Syntactically Correct Genetic Programming

Róbert Ványi[1] and Szilvia Zvada[2]

[1] Department of Theoretical Computer Science,
Friedrich-Alexander University
Martensstraße 3, D-91058 Erlangen, Germany
vanyi@cs.fau.de
[2] Department of Programming Languages,
Friedrich-Alexander University
Martensstraße 3, D-91058 Erlangen, Germany
szisza@cs.fau.de

Abstract. When common Genetic Programming is used to solve a problem, the so-called closure property must be fulfilled. That is any solution received by changing parse trees at any point has to be considered as a candidate and must receive a fitness value. Sometimes maintaining this property needs extra resources, since it may cause that several candidates have to be corrected or replaced.
Sometimes it is possible to describe the correct solutions using a grammar. This way the syntax of the accepted candidates can be given, and during the evolution solely valid individuals, that is possible solution candidates can be generated.
In this paper three methods for syntactically correct genetic programming are examined. The first one is Strongly Typed Genetic Programming, which extends Koza's original GP by assigning signatures to function symbols. The second is Grammatical Evolution, which defines the correct individuals by a context-free grammar. It is not exactly genetic programming using a strict definition, but also works with complex individuals, though using a simple bitvector representation. The last one is Derivation-Tree Based Genetic Programming that is similar to Grammatical Evolution but uses derivation trees to represent individuals. Some extensions for the latter one are also given in this paper.

1 Introduction

Genetic Programming [1] is a widely-used tool for optimizing complex objects. A great advantage is that it needs no exact knowledge of the problem, but it works using a blackbox-principle. That is the algorithm itself only needs a function – the so-called fitness function – that assigns a goodness value to each possible solution. However, constructing candidate solutions without guidance often means that incorrect or invalid individuals are produced. In some cases they can be penalized by giving them very bad fitness values. In other cases they cannot even be evaluated using the fitness function. Sometimes they can be corrected, but in any cases they cause a loss of resources, since new individuals

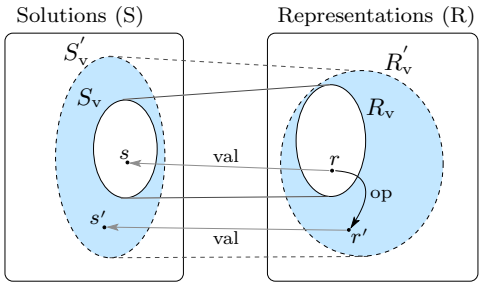


Fig. 1. Genetic Programming with the closure property

have to be generated to replace them, or because a correction procedure must be applied. The outline of this method can be seen in Figure 1. The set of possible solutions S is described by a set of representations R, usually via a many-to-one mapping. The valid solutions are elements of set S_v, and described by the elements of set R_v. When an operator creates an individual r' from the valid individual r, the new individual must be validated, and corrected or dropped if necessary. Other possibility is to extend the set of the valid representations to be closed under the evolutionary operators, but in case the set of valid solutions is extended as well.

There are several possibilities to overcome this problem. In this paper three of these possibilities are examined that use syntactical approaches to restrict the generated individuals. The first method is *Strongly Typed Genetic Programming* (*STGP*), that is defined by Montana [2], and mainly used to evolve expressions. Using STGP the operators have exact types, and they accept a given number of parameters of pre-defined types. That is each operator has an exact pre-defined *signature*. The operators are designed to respect these signatures.

The other two methods use *context-free grammars* (*cfg*s) to describe a *language*, that is the set of syntactically correct individuals. One of them is *Grammatical Evolution* (*GE*) developed by O'Neill, Ryan and others. [3] It uses bitvectors to describe left-hand derivations leading to the correct individuals. The last method is *Derivation-Tree Based Genetic Programming* (*DTGP*), which is similar to GE, but the derivations are represented by *derivation trees* instead of bitvectors. [4]

These methods are outlined in Figure 2. In contrast to common Genetic Programming the valid solutions are represented by a language and not by a simple set. These solutions are described syntactically, and the operators, if necessary, are constructed to respect the syntax. Thus no invalid individuals are created, and no extension of the solution space is necessary. In this paper the details of these methods will be discussed.

