Context-Aware Mutation: A Modular, Context Aware Mutation Operator for Genetic Programming

Hammad Majeed
Computer Science & Information Systems
University of Limerick,
Limerick, Ireland
hammad.majeed@ul.ie

Conor Ryan
Computer Science & Information Systems
University of Limerick,
Limerick, Ireland
conor.ryan@ul.ie

ABSTRACT

This paper introduces a new type of mutation, *Context-Aware Mutation*, which is inspired by the recently introduced context-aware crossover. Context-Aware mutation operates by replacing existing sub-trees with modules from a previously constructed repository of possibly useful subtrees.

We describe an algorithmic way to produce the repository from an initial, exploratory run and test various GP set ups for producing the repository. The results show that when the exploratory run uses context-aware crossover and the main run uses context-aware mutation, not only is the final result significantly better, the overall cost of the runs in terms of individuals evaluated is significantly lower.

Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Performance

Keywords

context-aware crossover, building blocks, modules, cascaded run, context, constructive, crossover, cache, fitness

1. INTRODUCTION

There has been much previous work [5, 1, 2, 9, 3, 4] done on the identification and subsequent reuse of useful modules for GP. There are two major difficulties with this. First, identifying the modules themselves, and second, getting GP to use them in a consistent (with how they were originally used) way. The major reason of these difficulties is the *structural complexity* of the tree representation. The tight bonding between the nodes of the trees make it very difficult to evaluate a subtree independent (out of context) of the tree containing it.

GECCO'07 July 7-11, 2007, London, England, United Kingdom.

We believe the technique introduced in [8] is more suitable for calculating the fitness of a subtree due to its context aware nature and can be used for selecting good subtrees from a population for encapsulation. Furthermore, these encapsulated subtrees (modules) when used in an effective way in the successive generations/run can result in a significant performance gain.

This paper is concerned with testing the above mentioned hypothesis and finding an effective method for the identification of the useful subtrees from a population and then their effective use afterwards.

We employed standard GP along with two new recombination operators, *context-aware mutation* and *module mutation* to test our hypothesis. The results obtained are quite promising and show a significant performance gain over the performance of standard crossover and context-aware crossover.

2. BACKGROUND

The idea of modularization is not new in Genetic Programming and many efforts have been made in the past with varying degrees of success. Koza in [5] introuduced the idea of automatically defined function (ADFs) in GP to solve complex problems. They are parameterized subroutines and solve complex problems by combining themselves in some useful manner. Koza demonstrated their usefulness on numerous complex problems in [5].

Angeline and Pollack in their Module Acquisition (MA) technique [1, 2] *randomly* selected subtrees from the population and compressed them for future use. This technique works by augmenting the evolutionary process with two additional operators, *compress* which selects a subtree of a tree and makes it immune to any further structural manipulation by compressing it into a module, and *expand* which decompresses the compressed modules and make them available for any future structural manipulation(s).

Rosca and Ballard addressed the random selection of the subtree of a tree in MA and introduced their Adaptive Representation through Learning (ARL) algorthim [9]. ARL extends the GP function set with the subtrees (modules) selected from the offsprings showing the best improvement over the fitness of their parents.

Dessi *et al* [3] showed that random selection of program sub-code for re-use is more effective than other heuristics across a range of problems. Furthermore, it was also shown that ARL does not produce highly modular solutions and once the contents of modules are allowed to evolve they become a form of ADF.

Howard employed frequency driven technique for subtree

encapsulation in [4] and combined modularization with multi-run. He started with many independent GP runs and after completion of fixed number of generations the best run was identified. The subtree database for the best run was then analyzed to group subtrees that are *operationally* equivalent. According to him, two subtrees are operationally equivalent if they evaluate to the same fitness for the given set of fitness cases. Lastly, the terminal set was augmented to include a terminal for each of randomly selected subtree from the group with size greater than one.

3. RECOMBINATION OPERATORS USED

In this study, four different recombination operators named standard crossover, context-aware crossover [6], *context-aware mutation* and *module-mutation* are used. Last two are introduced in this study and work on the *module repository* created at the end of a run. The instructions for creating a repository are discussed in detail in section 4.1. For the time being we will assume the existence of a module repository in order to introduce these new operators in the following sections.

3.1 Context-Aware Crossover

In [6] a new, context-aware crossover operator for GP was introduced. It works by placing a randomly selected subtree from one parent in its best possible context in the other parent. The detailed working of the operator is as follows:

Two parents are selected for crossover as normal, and crossover cannot take place at the root node. A subtree is chosen at random from one parent, and then all possible valid offspring are generated, that is, all those individuals that are within the depth limits, etc. Each of these individuals is then evaluated, and the best one introduced to the next generation. The same process is repeated for the second subtree and placed in the first selected parent.

