Grammar Design for Derivation Tree Based Genetic Programming Systems

Stefan Forstenlechner(✉), Miguel Nicolau, David Fagan, and Michael O'Neill

Natural Computing Research and Applications Group, School of Business, University College Dublin, Dublin, Ireland
stefan.forstenlechner@ucdconnect.ie,
{miguel.nicolau,david.fagan,m.oneill}@ucd.ie

Abstract. Grammar-based genetic programming systems have gained interest in recent decades and are widely used nowadays. Although researchers normally present the grammar used to solve a certain problem, they seldom write about processes used to construct the grammar. This paper sheds some light on how to design a grammar that not only covers the search space, but also supports the search process in finding good solutions. The focus lies on context free grammar guided systems using derivation tree crossover and mutation, in contrast to linearised grammar based systems. Several grammars are presented encompassing the search space of sorting networks and show concepts which apply to general grammar design. An analysis of the search operators on different grammar is undertaken and performance examined on the sorting network problem. The results show that the overall structure for derivation trees created by the grammar has little effect on the performance, but still affects the genetic material changed by search operators.

Keywords: Grammar design · Derivation trees · Genetic programming

1 Introduction

Grammars are an important representation in computer science, especially for programming languages and compilers. Grammars have gained popularity in genetic programming (GP) over time as they overcame some of traditional GP's limitations. Grammar based genetic programming is widely used nowadays [14]. A search space for a problem in GP can easily be defined with a grammar and even problem specific information can be added to bias the search in a certain region [18,27]. Much research has been conducted on different grammars that can be used in GP to solve certain problems. Nevertheless, how to design a grammar for GP remains an under explored research area. McKay et al. wrote in a survey about grammar-based genetic programming [14]:

> "While experienced practitioners of each representation form have some tacit understanding of how to choose grammars, there is little explicit

M. Heywood et al. (Eds.): EuroGP 2016, LNCS 9594, pp. 199–214, 2016.
DOI: 10.1007/978-3-319-30668-1_13

knowledge. More explicit knowledge may lead to more structured methodologies (and interactive software support) to incrementally find good representations for new problem domains, and even to partial or complete automation of the process."

However some limited studies have been undertaken, including Whigham [27], Hemberg [6], Murphy [17] and Nicolau [19].

This paper makes a step towards analysing grammar design and how it influences the search process. Section 2 gives an overview of how grammars have been used in evolutionary systems. Section 3.3 explains the experimental setup. The results are presented and discussed in Sect. 4. Finally, conclusion is presented in Sect. 5 as well as possible future work.

2 Grammars in Evolutionary Systems

2.1 Grammar Guided Genetic Programming

Many different grammar based genetic programming system have been introduced. The broad term Grammar Guided Genetic Programming (GGGP) will be used in this paper to address these systems.

Some important GGGP systems are mentioned in this section. Whigham's CFG-GP system [27] uses a context-free grammar to specify the syntax of solutions. He also defined specialized crossover and mutation operators for the search. The search operators manipulate the derivation trees that are created when a sentence is derived from a grammar.

Another well-known grammar-based system is grammatical evolution (GE), which also uses CFGs normally in Backus-Naur Form (BNF) [5,22], but has also been adapted to employ other grammars mentioned below.

In contrast to CFG-GP, GE uses an integer string as representation instead of the derivation tree. A mapping process is used to generate a derivation tree from the linear representation, which describes the phenotype. The search operators used in GE primarily operate on the integer string. Other systems integrated context-sensitive grammars into GP [25], logic grammars [8,28], tree adjoining grammars [7,16], attribute grammars [1]. Also shape grammars have been used for design by O'Neill et al. [21].

Depending on the grammar used, it might provide special features; context free grammars, for example, can be used to interpret non-terminals differently in different contexts.

Major benefits of GGGP systems are that the closure property is implicitly given by the grammar similar to strongly typed GP [15] and bias can easily be incorporated into the grammar. Bias is often an unwanted property of representations or operators. In grammars bias can be used as an advantage [27]. A grammar can be adapted to influence the search based on a priori knowledge about a problem. Bias can be subtle, by increasing the frequency of a symbol in a grammar or more definite by forcing a certain structure on all problems. For example, the first few bits of a binary string can be defined to be all zeros.

The risk of bias is that the grammar might not cover the whole search space or worse that the global optimum is not even in the search space any more [14].

GGGP systems can be classified by their representation into tree based grammar guided and linearised grammar guided systems. This paper mainly focuses on tree based genetic programming systems, but differences to linearised representations about grammar design are mentioned.