We proceed as follows. In Section 2 the previously mentioned three possibilities for syntactical restrictions are described. In Section 3 the differences of these method are discussed. An improved version of DTGP is presented in Section 4.

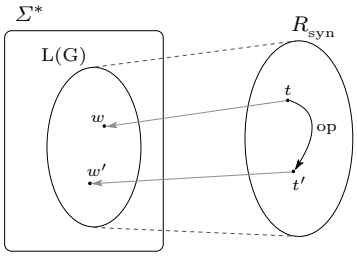


Fig. 2. Genetic Programming with syntactical restriction

In Section 5 some experimental results are shown. Finally conclusions are drawn and future plans are outlined in Section 6.

2 Syntactical Restrictions

When Koza introduced Genetic Programming, the operators were designed to work on parse trees representing Lisp S-expressions. To allow seamless evolutionary operators, it was assumed that each node of the parse tree, that is each operator in the expression accepts the same number of parameters and returns the same type. This is, unfortunately, not always acceptable, thus several methods were designed to overcome this problem.

2.1 Strongly Typed Genetic Programming

Strongly Typed Genetic programming was directly designed to solve this problem. To do this it pre-defines the number and types of the parameters for each operator or function, and the return type of the operator as well.

To represent the solutions STGP uses parse trees, which can be considered as expressions written in a tree structure. Thus, the individuals can be considered as the solutions themselves. This representation, however, allows only expressions to be evolved.

The difference between the original GP and STGP is that the so-called signature for each function is given in advance. That is each function has a given return type, a given arity, and the parameters also have a given type. Formally it can be defined as follows. Let $\mathcal{T}$ be a set of types and $\mathcal{F}$ be a set of function symbols. A signature σ is a triplet, that is an element of $\mathcal{T} \times \mathcal{F} \times \mathcal{T}^*$. A signature $\sigma = (t, f, t_1 t_2 \dots t_n)$ can be written as $t\ f(t_1, t_2, \dots, t_n)$.

An example for a set of signatures can be seen in Figure 3. These signatures describe a set of functions built from the usual arithmetical operators $(+, -, *, /)$, logical operators $(\neg, |, \&)$ and the so called ternary operator if, which takes a logical expression and two real expressions and depending on the value of the logical expression returns the value of the first or the second expression.

real	+	(real,real)
real	−	(real,real)
real	∗	(real,real)
real	/	(real,real)

a) real functions

bool	¬	(bool)
bool	|	(bool,bool)
bool	&	(bool,bool)
real	if	(bool,real,real)

b) boolean functions

real	0	()
real	1	()
bool	$false$	()
bool	$true$	()

c) constants

Fig. 3. Example set of signatures

2.2 Grammatical Evolution

The first aim behind Grammatical Evolution was to allow evolutionary optimization within arbitrary context-free languages. Thus the blackbox principle is generalized further. The evolutionary algorithm does not need an exact representation of the solution, only a context-free grammar is needed that describes the set of solutions. Of course, by describing the valid individuals with syntactical rules means a syntactical restriction is given on the solutions.

A context-free grammar is simply a 4-tuple $G = (N, \Sigma, P, S)$ with finite sets of nonterminal and terminal symbols N and Σ respectively, a finite set of productions P and a start symbol S. The rules describe how a certain nonterminal symbol can be replaced, that is which words can replace it. Words can be produced by taking the start symbol and applying rules until no more nonterminals exist in the word. If more rules can be applied then any of them can be taken, that is the process is non-deterministic. An example grammar for generating the set of the valid arithmetical expressions, can be $G_{ex} = (N_{ex}, \Sigma_{ec}, P_{ex}, expr)$ with

