TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

School of Mathematics, Statistics and Computer Science
Computer Science

Numerical-Node Building Block Analysis of Genetic Programming with Simplification

Phillip Wong, Mengjie Zhang

Technical Report CS-TR-06/15
December 2006

School of Mathematics, Statistics and Computer Science
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

School of Mathematics, Statistics and Computer Science
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Numerical-Node Building Block Analysis of Genetic Programming with Simplification

Phillip Wong, Mengjie Zhang

Technical Report CS-TR-06/15
December 2006

Abstract

This paper investigates the effects on building blocks of using simplification in a GP system to combat the problem of code bloat. The evolved genetic programs are simplified online during the evolutionary process using algebraic simplification rules and hashing techniques. A simplified form of building block (numerical-nodes) are tracked throughout several individual GP runs both when using and not using simplification. The results suggest that simplification disrupts existing potential building blocks during the evolution process. However, the results also suggest that simplification is capable of creating new building blocks which are used to form a more accurate solution than the standard GP. The effectiveness of GP systems utilising simplification can be correlated to the creation of these new building blocks.

keywords Simplification, building blocks, genetic programming, numerical-nodes

Author Information

Phillip Wong is a postgraduate student and mengjie zhang is an academic staff member in computer science in the school of mathematics, statistics and computer science, victoria university of wellington, new zealand.

1 Introduction

Genetic Programming (GP) ([13], [14]) is a relatively young form of evolutionary computing derived from genetic algorithms, whereupon programs are automatically constructed to perform a specific task using processes and operators derived from genetic biology. These genetic operators (e.g. crossover, mutation, selection, reproduction) are repeatedly invoked on an initial population of randomly generated genetic programs until some form of termination criteria is satisfied. The best performing program evolved in the evolutionary process (usually the best program in the final generation) is used as the system's solution.

One of the current problems in GP is that of *code bloat* ([13], [4], [21], [15]). Genetic programs tend to grow very quickly in size, easily out-pacing the improvements in program fitness as evolution progresses. This is because increasing amounts of code being formed is *redundant*, and does not provide any contribution towards the programs fitness measure. As a simple example of redundant code, consider the program `(+ x (- y y))`. The `(- y y)` subtree can be regarded as redundant, as it provides no extra functionality than if the subtree were removed. Programs containing a lot of these redundancies can be both inefficient in execution and harder to understand. As the programs grow exponentially, they can quickly exhaust a system's available resources, slowing down the evolution process and can even halt the system before a "good" solution can be found. This may limit GP's ability to scale to more complex problems requiring larger-sized solutions.

Simplification is one approach to combating code bloat. Simplification works by directly processing and removing redundant code from programs within a GP system. The *editing function* proposed in [13] is an example of a simplification component, intended to improve comprehensibility and execution efficiency of the final solution at the end of the evolutionary process. But, since code bloat is a problem that occurs throughout the evolutionary process, simplification is invoked *during* the evolutionary process. Simplification has been tested on several symbolic regression ([10], [5], [24]) and multi-class classification ([25], [24]) tasks. These early results have shown that simplification can significantly improve the efficiency of the GP system, producing smaller sized programs and requiring shorter training times to find solutions. It has also been found that GP systems are capable of producing more effective/accurate solutions when simplification is employed. [24] hypothesises that the simplification may destroy potential building blocks early on, but can also provide new building blocks at later stages of the GP process, which allows simplification to improve in effectiveness. However, these hypotheses have not yet been seriously examined through experimentation.

It is becoming an ever increasing thought in biology (Epigenetics [16], [12]) that there can be changes in a gene's phenotype (cell function) without change to its genotype (cell DNA). Similarly, it may be that the redundant code present in genetic programs serves a purpose other than purely functional. It has been theorised that these redundancies may shield programs from the destructive effects of GP crossover ([17], [22]) by reducing the chances of crossover/mutation points being selected within a critical portion of a program. The redundancies may also contain valuable building blocks that can contribute in the future (through recombination) to form better solutions. The removal of these redundancies may eliminate these potential building blocks from the GP population, denying further generations access to them, and thus hindering the GP system's ability to find a good solution.

1.1 Goals

This paper aims to investigate and analyse the effects on GP building blocks, of using online simplification during the evolutionary process (proposed in [24]). We will achieve this by tracking building blocks throughout several individual GP runs both with and without simplification, and then examining their behaviour through these runs. More specifically, we will look to do the following:

- Investigate whether utilising a simplification component in a GP system disrupts potential building blocks in the GP population.
- Determine whether simplification creates new building blocks in a GP population with which to form better solutions.
- Determine if the creation of new building blocks can overcome the negative effects of building blocks being disrupted, and result in more accurate GP solutions.

1.2 Structure

The remainder of this paper is structured as follows. Section 2 describes the simplification method used in the building block analysis. Section 3 discusses the brief history of building blocks in GP and the form of building blocks analysed in this paper. It also describes the experimentation tasks and GP settings used for this paper. Section 4 presents the results from these experiments, as well as discussions. Finally, section 5 concludes the paper, as well as outlining possible future directions.

2 Simplification

Simplification is the process of directly removing redundancy from a program, leaving behind a smaller program which is semantically the same as the original (yields the same outputs given the same inputs). Several different simplification approaches have been used in the past with varying results ([10], [25], [5], [24]). Since we are investigating the effects of simplification in general, it is beyond the scope of this paper to discuss the merits of one simplification method over another. For the purposes of this paper, we will use an *Algebraic Simplification* method described in GECCO 2006 [24]. A short overview of this simplification system is given in the following subsection.

2.1 Overview of the Simplification Process

The algebraic simplification method of GP programs [24] is motivated by the algebraic nature of the standard GP functions ($+, -, \times, \div$) and uses a set of *algebraic simplification rules* to remove redundancies. These rules are similar to STRIPS operators ([6]), and consist of two parts, a *precondition* which represents the state of the surrounding nodes that must be present for the rule to be applied, and a *postcondition* which represents the state that the surrounding nodes are in after additions and deletions are made. A number of these rules make up the *rule-set* for the simplification system.