2.2 Grammar Design

GGGP systems can be used similar to fixed and variable length GAs [23] as well as a generalization of GP, as shown by Whigham in his thesis [27]. A variable length GA representation for binary strings in BNF can be achieved by a grammar, as shown in Fig. 1.

```
<string> ::= <string><bit> | <bit>
<bit> ::= 0 | 1
```

Fig. 1. Variable length GA like representation for binary strings in BNF.

A fixed length representation in BNF is not more difficult than a variable length representation, but more rules are required, as for every position in the representation a separate rule has to be created, as shown in Fig. 2. A single rule with one production with a fixed number of `<bit>` non-terminal symbols could be used, but then crossover in a derivation tree based system could only change single bits. A linearised grammar guided system like GE can represent this grammar with exactly the same number of integers as bits are in the representation, because the mapping process only uses an integer when deciding which production to choose. As all rules except `<bit>` have only a single rule, exactly n (number of *stringparts*) integers are needed. In a derivation tree based system, crossover can only exchange subtrees with the same symbol, therefore the structure of the derivation trees non-terminal symbols will never change.

```
<stringpart0> ::= <stringpart1><bit>
<stringpart1> ::= <stringpart2><bit>
...
<stringpartN> ::= <bit>
<bit> ::= 0 | 1
```

Fig. 2. Fixed length GA representation for binary strings in BNF.

Standard GP consists of a function and terminal set, with the important property of closure, which means that every function has to be capable to evaluate any possible input it gets. Any GP representation with a function set

```
<tree> ::= f1 tree ... tree | f2 tree ... tree | ... | fx tree ... tree
         | t1 | t2 | ... | ty
```

Fig. 3. General standard GP grammar in BNF.

$f_1, f_2, ..., f_x$ and terminal set $t_1, t_2, ..., t_y$ can be represented with the general grammar in Fig. 3. Note that any function can have an arbitrary number of inputs and that only a single rule is required to represent the function and terminal set of GP due to the closure property.

For a detailed discussion about grammars for GA and GP representations as well as the schema theorem for such grammars, see Whigham's thesis [27].

2.3 Structure in Grammars

Creating variable length derivation trees requires direct or indirect recursion in a grammar as shown in Sect. 2.2. A direct left recursion, e.g. `<rule> ::= <rule><part> | <part>` does that trick. `<rule>` and `<part>` can be replaced with any rule like `<string>` and `<bit>` see Fig. 1, `<code>` and `<line>` such as Santa Fe Ant Trail problem [22], `<for>` and `<code>` for program synthesis [20], `<design>` and `<component>` for creating designs [13], `<int_constant>` and `<number>` for integer constant creation [4], etc.

If a derivation tree is drawn that has been created by this rule, it is more similar to a "list" than a tree, as depicted in Sect. 3.2 for a similar grammar in Fig. 6. It will be a list of `<part>` non-terminals. The reason is that it only expands in one direction (unless there is an indirect recursion from `<part>` to `<rule>`). The question about this fairly commonly used rule is, if it should be expressed in another way. As the operators applied to the derivation tree are tree based, the grammar might improve the search if it would express more tree-like structures as in standard GP. For this purpose, we choose a problem, described in Sect. 3.1, which can be expressed with a short grammar. Multiple grammars, discussed in Sect. 3.2 are presented. All of them cover the search space for the problem, but present different properties.

3 Experimental Setup

3.1 Sorting Network

For the grammars and the experiments in this study we use the sorting network problem [10,24]. The reason why we choose this problem is that it is a real-world problem and simple grammars can be written which cover all possible solutions. At the same time a grammar for sorting networks has properties that apply to other grammars and can be generalized from.

A sorting network can be seen as a sorting algorithm in hardware with a fixed number of inputs and outputs. The output of a sorting network is the input in sorted order. The sorting network consists only of wires and comparator modules.

Comparators take two wires as input and swap the values of these wires if they are not in the correct order, otherwise they return the input.

When drawing a sorting network, the wires are represented as straight lines and the comparator modules are connections between the wires. An optimal sorting network with four inputs is shown in Fig. 4. It is important to notice that the order of the connections makes a difference. For example, if you would put the last connection (on the right side) before the first connection, the sorting network would not return correct values for all inputs.

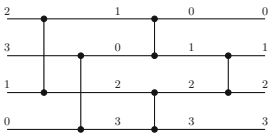

Fig. 4. Sorting network with four inputs and five comparators.

A sorting network has three properties. The number of inputs, the size or number of connections and its depth. The depth is the number of steps it takes to complete the network. One step can execute multiple comparisons at once, if the comparisons are independent of each other, which means they do not depend on the output of the other comparisons in the same step. For example, the network in Fig. 4 has 4 inputs, 5 connections and is of depth 3.