$$N_{ex} = \{expr, fact, tag, val, bexpr, bfact, btag, bval\}$$

$$\Sigma_{ex} = \{+, -, *, /, \neg, |, \&, if, 0, 1, true, false, (,)\}$$

$$\begin{aligned} expr &\rightarrow tag + tag \ ; \ tag \\ tag &\rightarrow fact * fact \ ; \ fact \\ fact &\rightarrow val \ ; \ (expr) \ ; \ if(bexpr, expr, expr) \\ val &\rightarrow 0 \ ; \ 1 \end{aligned}$$

$$\begin{aligned} bexpr &\rightarrow btag|btag \ ; \ btag \\ btag &\rightarrow bfact\&bfact \ ; \ bfact \\ bfact &\rightarrow bval \ ; \ (bexpr) \ ; \ \neg(bexpr) \\ bval &\rightarrow false \ ; \ true \ ; \ \neg bval \end{aligned}$$

An important property of GE that it works on the derivations leading to valid solutions, instead of working on the solutions themselves. It is well known that if a word can be derived using a context-free grammar, then it has a left-derivation, that is a derivation where always the leftmost nonterminal is replaced. Grammatical Evolution represents these left-derivations with a sequence of integers, which in turn are represented by bits. That is GE works with bitstrings and applies operators similarly to Genetic Algorithms.

2.3 Derivation-Tree Based Genetic Programming

Derivation-Tree Based Genetic Programming (DTGP) was designed to evolve solely syntactically correct individuals, and to solve some shortcomings of Grammatical Evolution. The two methods are similar in a sense that they work on derivations. However, while GE works on bitstrings representing left-derivations, DTGP uses derivation trees. Derivation trees are often-used to represent derivations, an example can be seen in Figure 4. Note that a derivation tree usually represents more derivations.

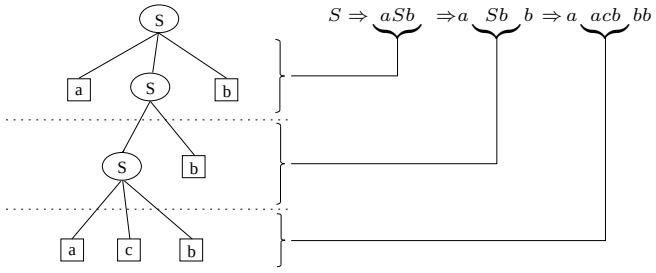


Fig. 4. Example for derivation tree

The advantage of DTGP over GE is that derivation trees contain the complete derivation and the derived word as well. This way it is ensured that the derivation is always finite and no derivation has to be done to receive the represented word. Disadvantage is the increased space-consumption and the complexity of the operators. On the other hand derivation trees can be extended onto attribute or context-free hyperedge grammars to describe words of a higher level language as well.

3 Main differences of the methods

For a detailed description of the introduced three methods the interested reader is kindly referred to the listed literature, however, the most important differences are described in this section.

3.1 Individual representation

Concerning the representation of the solutions the STGP is exactly the same as common GP. That is the individuals are parse trees, although using STGP the number of branches for a node having a given label is unambiguously determined.

Grammatical Evolution uses the simplest representation that is possible, namely it uses bitvectors. Certain number of bits are grouped together and considered as integer numbers like bytes or words. These integers are then used

to select a rule from an ordered set of the applicable rewriting rules during the derivation procedure. Thus, each sequence of bits describes a derivation, and aside from the rare cases of infinite derivations, it determines exactly one valid solution. This simple representation form enables easy and fast operators.

The most complex data structure is used by Derivation-Tree Based Genetic Programming. As the name says, the individuals are represented by derivation trees. By definition a string is an element of a language if it can be derived. If a string can be derived then a derivation tree exists as well, which is even unique in case of many practically used grammars. Unfortunately the derivation tree is rather large, since not only the "useful" symbols, but the nonterminal symbols are also present, in spite of that the latter symbols appear in a derivation only temporarily. Fortunately tree operators can be carried out easily, although there are some important drawbacks of this representation form, which will be detailed in the next section. An example for representing the expression $a + b * 2$ by a parse tree used for STGP and by a derivation tree used for DTGP can be seen in Figure 5.

