Scalability of the Coevolutionary Automated Software Correction System

Josh L. Wilkerson
Natural Computation Laboratory
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri, U.S.A.
jwilkerson@acm.org

Daniel R. Tauritz
Natural Computation Laboratory
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri, U.S.A.
dtauritz@acm.org

ABSTRACT

The Coevolutionary Automated Software Correction system addresses in an integral and fully automated manner the complete cycle of software artifact testing, error location, and correction phases. It employs a coevolutionary approach where software artifacts and test cases are evolved in tandem. The test cases evolve to better find flaws in the software artifacts and the software artifacts evolve to better behave to specification when exposed to the test cases, thus causing an evolutionary arms race. Experimental results are presented which demonstrate the scalability of the Coevolutionary Automated Software Correction system by establishing correlations between program size and both success rate and estimated convergence rate that are at most linear.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging; I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search

General Terms: Algorithms, Experimentation

Keywords: Automated Debugging, Repair, Coevolution, Genetic Programming, Search-Based Testing

1. INTRODUCTION

For a given software artifact, testing, locating the errors identified, and correcting those errors is a critical and time consuming process in software development. The Coevolutionary Automated Software Correction (CASC) system [4] views the problem of correcting a given software artifact as a search in the space of all software artifacts with a starting point of the given software artifact and a goal point of the desired corrected version. Finding high-quality test cases is a very hard problem in and of itself, so to perform a sampling biased towards high-quality test cases can be viewed as another search problem. These two searches are interdependent. Namely, the fitness of a software artifact is dependent on the test case sample and vice versa. This relationship is implemented using competitive coevolution. The test cases evolve to better find flaws in the software artifacts and the software artifacts evolve to better behave to specification when exposed to the test cases, thus causing an evolutionary arms race with as ultimate result a corrected version of the software artifact originally fed into the system. CASC exploits the reduced complexity of the fitness function relative to the software artifact to be corrected.

The search space increases exponentially with program length, which, if not reduced, limits CASC's practicality. A significant reduction can be made through the assumption that the buggy source program is not dramatically different from the correct program [3]. This limits the search to neighborhood modifications, which results in a polynomial search space [1]. Additional assumptions can be made to further reduce the search space; however, each assumption decreases CASC's applicability. The empirical study presented in this paper investigates how the number of atomic code elements in a program affects both CASC's success rate and estimated convergence rate.

GECCO'11, July 12–16, 2011, Dublin, Ireland.
ACM 978-1-4503-0690-4/11/07.

2. CASC OVERVIEW

The initial program population is based on the buggy source program, which is parsed and transformed into an evolvable parse tree. The CASC parser supports nearly all features of the C++ programming language. The general section of code containing the bug must be enclosed by two specific guard statements, designating the code block as evolvable code to the system (this code section will be referred to as the evolvable section of the seed program). Initial population diversity is achieved by mutating the source program. The CASC test case generation module generates test cases in the form of one or more lists of values, in a method appropriate for the problem being addressed. CASC also allows for the seeding of test case values, which are guaranteed preservation throughout the evolutionary process.

New programs are produced by applying one of five GP operators: copy, reset, crossover, mutation, and architecture alteration, with selection chance based on a user supplied probability distribution; however, the probabilities for mutation and crossover are adaptively updated by the system. Compile time errors are checked for in the event that a mutation or crossover resulted in an incorrect program. Test case reproduction uses 100% crossover rate. Uniform crossover is used with a 10% bias towards the fitter parent. The mutation rate is 10%. The reproduction operators are applied until the specified number of offspring have been created.

In order to maximize population exposure, each individual is executed against all opponents in the competing population. Repeat program-test case pairings are not re-executed; the results of such pairings are retrieved from a lookup table (implemented as a hash table). The remaining pairings are divided and executed in parallel on a computing cluster

employing MPI. Each computing node runs the executions concurrently in threads. Executions are scored using the fitness function for the problem. Run time errors and program timeouts are monitored during execution. If any errors occur, the offending program is assigned an arbitrarily low fitness. The overall fitness for each individual is calculated as the average of all the pairings from the current generation.

$(\mu + \lambda)$ survival selection is performed using an inverse fitness proportional k-tournament. This process is repeated until the populations are at the specified sizes. CASC terminates after a specified number of generations are completed.