The system also makes use of an *Algebraic Equivalence* hashing function to determine whether two subtrees are functionally equivalent. In order to achieve this, a hash value is calculated for each subtree of a program. If two subtrees have the same hash value, they are considered to be equivalent and are treated as such. A *hashing order* (p) is used to denote the total number of possible hash values. Terminals are allocated hash values depending on their type. Floating point constant terminals are transformed into hash values using a *constant precision* parameter, whereas variable terminals are given a random hash value (which is kept constant throughout the evolutionary process). Hash values of child nodes can be easily combined to give the hash value of the parent node, limiting the amount of work needed to calculate the hash value of every subtree.

In order to simplify a program, the rule-set is applied using a "greedy" engine. It recursively traverses the program tree in a postfix bottom-up fashion. For each node it processes in this way, it checks the precondition of each rule in the rule-set. If any of the rules match, then it is

applied to that portion of the tree. It continues to check preconditions until none of the rules in the rule-set can be applied, in which case the algorithm moves on to the next node.

A simple example of the algebraic simplification on a small program is given in figure 1.

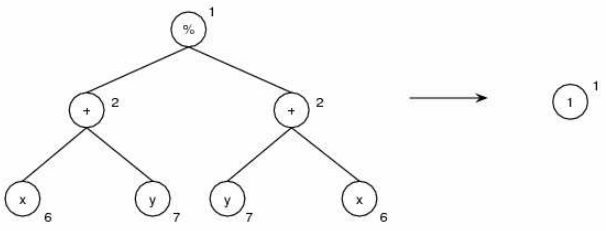

Figure 1: Simple example of the algebraic simplification using the rule: $\frac{A}{A} \rightarrow 1$. The numbers denote example algebraic equivalence hash values in a field of order 11 (the hash order). Note how through hashing `(+ x y)` and `(+ y x)` are deemed equivalent and so the rule can be applied.

In this example, `x` has been randomly assigned the hash value 6, while `y` has been assigned the hash value 7. For the '+' nodes, the children are added together to make 13. Since the hash order is 11, the hash value is calculated to be *13 mod 11 = 2*. For the root node '%' (the same as $\div$), the rule $\frac{A}{A} \rightarrow 1$ is found to apply, since both left and right child nodes have the same hash value. The rule is applied and the result is a single numerical-node, with a value of `1`.

3 Building Block Analysis

The concept of *building blocks* can be traced back to Genetic Algorithms (GA), where it was hypothesised in [8] that *"Short, low order, and highly fit schemata are sampled, recombined [crossed over], and re-sampled to form strings of potentially higher fitness"*. This hypothesis is regarded as the process that GA utilises to solve given problems ([9], [8], [7]), and why it is so powerful in doing so. Conceptually, building blocks are simply small components solution that can be formed into larger, more fit components through the use of genetic operators.

From the outset, there have been several attempts to bring a similar building block theory to the realm of GP ([13], [2], [23], [19], [18]). These have themselves spawned several ways to describe building blocks: a subtree to a solution tree ([1]), a rooted subtree ([20]), a block of code ([11]). Needless to say, it is still difficult for one to define exactly what a "GP building block" is.

3.1 Numerical-Nodes as "Simple" Building Blocks

For this paper, we need to track building blocks throughout a GP run in order to observe their behaviour and how they are affected by simplification. For this purpose, we narrow our focus onto the simplest form of what has generally been considered a building block (the subtree): *Single numerical terminal nodes* (which we term simply as *numerical-nodes*). These are single nodes present in a program which are usually represented by a single floating point number. They are often used in GP tasks to contribute additional numerical constants toward a GP solution. In the standard GP, these are usually regarded as "constant" terminals. In GP systems using simplification, these terminals are no longer constant as they can be altered by the simplification process.

There are many reasons to focus on these numerical-nodes. Numerical-nodes may be combined with other numerical-nodes to build larger subtrees that contribute to a more accurate solution. Like other GP subtrees, numerical-nodes may either contribute towards a solution or merely act as noise, guiding the system away from the solution. Most importantly, they can be combined or removed by the simplification component used in a GP system and as such can possibly be *disrupted*.

Numerical-nodes are also easily individually tracked during a GP run so that their behaviour can be monitored. While larger building blocks could also be tracked, their analysis would be far more difficult. By considering only single-nodes, and additional complexity in analysis that can be introduced by crossover taking place within a tracked building block is removed.

Essentially, numerical-nodes exhibit nearly all the characteristics of larger building blocks, but with reduced complexity, allowing for easier tracking and analysis. They provide a good "first step" into analysing the influence simplification has on GP building blocks. We will consider and analyse more "general" building blocks later in future work.

3.2 Experimental Setup

For initial experimentation, we used a symbolic regression task. Symbolic regression is the process of determining a model that fits a given set of data. In this case, given a set of data values, what mathematical function would best fit those values. This task was chosen because it is a commonly used task and can be tuned to be heavily reliant on numerical-nodes.

The task is made up of 200 data values, uniformly distributed between the values of -10.0 and 10.0, generated using the mathematical polynomial $f(x) = 11x + 50.0$. The coefficients of 11 and 50 were chosen because they are fairly distant from each other and are not multiples of each other. This means that at least two different recombinations of the numerical-nodes are needed to correctly regress this function. A linear (single variable) equation was selected to place more emphasis on the numerical constants, so that the GP system would need to process numerical-nodes more often.

In both the "standard GP" and "GP with simplification" approaches, a tree-based structure is used to represent the genetic programs. The ramped half-and-half method was used for generating the initial programs, and also for the mutation operator ([3]). Proportional selection and reproduction crossover and mutation were used as the genetic operators in the GP process.

