On the search space of genetic programming and its relation to nature's search space

Marc Ebner
Eberhard-Karls-Universität Tübingen
Wilhelm-Schickard-Institut für Informatik
Arbeitsbereich Rechnerarchitektur
Köstlinstraße 6, 72074 Tübingen, Germany
ebner@informatik.uni-tuebingen.de

Abstract- The size of the search space has been analyzed for genetic programming and genetic algorithms. It is highly unlikely to find any single individual in this huge search space. However, genetic programming with variable length structures differs from standard genetic algorithms where fixed size bit strings are used in that usually many different individuals show the same phenotypical behavior due to introns. Therefore, finding any given behavior is not as difficult as the size of the search space suggests. A quantitative analysis is presented for the number of individuals that code for the identity function. The identity function is important in the analysis of the search space because it can be used to construct individuals showing the same behavior as any given individual. Finally, an analogy is drawn to nature's sequence space which suggests possible directions for future research. The representation should be chosen such that all possible behaviors are reachable within a comparatively small number of steps from any given behavior and the individuals coding for any given behavior should be distributed randomly in the search space. In addition, long paths of neutral mutations should lead to individuals which code for the same behavior.

1 Motivation

The theory of genetic programming [10, 11, 3] is still in its infancy compared to the theory of standard genetic algorithms [7, 5]. However, the number of experimental studies which analyze the behavior of genetic programming runs is growing. The experimental studies are important because they can help to create evolutionary algorithms that perform more efficient. In addition, the theoretical and experimental results can uncover paths on how to scale the algorithms to solve larger and more difficult problems. In particular, the phenomenon known as bloat, that is the increase of the size of the individuals due to introns, has been analyzed in detail [3, 12, 13]. Goldberg and O'Reilly [6] analyzed the influence of contextual semantics on the structure of the individuals. Koza [10] analyzed the search space of genetic programming by randomly generating individuals. He looked at the difficulty of finding an 11-multiplexer, a 6-multiplexer, a 3-multiplexer, the odd-3-parity function and the exclusive-or function by random search. Koza also performed a detailed study on the search space of boolean functions with 2 and 3 arguments [10]. In this case, individuals consisting of 20 internal and 21 external nodes were randomly generated. The functions with 2 and 3 arguments can be partitioned into different equivalence classes. For each class Koza counted the number of individuals that are an instance of the corresponding class. Some functions were found easier than others.

Most experiments in genetic programming are performed at a rather abstract level with not much relation to natural evolution. In contrast to this, Banzhaf [2] developed a genetic programming paradigm which mimics natural evolution. He used a genotype-phenotype mapping which allows for a high degree of redundancy and neutral mutations. He evolved bit strings which are transcribed, edited and finally translated to yield the phenotype. Keller and Banzhaf [9] showed that this paradigm performed better on a symbolic regression problem than a common genetic programming approach.

In this paper we present results on characteristics of the search space of genetic programming. We show that although it is highly unlikely to find any single individual in this search space it is much easier to locate an individual showing the same behavior. This assumes that a representation is used where introns can occur. In particular, we counted the number of individuals that code for the identity function. Finally, we draw an analogy between genetic programming and nature's sequence space. This analogy could promote the use of a certain type of representation that makes the search much easier. To simplify the following discussion we call evolutionary algorithms that work with the traditional fixed length bit strings genetic algorithms [7, 5] and evolutionary algorithms that evolve programs or variable length structures genetic programming [10, 11, 3].

2 The size of the search space of genetic programming

One of the major differences between genetic algorithms and genetic programming is the size and the characteristics of the search space. The search space of genetic programming is usually much larger than the search space of genetic algorithms. Let l be the length of a bit string which is used to code an individual in the genetic algorithm paradigm. Then the search space of this representation contains 2^l different individuals (assuming a binary representation and a one-to-one

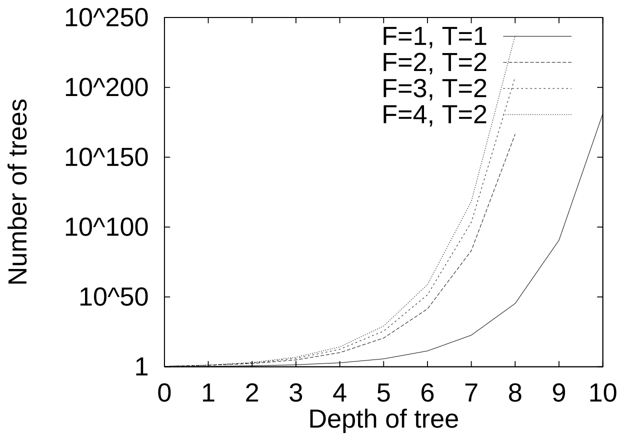

Figure 1: Number of different trees with maximum depth d. The graph has a logarithmic scale.