It has been shown [7] that this context-aware crossover can attain significantly higher fitness than standard crossover while processing significantly *fewer* individuals. Although intuitively it may seem that context-aware crossover should process more individuals, because of the manner in which the offspring are exhaustively generated, it has been demonstrated that it can use much smaller populations, which means that, although individual crossover operators are more expensive, over the course of a run far fewer evaluations are carried out.

Context-aware crossover is so successful because it is so constructive, while standard crossover is mostly destructive [7]. However, it has been shown that while context-aware crossover makes the best use of available subtrees, it does rely on having access to subtrees that are reasonably good. Studies have shown that, because of this, context-aware crossover works best as an *exploitation* operator, and runs using it benefit from having first using only standard crossover, which is a good *exploration* operator, and then having context-aware crossover used as the only crossover operator after around 80% of a run.

This is effectively taking advantage of the tension that exists in GP between exploration and exploitation. Early on in runs [10] GP is free to create new subtrees, because with unfit populations there is less pressure to bloat or make incremental changes.

However, at the same time, GP is also driven to explore the use of these subtrees, and once fit individuals begin appearing, it is very difficult for it to change the underlying subtrees.

A common example of this is the manner in which GP synthesises constants. When using the ephemeral real constant $\Re$ one rarely, if ever, is fortunate enough to have the exact constant required appear in the initial generation, so GP synthesises it using those that did appear. For example, if one was trying to produce a function $2.1369X + 2.1369$, one would expect to see GP use a combination of constants and arithmetic expressions to construct the crucial 2.1369 constant. Once something *reasonably* close has been discovered and used in more than one place in individuals, it is very difficult to change it.

Context-aware crossover embraces this tension by explicitly dividing a GP run into two distinct parts, the explorative phase which uses standard crossover, and the exploitative phase that uses context-aware crossover.

3.2 Context-Aware Mutation operator

We introduce a new mutation operator that is inspired by context-aware crossover. This mutation operator attempts to introduce a new subtree into its best possible context, in a similar manner to context-aware crossover. The crucial difference is that this is a *new* subtree rather than one taken from another individual.

Given that context-aware crossover works best as an exploration operator, it makes sense that the subtrees introduced in this way should be reasonably good, which raises the issue of where to get these good subtrees from. We assume the existence of *repository* of *potentially useful* subtrees, and always choose the new subtree from there, rather than creating it from scratch.

3.3 Module Mutation operator

Module mutation is a modified form of standard crossover which operates on the module repository. Like the context-aware mutation operator, it also selects a subtree from the module repository using some selection scheme and then places randomly in the selected parent. The random placement of the selected subtree means that it is likely to suffer from the same mainly destructive nature as standard crossover.

4. CALCULATING THE CONTRIBUTION OF A SUBTREE

Majeed *et al* in [8] described a way to calculate the fitness contribution of any subtree within a tree (container-tree). This is a three step process. In the first step, an individual containing the subtree is evaluated, before the subtree is replaced with an *identity* node. The identity node acts as an intron in the container-tree and cancels the effect of the subtree. After replacement, the individual is then re-evaluated. The difference between the two fitness values is the contribution of the subtree in the container-tree.

The working principle of an identity node is similar to the identity function in set theory. An identity node always replaces some node or subtree. This replacement cancels out the effect of the replaced node or subtree. For example, in (* (+ X X) Y) the fitness contribution of subtree (+ X X) is calculated in the following manner.

1. Tree is evaluated
2. (+ X X) is replaced with the identity function of its parent node "*", which is ONE
3. The new tree is reevaluated
4. The difference between the two fitness values of the trees is the fitness contribution of (+ X X)

Note, - and / are special cases of + and * respectively. As $a - b = a + (-b)$ and $a / b = a * (1 / b)$

4.1 Identification of modules and repository creation

Many researchers have devised different methods to create their module repositories and used them in the subsequent runs, and have experienced varying degrees of success. We believe that the contribution of a subtree towards its container tree and the number of times it appeared in the population (hit count) are good measures to calculate its significance in the population.

In this study, we create the module repository containing unique modules by selecting the individuals from the final generation of a run with fitness higher than the average fitness of the generation, this ensures the selection of the fit individuals for further analysis. The maximum size[1] of the module repository is fixed to the population size of the cascaded run. The repository is created by employing two different methods. In the first method, subtrees with fitness contribution greater than zero, node count greater than one and parent nodes containing only binary functions are allowed to enter the repository. In the second approach, the subtrees with node count greater than one and parent nodes containing only binary functions are allowed to enter the repository. Note, the second approach is considerably cheaper due to elimination of the calculation of the fitness contribution of the subtrees which requires many reevaluations of the same individual.