The remainder of this section details the aspects and parameters used for the GP systems. The terminal set consists of the single variable x, along with a variable number of randomly generated floating point valued numerical-nodes. Each numerical-node is generated during the initial program population generation, and is chosen from a range of $[-1.0, 1.0]$. The use of a small range (in the context of the symbolic regression problem being used) is intentional, in order to make it clearer if simplification is able to create new numerical-nodes. While in the standard GP, these terminals remain untouched and thus *constant* throughout the evolution process, in the GP system using simplification they may be altered by the simplification system.

$$\text{Terminal Set} = \{x, r_0, r_1, ..., r_n\}.$$

The function set is simply made up of the four arithmetic functions. Addition, subtraction and multiplication have their usual meanings. $\div$ takes the form of *protected* division, which removes the undefined case by defining division by 0 to be 0. Each of these functions takes two arguments and returns a single result.

$$\text{Function Set} = \{+, -, \times, \div\}.$$

For this symbolic regression task, the fitness of a program is determined by measuring the mean squared error between the desired output and the actual output of the program on all patterns of each data set. This results in GP programs with lower fitness being regarded as "fitter". The ideal program would therefore have a fitness of 0 (as negative mean squared errors are unobtainable), meaning that there is no difference between the desired output and the output given by the program for all the 200 data values.

The settings used in the Genetic Programming systems are detailed in table 1. Both the standard GP and GP with simplification share the same parameter values. No early stopping criteria was used in order to monitor each GP system run for the full 100 generations.

Parameter	Value
Mutation	30%
Elitism	10%
Crossover	60%
Population Size	500
Generations	100
Maximum Depth Limit	6

Table 1: Settings/Parameters used for all Genetic Programming systems

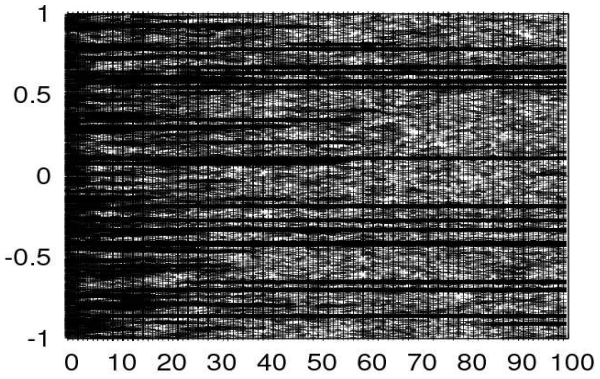

Figure 2: Numerical-node value plot of a GP system run without simplification, re-plotted into the range of $[-1.0, 1.0]$

In addition to the parameters in table 1, GP with simplification requires another four parameters. *Proportion* controls how many programs in the GP population are simplified each time the simplification process is invoked. *Frequency* dictates how often simplification is invoked. A frequency value of 1 means that simplification is invoked *every* generation, while a frequency value of 7 would mean that simplification would be invoked every seventh generation, with 6 standard GP generations in between. *Hash order (p)* for the hashing process is set to 1000077157, and the *constant precision (δ)* is set to 1000000.

4 Results and Discussion

For the results several aspects of the GP systems were recorded and each numerical-node was tracked in each generation of each individual GP run. The values and quantity of these numerical-nodes as well as the mean squared error of the "best" program in each generation, were recorded throughout 10 individual GP runs for each simplification frequency used (without, every 1, every 5 and every 10). The full results of these recordings are given in figures 6, 7 and 8, which are located in the appendices. For the numerical-value graphs (figure 6), each small '+' symbol in each graph represents the value (y-axis) of a numerical-node within the population of programs at the given Generation (x-axis). A small amount of Gaussian random noise has been added to each numerical-node value for these graphs, purely for improved visualisation purposes. This is done so that identically valued numerical-nodes (occurring in the same generation) can both been seen on the same figure. This also allows for highly populated values to show a higher density than lowly populated values. Graphs in figure 7 display the Mean Squared Error of the fittest program in each GP generation, measured for each system for each GP run analysed. Graphs in figure 8 show the number of *distinct* numerical-nodes that are present in each generation of programs during each GP runs.

In figure 6, the simplification frequency is quite apparent in the numerical-node value graphs,

with changes in the numerical-nodes occurring whenever simplification occurs. The numerical-node values in the standard GP are incapable of existing outside of the original $[-1.0, 1.0]$ range and so the node-values are difficult to view in the depicted graphs. A zoomed version of the graph *0A* is shown in figure 2, which shows how initially, the numerical-nodes are uniformly distributed within the $[-1.0, 1.0]$ range. However, during the evolution process numerical-nodes are eliminated during GP selection and mutation, leaving only a few major clusters of numerical-nodes at the end of evolution.

