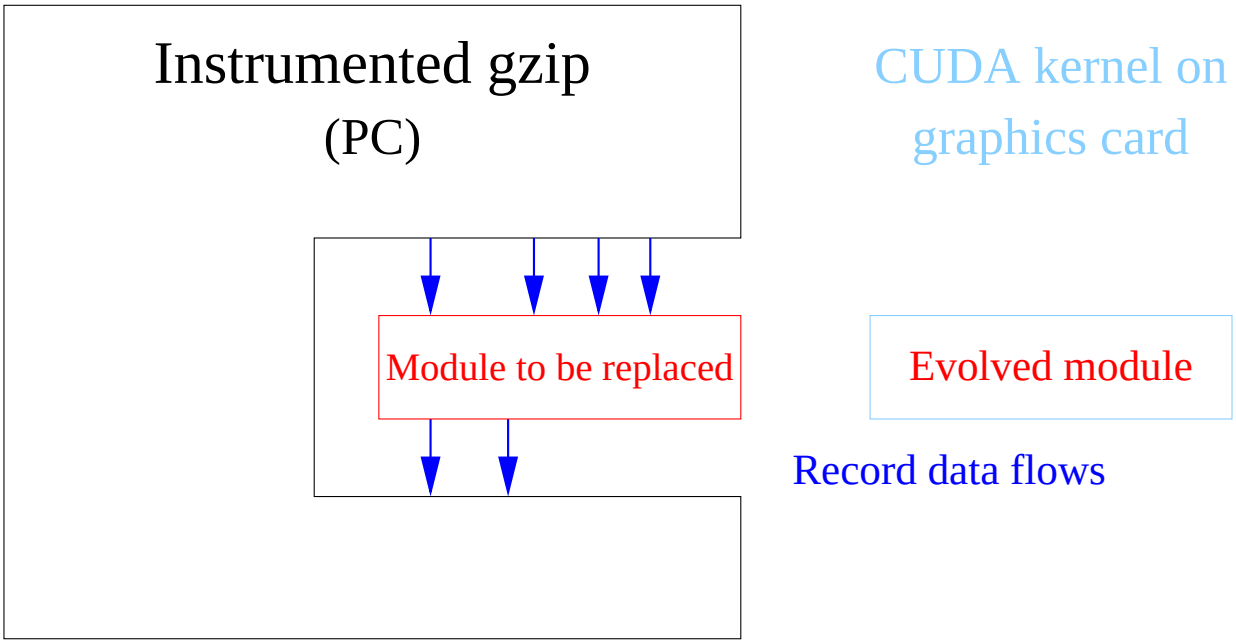

Figure 17: Evolving a replacement module of gzip. Existing code (left) used to specify correct output for each set of inputs to the module to be replaced. Replacement module was evolved with fitness given by similarity between evolved answer and correct answer. After 55 generations a functionally identical module written in CUDA was evolved.

GP offers a potential way of moving existing applications to mobile platforms [76] where not only is the hardware radically different but so too are user interfaces and expectations. Compiler based re-optimisation in not sufficient but evolutionary computation may be able to provide the more radical changes to the program sources needed.

6.2 Less Explored areas for Evolving Software

Another potential software application is to use evolutionary computation to produce diverse variants of programs. (Indeed there has already been some work in this direction [20].) While multiplicity computing [8] currently considers a few different versions of programs, GP can already produced populations of variants. Whilst some of the population are not suitable for use, if evolution is allowed to continue after it finds the first solution, in subsequent generations the population can contain hundreds of solutions [3]. Obviously the number of program variants could be increased still further. Additional non-functional criteria might be introduced into fitness selection. E.g. to ensure a certain minimum level of variation [77] or to ensure mutants were not too extreme. In some cases it might even be possible to ensure every user had their own variant of the program. Uses of diverse software might include resilience, both to attack and faults, and watermarking.