In the first approach, after filling the repository, the *final fitness* of each module of the repository is calculated by using the following equation.

$$\frac{0.7 \times contrib}{\sum_{i=1}^{cnt} contrib_i} + \frac{0.1 \times hits}{\sum_{i=1}^{cnt} hits_i} + 0.2 \times (1 - \frac{node\_count}{\sum_{i=1}^{cnt} node\_cnt_i})$$

where *contrib* , *node_cnt* and *cnt* are average fitness contribution of the subtree, node count of the subtree and the size of the repository, respectively. Note that the selection of the modules can be influenced in different ways by adjusting the weights assigned to the used parameters. The proportions used here were simply guesses, and no effort has been made to optimise them.

In this study we only conduct two cascaded runs, i.e. one run followed by the initial run, as this study is only a proof of concept. In practice, a module repository needs to be created only once after the initial run and multiple "second runs" can be done by using it. To make things fair and transparent in this study, the cost of the cascaded run for all the experiments include the cost of the creation of the module repository after the initial run.

[1]Actual size of the repository can be less than the size of the population depending on the availability of the subtrees fulfilling the criteria of module selection.

4.2 Selection of modules

We believe that the selection of the right module from the module repository is a critical task and can have significant effect on the outcome of the cascaded run. To check this hypothesis, two module selection methods were used in this paper. The first one is based on fitness proportionate selection and uses a roulette wheel for its working. It works by first creating a roulette wheel by using the final fitness of each module and then selecting the module by spinning the wheel. The second selection method is the simple random selection of modules from the repository. These two selection methods are introduced to examine the effect of the selection methods on the final outcome of a run.

5. EXPERIMENTAL SETUP

For this study Koza's Quartic Polynomial Symbolic Regression problem was used and three sets of experiments, involving different recombination operators were conducted. For each experiment, two *cascaded* runs were performed. After completion of the first run, a repository of the useful modules was created as explained in section 4.1 and used in the subsequent run. In this fashion 50 independent runs were conducted. The initial population was generated using the ramped half and half method with initial tree depth varying from 2-6 and the maximum tree depth was set to 17.

In the first set of experiments, two tests were conducted. In the first test, during the first run only context-aware crossover was used and in the cascaded run only context-aware mutation operator was used. This was to check the usefulness of the tree distribution of the identified modules in the cascaded run. This test was repeated for fitness proportionate and random module selection methods. This was to check the effect of the introduction of the fit and randomly selected modules on the performance of the system.

In the second test, context-aware crossover along with context-aware mutation was used in the cascaded run. This was to check the performance of context-aware crossover in the presence of the subtrees from a different tree distribution. As before, this test was repeated for fitness proportionate and random selection methods. For these experiments a population of size 200 was allowed to complete 25 generations. In the second test, context-aware crossover and context-mutation were used with equal probability, i.e. 0.5.

In the second set of experiments, for each experiment, two tests were conducted as before. In the first test, standard crossover was used in the first run followed by the use of only module mutation in the successive run. This test was repeated for fitness proportionate and random selection methods. In the second test, the first run was completed by only using standard crossover and in the cascaded run standard crossover was used along with module mutation. This was to check the effect of the interaction between these operators on the performance of the system. For these experiments a population of size 1000 was allowed to complete 100 generations. In the second test, standard crossover and modified mutation were used with equal probability, i.e. 0.5.

In the last set of experiments, for each experiment, again two tests were conducted. In the first test, standard crossover was used in the first run followed by the use of only context-aware mutation operator. This was to check the behavior of context-aware mutation using the modules generated by

standard crossover. In the second test, the first run was completed by using standard crossover and in the successive run context-aware mutation and module mutation operators were used. This was to check the efficiency of the system using both these operators together. For these experiments the population size and number of generations were reduced in the second run to make the evaluation count comparable between the two runs. The population size and number of generations were set to 1000 and 100 respectively for the first run and reduced to 200 and 10 respectively, in the second run.

5.1 Does good first run guarantee good second run ?

To answer this question, we examined the final generations of the first and second run and tried to look for the existence of any relationship between them. To accomplish this, *coefficient of determination* (r^2) between the final generations of the first and second run was calculated. Recall, coefficient of determination gives the proportion of the variance (fluctuation) of one variable that is predictable from the other variable, so if the success of second run is dependent on the success of first run then the coefficient of determination should be high (r^2 close to one means tight relationship). We calculated r^2 for all the experiments and for all the cases the relationship between the first and second run was quite weak (r^2 was quite close to zero in almost all the cases). This shows that the success or otherwise of the second run is mostly dependent on the selection of the fit repository and the way it was used in the successive run.

6. RESULTS

All the results presented here are averaged over 50 runs. On the x-axis, the cumulative number of evaluations done by each system is shown. The cumulative number of evaluations is a better measure of comparing the performance of different recombination operators as the number of evaluations done by each per generation can vary drastically. For the first run of all the experiments, the successive run was repeated for fitness proportionate and random selection methods. As the experiments involving fitness proportionate selection method require additional number of evaluations at the end of the first run therefore for these experiments, the cost of the creating the repository is added to the cascaded run.

