SSBSE 2013, Guenther Ruhe and Yuanyuan Zhang *eds.*, Saint Petersburg, 24-26 Aug.

Applying Genetic Improvement to MiniSAT

Justyna Petke, William B. Langdon, Mark Harman

CREST Centre, University College London,
Gower Street, London,
WC1E 6BT,
United Kingdom

Abstract. Genetic Programming (GP) has long been applied to several SBSE problems. Recently there has been much interest in using GP and its variants to solve demanding problems in which the code evolved by GP is intended for deployment. This paper investigates the application of genetic improvement to a challenging problem of improving a well-studied system: a Boolean satisfiability (SAT) solver called MiniSAT. Many programmers have tried to make this very popular solver even faster and a separate SAT competition track has been created to facilitate this goal. Thus genetically improving MiniSAT poses a great challenge. Moreover, due to a wide range of applications of SAT solving technologies any improvement could have a great impact. Our initial results show that there is some room for improvement. However, a significantly more efficient version of MiniSAT is yet to be discovered.

Keywords: Genetic Improvement, GISMOE, SAT

1 Introduction

Genetic improvement [2,8,9,12,15] seeks to use SBSE to automatically improve programs according to one or more fitness function. Typically, an evolutionary algorithm has been used based on genetic programming [2,12,15] or a hybrid of genetic programming and other techniques [8,9].

This paper investigates the application of genetic improvement to a challenging problem of improving the MiniSAT [5] system. This is a significant challenge, because MiniSAT has been iteratively improved over many years by expert human programmers, to address the demand for more efficient SAT solvers and also in response to repeated calls for competition entries in the MiniSat hack track of SAT competitions [1].

We therefore chose MiniSAT because it represents one of the most stringent challenges available for automated genetic improvement using SBSE. Our goal is to investigate the degree to which genetic improvement can automatically improve a system that has been very widely and well studied and for objectives that have been repeatedly attacked by expert humans.

We report initial results of experiments aimed at the genetic improvement of MiniSAT using the GISMOE approach to genetic improvement [8]. Our primary findings are that one can achieve a more efficient version of MiniSAT by simply getting rid off assertions and statements related to statistical data. Moreover, deleting certain optimisations leads to faster runs on some SAT instances. However, a significantly more efficient version of the system is yet to be discovered.

2 MiniSAT

MiniSAT is a well-known open-source C++ solver for Boolean satisfiability problems (SAT). It implements the core technologies of modern SAT solving, including: unit propagation, conflict-driven clause learning and watched literals [13], to name a few. The solver has been widely adopted due to its efficiency, small size and availability of ample documentation. It is used as a backend solver in several other tools, including Satisfiability Modulo Theories (SMT) solvers, constraint solvers and solvers for deciding Quantified Boolean Formulae (QBF) . MiniSAT has also served as a reference solver in SAT competitions.

In the last few years progress in SAT solving technologies involved only minor changes to the solvers' code. Thus in 2009 a new track has been introduced into the SAT competition, called MiniSAT hack track. In order to enter this track one needs to modify the code of MiniSAT. This solver has been improved by many expert human programmers over the years, thus we wanted to see how well an automated approach scales. We used genetic improvement in order to find a more efficient version of the solver. In our experiments we used the latest version of the solver - MiniSAT-2.2.0[1].

3 Our Approach to the Genetic Improvement of MiniSAT

Our objective is to find a version of MiniSAT that is correct, i.e answers whether an instance is satisfiable or not, and that is faster than the unmodified solver. We used test cases from SAT competitions[2]. The training test suite was divided into five groups: satisfiable/unsatisfiable instances on which MiniSAT runs for less than 1 second, satisfiable/unsatisfiable instances on which MiniSAT runs for between 1 and 10 seconds and a mixture of satisfiable and unsatisfiable SAT instances on which MiniSAT runs for between 10 and 20 seconds.

We modified the SAT solver at the level of source code. We used a specialised BNF grammar to make sure that the evolved code is syntactically correct. Thus individuals produced have a good chance of compiling and thus high chances of running. We used time-outs to force termination of individuals which run significantly longer than the unmodified solver. We changed the code by using three operations:

- *copy* : copies a line of code in another place,
- *replace* : replaces a line of code with another line of code,
- *delete* : deletes a line of code.

There were a few special cases involving loops and `if` conditions, namely the same three operations (*copy*, *replace* and *delete*) were applied to conditions in `if` statements, `while` and `for` loops[3].

[1] Solver available at: `http://minisat.se/MiniSat.html`.

[2] Instances available at: `http://www.satcompetition.org/`.

[3] Note here, however, that a part of a `for` loop, for instance, could have only been replaced with the same part of another `for` loop. For instance, '$i++$' could have been replaced with '$j++$', but not '$j = 0$'.