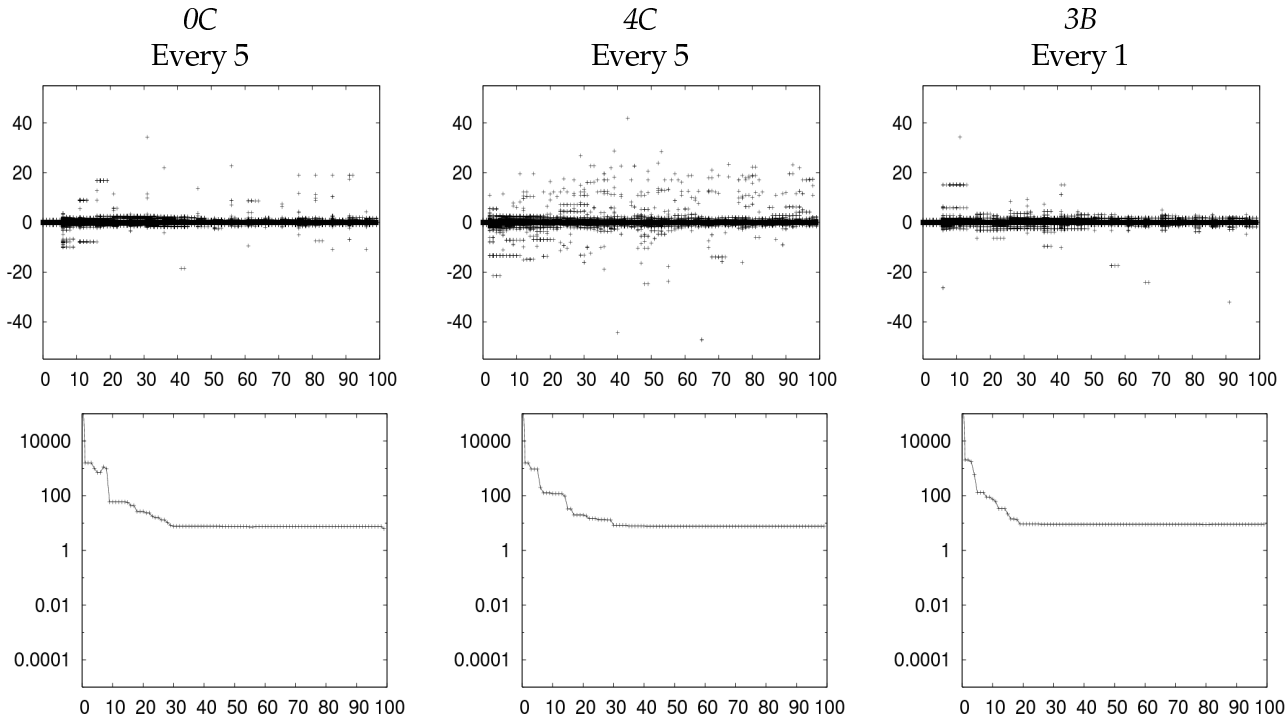

Figure 3: Selected Numerical-node value and Error plots of poorly performing GP systems for $f(x) = 11x + 50$.

4.1 Analysis of the Disruption of Existing Building Blocks

Figure 3 shows numerical-node value plots and corresponding mean squared error plots from a few selected GP system runs which use simplification. These selected runs are typical of those which produced solutions that performed poorly compared to the standard GP system, and even when compared to the other two GP systems using simplification (same initial seed, but different frequencies).

As can be seen from the three numerical-value node graphs (*3B*, *4C* and *0C*) in figure 3, it can be seen that there are no sustained clusters of newly created numerical-nodes. Many of the newly created numerical-nodes only last a few generations before being eliminated from the genetic population. This is because the GP evolution process does not deem programs that are using these new numerical-nodes to be particularly fit solutions, and so eventually these programs are discarded from the population along with the new numerical-nodes. This suggests that the new numerical-nodes being created in these cases are not of great contribution towards the final solution.

The fact that these particular runs perform poorly when compared to the standard GP, even though they are both given the same initial conditions, suggests that any building blocks that are present early on in the evolutionary process are being disrupted by the simplification procedure. Without these building blocks, the GP systems have a harder time forming a "good" solution to

the symbolic regression task, and so perform poorly.

4.2 Analysis of the Construction of New Building Blocks

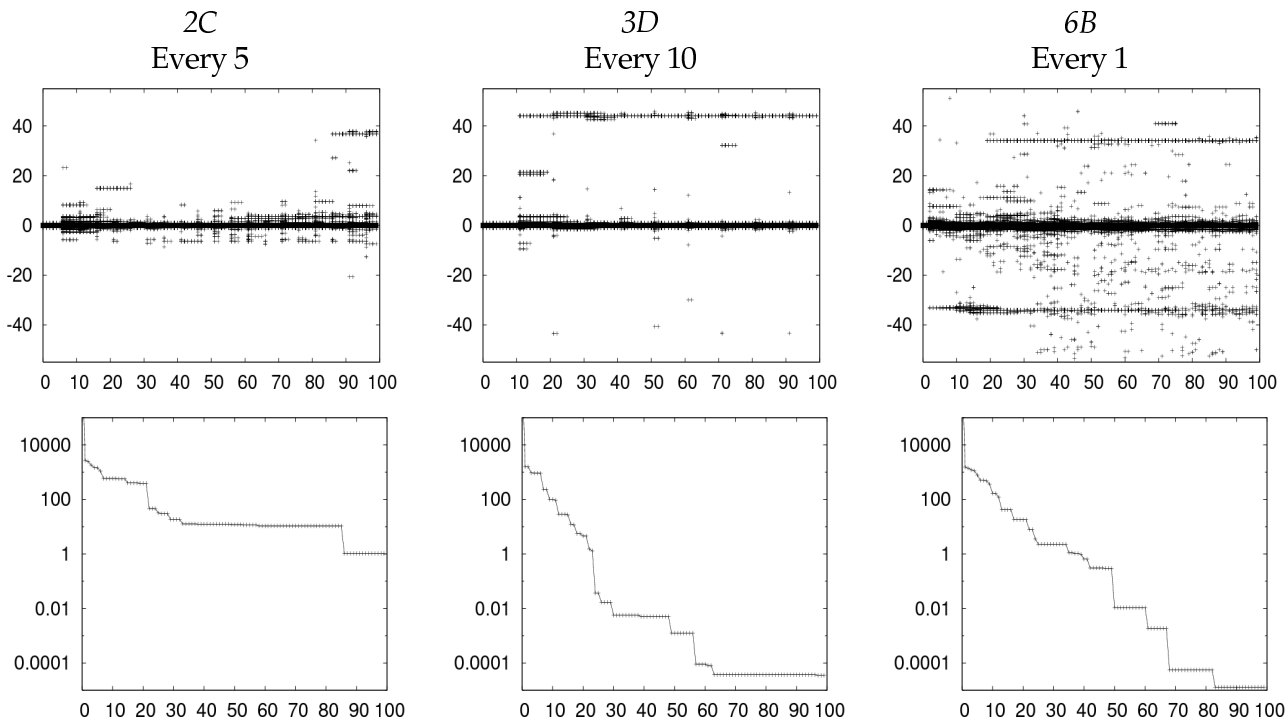

Figure 4: Selected Numerical-node value and Error plots of well performing GP systems for $f(x) = 11x + 50$.