To better understand the effect of the module selection methods on the performance of the system, the modules selected by the selection methods during mutation operation were examined and their mean and standard deviation of the fitness contributions were noted for further analysis. Mean fitness contribution tells us about the goodness of the selected modules and standard deviation the variations in the fitness values of the selected modules.

Due to the space restriction, only mean best plots for all the setups are shown and discussed.

6.1 Use of standard crossover in the first run

Figure 1 (Left) shows the performance of the system employing standard crossover in the first run and module mutation in the second cascaded run. The plots involving fitness proportionate selection are shifted right due to the inclusion of the cost of creating the module repositories for these runs. Recall, the fitness proportionate selection method uses fitness contribution of the modules which requires numerous re-evaluations of the individuals before starting the second run. Both the successive runs show an exponential gain over the performance of standard crossover and reach their respective maximum fitness values quickly before flattening out. This shows the resilience of the modules used in the cascaded runs and their ability to overcome the destructive effects of module mutation. Notice that the method of selection of modules for the repository has little effect on the final outcome of the run, we believe this is due to the inability of module mutation to use the fit modules in a best possible way in the selected parents.

Figure 1 (Right) shows the performance of the system using standard crossover in the first run and module mutation along with standard crossover in the second run. The plots involving fitness proportionate selection are shifted right due to the inclusion of the cost of creating the module repositories for these runs. As above, the performance of both the runs using modules is far better than the performance of the first run. Both the successive runs show a slight drop in fitness prior to 20,000 evaluations and take little longer time to reach their respective maximum fitness values. We believe this is due to the destructive effects of the standard crossover during the run.

6.1.1 Analysis

Figure 2 shows the detailed statistics of the selected modules from the repository by module mutation operator during the runs. On x-axis, independent runs are shown and on y-axis the mean and standard deviation of the selected modules during the run are plotted.[2].

Clearly, fitness proportionate selection selects quite fit (high mean) and diverse (high standard deviation) modules compared to random selection during the mutation operation. Surprisingly, the selection of fit and diverse modules does not show any significant effect on the performance of the system and mod_mut_fit in figure 1 performs worse than mod_mut_rand. This is due to the inability of module mutation to use the selected modules in the most effective way in the selected parents.

Figure 3 shows the performance of the experiments involving context-aware mutation in the successive run. Recall, for these experiments, the second run was reduced to 10 generations and population size was set to 200 to make evaluations count comparison fair between the first and second run. Due to the small sized population, the module repository size was also reduced to 200. The cost of creating the repository is included in ctxt_mut_fit and ctxt_mut_mod_mut_fit plots, inclusion of the cost in these plots is not noticeable due to the low cost of creating the reduced size repositories.

Figure 3 (Left) shows the performance of the system using standard crossover in the first run and context-aware mutation in the second run. ctxt_mut_fit shows significant improvement over ctxt_mut_rand and std_xover in the early stages of the run due to the use of fitness proportionate selection method. ctxt_mut_rand matches std_xover prior to 20,00 evaluations and then improves to reach ctxt_mut_fit beyond 60,000 evaluations. Although, ctxt_mut_fit shows slow progress in the early stages of the run than mod_mut_rand and mod_mut_fit in figure 1 but does not suffer from the con-

[2]The runs with zero standard deviation have selected either the same module or modules with same fitness value all the times.