Testing the correctness of sorting networks is a computational expensive task. Fortunately not all possible combinations of inputs have to be tested, which would result in a runtime complexity of $n!$. The zero-one principle [9] says that if all combinations of 0, 1 as inputs are sorted correctly by a network, all other arbitrary values will be sorted correctly. This reduces runtime complexity to 2^n.

3.2 Experimental Grammar Design

In this section we present multiple grammars which can be used in GGGP systems to evolve sorting networks. The phenotype the grammars create are lists with an even amount of numbers and at least one pair of numbers, e.g. 0 2 1 3 0 1 2 3 1 2 represents the sorting network in Fig. 4. Therefore, a grammar needs a recursive rule to create arbitrary amount of pairs of numbers and always has to append two numbers to the phenotype at once.

Section 2.3 showed a common grammar used to address such a grammar design problem, but this grammar may hinder the search operators by forcing them to exchange large parts of individuals, due to the list like structure of the derivation trees. To address this issue, we assume that grammars that describe more tree like structures, may be beneficial to find optimal solutions, as derivation tree based operators can exchange variable amounts of genetic material anywhere in an individual.

The grammars are context free grammars and written in (BNF). Note that all grammars have a rule `<node>`, which can be expressed as a number from $0-3$ as the sorting network presented in Sect. 3.1 and for reasons of simplicity. Depending on the number of inputs of the network this rule has to be adapted.

Grammar 1 (G1). A simple grammar is depicted in Fig. 5. The first production of `<snet>` is a left recursion and the second production stops the recursion, as shown in Sect. 2.3. In both productions two non-terminals `<node>` are used so that two numbers are added in every recursion step. This grammar consist of only two rules and one production which creates the structure of the derivation tree. An example of a derivation tree created with G1 is shown in Fig. 6. The derivation tree has more in common with a list of pairs of numbers than a tree. The rule `<snet>` is only responsible for creating the structure or more specifically the length of the list.

```
<snet> ::= <snet> <node> <node> | <node> <node>
<node> ::= 0 | 1 | 2 | 3
```

Fig. 5. Simple grammar, which works similar to a list (G1).

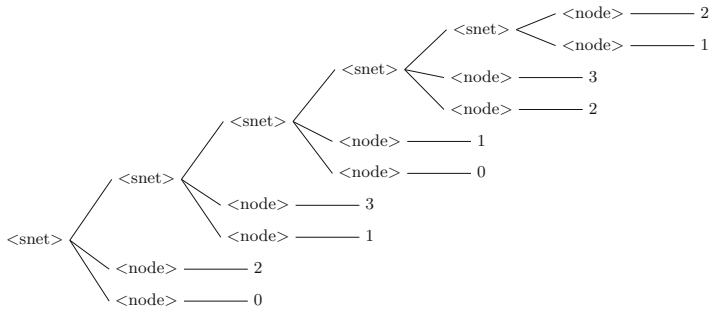

Fig. 6. Grammar 1 derivation tree for the optimal sorting network with four inputs, see also Fig. 4.

Crossover can only exchange single numbers, if applied to `<node>`. If applied to `<snet>` all pairs of numbers from the first parent will be in the child up to the crossover point, where all the pairs of numbers from the second parent will be inserted. In case of the phenotype, subtree crossover on G1 is similar to single-point crossover in genetic algorithms, limited to crossover after an even amount of numbers. It is not possible to exchange sorting network comparisons somewhere in the middle of a list of numbers with subtree crossover in G1.

Grammar 2 (G2). The next grammar is very similar to G1, see Fig. 7. Again the structure the derivation tree creates, looks more like a list than a tree. The difference to G1 is that every pair of numbers is encapsulated in a separate rule `<nodes>`, which represents a single compare operation.

```
<snet>  ::= <snet> <nodes> | <nodes>
<nodes> ::= <node> <node>
<node>  ::= 0 | 1 | 2 | 3
```

Fig. 7. Simple grammar, which works similar to a list with the benefit that pairs of nodes can be exchanged (G2).

The benefit of G2 is that crossover can exchange a single compare operation between parents, even if located in the middle of the list of comparisons. Mutation can also replace a comparison, whereas in G1 it either changes a single number or every number after the mutation point in the tree, which can be very destructive.

Grammar 3 (G3). A grammar that can generate more tree-like derivation structures is shown in Fig. 8. Additionally to the left recursion, its complement a right recursion has been added, as well as production that can have two child `<snet>` nodes. Therefore, binary trees can be created with G3. Note that every production of `<snet>` also creates a pair of numbers. An example of a derivation tree created with G3 is shown in Fig. 9.