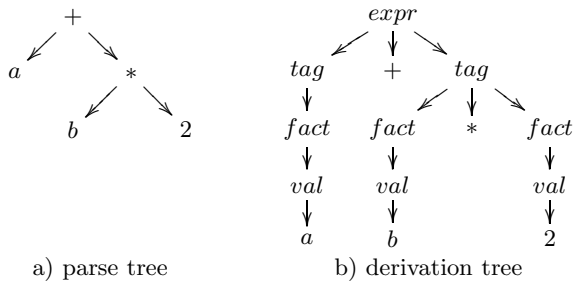


Fig. 5. Different representations of an expression

3.2 Evolutionary operators

The important difference between common GP and STGP lies in the operators. To keep the syntactical correctness of the evolved individuals, the operators must ensure that the nodes representing functions have the prescribed signatures. That is subtrees can only be replaced by subtrees with the same return type. For random tree, or random subtree generation it means that when the tree is build at a given position only a subset of the operators may be used. In the case of subtree exchange for example for crossover, the inserted subtree can only be selected among subtrees having the given return type. These restrictions unfortunately can make the operators more difficult, and also slower.

Grammatical Evolution is really simple, not only regarding the representation form, but with respect to the operators as well. One can use the bit-operators well-known from the field of genetic algorithms. Furthermore other operators can also be defined. [5] However, the parts of the bitstring have different influence on

the derived word, thus changing even one bit may result in a completely different solution. Because of these "hidden" representation, these operators cannot be parameterized in the way as the tree operators.

Derivation-Tree Based Genetic Programming uses almost the same operators as the original Genetic Programming. To ensure that the resulting trees are valid derivation trees, the nodes of the inserted subtrees must have the same label in the root as the deleted subtree.

3.3 Evaluation

In each evolutionary step each individual has to be evaluated, to get the represented solution, and then this solution must be qualified using the fitness function. In case of parse trees, the evaluation is very easy, since the solution, that is the expression is the tree itself.

In the case of the other two methods the individuals represent derivations. This means that if Grammatical Evolution is used, a complete derivation has to be done, selecting the appropriate rules of the grammar with the help of the individual. In case of DTGP the represented solution can be found in the frontier of the tree, thus only the leaves have to be read from left to right. Notice, that this might be a non-trivial task as well.

4 Improving DTGP

As it was mentioned previously, the operators for DTGP can be really problematic, since not only complex structures must be modified, but the operators must also consider several constraints. This may cause that the DTGP operators consume a huge amount of processing resources, and because they are frequently applied, they make the evolutionary step use significantly more time.

4.1 Attributed DTGP

The most important tree operators may be enhanced by storing several attributes in the nodes. For example it was shown that a random node selection can be done in a logarithmic number of steps with respect to the size of the tree. [6] Using this method the selection weights, which are proportional to the selection probability, but fall not necessarily between 0 and 1, must be stored at the nodes. Furthermore the accumulated weights for each subtree must be stored as well. Fortunately when the tree is changed, the accumulated weights can be updated in a logarithmic number of steps. In the simplest case the selection weight is 0 for leafs and 1 for every other node, but it can involve many parameters like depth or breadth of the subtree without significant extra costs.

The random node selection starts at the root node, and uses the accumulated weights to randomly decide in which subtree to continue the search. With this method the random node selection, subtree deletion and subtree insertion can be significantly faster, since for example in case of a balanced binary tree with

1024 leaves they need not more than 10 steps, which is much smaller than 1024 (or 2047). Although it is still greater than 1. The attributes may also be used to store parts of the result, or parts of the fitness value, thus the evaluation may be aided by checking the root node of the tree and reading the appropriate attribute values.