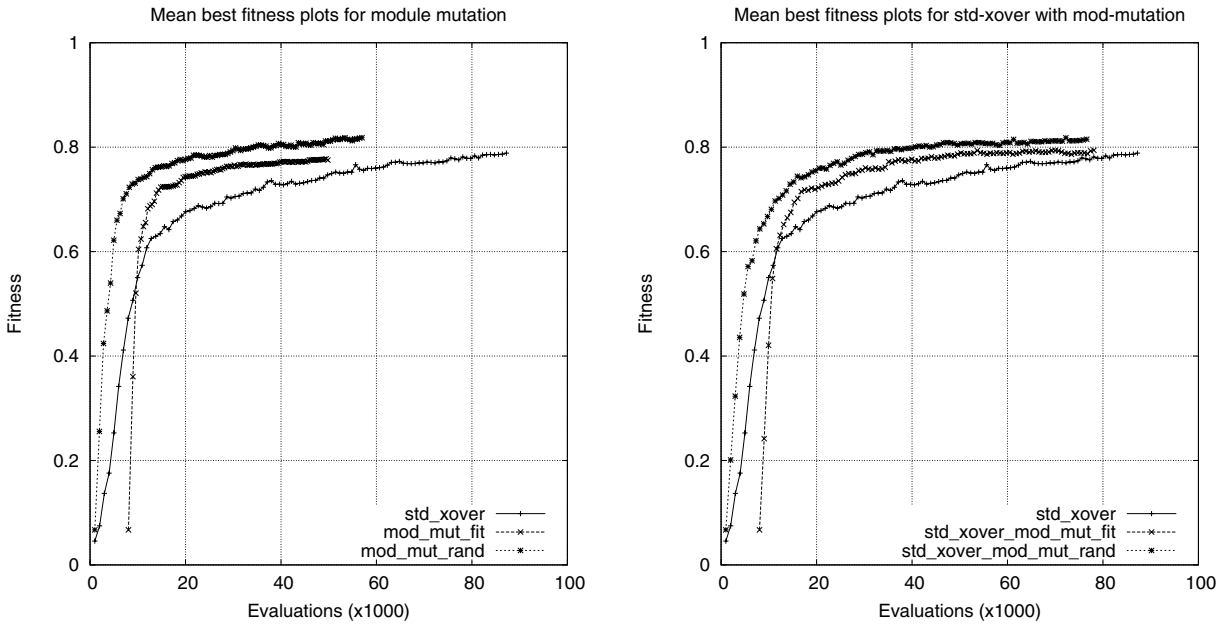

Figure 1: ***Left:*** **Plots for the system using standard crossover in the first run and module mutation in the second run. The run using fitness proportionate selection is labeled as mod_mut_fit and random selection is labeled as mod_mut_rand.** ***Right:*** **Plots for the system using standard crossover in the first run and module mutation along with standard crossover in the second run. The run using fitness proportionate selection is labeled as std_xover_mod_mut_fit and random selection is labeled as std_xover_mod_mut_rand.**

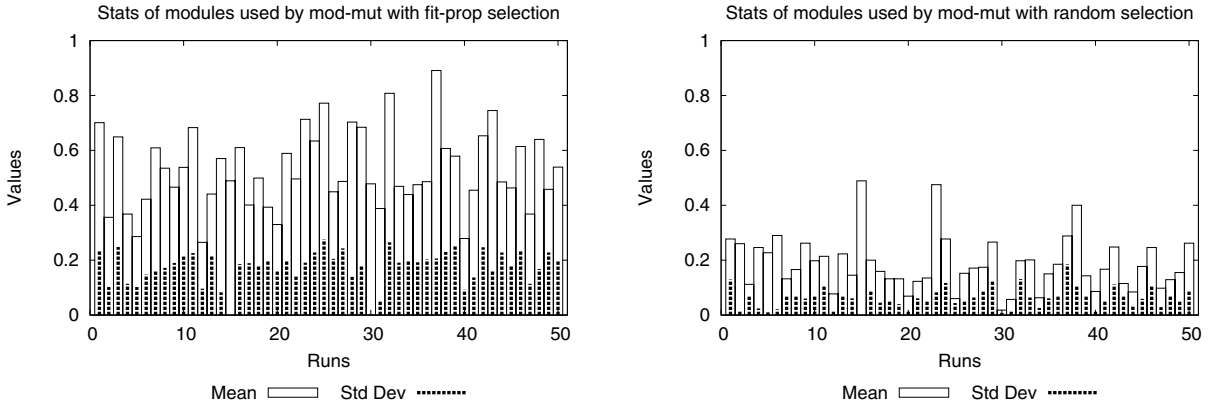

Figure 2: Mean and standard deviation of the modules selected for mutation during independent runs. ***Left:*** **Shows the statistics for fitness proportionate selection.** ***Right:*** **Shows the statistics for random selection.**

vergence to a low maximum fitness and keeps on improving in the later stages of the run.

Figure 3 (Right) shows the performance of the system using standard crossover in the first run and context-aware mutation along with module mutation in the second run. The use of fitness proportionate selection results in a dramatic improvement in fitness in the early stages of the run without showing any sign of flattening and keeps on improving with time. ctxt_mut_mod_mut_rand is slow to respond but very soon it catches up with ctxt_mut_mod_mut_fit and keeps on improving after that.

6.1.2 Analysis

The statistics for the modules selected in this experiment are quite similar to the ones shows in figure 2, therefore we will use the same figure to analyze the results presented here. In figure 3 (Left), ctxt_mut_fit performed better than the rest due to the effective (due to its context-aware nature) use of the fit modules selected by fitness proportionate selection. ctxt_mut_rand, on the other hand, was provided with the inferior modules and failed to find good placements for them in the random population, however after a few generations it was able to use the selected modules effectively in the relatively matured population.