The benefit of G3 is that the tree structure provides subtree crossover with more possibilities for crossover points. Instead of operating on a "list", where

```
<snet> ::= <snet> <node> <node> | <node> <node> <snet>
         | <snet> <node> <node> <snet> | <node> <node>
<node> ::= 0 | 1 | 2 | 3
```

Fig. 8. Tree-like grammar, where every production contains a pair of nodes (G3).

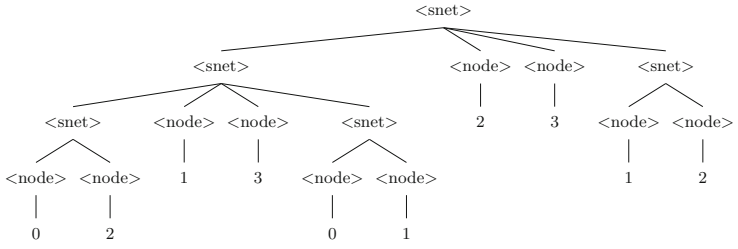

Fig. 9. Grammar 3 derivation tree for the optimal sorting network with four inputs, see also Fig. 4.

crossover takes the first part of the first parent and the second part from the second parent, crossover on G3 can exchange any number of compare-exchange operations anywhere in the tree. Furthermore, mutation might not be as destructive as in G1 and G2 any more, because smaller subtrees might be mutated instead of the whole tail of a "list".

Grammar 4 (G4). The fourth grammar provides the same tree structure as G3, but with the same modification that has been added in G2. A separate rule `<nodes>` has been added which encapsulates a pair of numbers, to easily exchange a single compare operation anywhere in the tree (Fig. 10).

```
<snet>  ::= <snet> <nodes> | <nodes> <snet>
          | <snet> <nodes> <snet> | <nodes>
<nodes> ::= <node> <node>
<node>  ::= 0 | 1 | 2 | 3
```

Fig. 10. Binary tree-like grammar, where every node contains a pair of nodes and a pair of nodes can be exchanged individually (G4).

Grammar 5 (G5). The last grammar is short and simple, but it can also create binary trees. It does not need a separate rule `<nodes>` to exchange single comparisons, because every pair of nodes can already be exchanged on its own when their parent node is exchanged (Fig. 11).

```
<snet> ::= <snet> <snet> | <node> <node>
<node> ::= 0 | 1 | 2 | 3
```

Fig. 11. Binary tree-like grammar. Nodes are not in the structure of the tree and pairs of nodes can still be exchanged individually (G5).

G5 probably provides the easiest and most readable way to create binary trees with a grammar. Additionally, it can easily be adapted to any n-ary trees by adding any number of non-terminal `<snet>` as production to the rule `<snet>`.

Derivation Tree Sizes. As grammars define the structure of the derivation trees, they also define the number of nodes and the depth of derivations trees needed to form a solution. In the grammars above, a pair of numbers represents a comparison operation in a sorting network. The minimum number of nodes and the minimum depth required for representing a certain number of comparisons is given in Table 1, which will be used in Sect. 3.3. Note that every production will be treated as a single node with one child node for every non-terminal in the production. This does not change the behaviour of the derivation trees or search operators, but decreases the number of nodes in a tree.

Table 1. Minimum number of nodes and minimum depth for each grammar given a certain number of comparisons (c)

Grammar	Number of nodes	Depth
G1	$3 * c$	$c + 1$
G2	$4 * c$	$c + 2$
G3	$3 * c$	$\lceil \log_2(c + 1) \rceil + 1$
G4	$4 * c$	$\lceil \log_2(c + 1) \rceil + 2$
G5	$4 * c - 1$	$\lceil \log_2(c) \rceil + 1$

Grammar Design Details. The grammars presented in this section have been written concerning the derivation tree they will create and how search operations which use the derivation tree might behave on them. These grammars can also be used by other GGGP systems which have a linear representation like GE, but keep in mind that systems with a linear representation do behave differently. For example, G1 and G2 should yield the same results in GE, because the rule `nodes` that has been added, has only one production rule. The mapping process in GE automatically replaces non-terminals with its production rule, if only one is available. The same applies for G3 and G4.

One additional change that may improve the grammars would be to change the rule `<nodes>` to all possible comparisons for a given number of inputs of a sorting network as depicted in Fig. 12. All given grammars can represent all combinations of pairs of numbers which are n^2. If only all possible compare-exchange operations are used, then it reduces the number of pairs to $\frac{n^2-n}{2}$, because duplicates can be removed (for example, 0 1 and 1 0 represent the same operation). And pairs with the same number twice are ignored as comparisons with the same input would not do anything. It might still be beneficial to allow one pair with the same number or an empty production of `<nodes>`, so that a compare-exchange operation can be deleted.

```
<nodes> ::= 0 1 | 0 2 | 0 3
          | 1 2 | 1 3
          | 2 3
```

Fig. 12. All possible comparisons in a sorting network with four inputs.