Software product lines [78] present a slightly different need. Instead of wanting different versions of the same program, software product lines considers ways to create functionally different versions of a program to meet different customer needs. This might be for customers who speak different languages (internationalisation) but also covers things like embedded controllers in similar but not identical hardware. E.g. a deluxe microwave oven with many features not available in the economy version. Both have the same controller chip but it needs different software in the two cases. Current approaches mostly consider only enabling and disabling parts of the source code but in future these parts might form the components of more flexible systems glued together by evolutionary computing.

7 Discussion

The GISMOE approach requires the program to be optimised to be tested. Non-trivial programs may take a long time to run. However during evolution fitness measurement does not need to be precise, all that we require is to be able to tell a good variation from a bad one. Indeed evolution can even cope with some noise in this choice. We have already started to explore ways of limiting the volume of testing used during optimisation. Future work may also consider ways of reducing the resources consumed by testing by deliberately targeting testing at the most recently introduced patches. This can build on test case prioritisation work.

With many mature software systems there is a large volume of existing tests which might be used by GSIMOE. In other cases, GISMOE might be extended to use automated test case generation. As the existing system can be used as its de facto specification, the GISMOE approach side steps the "test oracle problem" by using the existing code to specify a "correct" answer for every test case, simply by running it on that test case. This is true whether the test case already exists or is newly automatically generated. We have not yet tried using GISMOE with non-deterministic or buggy systems.

If a line of code is unused, any effort spent on improving it is wasted. Unlike a traditional compiler, we can keep track of resource consumption by particular parts of the code and so concentrate optimisation on resource hungry lines of code.

8 Conclusions

Section 3.6 (page 9) reports a new type of power law relationship in the number of repetitions of lines of code, rather than in the number of low level syntactic elements or the size of repeats (known as code clones). We find here the number of exact repetitions follows a Zipf's law.

Considerable manual effort is needed to create programs. Today even identifying new operating points for tools that are suitable for new circumstances or new user requirements is labour intensive and few can afford to even explore more than one possibility by hand. Automated software production offers the prospect of exploring complete Pareto trade-off surfaces, for example, between functionality and speed.

With this in mind for the first time we have evolved specific improvements to substantial multi-file C++ code using a fitness function which compares the output of new code with that of the old to ensure it maintains or improves functionality and improves non-functional requirements. On out of sample examples, the evolved version of Bowtie2 on average on the targeted data yields slightly better answers and is more than 70 times faster.

Acknowledgements

I would like to thank Ben Langmead and Sean Davis.

Funded by EPSRC grant EP/I033688/1.

References

[1] Andrea Arcuri, David Robert White, John Clark, and Xin Yao, "Multi-objective improvement of software using co-evolution and smart seeding," in *Proceedings of the 7th International Conference on Simulated Evolution And Learning (SEAL '08)*, Xiaodong Li, Michael Kirley, Mengjie Zhang, David G. Green, Victor Ciesielski, Hussein A. Abbass, Zbigniew Michalewicz, Tim Hendtlass, Kalyanmoy Deb, Kay Chen Tan, Jürgen Branke, and Yuhui Shi, Eds., Melbourne, Australia, Dec. 7-10 2008, vol. 5361 of *Lecture Notes in Computer Science*, pp. 61–70, Springer.

[2] Mark Harman, William B. Langdon, Yue Jia, David R. White, Andrea Arcuri, and John A. Clark, "The GISMOE challenge: Constructing the pareto program surface using genetic programming to

find better programs," in *The 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 12)*, Essen, Germany, Sept. 3-7 2012, ACM.

[3] W. B. Langdon and M. Harman, "Evolving a CUDA kernel from an nVidia template," in *2010 IEEE World Congress on Computational Intelligence*, Pilar Sobrevilla, Ed., Barcelona, 18-23 July 2010, pp. 2376–2383, IEEE.

[4] Michael Orlov and Moshe Sipper, "Flight of the FINCH through the Java wilderness," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 2, pp. 166–182, Apr. 2011.