In figure 3 (Right) the use of context-aware mutation with

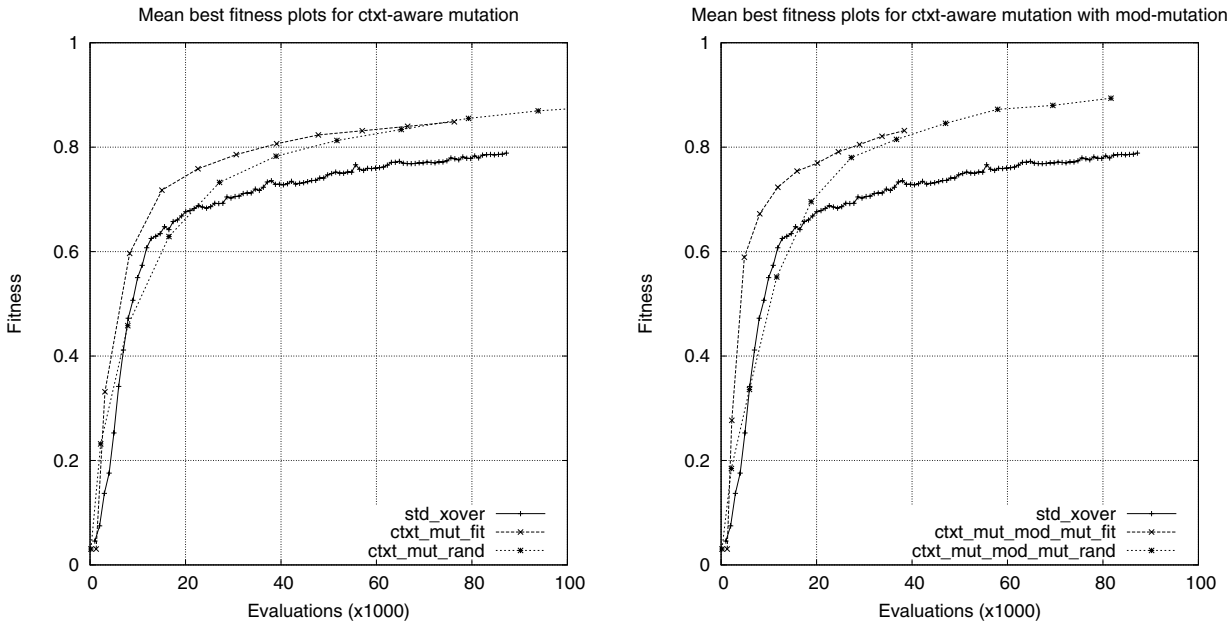

Figure 3: ***Left:*** **Plots for the system using standard crossover in the first run (std_xover) and context-aware mutation in the second run using fitness proportionate (ctxt_mut_fit) and random (ctxt_mut_rand) selection methods.** ***Right:*** **Plots for the system using context-aware crossover in the first run (std_xover) and context-aware mutation along with mod-mutation in the second run using fitness proportionate (ctxt_mut_mod_mut_fit) and random (ctxt_mut_mod_mut_rand) selection methods.**

module mutation has proven to be quite useful. As previously discussed, context-aware crossover is a good exploitation operator while standard crossover is considered to be good for exploration of the search space. We believe that in this experiment module mutation played the role of standard crossover and context-mutation operator improves over that by applying small tweaks to the population.

6.2 Use of Context-Aware crossover in the first run

Figure 4 (Left) shows the mean best fitness of the system using context-aware crossover in the first run and context-aware mutation in the second run. Both ctxt_mut_fit and ctxt_mut_rand show a significant gain over ctxt_aware_xover prior to 50,000 evaluations. This clearly shows the effect of use of the modules on the fitness of the early generations of the run. The use of different module selection methods does not have a significantly different effect on the performance of the two systems and both of them show same performance throughout the run.

Figure 4 (Right) shows the mean best fitness of the system using only context-aware crossover in the first run and context-aware mutation along with context-aware crossover in the successive run. In general, the use of context-aware crossover (not mutation) in the second run results in a slightly less fit population. We believe that the deterioration in the performance is due to random selection of the subtree from the parent by context-aware crossover, which can result in the disruption of the useful modules introduced by context-aware mutation operator in the past, while the runs that only use context-aware mutation in the second run only ever choose from the repository.

6.2.1 Analysis

Figure 5 shows the mean fitness and standard deviation in the fitness of the modules selected in the cascaded run by fitness proportionate (Left) and random selection methods (Right). The plots for the second experiment (figure 4 (Right)) are quite similar to the ones shown in figure 5, therefore the explanation for the first is applicable to the second without any exception.

As evident from figure 5, the modules selected by fitness proportionate and random selections are not very different from each other. Granted, the mean fitness of the modules selected by fitness proportionate selection method is slightly higher than the ones selected by random selection but still this difference is not as big as shown by the experiments using standard crossover in the first run (look at figure 2). We believe this is due to the creation of a fit repository at the end of the first run employing context-aware crossover. The selection of the similar modules can be attributed as one of the plausible reasons behind the similar performance of the two systems using different module selection methods.

7. DISCUSSION

In general, the use of standard crossover in the first run and module mutation in the successive run has resulted in an exponential gain in the performance and in attaining a high fitness in no time. Unfortunately, after converging to the high fitness value it stuck there and showed no sign of improvement despite the prolonged runs. This was mainly due to the *neutral* mutation events at the end of the run which were caused by the bloated trees.