This and further optimizations of the rule `<nodes>` have not been investigated, because they would be problem specific, whereas changing the structure of the derivation tree through the grammar and encapsulating non-terminals into new rules can be used in any grammar.

3.3 Experiments

The experiments performed for this paper have been executed with HeuristicLab [26] and a plugin which we added that can be found on GitHub[1].

Two experiments are performed on the grammars presented in Sect. 3.2. As the grammars define different structures, the derivation tree operators, crossover and mutation, may behave differently. They are going to add and remove different amounts of genetic material and choose other nodes to exchange material.

Experiment 1. The first experiment is to analyse the grammars and the difference of genetic material that gets exchanged between individuals depending on the grammars. One time we only use crossover, the second time we only use mutation and the last time we use both operators. No evaluation is performed and selection is done randomly as we are only interested how the search operators behave. In all three cases we use 100 % probability for crossover and mutation. The derivation trees are limited to 50 compare-exchange operations for this experiment, see Table 1 for the number of nodes.

Experiment 2. In the second experiment, we want to know if any of the grammars has a performance advantage over the others. Therefore, fitness is measured and tournament selection is used. We choose a sorting network with twelve inputs as problem to compare the grammars. The optimal number of comparators is not yet known, but it has been proven that it lies between 37 to 39 comparators [2]. Due to the zero-one principle the 12 input sorting network has 4096 training cases. As fitness function we minimize the number of incorrect sorted inputs plus the number of used comparators divided by 100, as in Koza et al. [11]. Therefore, the main objective will be to sort the inputs correctly and the subsidiary goal is to minimize the size of the sorting network. We limit the maximum number of comparators to 59, which is 1.5 times the upper bound rounded up.

General Settings. The settings of the experiments are summarized in Table 2. The differences between the experiments are marked with superscripts.

The initialisation of the individuals is done with the Probabilistic Tree-Creation 2 (PTC2) [12], because PTC2 gives us the possibility to limit the number of nodes of the initial trees and not only the depth. Setting a max depth for the initialisation, like it is done for ramped half-and-half initialisation and also grow or full method, would not give a fair comparison between the grammars. The grammars produce different structures and therefore derivation trees can have completely different amount of nodes for a certain depth, which would make it impossible to compare the results. Additionally, Daida et al. [3] showed that standard GP with binary trees searches rather sparse than dense trees. Therefore, we decided to define the number of compare-exchange operations that are allowed and calculated the required number of nodes in a derivation tree for every grammar. So we set the number of nodes for every experiment individually depending on the grammar, according to Table 1. Therefore, the

[1] https://github.com/t-h-e/HeuristicLab.CFGGP.

Table 2. Experimental parameter settings. [1]Experiment 1. [2]Experiment 2.

Parameter	Setting
Runs	100^1, 50^2
Generations	100
Population size	1000
Population initialisation	PTC2 [12]
Tournament size	7
Internal crossover probability	0.9
Mutation probability	$100\,\%^1$, $5\,\%^2$
Elite size	0^1, 1^2
Maximum compare-exchange operations	50^1, 59^2

structure of the derivation tree is not limited by depth and arbitrary trees only limited by the number of nodes can be created.

4 Results

This section presents the results of the experiments. Note that no fitness evaluation was used in experiment one, because only the behaviour of the search operators was observed.

4.1 Experiment 1

Changing grammars to more tree-like structures has an obvious effect on crossover and mutation, which is that smaller amounts of genetic material can be exchanged as Fig. 13 shows. The plots for the experiments where only crossover or mutation was used are omitted, as they are quite similar. The only difference is that the trees are shrinking over time in the experiments where only crossover is used. The reason is that the trees are limited by a maximum number of nodes. When crossover selects a subtree from the second parent, it has to select a subtree which does not violate this limit. Therefore, the chance of creating an overall smaller tree is more likely, when the tree is already rather big. But the size of the trees stabilizes after 30 generations. When using mutation only, the same amount of genetic material is removed and added again. In the case of using crossover and mutation, this phenomena is only observed up to the tenth generations, but crossover continues to remove more genetic material than it is adding. Mutation counteracts crossover by adding more material.