[5] David R. White, Andrea Arcuri, and John A. Clark, "Evolutionary improvement of programs," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 4, pp. 515–538, Aug. 2011.

[6] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee, *A field guide to genetic programming*, Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008, (With contributions by J. R. Koza).

[7] David R. White, *Genetic Programming for Low-Resource Systems*, Ph.D. thesis, Department of Computer Science, University of York, Dec. 2009.

[8] Cristian Cadar, Peter Pietzuch, and Alexander L. Wolf, "Multiplicity computing: a vision of software engineering for next-generation computing platform applications," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, Kevin Sullivan, Ed., Santa Fe, New Mexico, USA, 7-11 Nov. 2010, FoSER '10, pp. 81–86, ACM.

[9] Mark Harman and Bryan F. Jones, "Search based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, Dec. 2001.

[10] F. G. Freitas and J. T. Souza, "Ten years of search based software engineering: A bibliometric analysis," in 3^{rd} *International Symposium on Search based Software Engineering (SSBSE 2011)*, 10th - 12th September 2011, pp. 18–32.

[11] M. Harman, A. Mansouri, and Y. Zhang, "Search based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, 2012, to appear.

[12] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test-case generation," *IEEE Transactions on Software Engineering*, 2010, To appear.

[13] Wasif Afzal, Richard Torkar, and Robert Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, June 2009.

[14] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman, "Search based requirements optimisation: Existing work and challenges," in *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'08)*, Montpellier, France, 2008, vol. 5025, pp. 88–94, Springer LNCS.

[15] O. Räihä, "A survey on search–based software design," *Computer Science Review*, vol. 4, no. 4, pp. 203–249, 2010.

[16] Mark Harman, "Software engineering meets evolutionary computation," *Computer*, vol. 44, no. 10, pp. 31–39, Oct. 2011, Cover feature.

[17] Wasif Afzal and Richard Torkar, "On the application of genetic programming for software engineering predictive modeling: A systematic review," *Expert Systems with Applications*, vol. 38, no. 9, pp. 11984–11997, 2011.

[18] William B. Langdon, Mark Harman, and Yue Jia, "Efficient multi-objective higher order mutation testing with genetic programming," *Journal of Systems and Software*, vol. 83, no. 12, pp. 2416–2430, Dec. 2010.

[19] Stefan Wappler and Joachim Wegener, "Evolutionary unit testing of object-oriented software using strongly-typed genetic programming," in *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, Eds., Seattle, Washington, USA, 8-12 July 2006, vol. 2, pp. 1925–1932, ACM Press.

[20] Robert Feldt, "Generating diverse software versions with genetic programming: an experimental study," *IEE Proceedings - Software Engineering*, vol. 145, no. 6, pp. 228–236, Dec. 1998, Special issue on Dependable Computing Systems.

[21] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan.-Feb. 2012.

[22] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *34th International Conference on Software Engineering (ICSE 2012)*, Martin Glinz, Ed., Zurich, June 2-9 2012, pp. 3–13.

[23] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest, "Automatically finding patches using genetic programming," in *International Conference on Software Engineering (ICSE) 2009*, Stephen Fickas, Ed., Vancouver, May 16-24 2009, pp. 364–374.

[24] Andrea Arcuri, *Automatic software generation and improvement through search based techniques*, Ph.D. thesis, School of Computer Science, University of Birmingham, UK, Aug. 2009.

[25] Andrea Arcuri, "On the automation of fixing software bugs," in *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, Leipzig, Germany, 2008, pp. 1003–1006, ACM, Doctoral symposium session.

[26] Andrea Arcuri and Xin Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *2008 IEEE World Congress on Computational Intelligence*, Jun Wang, Ed., Hong Kong, 1-6 June 2008, IEEE Computational Intelligence Society, IEEE Press.

[27] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biology*, vol. 10, no. 3, pp. R25, 2009.