An example for an attributed tree can be seen in Figure 6. Under the labels, which are used here for demonstration purposes only, the selection weight and the accumulated selection weight can be seen. Under them the breadth and the depth of the subtree is also given. Selection weights are 0 for the leafs, the depth is 0 as well, and the breadth is 1. Thus these parameters are not shown for leafs. When the algorithm starts at the root and steps down to the node labelled by U, a decision is made according to the accumulated weights. From 80 cases on the average subtree X is selected 15 times, J is selected 15 times and C is selected 50 times. U itself will not be selected, since its selection weight is 0.

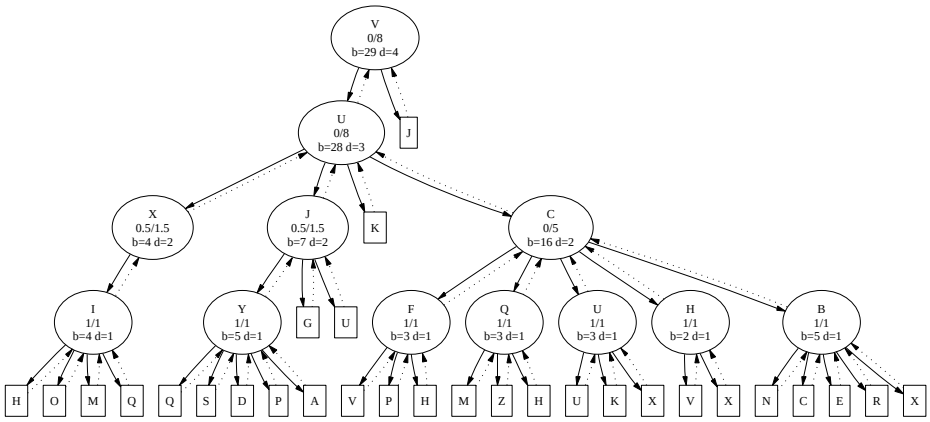


Fig. 6. Attributed tree

There is, however, one operator that cannot easily be improved by using attributes. It is the crossover operator, because it needs a special node selection, since the labels of the subtree roots must be the same. Thus the crossover operator must be modified.

4.2 Modified crossover

One possibility would be to store information in the nodes about the distribution of the labels. This would, however, mean an increased storage consumption, since in each node this information should be stored for each possible label. Furthermore, this information cannot be combined with other attributes. But storing weights for each possible combination of attributes is practically impossible.

Another possibility is to redesign the operator. The importance of crossover is that good candidates may contribute their parts into the next generation. Thus in the first step subtrees are selected, removed from the parents and inserted into a *contribution pool*. This means only random node selection, which, as seen previously, can be done in logarithmic time. When nodes, that is subtrees are selected, the label, or any attribute of the root node can be checked, and according to this information the selected subtrees can be categorized. In the second step these subtrees are inserted back into the parents but in a random order. Insertion means logarithmic time again, because of the updates. Selecting subtrees from the appropriate categories ensures the correctness of the operator. This modified crossover has the same effect as the original crossover, but it works globally and not locally. The outline of this operator can be seen in Figure 7.