Figure 4 show numerical-node value plots and corresponding mean squared error plots from a few different GP system runs, again using simplification. These cases show typical GP runs where evolved solutions were of superior fitness to those produced by the standard GP system.

As can be seen in the mean squared error graphs in figure 7, it can be seen that systems employing simplification can in most cases outperform the standard GP system. In many of the instances where a system using simplification does produce a more accurate solution, the corresponding numerical-node value graph depicts at least one or more new numerical-nodes being formed (graphs *3D* and *6B* in figure 4 are examples of this). Unlike the new nodes that are created in poorly performing cases, these nodes are sustained over many generations during the run, and often until the end of evolution. This suggests that these new numerical-nodes are contributing toward a more accurate solution, and as such are acting as new *building blocks*. A good example of the correlation between the creation of these new numerical-nodes and an improvement in accuracy is in graph *2C* (figure 4). The error graph for this system's run shows a sudden reduction in error occurring just after 80 generations. This coincides with a new formation of numerical-nodes with values of around 40, which is sustained till the end of the GP run. It is easy to see why a new numerical-node of 40 would be of use when trying to regress the function $11x + 50$, as it greatly contribute to trying to build the value 50.

This behaviour shows that even though simplification disrupts building blocks already existent in the GP population (negatively impacting the potential effectiveness of the GP system), in a majority of cases the GP system is able to use new building blocks constructed by simplification to evolve programs which work just as effectively, or better, than those produced by standard GP. The effect of constructing new building blocks is able to overcome the negative effects of building blocks being disrupted.

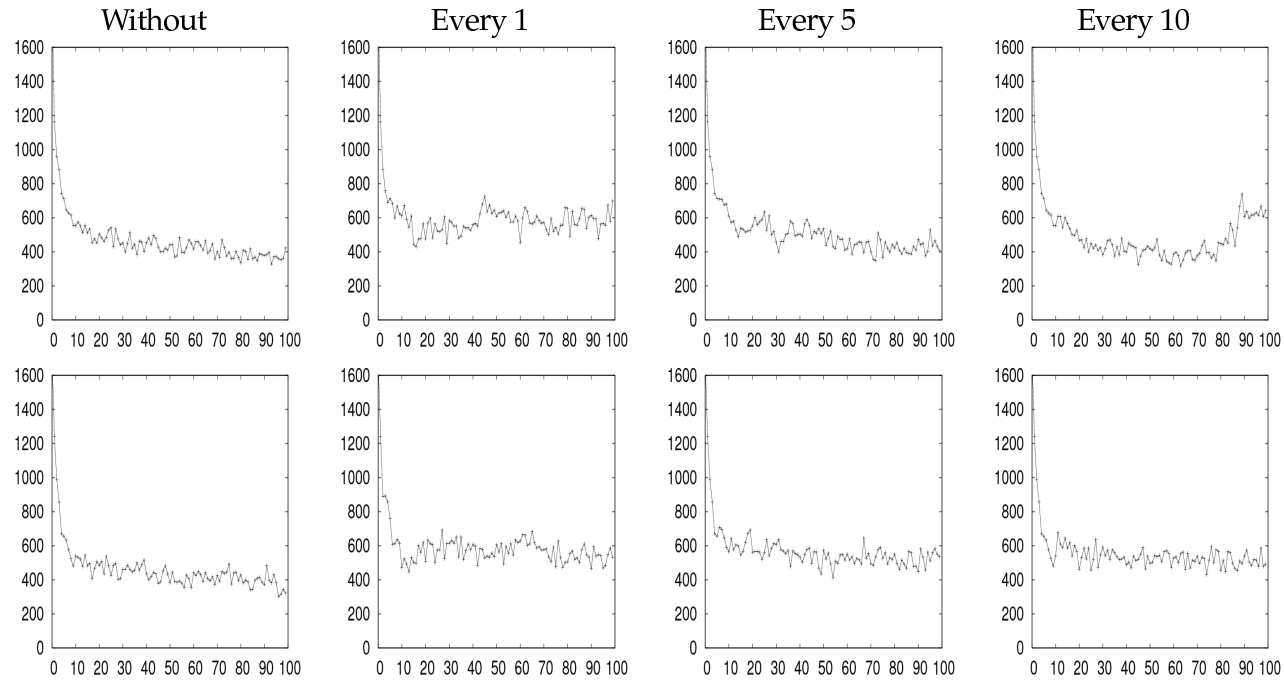

Figure 5: Selected Distinct Numerical-Node Count plots for $f(x) = 11x + 50$.

Figure 5 shows graphs displaying the number of *distinct* numerical-nodes present in each generation of two GP runs. Each of the four graphs in each row are for each of the four systems tested (using the same initial random seed). They show that the trend of all the GP systems is to have a large initial number of numerical-nodes which are quickly refined to a much smaller number by the GP operators. In most of the cases which use simplification, the number of distinct numerical-nodes present throughout the run is actually higher than in the standard GP. This is caused by the simplification procedure's ability to create new numerical-nodes and new building blocks, which in turn creates a more diverse population of numerical-nodes. Even though standard GP has an overall larger pool of nodes available (as the average program size is larger in standard GP), systems using simplification have access to more variations in genetic material. By having a more diverse population of genetic materials available, the GP system is able to access more parts of the of the genetic search space and discover better solutions. This helps explain why GP systems employing simplification are often able to outperform the standard GP system.

5 Conclusions

This paper sought to achieve three goals. The first goal was to determine whether the use of simplification resulted in the disruption of building blocks during a GP system's run. The second goal was to determine whether using simplification could result in new building blocks being created. The third goal was to investigate whether the positive effects of the new constructed building blocks would outweigh the negative effects of having existing building blocks disrupted.