[28] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman, "Gapped BLAST and PSI-BLAST a new generation of protein database search programs," *Nucleic Acids Research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[29] Ben Langmead and Steven L Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, 4 March 2012.

[30] W. B. Langdon, "Genetic improvement of programs," in *18th International Conference on Soft Computing, MENDEL 2012*, Radomil Matousek, Ed., Brno, Czech Republic, 27-29 June 2012, Brno University of Technology, Invited keynote.

[31] John R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, USA, 1992.

[32] William B. Langdon, *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, vol. 1 of *Genetic Programming*, Kluwer, Boston, 1998.

[33] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in 3^{rd} *International Workshop on Search-Based Software Testing (SBST 2010)*, Paris, France, April 2010.

[34] Elaine J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, Nov. 1982.

[35] W. B. Langdon, "A many threaded CUDA interpreter for genetic programming," in *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, Anna Isabel Esparcia-Alcazar, Aniko Ekart, Sara Silva, Stephen Dignum, and A. Sima Uyar, Eds., Istanbul, 7-9 Apr. 2010, vol. 6021 of *LNCS*, pp. 146–158, Springer.

[36] Sidney R. Maxwell III, "Experiments with a coroutine model for genetic programming," in *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, Orlando, Florida, USA, 27-29 June 1994, vol. 1, pp. 413–417a, IEEE Press.

[37] Astro Teller, "Genetic programming, indexed memory, the halting problem, and other curiosities," in *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium*, Pensacola, Florida, USA, May 1994, pp. 270–274, IEEE Press.

[38] Ben Langmead, *Bowtie 2*, 2.0.0-beta7 edition, 2012, Accessed 2 Aug 2012.

[39] Mark Harman, Yue Jia, and William B. Langdon, "Strong higher order mutation-based test data generation," in *8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011)*, Andreas Zeller, Ed., Szeged, Hungary, September 5-9 2011, pp. 212–222, ACM.

[40] Mark Gabel and Zhendong Su, "A study of the uniqueness of source code," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, Santa Fe, New Mexico, USA, 7-11 Nov. 2010, pp. 147–156, ACM.

[41] Hongyu Zhang, "Exploring regularity in source code: Software science and Zipf's law," in *15th Working Conference on Reverse Engineering, WCRE'08*, Antwerp, Belgium, oct. 2008, pp. 101–110, IEEE.

[42] George Kingsley Zipf, *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*, Addison-Wesley Press Inc., Cambridge 42, MA, USA, 1949.

[43] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos, "Power laws in software," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 1, pp. 2:1–2:26, Oct. 2008.

[44] William B. Langdon and Wolfgang Banzhaf, "Repeated sequences in linear genetic programming genomes," *Complex Systems*, vol. 15, no. 4, pp. 285–306, 2005.

[45] W. B. Langdon and W. Banzhaf, "Repeated patterns in genetic programming," *Natural Computing*, vol. 7, no. 4, pp. 589–613, Dec. 2008.

[46] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *International Conference on Software Maintenance (ICSE'98)*, 1998, pp. 368–377.

[47] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, "CCFinder: A multi-linguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 6, pp. 654–670, 2002.

[48] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[49] Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsushita, "KClone: A proposed approach to fast precise code clone detection," in 3^{rd} *International Workshop on Software Clones (IWSC'09)*, Kaiserslautern, Germany, March 2009.

[50] Mark Harman, "Automated patching techniques: The fix is in," *Communications of the ACM*, vol. 53, no. 5, pp. 108, June 2010.

[51] David Orvosh and Lawrence Davis, "Shall we repair? genetic algorithms combinatorial optimization and feasibility constraints," in *Proceedings of the 5th International Conference on Genetic Algorithms*, Stephanie Forrest, Ed., University of Illinois at Urbana-Champaign, 17-21 July 1993, p. 650, Morgan Kaufmann.