The use of standard crossover along with module mutation in the second run has deteriorated the fitness of the

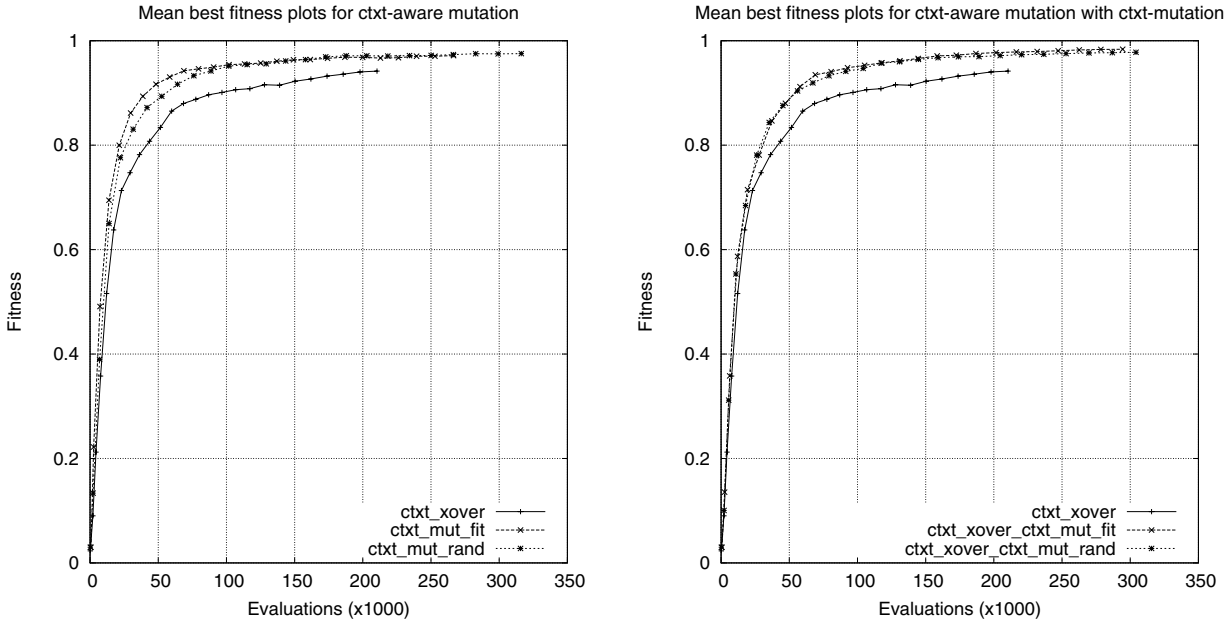

Figure 4: ***Left:*** **Plots for the system using context-aware crossover in the first run (ctxt_xover) and context-aware mutation in the second run using fitness proportionate (ctxt_mut_fit) and random (ctxt_mut_rand) selection methods.** ***Right:*** **Plots for the system using context-aware crossover in the first run (ctxt_xover) and context-aware crossover along with context-mutation in the second run using fitness proportionate (ctxt_xover_ctxt_mut_fit) and random (ctxt_xover_ctxt_mut_rand) selection methods.**

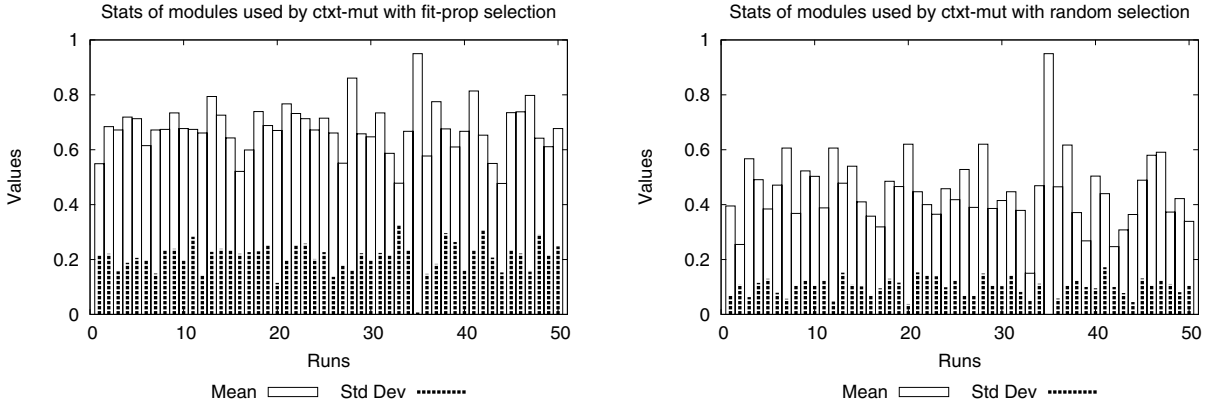

Figure 5: Mean and standard deviation of the modules selected for mutation during independent runs. ***Left:*** **Shows the statistics for fitness proportionate selection.** ***Right:*** **Shows the statistics for random selection.**

system slightly and made it took bit longer to attain the high fitness value. This was due to the destructive effects of standard crossover operator. We believe that module mutation though works like standard crossover but still is less destructive. Standard crossover works by selecting a subtree randomly from the first selected parent and places it again randomly in the other parent, which increases the probability of the disruption of the good module present in the parent. Module mutation, on the other hand copies the whole module without any disruption in the selected parent hence making it less destructive.