In the case of G1, crossover and mutation take place on `<snet>` most of the time, as this is the node which is most frequent in the trees. The most frequent exchanged symbols by crossover with G2 are `<snet>` and `<nodes>`. The frequency of `<nodes>` is slightly higher as there is always one more `<nodes>` than `<snet>` non-terminal. For G3, G4 and G5, it is obviously `<snet>` and the child node

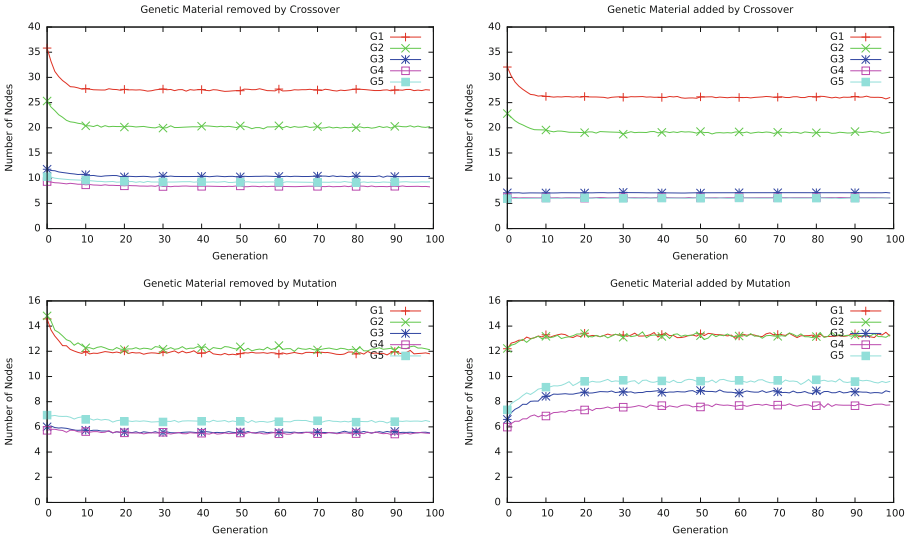

Fig. 13. Genetic material added and removed when using crossover and mutation.

(from the second parent) is mostly `<node> <node>`. The reason is that crossover chooses uniformly from all nodes in the tree and therefore smaller trees are more likely to be selected, as there is a higher amount of smaller subtrees. Because we use Koza's crossover where a probability is used to decide whether an internal node or a leaf node should be chosen, with a 90 % probability for internal nodes. As crossover still favours smaller subtrees, `<node> <node>` gets exchanged most frequently. For mutation there is no such probability, which explains why leaf nodes are changed more frequently.

The first experiment showed that crossover on grammars which create tree-like structures can exchange smaller amounts of genetic material between individuals. Mutation changes smaller amounts as well. The use of the extra rule `<nodes>` had also on effect on the change of genetic material, see e.g. G1 and G2 in Fig. 13, but not as much as the tree-like structure.

4.2 Experiment 2

When we look at the results in Table 3, we can see that G2, G4 and G5 are doing better than G1 and G3. G2 and G4 have the extra rule `<nodes>` to be able to exchange comparators individually, which G5 can also do without that extra rule. The tree structure of the grammar seems to give G3 a slight advantage over G1, but the difference is not statistically significant, similar to the difference between G4 and G2. Although G5 also has a tree structure, it is not doing better than G2.

A more difficult sorting network with 14 inputs is used to repeat the same experiment with G2, G4 and G5, to check if it might create a statistically

Table 3. Results for sorting networks with 12 inputs with the average best fitness, standard deviation, median, best individual and success ratio over 50 runs.

	Average best fitness ± Std dev	Median	Best	Success ratio
G1	82.604 ± 42.794	77.58	18.56	0 %
G2	31.851 ± 21.662	26.58	0.58	4 %
G3	65.600 ± 47.892	51.57	14.58	0 %
G4	23.528 ± 15.472	19.57	0.53	8 %
G5	34.130 ± 28.255	28.59	0.55	2 %

significant difference. The results, shown in Table 4, are very similar. On the one hand, G4 was again doing slightly better. On the other hand, G5 is doing worse than G2. So there is no way to say that the tree structure improves the results.

Table 4. Results for sorting networks with 14 inputs with the average best fitness, standard deviation, median and best individual over 50 runs. No correct sorting network was found.