[52] William B. Langdon, Terry Soule, Riccardo Poli, and James A. Foster, "The evolution of size and shape," in *Advances in Genetic Programming 3*, Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, Eds., chapter 8, pp. 163–190. MIT Press, Cambridge, MA, USA, June 1999.

[53] T.F. Smith and M.S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[54] Richard M. Durbin, *et al.*, "A map of human genome variation from population-scale sequencing," *Nature*, vol. 467, no. 7319, pp. 1061–1073, 28 Oct 2010.

[55] Jose Javier Dolado, "A validation of the component-based method for software size estimation," *IEEE Transactions on Software Engineering*, vol. 26, no. 10, pp. 1006–1021, Oct. 2000.

[56] Maria Cláudia Figueiredo Pereira Emer and Silvia Regina Vergilio, "GPTesT: A testing tool based on genetic programming," in *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, Eds., New York, 9-13 July 2002, pp. 1343–1350, Morgan Kaufmann Publishers.

[57] Peter Martin, "Genetic programming for service creation in intelligent networks," in *Genetic Programming, Proceedings of EuroGP'2000*, Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, Eds., Edinburgh, 15-16 Apr. 2000, vol. 1802 of *LNCS*, pp. 106–120, Springer-Verlag.

[58] Pablo Rodriguez-Mier, Manuel Mucientes, Manuel Lama, and Miguel I. Couto, "Composition of web services through genetic programming," *Evolutionary Intelligence*, vol. 3, no. 3-4, pp. 171–186, 2010.

[59] Lidia Yamamoto and Christian F. Tschudin, "Experiments on the automatic evolution of protocols using genetic programming," in *Autonomic Communication, Second International IFIP Workshop, WAC 2005, Revised Selected Papers*, Ioannis Stavrakakis and Michael Smirnov, Eds., Athens, Greece, Oct. 2-5 2005, vol. 3854 of *Lecture Notes in Computer Science*, pp. 13–28, Springer.

[60] Thomas Weise and Ke Tang, "Evolving distributed algorithms with genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 16, no. 2, pp. 242–265, Apr. 2012.

[61] Daniar Hussain and Steven Malliaris, "Evolutionary techniques applied to hashing: An efficient data retrieval method," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, Eds., Las Vegas, Nevada, USA, 10-12 July 2000, p. 760, Morgan Kaufmann.

[62] Patrick Berarducci, Demetrius Jordan, David Martin, and Jennifer Seitzer, "GEVOSH: Using grammatical evolution to generate hashing functions," in *Proceedings of the Fifteenth Midwest Artificial Intelligence and Cognitive Sciences Conference, MAICS 2004*, Eric G. Berkowitz, Ed., Chicago, USA, Apr. 16-18 2004, pp. 31–39, Omnipress.

[63] Cesar Estebanez, Julio Cesar Hernandez-Castro, Arturo Ribagorda, and Pedro Isasi, "Evolving hash functions by means of genetic programming," in *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, Eds., Seattle, Washington, USA, 8-12 July 2006, vol. 2, pp. 1861–1862, ACM Press.

[64] Jan Karasek, Radim Burget, and Ondrej Morsky, "Towards an automatic design of non-cryptographic hash function," in *34th International Conference on Telecommunications and Signal Processing (TSP 2011)*, Budapest, 18-20 Aug. 2011, pp. 19–23.

[65] Jose L. Risco-Martin, David Atienza, J. Manuel Colmenar, and Oscar Garnica, "A parallel evolutionary algorithm to optimize dynamic memory managers in embedded systems," *Parallel Computing*, vol. 36, no. 10-11, pp. 572–590, 2010, Parallel Architectures and Bioinspired Algorithms.

[66] Michael O'Neill and Conor Ryan, "Grammatical evolution," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 4, pp. 349–358, Aug. 2001.

[67] W. B. Langdon and R. Poli, "Why ants are hard," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, Eds., University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998, pp. 193–201, Morgan Kaufmann.