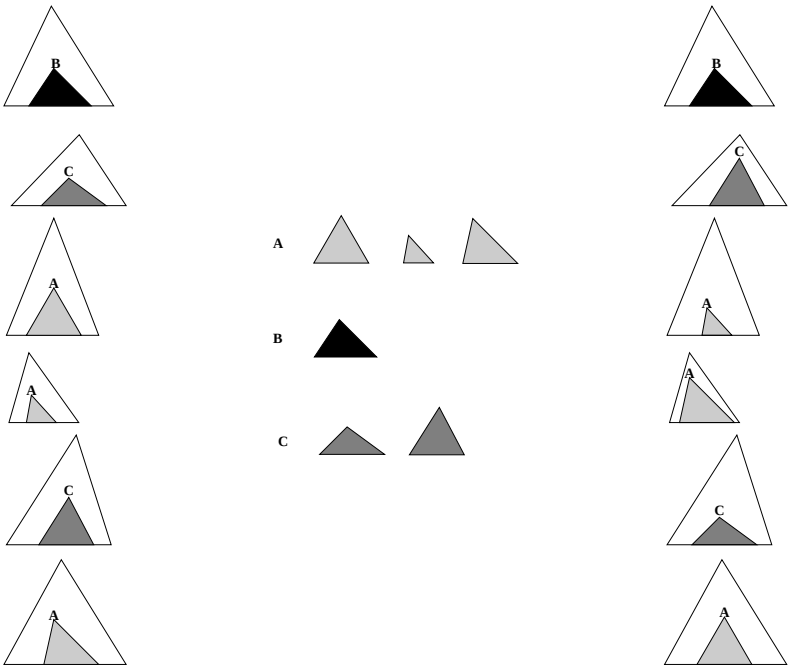


Fig. 7. Global tree-crossover

As it can be seen in the figure, this operator is similar to crossover, but subtrees are swapped among more trees (A). It may happen, that from a certain type of subtree only one can be found in the contribution pool (B). In this case this subtree is automatically inserted back to the tree where it comes from. In

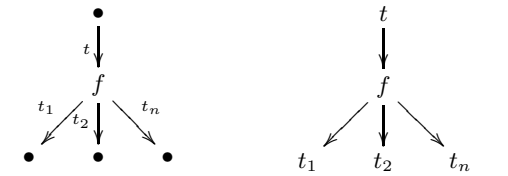
several cases the effect of the global crossover is the same as the effect of the usual crossover (C).

4.3 Simulating STGP with DTGP

If one examines both Strongly Typed Genetic Programming and Derivation-Tree Based Genetic Programming, it is easy to find a similarity between how the trees are built in both of these methods. The difference is only in the rules that determine the structure. In case of parse trees the rules are the signatures, in case of derivation trees, the rules are the rewriting rules of the grammar. Let us consider an arbitrary function with the appropriate signature

$$t\ f(t_1, t_2, \ldots, t_n).$$

This signature creates the subtree depicted in Figure 8a) where dots represent attachment points of a given type. These attachment points can be however replaced by real nodes as it can be seen in Figure 8b). These nodes in turn stand



a) in a parse tree b) in a derivation tree

Fig. 8. Signatures as subtrees

for nonterminal symbols in case of a derivation tree. Thus the given signature can be represented by the rewriting rules

$$T \to F, \text{ and } F \to T_1 T_2 \ldots T_n.$$

An example for a complete grammar for the previously mentioned expression example is $G_{stgp} = (N_{stgp}, \Sigma_{stgp}, P_{stgp}, REAL)$ with

$$N_{stgp} = \{REAL, +, -, *, /, \neg, |, \&, if\}$$

$$\Sigma_{stgp} = \{0, 1, true, false\}$$

$$REAL \to +\ ;\ -\ ;\ *\ ;\ /\ ;\ 0\ ;\ 1 \quad BOOL \to \neg\ ;\ |\ ;\ \&\ ;\ if$$

$$\begin{array}{ll} + \to REAL\ REAL & \neg \to BOOL \\ - \to REAL\ REAL & | \to BOOL\ BOOL \\ * \to REAL\ REAL & \& \to BOOL\ BOOL \\ / \to REAL\ REAL & if \to BOOL\ REAL\ REAL \end{array}$$

To make the nodes that represent functions keep the given signature, the evolutionary operators must not change a tree at such nodes. Thus, evolutionary change can only happen at nodes representing types. This modification, however,

can easily be implemented by setting the selection weight to 0 in case the node is representing a function.

It is easy to see that when the rewriting rules are given in the described way and the selection weight is set to 0 for function nodes, the DTGP will have practically the same behavior as the original STGP. This approach use larger individuals, though only twice as large, thus the memory consumption is scaled up only by a factor.