	Average best fitness ± Std dev	Median	Best
G1	769.833 ± 303.508	724.74	256.74
G2	245.515 ± 143.939	214.75	26.77
G3	575.554 ± 300.543	539.76	102.75
G4	230.196 ± 119.649	201.76	8.73
G5	297.675 ± 197.856	224.75	40.75

After examining the results from the experiments, we noticed that the main reason G2, G4 and G5 are doing better is that they can exchange comparators individually. The Koza style crossover favours internal nodes over leaf nodes, but that does not change the fact that smaller trees are exchanged more often. Therefore single comparators are exchanged quite frequently when using these grammars, whereas leaf nodes rarely get exchanged. So we performed an additional experiment where crossover was changed to select nodes in the tree with an equal probability to see the effect of exchanging even smaller bits of information. The results are shown in Table 5. Now that single nodes can be exchanged more frequently, the results have completely changed and improved for all grammars.

Using crossover which selects from all nodes with equal probability is useful for this specific problem as single numbers are exchanged frequently. If more rules are used which created bigger subtrees in the non-recursive part, this crossover might have a negative effect.

Experiment 2 indicates that the structure of the grammar has only little influence in performance, as G2 shows similar results as G4 and G5. Adding the

Table 5. Results for sorting network with 12 inputs with subtree crossover that chooses from all nodes in the tree with equal probability.

	Average best fitness ± Std dev	Median	Best	Success ratio
G1	9.719 ± 8.621	8.58	0.50	20 %
G2	11.809 ± 11.209	8.58	0.53	22 %
G3	11.767 ± 12.827	8.58	0.53	18 %
G4	12.725 ± 11.410	11.56	0.53	16 %
G5	13.286 ± 11.907	11.55	0.52	16 %

extra rule `<nodes>` improved the performance, because it encapsulated a small piece of information for the problem and was exchanged more often than leaf nodes. Changing the crossover improved the performance on the sorting network problem, but for grammars where the non-recursive part might express a deeper derivation tree, it might not have any effect.

5 Conclusion and Future Work

This paper presented some general concepts on how to design a grammar, especially possibilities on how to write grammars that produce variable length phenotypes, so that the derivation tree does not become "list-like". The grammars were analysed in terms of the behaviour of the applied search operators. Crossover and mutation were able to exchange arbitrary amounts of genetic material within these trees in grammars that created tree-like structures. The second set of experiments analysed the impact of grammar design for tree based GGGP on performance on sorting networks, particularly in the definition of recursive rules for derivation tree based operators. Although the Koza style crossover helps exchange bigger amounts of genetic material, it interferes with the search in this problem instance. If Koza's crossover should be used for tree based grammar guided GP systems, cannot be inferred by these experiments alone. It might be beneficial for problems which create bigger subtrees in the non-recursive part of a grammar.

Nevertheless, the results of the experiments showed that the structure of the underlying derivation tree created by a grammar seems to have no or only little effect on the search given the search operators employed in this study, if the grammar is not biased towards certain solutions and the language of the grammars is equivalent. This conclusion can be seen positive, as this means that no particular attention has to be paid to this aspect, when designing grammars.

Further investigation is needed, if grammars with non-recursive parts that create bigger subtrees than the rule `<nodes>`, show the similar results.

Acknowledgments. This research is based upon works supported by the Science Foundation Ireland, under Grant No. 13/IA/1850.