We modified two C++ files: Solver.cc, containing the core solving algorithm (321 out of 582 lines of code), and SimpSolver.cc, which simplifies the input instance (327 out of 480 lines of code).

Furthermore, we were evolving a list of changes, that is, a list of *copy*, *replace* and *delete* instructions. We only kept such lists in memory, instead of multiple copies of an evolved source code.

For each generation the top half of the population was selected. These were either mutated, by adding some of the three operations mentioned above, or crossover was applied, which simply merged two lists of changes together. Mutation and crossover took place with 50% probability each. New individuals were created by selecting one of the three mutation operations.

For each generation five problems were randomly chosen from the five groups of test cases. Fitness was evaluated as follows: if correct answer was returned by an individual, 2 points were added; if, additionally, the modified program was faster, 1 more point was added. Only individuals with 10 or more points were considered for selection. In order to avoid environmental factors, we counted the number of lines used to establish whether a mutated program was more efficient than the original one. The whole process is presented in Figure 1.

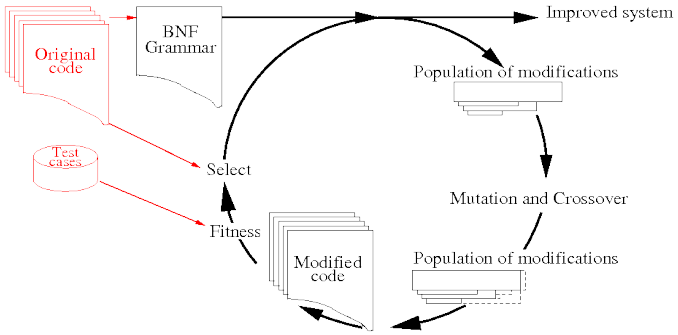

Fig. 1. GP improvement of MiniSAT.

4 Initial Results

A summary of our results is shown in Table 1. We refer to versions of MiniSAT that run faster than the unmodified solver on the maximum set of instances as 'best individuals'.

We ran our experiments on a test suite with 71 test cases taken from the 2011 SAT competition. Each generation contained 20 individuals. Time limit was set to 25 seconds and it took 14 hours to produce 100 generations. We only modified the Solver.cc file, containing the core solving algorithm. Of all programs generated 73% of them compiled. The best one was more efficient than the unmodified solver on 70 SAT instances, in terms of lines of code used. However, the modified versions mostly just removed assertions as well as some statistical data. Some optimisations have also been deleted, but these in turn led to longer runtimes on certain instances.

Next, we selected the test cases from only the application tracks of SAT competitions. MiniSAT was able to find an answer for 107 problems out of 500 instances tested

Test cases	Type	Population size	Generations	Compiles	Improved	Best improvement
71	various	20	100	73%	70	0.937%
107	application	20	100	73%	107	0.859%
107	application	100	20	66%	106	0.858%

Table 1. Results of genetically improving MiniSAT. The 'Improved' column shows the number of test cases on which the best generated version of the solver was more efficient. The 'Best improvement' column shows the highest decrease in lines used for some test case, not necessarily achieved by the best individual.

within the time limit, which was set to 25 seconds for each instance. Therefore, the 107 SAT problems were used for GP. Again we set population size to 20 and the number of generations to 100 and around 73% of individuals compiled. In 34 generations there was an individual that was more efficient than the unmodified solver on all five randomly selected test cases. The best one was more efficient on all 107 instances. However, it only removed assertions and operations on variables used for statistical purposes. We also ran the experiments with population size 100 for 20 generations and achieved similar results (with the exception that 66% of modified programs compiled). In all cases the number of lines used by a 'better' version of MiniSAT generated was less by at most 1% and the average number of lines used during each solver run was in the order of 10^{10}. None of the individuals produced led to large performance improvements. Most of the changes involved deletion of assertions, operations used for gathering statistical data or deletion of minor optimisations.

To sum up, in our experiments genetic improvement has mostly found ways to pare down MiniSAT implementation. This was achieved by removing non-essential code like assertions. Another type of change performed by GP was removal of minor optimisations. We will provide an example: A SAT instance is composed of constraints called *clauses*, hence SAT solvers try to find a variable assignment that satisfies all the clauses. MiniSAT contains a function called `satisfied` that checks the satisfiability of a clause and removes it from the database if it's already satisfied by some variable assignment that cannot be changed. GP disabled this function by setting the second part of the main `for` loop to zero. Thus, during a run of such a modified solver at each variable assignment all clauses were checked for satisfiability, even though some of them could have already been satisfied. On the other hand, the main body of the `satisfied` function was not executed.

5 Related Work

Genetic Programming (GP) has long been applied to several SBSE problems including project management and testing.