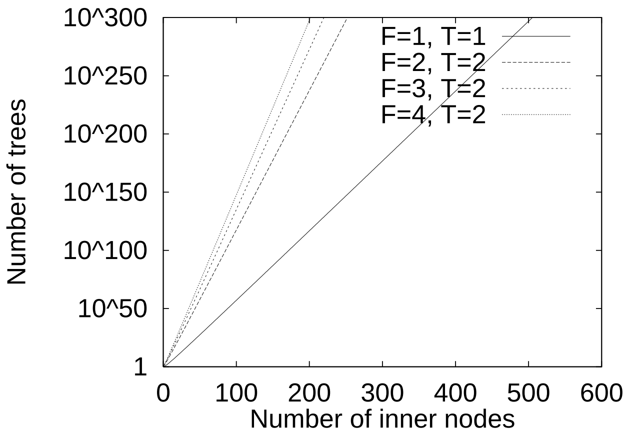

Figure 2: Number of different trees with n inner nodes. The graph has a logarithmic scale.

correspondence between genotypes and phenotypes). Now let us take a look at the size of the search space of tree-based genetic programming after which we will look at the characteristics of the search space. For simplicity we only consider binary trees. That is, we are limiting the search space to functions with only two arguments. Let F be the number of different primitive functions and let T be the number of different terminal symbols. Let Trees(d) be the number of different trees that can be represented with a maximum depth of d. A tree which has only one node, its root, has depth 0. We can calculate the number of trees that can be represented with a maximum depth of $d+1$ by choosing an arbitrary function as the root node and all possible subtrees with a maximum depth d. To this number we have to add the number of trees that consist of a single terminal symbol. Therefore, the number of different trees with a maximum depth d is given by the following recursive equation:

$$\text{Trees}(0) = T \quad (1)$$
$$\text{Trees}(d) = F \cdot \text{Trees}^2(d-1) + T \quad (2)$$

The number of different trees with a maximum depth d can also be written as

$$\text{Trees}(d) = \sum_{i=0}^{2^d-1} t_{d,i} F^i T^{i+1} \quad (3)$$

where $t_{d,i}$ is the number of trees that can be represented with i primitive functions and which have a maximum depth d. The number of different structures is obtained by setting F and T to one. One can get a grasp on the growth of the function Trees(d) by looking at the last term, the number of full trees with depth d. There exist $F^{2^d-1}T^{2^d}$ different trees that have the structure of a full binary tree.

The number of different trees for a given number of elementary functions and terminal symbols is shown in Table 1. We are using symbolic regression [10] as an example. Many optimization problems can in fact be formulated as symbolic regression. Therefore, symbolic regression is very suitable to analyze the search space of genetic programming. If one is using addition (+), subtraction (-), multiplication (*) and division (/) as elementary functions and the variable X and one constant as terminal symbols we have $F=4$ and $T=2$. Table 1 shows that already for small depths, small numbers of primitive functions and terminal symbols the search space grows rapidly. Figure 1 shows the number of different trees as a function of maximum depth.

Usually the maximum depth of a tree is not the only parameter that limits the search space of genetic programming. The maximum number of nodes of an individual is often also limited. The number of different trees that can be represented with exactly n inner nodes is specified by the following recursive equation.

$$\text{Trees}(0) = T \quad (4)$$
$$\text{Trees}(n) = \sum_{i=0}^{n-1} F \cdot \text{Trees}(n-1-i) \cdot \text{Trees}(i) \quad (5)$$

A tree with n inner nodes can be constructed by using all possible combinations of two subtrees that together have $n-1$ inner nodes. The root of this new tree can be any primitive function. We have Trees$(n) = t_{n,n}F^nT^{n+1}$. The number of different binary trees with m nodes is $\frac{1}{m+1}\binom{2m}{m}$ [4]. Therefore $t_{n,n} = \frac{1}{2n+2}\binom{4n+2}{2n+1}$ with m=2n+1. The number of different trees with a maximum of n inner nodes for different numbers of primitive functions and terminal symbols is shown in Figure 2. Table 2 shows the number of different trees with a maximum of 1000 inner nodes, a number that is commonly used in genetic programming experiments.