5 Experimental Results

To see the importance of the modified selection operator the comparison of the running times using the linear and the logarithmic random node selections can be seen in Figure 9. Three running times are plotted against the breadth of

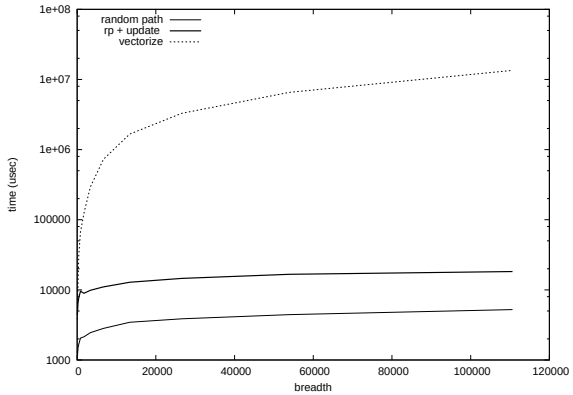


Fig. 9. Running times for linear and logarithmic random node selections

the tree using a logarithmic scale. The first method is the described random path traversal, the next one is random path traversal and attribute update together. The third one is the original selection method, where each node had to be checked, and the candidate nodes had to be put into a vector. [6]

It is obvious that even though the random node selection can be done quickly, the time needed to carry out the original crossover operator has the same order as the time needed for a tree vectorizing. Thus without the modified crossover operator the new random node selection could not save much time. Fortunately the newly-designed global crossover needs a similar amount of time as the node selection. This was tested on 1000 trees having depths between 5 and 25. The time needed for 10000 node selections and the time needed to create a new individual 10000 times using global crossover were plotted against the breadth of the trees in Figure 10.

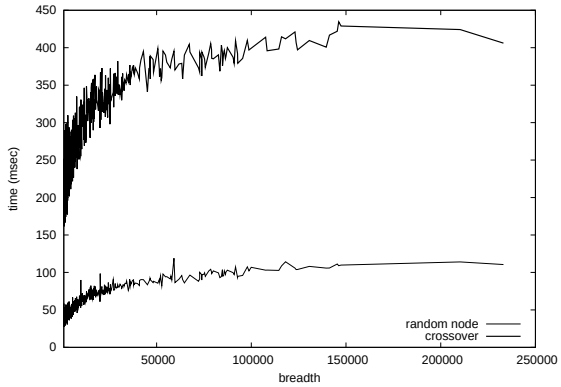


Fig. 10. Running times for random node selection and crossover

6 Conclusion

In this paper three methods for evolving syntactically correct objects were compared, the main ideas, representations and the operators were examined. The Derivation-Tree Based Genetic Programming was described in more detail, and it was shown, how this method can be improved by using a newly-designed crossover operator. Furthermore, it was presented how Strongly-Typed Genetic Programming can be simulated by DTGP which shows that DTGP can be used as a drop-in replacement for STGP.

The most important future work is to experimentally compare DTGP and GE, which was previously not possible, since with the original crossover operator the DTGP method was for larger problems practically unusable. It would be also interesting to see whether the evolutionary runs of STGP and DTGP differ, and to test DTGP for many real-world problems to prove its usability.

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, Massachusetts (1992)
2. Montana, D.J.: Strongly typed genetic programming. Technical report, Bolt Banek and Newman Inc. (1993)
3. O'Neill, M., Ryan, C.: Grammatical Evolution - Evolving Programs in an Arbitrary Language. Kluwer Academic Publishers (2003)
4. Ványi, R., Zvada, Sz.: Avoiding syntactically incorrect individuals via parameterized operators applied on derivation trees. In: Proceedings of the 2003 Congress on Evolutionary Computation CEC2003, Canberra, IEEE Press (2003) 2791–2798
5. Keijzer, M., Ryan, C., O'Neill, M., Cattolico, M., Babovic, V.: Ripple crossover in genetic programming. In Miller, J., et al., eds.: Proceedings of EuroGP'2001. Volume 2038 of LNCS., Lake Como, Italy (2001) 74–86
6. Zvada, Sz., Ványi, R.: Improving grammar based evolutionary algorithms via attributed derivation trees. In Keijzer, M., et. al., eds.: Genetic Programming 7th European Conference, EuroGP 2004, Proceedings. Volume 3003 of LNCS., Coimbra, Portugal, Springer-Verlag (2004) 208–219