These goals were achieved by tracking *numerical-nodes* during several GP system runs on a heavily constant-dependent symbolic regression task. Both the standard GP system without simplification and GP systems employing a form of Algebraic Simplification were tested, with different frequencies of simplification being used. The values of the numerical-nodes at each generation, as well as a count of distinct numerical-node values and mean squared error were recorded and analysed.

In line with the first goal, it was found that, in general, simplification does in fact disrupt existing building blocks, which can make it more difficult for a system to find a highly fit solution. This can result in some GP systems using simplification performing more poorly than the standard GP.

In investigating the second goal, we found that simplification was able to create new numerical-nodes. While not all of these numerical-nodes were helpful towards creating a solution, there were often clusters of nodes that did contribute. These contributing numerical-nodes constitute new building blocks being created by the simplification system. These new building blocks can often be used to create more accurate solutions than the standard GP.

The final goal aimed to determine whether the benefits from the construction of new building blocks was able to overcome the negative effects of building block disruption, both caused by simplification. The results from all individual runs suggest that in most cases, new building blocks were able to be constructed, and GP systems using simplification were able to recover from the disruption of building blocks and even outperform the standard GP. There were only a few cases when new building blocks were unable to be constructed, and those systems tended to perform poorly. This suggests that the performance of a GP system using simplification is reliant on whether these new building blocks are created during the evolution process.

While analysis on numerical-nodes has provided some insight into the effects of simplification on GP building blocks, it is clear that these interactions need to be further investigated. More general and more complex forms of building blocks need to be analysed to see if trends exhibited by the numerical-node building blocks are continued.

References

[1] Lee Altenberg. The evolution of evolvability in genetic programming. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, pages 47–74. MIT Press, 1994.

[2] Lee Altenberg. The Schema Theorem and Price's Theorem. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 23–49, Estes Park, Colorado, USA, 31 –2 1994 1995. Morgan Kaufmann.

[3] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[4] Tobias Blickle and Lothar Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany, 1994. Max-Planck-Institut für Informatik (MPI-I-94-241).

[5] Aniko Ekart. Shorter fitness preserving genetic programs. In *AE '99: Selected Papers from the 4th European Conference on Artificial Evolution*, pages 73–83, London, UK, 2000. Springer-Verlag.

[6] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[7] Stephanie Forrest and Melanie Mitchell. Relative building-block fitness and the building-block hypothesis. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 109–126. Morgan Kaufmann, San Mateo, CA, 1993.

[8] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[9] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[10] Dale Hooper and Nicholas S. Flann. Improving the accuracy and robustness of genetic programming through expression simplification. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 428, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[11] Hitoshi Iba and Hugo de Garis. Extending genetic programming with recombinative guidance. pages 69–88, 1996.

[12] Bird A Jaenisch R. Epigenetic regulation of gene expression: how the genome integrates intrinsic and environmental signals. *Nature Genetics*, 33 (Suppl):245–254, March 2003.

[13] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, December 1992.

[14] John R. Koza. Genetic programming: On the programming of computers by means of natural selection. *Statistics and Computing*, 4(2), 1994.

[15] W. B. Langdon. Quadratic bloat in genetic programming. In Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 451–458, Las Vegas, Nevada, USA, 10-12 2000. Morgan Kaufmann.

[16] Joshua Lederberg. The meaning of epigenetics. *The Scientist Magazine*, 15(18):page 6, September 2001.

[17] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 6–22, Tahoe City, California, USA, 9 July 1995.

[18] Una-May O'Reilly. *An analysis of genetic programming*. PhD thesis, 1995. Adviser-Franz Oppacher.

[19] Riccardo Poli and W. B. Langdon. An experimental analysis of schema creation, propagation and disruption in genetic programming. In Thomas Back, editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 18–25, Michigan State University, East Lansing, MI, USA, 19-23 1997. Morgan Kaufmann.

[20] Justinian P. Rosca. Analysis of complexity drift in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 286–294, Stanford University, CA, USA, 13-16 1997. Morgan Kaufmann.

[21] Terence Soule, James A. Foster, and John Dickinson. Code growth in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223, Stanford University, CA, USA, 28–31 1996. MIT Press.

[22] Matthew J. Streeter. The root causes of code growth in genetic programming. In *Genetic Programming: 6th European Conference, EuroGP 2003, Essex, UK, April 14-16, 2003. Proceedings*, pages 443–454. Springer-Verlag, 2003.

[23] P. A. Whigham. A schema theorem for context-free grammars. In *1995 IEEE Conference on Evolutionary Computation*, volume 1, pages 178–181, Perth, Australia, 29 - 1 1995. IEEE Press.

[24] Phillip Wong and Mengjie Zhang. Algebraic simplification of gp programs during evolution. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 927–934, New York, NY, USA, 2006. ACM Press.

[25] Mengjie Zhang, Yun Zhang, and William D. Smart. Program simplification in genetic programming for object classification. In Rajiv Khosla, Robert J. Howlett, and Lakhmi C. Jain, editors, *Knowledge-Based Intelligent Information and Engineering Systems, 9th International Conference, KES 2005, Proceedings, Part III*, volume 3683 of *Lecture Notes in Computer Science*, pages 988–996, Melbourne, Australia, September 14-16 2005. Springer.