3 The impact of introns on the search space

We now take a look at some characteristics of the search space of genetic programming. The characteristics of the search

Maximum depth of tree	0	1	2	3	4	5	...	12
$F = 1, T = 1$	1	2	5	26	677	458330	...	4.27·10^{724}
$F = 2, T = 2$	2	10	202	81610	1.33·10^{10}	3.55·10^{20}	...	4.35·10^{2668}
$F = 3, T = 2$	2	14	590	1.04·10^{6}	3.27·10^{12}	3.21·10^{25}	...	2.80·10^{3325}
$F = 4, T = 2$	2	18	1298	6.74·10^{6}	1.82·10^{14}	1.32·10^{29}	...	7.96·10^{3803}

Table 1: Number of different trees with a maximum depth d and given number of primitive functions and terminal symbols.

Maximum number of inner nodes	0	1	2	3	4	5	...	1000
$F = 1, T = 1$	1	2	4	9	23	65	...	2.73·10^{597}
$F = 2, T = 2$	2	10	74	714	7882	93898	...	5.01·10^{1199}
$F = 3, T = 2$	2	14	158	2318	38606	691790	...	6.05·10^{1375}
$F = 4, T = 2$	2	18	274	5394	120082	2.87·10^{6}	...	5.20·10^{1500}

Table 2: Number of different trees with a maximum of n inner nodes and given number of primitive functions and terminal symbols.

space of genetic programming are different from the characteristics of the search space of genetic algorithms. With genetic algorithms one usually uses a coding where each phenotype has exactly one corresponding individual. Therefore, there usually exists only one individual which codes for the solution to the problem. In contrast to genetic algorithms with genetic programming there usually exist many different individuals in the search space that map to the same phenotype. The different individuals are a result of program code that does not add to the behavior of the phenotype. These segments are called introns in analogy to natural evolution. Introns are parts of the genotype that are not expressed in the phenotype of the individual. Banzhaf et al. [3] give the following definition for introns. Introns are parts of the genotype that emerge as a result of the evolution of individuals with a variable length representation and have no influence on the survival of the individual.

Experiments have shown that the growth of introns are a result of the use of genetic operations such as mutation, crossover and selection [3, 12, 13]. Although introns have no influence on the survival of the individual they do have an influence on the survival rate of their offspring [3]. If a crossover operation is performed, each of two highly fit individuals is segmented into a subtree and remaining part of the individual. The segments are swapped and merged to produce an offspring. It is hoped that this operation will produce another highly fit individual. Most of the time, however, an individual is torn to pieces and the resulting program is of no use at all. If an intron exists in the genotype of an individual then it can happen that the crossover operation happens inside this segment of the genotype. In this case the offspring has the same fitness as its parent. Therefore, the probability that an offspring has the same fitness as its parent increases with the number and the size of its introns. The individuals bloat because this is the only way to increase their effective fitness. The effective fitness of an individual is the fitness that takes the negative influence of genetic operators into account. Evolution stagnates if the fraction of introns grows very much. The fraction of introns of an individual could be used as an indicator which shows that a local optimum has been reached.

There is a difference between the type of introns that can occur with tree-based genetic programming and linear genetic programming. See Banzhaf et al. [3] for an introduction to genetic programming with linear genotypes. For the following discussion we are again assuming that we are doing symbolic regression as a sample problem. The program that is evolved using linear genetic programming can contain additional code to the part which codes for the symbolic expression. This code is called dead or useless code, or if jumps are used, it is called unreachable code [1]. An example of an individual that contains dead code is shown in Figure 3. On the right side of the figure the individual is shown as it could appear in tree-based genetic programming. The segment shown in light gray does not add to the behavior of the individual. On the left an individual from linear genetic programming is shown. This individual shows the same behavior as the individual on the right. In addition to the fragment shown in light gray the individual also contains code fragments which represent useless code. These fragments are drawn in dark gray. They change variables which are no longer used or which are calculated again before they are used.