[68] W. B. Langdon and J. P. Nordin, "Seeding GP populations," in *Genetic Programming, Proceedings of EuroGP'2000*, Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, Eds., Edinburgh, 15-16 Apr. 2000, vol. 1802 of *LNCS*, pp. 304–315, Springer-Verlag.

[69] Eduard Lukschandl, Magus Holmlund, and Eirk Moden, "Automatic evolution of Java bytecode: First experience with the Java virtual machine," in *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming*, Riccardo Poli, W. B. Langdon, Marc Schoenauer, Terry Fogarty, and Wolfgang Banzhaf, Eds., Paris, France, 14-15 Apr. 1998, pp. 14–16, CSRP-98-10, The University of Birmingham, UK.

[70] Eric Schulte, Stephanie Forrest, and Westley Weimer, "Automated program repair through the evolution of assembly code," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, Antwerp, 20-24 Sept. 2010, pp. 313–316, ACM.

[71] Pitchaya Sitthi-amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence, "Genetic programming for shader simplification," *ACM Transactions on Graphics*, vol. 30, no. 6, pp. article:152, Dec. 2011, Proceedings of ACM SIGGRAPH Asia 2011.

[72] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *SIGSOFT FSE*, Szeged, 5-9 September 2011, pp. 124–134.

[73] Josh L. Wilkerson and Daniel Tauritz, "Coevolutionary automated software correction," in *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, Juergen Branke, Martin Pelikan, Enrique Alba, Dirk V. Arnold, Josh Bongard, Anthony Brabazon, Juer-

gen Branke, Martin V. Butz, Jeff Clune, Myra Cohen, Kalyanmoy Deb, Andries P Engelbrecht, Natalio Krasnogor, Julian F. Miller, Michael O'Neill, Kumara Sastry, Dirk Thierens, Jano van Hemert, Leonardo Vanneschi, and Carsten Witt, Eds., Portland, Oregon, USA, 7-11 July 2010, pp. 1391–1392, ACM.

[74] Thomas Ackling, Bradley Alexander, and Ian Grunert, "Evolving patches for software repair," in *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, Natalio Krasnogor, Pier Luca Lanzi, Andries Engelbrecht, David Pelta, Carlos Gershenson, Giovanni Squillero, Alex Freitas, Marylyn Ritchie, Mike Preuss, Christian Gagne, Yew Soon Ong, Guenther Raidl, Marcus Gallager, Jose Lozano, Carlos Coello-Coello, Dario Landa Silva, Nikolaus Hansen, Silja Meyer-Nieberg, Jim Smith, Gus Eiben, Ester Bernado-Mansilla, Will Browne, Lee Spector, Tina Yu, Jeff Clune, Greg Hornby, Man-Leung Wong, Pierre Collet, Steve Gustafson, Jean-Paul Watson, Moshe Sipper, Simon Poulding, Gabriela Ochoa, Marc Schoenauer, Carsten Witt, and Anne Auger, Eds., Dublin, Ireland, 12-16 July 2011, pp. 1427–1434, ACM.

[75] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of 16th International Conference on Software Engineering, ICSE-16*, May 1994, pp. 191–200.

[76] Alban Cotillon, Philip Valencia, and Raja Jurdak, "Android genetic programming framework," in *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012*, Alberto Moraglio, Sara Silva, Krzysztof Krawiec, Penousal Machado, and Carlos Cotta, Eds., Malaga, Spain, 11-13 Apr. 2012, vol. 7244 of *LNCS*, pp. 13–24, Springer Verlag.

[77] John R. Koza, Forrest H Bennett III, and Oscar Stiffelman, "Genetic programming as a Darwinian invention machine," in *Genetic Programming, Proceedings of EuroGP'99*, Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, Eds., Goteborg, Sweden, 26-27 May 1999, vol. 1598 of *LNCS*, pp. 93–108, Springer-Verlag.

[78] Hassan Gomaa and Michael E. Shin, "Automated software product line engineering and product derivation," in *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, jan. 2007, p. 285a.