References

1. Cleary, R., O'Neill, M.: An attribute grammar decoder for the 01 multiconstrained knapsack problem. In: Raidl, G.R., Gottlieb, J. (eds.) EvoCOP 2005. LNCS, vol. 3448, pp. 34–45. Springer, Heidelberg (2005)
2. Codish, M., Cruz-Filipe, L., Frank, M., Schneider-Kamp, P.: Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). CoRR (2014)
3. Daida, J., Hilss, A.: Identifying structural mechanisms in standard genetic programming. In: Cantú-Paz, E., et al. (eds.) Genetic and Evolutionary Computation — GECCO 2003. LNCS, vol. 2724, pp. 1639–1651. Springer, Heidelberg (2003)
4. Dempsey, I., O'Neill, M., Brabazon, A.: Constant creation in grammatical evolution. Int. J. Innovative Comput. Appl. **1**, 23–38 (2007)
5. Dempsey, I., O'Neill, M., Brabazon, A.: Grammatical evolution. In: Dempsey, I., O'Neill, M., Brabazon, A. (eds.) Foundations in Grammatical Evolution for Dynamic Environments. SCI, vol. 194, pp. 9–24. Springer, Heidelberg (2009)
6. Hemberg, E.: University College, D.S.o.C.S.I. An exploration of grammars in grammatical evolution. Ph.D. thesis, University College Dublin, Ireland (2010)
7. Hoai, N.X., McKay, R., Essam, D.: Representation and structural difficulty in genetic programming. IEEE Trans. Evol. Comput. **10**(2), 157–166 (2006)
8. Keijzer, M., Babovic, V., Ryan, C., O'Neill, M., Cattolico, M.: Adaptive logic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), California, USA, pp. 42–49, 7–11 July 2001
9. Knuth, D.E.: The Art of Computer Programming. Sorting and Searching, vol. 3, 2nd edn. Addison Wesley Longman Publishing Co. Inc, Redwood City (1998)
10. Koza, J.R., Andre, D., Bennett, F.H., Keane, M.A.: Genetic Programming III: Darwinian Invention & Problem Solving, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (1999)
11. Koza, J.R., Bennett I, F.H., Hutchings, J., Bade, S., Keane, M.A., Andre, D.: Evolving sorting networks using genetic programming and the rapidly reconfigurable xilinx 6216 field-programmable gate array. In: Conference Record of the Thirty-First Asilomar Conference on Signals, Systems amp; Computers, vol. 1, pp. 404–410, November 1997
12. Luke, S.: Two fast tree-creation algorithms for genetic programming. IEEE Trans. Evol. Comput. **4**(3), 274–283 (2000)
13. McDermott, J., Swafford, J.M., Hemberg, M., Byrne, J., Hemberg, E., Fenton, M., McNally, C., Shotton, E., O'Neill, M.: String-rewriting grammars for evolutionary architectural design. Environ. Plann. B Plann. Des. **39**(4), 713–731 (2012)
14. McKay, R., Hoai, N., Whigham, P., Shan, Y., ONeill, M.: Grammar-based genetic programming: a survey. Genet. Program. Evolvable Mach. **11**(3–4), 365–396 (2010)
15. Montana, D.J.: Strongly typed genetic programming. Evol. Comput. **3**(2), 199–230 (1995)
16. Murphy, E., O'Neill, M., Galvapez, E., Brabazon, A. : Tree-adjunct grammatical evolution. In: 2010 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8 (2010)
17. Murphy, E.: An exploration of tree-adjoining grammars for grammatical evolution. Ph.D. thesis, University College Dublin, Ireland, 6 December 2014
18. Murphy, E., Hemberg, E., Nicolau, M., O'Neill, M., Brabazon, A.: Grammar bias and initialisation in grammar based genetic programming. In: Moraglio, A., Silva, S., Krawiec, K., Machado, P., Cotta, C. (eds.) EuroGP 2012. LNCS, vol. 7244, pp. 85–96. Springer, Heidelberg (2012)

19. Nicolau, M.: Automatic grammar complexity reduction in grammatical evolution. In: GECCO 2004 Workshop Proceedings, Seattle, Washington, USA (2004)
20. O'Neill, M., Nicolau, M., Agapitos, A.: Experiments in program synthesis with grammatical evolution: A focus on integer sorting. In: 2014 IEEE Congress on Evolutionary Computation (CEC), pp. 1504–1511, July 2014
21. O'Neill, M., McDermott, J., Swafford, J.M., Byrne, J., Hemberg, E., Brabazon, A., Shotton, E., McNally, C., Hemberg, M.: Evolutionary design using grammatical evolution and shape grammars: designing a shelter. Int. J. Des. Eng. **3**(1), 4–24 (2010)
22. O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Kluwer Academic Publishers, Berlin (2003)
23. Ryan, C., Nicolau, M., O'Neill, M.: Genetic algorithms using grammatical evolution. In: Foster, J.A., Lutton, E., Miller, J., Ryan, C., Tettamanzi, A.G.B. (eds.) EuroGP 2002. LNCS, vol. 2278, pp. 278–287. Springer, Heidelberg (2002)
24. Sekanina, L., Bidlo, M.: Evolutionary design of arbitrarily large sorting networks using development. Genet. Program. Evolvable Mach. **6**(3), 319–347 (2005)
25. Tanev, I.: Incorporating learning probabilistic context-sensitive grammar in genetic programming for efficient evolution and adaptation of snakebot. In: Keijzer, M., Tettamanzi, A.G.B., Collet, P., van Hemert, J., Tomassini, M. (eds.) EuroGP 2005. LNCS, vol. 3447, pp. 155–166. Springer, Heidelberg (2005)
26. Wagner, S., et al.: Architecture and design of the heuristiclab optimization environment. In: Klempous, R., Nikodem, J., Jacak, W., Chaczko, Z. (eds.) Advanced Methods and Applications in Computational Intelligence. TIEI, vol. 6, pp. 193–258. Springer, Heidelberg (2013)
27. Whigham, P.A.: Grammatical bias for evolutionary learning. Ph.D. thesis, New South Wales, Australia, Australia (1996)
28. Wong, M.L., Leung, K.S.: Evolutionary program induction directed by logic grammars. Evol. Comput. **5**(2), 143–180 (1997)