Because of introns a number of individuals correspond to the same solution in genetic programming. Therefore, depending on the set of primitive functions and the set of terminal symbols, it could happen that quite a large number of solutions occur in the search space. In fact, any solution is represented infinitely often in the search space provided that no limit is set on the tree depth and no limit is set on the number of nodes. This reduces the difficulty of the problem considerably. Again we are using the problem of doing symbolic regression. As primitive functions we used addition (`+`), subtraction (`-`), multiplication (`*`) and protected division (`/`). Protected division returns 1 if the absolute value of the divisor is less than 10^{-10}. As terminal symbols we used the variable `X` and the constant `1`. We made a brute force search and counted the number of individuals that represent

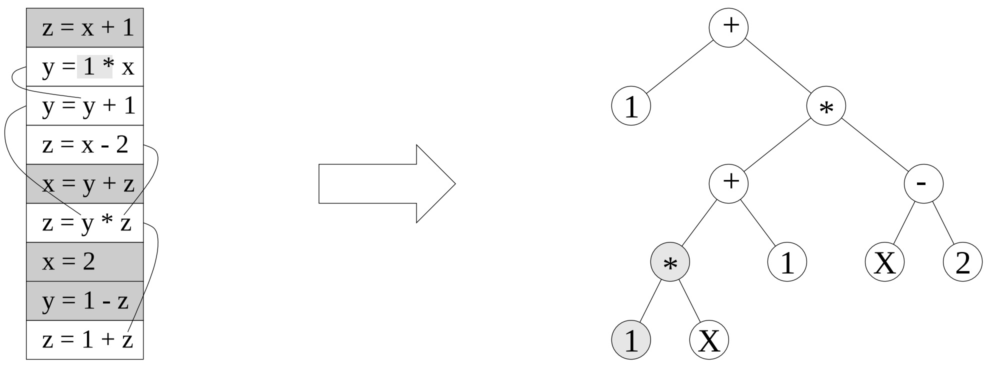

Figure 3: The types of introns in tree-based genetic programming differ from those of linear genetic programming. It is again assumed that one is doing a symbolic regression. The tree on the right is an example of an individual from tree-based genetic programming. The tree also contains a part that does not add to the behavior of the individual (shown in light gray). The tree on the left is an example of the same individual as it could appear in genetic programming with a linear genotype. Both individuals show the same phenotypical behavior. For the individual shown on the left it is assumed that x, y, and z are registers which can be used by the program. Input is supplied to the program via the x register and the result of the computation is read from the z register. In addition to the code fragment shown in light gray the parts shown in dark gray also do not add to the behavior of the individual.

Maximum depth	0	1	2	3
$F=2, T=2$, and primitive functions: `+-`	1	1	13	3173
$F=3, T=2$, and primitive functions: `+-*`	1	3	44	27586
$F=3, T=2$, and primitive functions: `+-/`	1	2	40	28648
$F=4, T=2$, and primitive functions: `+-*/`	1	4	108	185709

Table 3: Number of functions with $f(x) \approx x$.

the function $f(x) \approx x$, that is if $|f(x) - x| < 10^{-10}$ for all $x \in \{\pi, e, 1, 2, 3, 7, 12\}$ we say that $f(x) \approx x$. The number of trees grows very rapidly with the depth of the trees, therefore we could only perform the search for small depths. The number of trees that represent the identity function are shown in Table 3 as a function of depth for different subsets of the four primitive functions. Figure 4 shows the fraction of functions that code for the identity function. Let $I(d)$ be the number of individuals that code for the identity function. Assuming that a solution occurs only once in the search space of depth d_s the same behavior can at least be found $I(d - d_s)$ times in a search space with a maximum depth d with $d \geq d_s$. This follows from the fact that we can simply replace the terminal X in the individual coding for the identity function with the required solution.