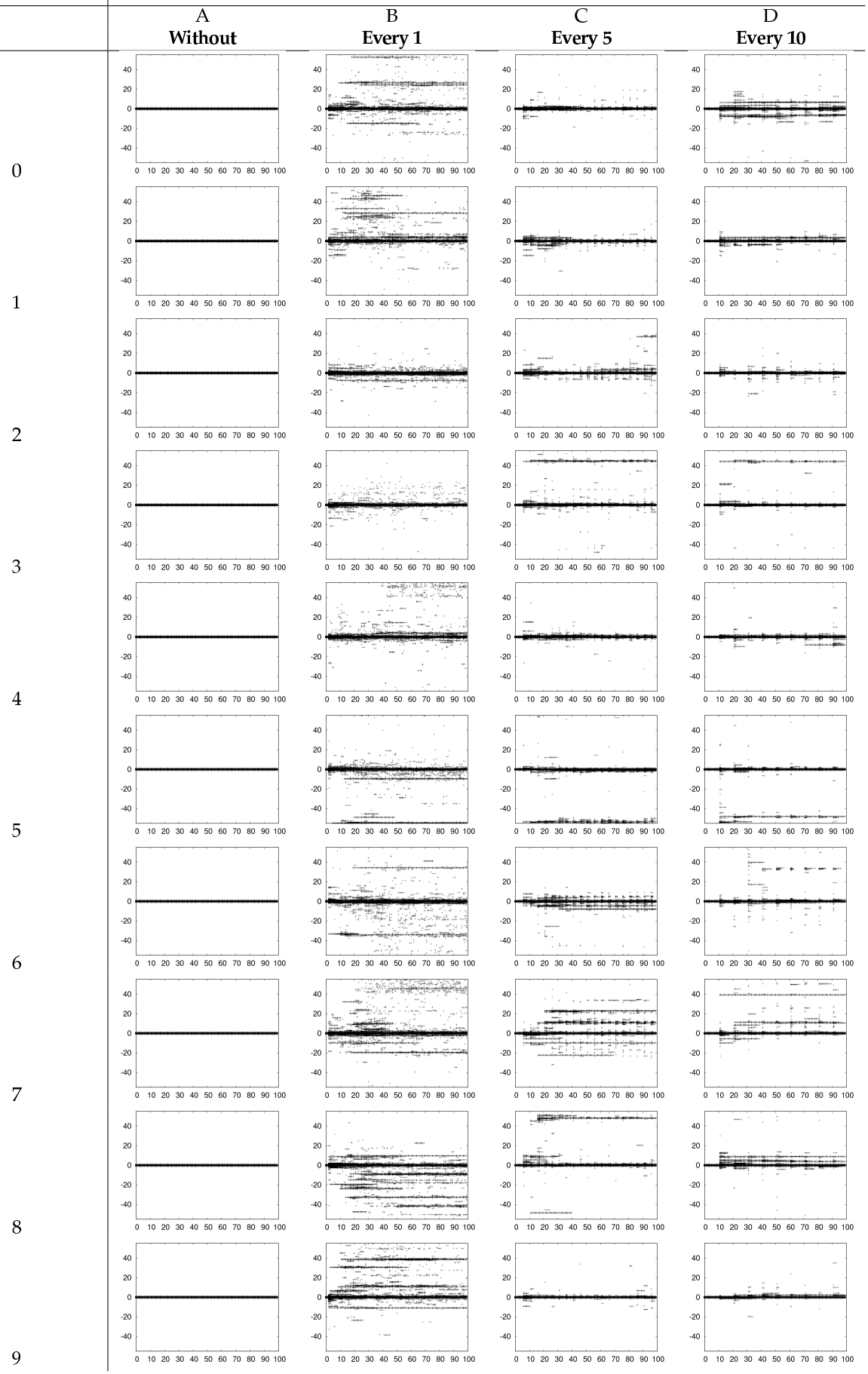

Figure 6: Scatter-gram plot of Numerical Constant Values vs. Generation for the symbolic regression problem $f(x) = 11x + 50$. Performed for 10 individual runs using four different settings for simplification frequency: (a) Without, (b) Every generation, (c) Every 5 generations and (d) Every 10 generations.