More recently, there has been much interest in using GP and variants and hybrids of GP to solve demanding problems in which the code evolved by GP is intended for deployment, rather than merely as a source of decision support (but not ultimate deployment as a working software system). Much of the recent upsurge in interest in GP can be traced back to the seminal work of Arcuri and Yao on bug fixing using GP [3] and the development of this agenda into practical, scalable systems for automated program repair [11,14]. Recent results indicate that these automated repairs may prove to be as

maintainable as human generated patches [6] and that the patches can be computed cheaply using cloud computing [7].

While previous work on bug fixing has already scaled to large real world systems, work on whole system genetic improvement has not previously scaled as well. However, recently Langdon and Harman [10] demonstrated scalability of whole program genetic improvement for a system of 50,000 Lines of Code on a real-world bioinformatics system. They were able to use a GP hybrid to find new evolved versions of the DNA sequence analysis system Bowtie2 that are, on average, 70 times faster than the original (and semantically slightly improved) when applied to DNA sequences from the 1,000 genome dataset.

A general framework of genetic improvement in set out on the ASE 2012 keynote paper by Harman et al. [8]. In the work reported here we adapt the approach developed by Langdon and Harman [10] applied to the Bowtie2 system to seek to optimise the MiniSAT system. Any improvements for implementations of SAT solving that we are able to achieve may have benefits for the wide and diverse applications of SAT solving. Even if we can only optimise a SAT solver for a subdomain of application (such as all constraints of a particular type), then this may allow us to use genetic improvement to achieve a kind of partial evaluation [4]. Such partial evaluation of SAT solving by genetic improvement may be useful in specific applications for which a known subset of formulae of the desired type are prevalent.

6 Conclusions and Future Work

Genetic improvement has successfully been applied to systems such as Bowtie2, leading to significant speed-ups. Therefore, we wanted to investigate if this could be achieved on a well-known software system that is easy to analyse and has been engineered by many expert human programmers. Hence we chose MiniSAT, a very popular Boolean satisfiability (SAT) solver that has been thoroughly studied. MiniSAT hack track of SAT competitions was specifically designed to encourage people to make minor changes to MiniSAT code that could lead to significant runtime improvements, and hence some new insights into SAT solving technology. We wanted to check how an automated approach scales.

If Genetic Programming (GP) is allowed to only apply mutations and crossover at the level of lines of source code, it turns out that little can be done to improve the current version of MiniSAT. Most changes simply pare down MiniSAT implementation. These involve deletion of assertions as well as operations used for producing statistical data. Some minor optimisations have also been removed by GP. A version of the solver that is significantly more efficient than the unmodified MiniSAT solver is yet to be discovered. We intend to conduct further experiments. We plan to remove assertions from the GP process and also conduct mutations on smaller constructs than a line of code. One possibility is to mutate mathematical expressions. Further experiments with varying population and generation size are also desirable. Furthermore, using a certain type of test cases, exhibiting, for instance, similar structure, could help find improvements specific for such classes of problems. We have already started experiments in this direction by considering test cases from the application tracks of SAT competitions. However, other classes of problems are yet to be investigated.

References

1. MiniSAT hack competition, 2013. In 2013 this is part of the 16th International Conference on Theory and Applications of Satisfiability Testing.
2. A. Arcuri, D. R. White, J. A. Clark, and X. Yao. Multi-objective improvement of software using co-evolution and smart seeding. In X. Li, M. Kirley, M. Zhang, D. G. Green, V. Ciesielski, H. A. Abbass, Z. Michalewicz, T. Hendtlass, K. Deb, K. C. Tan, J. Branke, and Y. Shi, editors, 7^{th} *International Conference on Simulated Evolution and Learning (SEAL 2008)*, volume 5361 of *Lecture Notes in Computer Science*, pages 61–70, Melbourne, Australia, December 2008. Springer.
3. A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In J. Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.
4. D. Bjørner, A. P. Ershov, and N. D. Jones. *Partial evaluation and mixed computation*. North–Holland, 1987.
5. N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004.
6. Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis (ISSTA'12)*, Minneapolis, Minnesota, USA, July 2012. To appear.
7. C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland, 2012.
8. M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The GISMOE challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In 27^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, Essen, Germany, September 2012.
9. W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In P. Sobrevilla, editor, *2010 IEEE World Congress on Computational Intelligence*, pages 2376–2383, Barcelona, 18-23 July 2010. IEEE.
10. W. B. Langdon and M. Harman. Genetically improving 50000 lines of C++. Research Note RN/12/09, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK, 19 Sept. 2012.
11. C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.
12. M. Orlov and M. Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, Apr. 2011.
13. J. P. M. Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.
14. W. Weimer, T. V. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE 2009)*, pages 364–374, Vancouver, Canada, 2009.
15. D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, 2011.