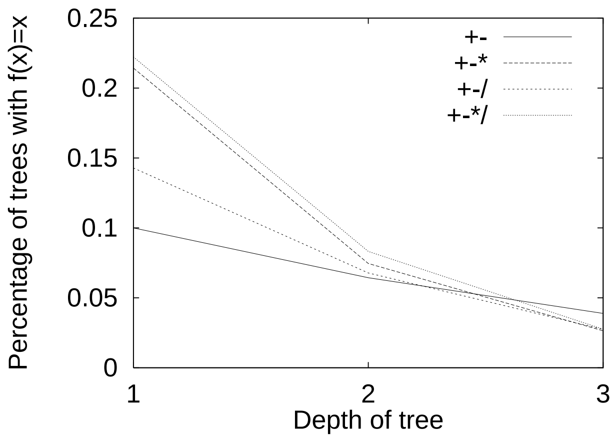

Figure 4: Fraction of number of functions that have a maximum depth of d that code for $f(x) \approx x$.

4 Analogies between the search space of genetic programming and sequence space

Again we can draw an analogy to natural evolution. Sequence space is spanned by DNA-, RNA- and protein sequences. Kauffman [8] and Schuster [14] analyzed sequence space. Kauffman stated that a limited number of enzymes can cover sequence space completely. Therefore, the coding has to be highly redundant. Sequences that code for all common shapes are located in the vicinity of any randomly selected sequence [14]. There are few shapes that are common and many shapes are rare. In addition, different sequences that code for a specific shape are randomly distributed in sequence space. Long paths of neutral mutations lead to sequences which code for identical shapes. The number of different shapes and chemical reactions is much less than the number of possible sequences. From this fact it follows, as Kauffman states [8], that the evolution of life is not as unlikely as the number of possible sequences suggests.

5 Conclusion

Just like in nature the coding of individuals in genetic programming is usually highly redundant. The above analysis (Table 3 and Figure 4) has shown that a large number of individuals exist which code for the same solution. Therefore, finding a solution using genetic programming is not as unlikely as the size of the search space suggests.

The comparison of the search space of genetic programming to nature's search space could provide directions for future research. It could be beneficial to use a representation such that any random behavior can be reached within a limited distance from any given behavior. In addition, individuals which code for the same behavior should be distributed randomly in the search space and long paths of neutral mutations should lead to individuals which code for the same behavior.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[2] W. Banzhaf. Genotype-phenotype-mapping and neutral variation – a case study in genetic programming. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature – PPSN III. Proceedings of the International Conference on Evolutionary Computation*, pages 322–332, Berlin, 1994. Springer-Verlag.

[3] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming - An Introduction: On The Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers, San Francisco, California, 1998.

[4] T. H. Cormen and C. E. Leiserson und R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[6] D. E. Goldberg and U.-M. O'Reilly. Where does the good stuff go, and why? how contextual semantics influences program structure in simple genetic programming. In W. B., R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Genetic Programming: Proceedings of the First European Workshop, EuroGP'98, Paris, France, April 14-15*, pages 16–36, Berlin, 1998. Springer-Verlag.

[7] J. H. Holland. *Adaptation in natural and artifical systems: an introductory analysis with applications to biology, control, and artificial intelligence*. The MIT Press, Cambridge, Massachusetts, 1992.

[8] S. A. Kauffman. *The Origins of Order. Self-Organization and Selection in Evolution*. Oxford University Press, Oxford, 1993.

[9] R. E. Keller and W. Banzhaf. Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996, Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*, pages 116–122, Cambridge, Massachusetts, 1996. The MIT Press.

[10] J. R. Koza. *Genetic Programming, On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachusetts, 1992.

[11] J. R. Koza. *Genetic Programming II, Automatic Discovery of Reusable Programs*. The MIT Press, Cambridge, Massachusetts, 1994.

[12] W. B. Langdon and R. Poli. Fitness causes bloat: Mutation. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Genetic Programming: Proceedings of the First European Workshop, EuroGP'98, Paris, France, April 14-15*, pages 37–48, Berlin, 1998. Springer-Verlag.

[13] W. B. Langdon and R. Poli. Genetic programming bloat with dynamic fitness. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Genetic Programming: Proceedings of the First European Workshop, EuroGP'98, Paris, France, April 14-15*, pages 97–112, Berlin, 1998. Springer-Verlag.

[14] P. Schuster. Extended molecular evolutionary biology: Artificial life bridging the gap between chemistry and biology. In C. G. Langton, editor, *Artificial Life: An Overview*, pages 39–60, Cambridge, Massachusetts, 1995. The MIT Press.