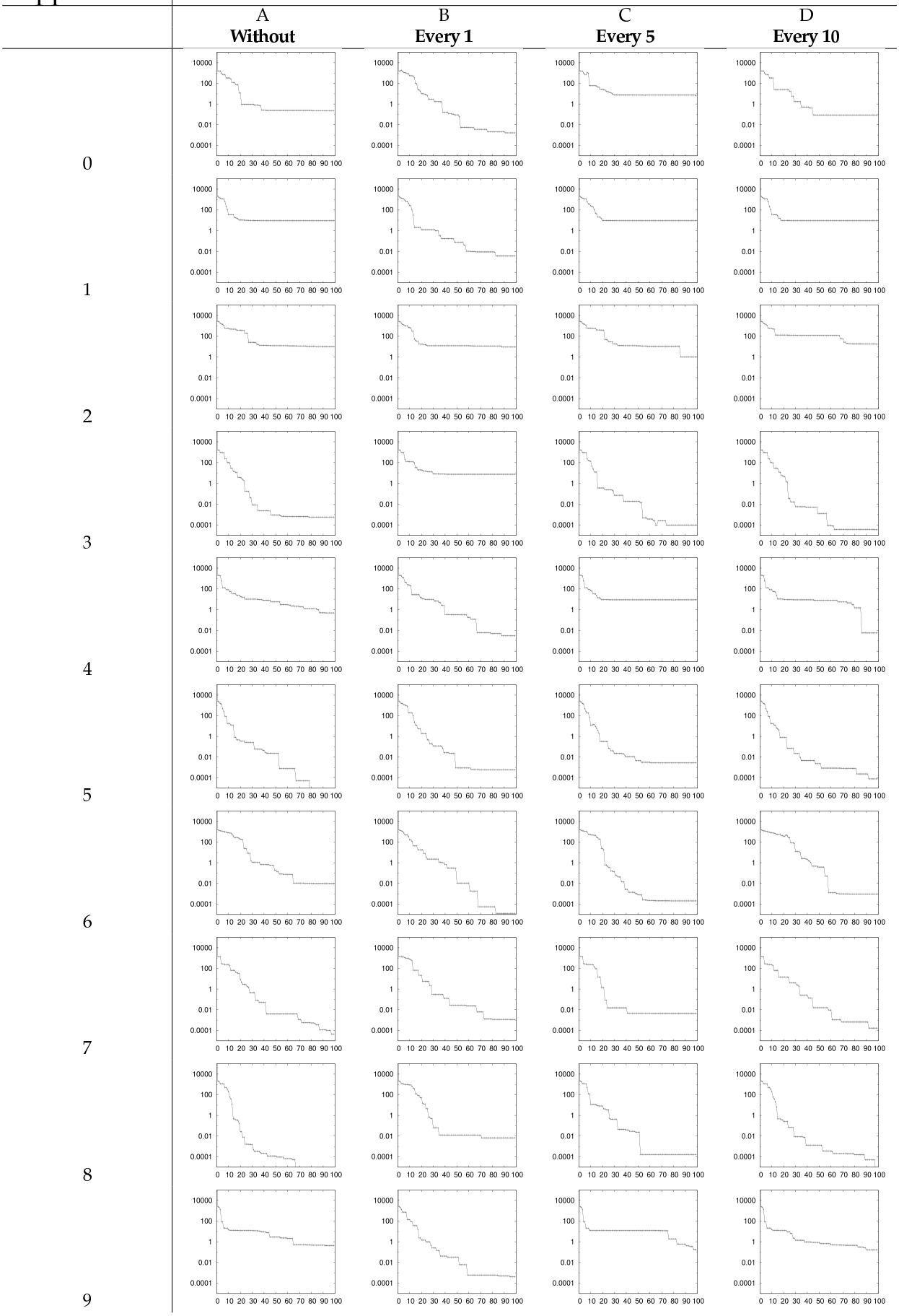

Figure 7: Line plot of Mean Squared Error vs. Generation for the symbolic regression problem $f(x) = 11x + 50$. Performed for 10 individual runs using four different settings for simplification frequency: (a) Without, (b) Every generation, (c) Every 5 generations and (d) Every 10 generations.

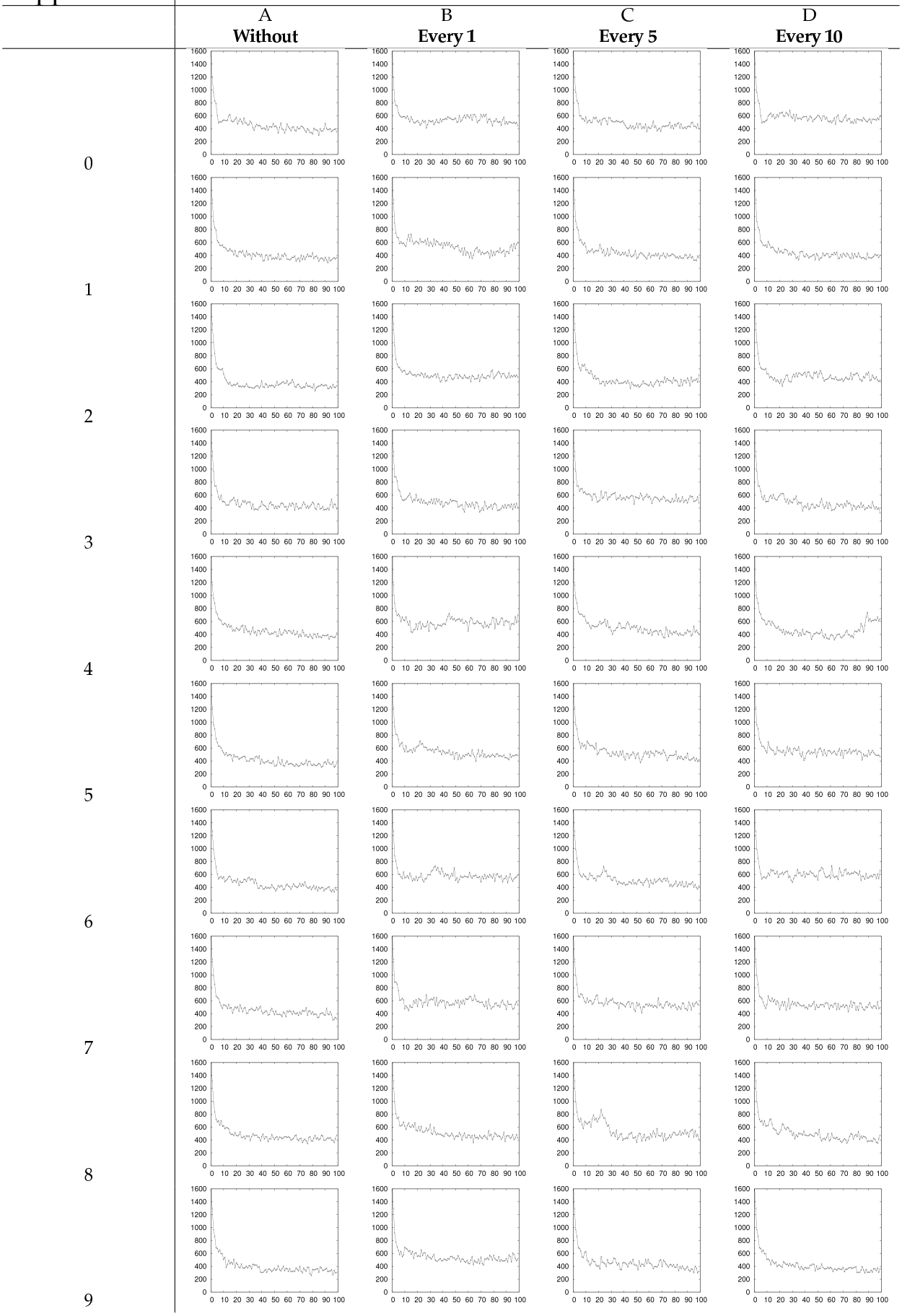

Figure 8: Line plot of the Number of Distinct Numerical Constant Values vs. Generation for the symbolic regression problem $f(x) = 11x + 50$. Performed for 10 individual runs using four different settings for simplification frequency: (a) Without, (b) Every generation, (c) Every 5 generations and (d) Every 10 generations.