The selection of fit or less fit modules from the repository has a minimal effect on the outcome of the runs involving standard crossover and module mutation. This is due to the ignorance of the standard and module mutation towards the importance of the context of the exchanged modules in the parent.

The use of standard crossover in the first run and context-aware mutation in the second run has solved the problem of convergence of the system to some fitness value. For this setup, the selection of fit modules has shown a significant improvement in the performance over the performance shown by random selection of the modules from the repository. We believe it was mainly due to the context-aware nature of the mutation operator used and used the provided fit module in the best possible way. The introduction of module mutation

in the second run has improved the results dramatically. We believe this was due to the mutual cooperation of the used recombination operators, in which module mutation acts as an exploration operator and context-aware mutation as an exploitation operator. Again due to the context-aware nature of the context-aware mutation operator the selection of fit modules during the run resulted the maximum fitness gain.

The system employing context-aware crossover in the first run and context-aware mutation operator in the second run has shown the best performance. In the cascaded run context-aware mutation has used the selected modules in the best possible way and improved the performance of the system significantly. The module selection methods have minimal effect on the outcome of the run and random and fit proportionate selection methods performed equally well. This was due to the generation of the fit module repository at the end the first run and the selection of the equally fit modules by both the selection methods from it. The introduction of context-aware crossover in the second run has deteriorated the performance of the system slightly. We believe that this behavior is due to the random selection of subtrees from the selected parents by context-aware crossover which increases the chance of the disruption of the useful modules introduced by context-aware mutation previously.

8. CONCLUSION & FUTURE WORK

This paper discusses the significance of context aware fitness evaluation of subtrees in selection of a fit module repository at the end of a run and improving the performance of the subsequent run employing the created repository.

In general, we have recorded an exponential gain in the performance of the cascaded run using the modules over the performance of the first run. The module selection methods has shown a minimal effect on the performance of the system employing only standard crossover or module mutation operator due to their ignorance of the importance of the context of the exchanged modules in the selected parents. Fitness proportionate selection method has performed significantly well when used with context-aware crossover or context-aware mutation. This was due to the ability of these operators to use the provided module in the best possible way in the selected parent. The best performance was recorded by the system using context-aware crossover in the first run and context-aware mutation with fitness proportionate module selection in the cascaded run.

In this study we have only conducted one cascaded run followed by the initial run, in future we are planning to perform multiple second runs after creating module repository. This approach will distribute the cost of creating of the module repository over multiple second runs. Also, in this study only one problem is examined, in future we are planning to apply the same methods to other problems in an effort to check its robustness.

9. REFERENCES

[1] P. J. Angeline and J. B. Pollack. Coevolving high-level representations. July Technical report 92-PA-COEVOLVE, Laboratory for Artificial Intelligence. The Ohio State University, 1993.

[2] Peter J. Angeline and Jordan Pollack. Evolutionary module acquisition. In D. Fogel and W. Atmar, editors, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 154–163, La Jolla, CA, USA, 25-26 February 1993.

[3] Antonello Dessi, Antonella Giani, and Antonina Starita. An analysis of automatic subroutine discovery in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 996–1001, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[4] Daniel Howard. Modularization by multi-run frequency driven subtree encapsulation. In Rick L. Riolo and Bill Worzel, editors, *Genetic Programming Theory and Practise*, chapter 10, pages 155–172. Kluwer, 2003.

[5] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[6] Hammad Majeed and Conor Ryan. A less destructive, context-aware crossover operator for GP. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 36–48, Budapest, Hungary, 10 - 12 April 2006. Springer.

[7] Hammad Majeed and Conor Ryan. Using context-aware crossover to improve the performance of GP. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 1, pages 847–854, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[8] Hammad Majeed, Conor Ryan, and R. Muhammad Atif Azad. Evaluating GP schema in context. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1773–1774, Washington DC, USA, 25-29 June 2005. ACM Press.

[9] Justinian P. Rosca and Dana H. Ballard. Discovery of subroutines in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–202. MIT Press, Cambridge, MA, USA, 1996.

[10] Conor Ryan and Maarten Keijzer. An analysis of diversity of constants of genetic programming. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 404–413, Essex, 14-16 April 2003. Springer-Verlag.