3. EXPERIMENTS

A buggy version of bubble sort presented in [2] as bug 4 was used as the base program in the presented experiments. Nine versions of this program were created in which each version has an additional line included between the guard statements marking the buggy code. Table 1 shows the node counts N for the generated source programs. A linear trend line can be generated for the N values with an R^2 value of 0.9807, indicating that problem size increases at a linear rate as the lines increase.

Table 1: Program Node Count and Success Rate

Lines	1	2	3	4	5	6	7	8	9
N	4	11	15	23	37	49	53	58	62
Success Pctg	100	98	88	78	68	76	64	62	60

For each version of the source program, 50 experimental runs were executed. Each experiment executed 400 generations in which each population contained 50 individuals. 50 new programs and 25 new test cases were created each generation. Selection tournament size for programs was 5, while all other tournament sizes were 10. The probability distribution of genetic operators for program reproduction were copy: 2.5%, reset: 5%, arch. alter: 2.5%, crossover: 45%, mutation: 45%; crossover and mutation were updated every 10 generations with a 2.5% addition to the probability of the more successful of the two at the cost of the other. The configuration values used are based on succes values found in literature and hand tuned for CASC.

The results for the experiments are shown in Table 1 and Figure 1. For the experiment success rate linear and logarithmic trend lines can be generated with R^2 values of 0.9076 and 0.9075, respectively. Similarly, linear and logarithmic trend lines can be generated for the average birth generation of valid solutions whose R^2 values are 0.7978 and 0.8050, respectively. Based on these results it can be hypothesized that the relationships between success rate and problem size and between estimated convergence rate and problem size are at most linear and very possibly sub-linear.

Figure 1 gives an overview of the distribution of birth generations for valid solutions (based on the minimum and maximum generations observed and first, second, and third quartiles). The fourth line added to the evolvable section is a branch statement determining if the other three lines would be executed, implying that modifications made to the fourth line could completely undermine changes made to the other three lines. This addition made convergence take approximately four times as long in 75% of the experiments, as can be seen in Figure 1. This observation indicates future considerations that need to be taken in the study of CASC's scalability as well as possible additions to CASC to account for similar situations. The performance of the eight line experiment (and the six line, to a lesser degree) is notably different from the apparent trend in the results. These results indicate a need for additional experimentation in order to be certain statistical stability has been achieved.

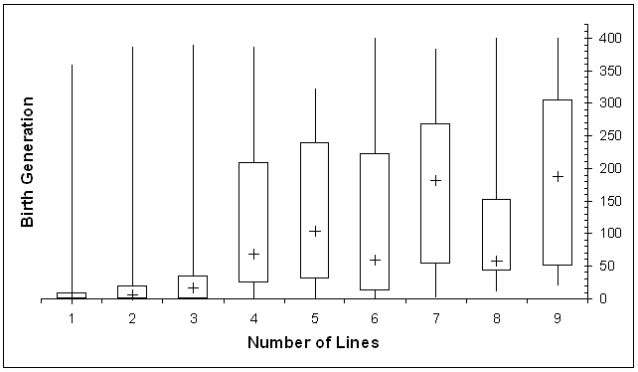

Figure 1: Box Plot of Solution Birth Generations for Successful Experiments

4. CONCLUSIONS AND FUTURE WORK

The results of an empirical study into the impact that problem size has on CASC's success rate and estimated convergence rate are presented in this paper. These results show that problem size has at worst a linear effect on both the success rate and the estimated convergence rate of the CASC system. Results are also presented that show that problem size may have a sub-linear effect on these performance measures. Additional experimentation is needed to be certain of the statistical stability of the results, as well as to further investigate the possibility of sub-linear relationships between problem size and success and estimated convergence rates. Also, the effect that various statement types in the evolvable section have on the CASC system needs to be investigated.

5. ACKNOWLEDGMENTS

This work was funded by the Missouri S&T Intelligent Systems Center.

6. REFERENCES

[1] A. Arcuri. *Automatic software generation and improvement through search based techniques.* PhD thesis, University of Birmingham, 2009.

[2] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of IEEE CEC 2008*, pages 162-168, June 2008.

[3] R. DeMillo, R. Lipton, and F. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*. 11(4):34-71, 1978.

[4] J. Wilkerson and D. Tauritz. Coevolutionary Automated Software Correction. In *Proceedings of GECCO 2010*, pages 1391-1392, 2010.