Detecting and Pruning Introns for Faster Decision Tree Evolution

Jeroen Eggermont and Joost N. Kok and Walter A. Kosters

Leiden Institute of Advanced Computer Science
Universiteit Leiden
P.O. Box 9512, 2300 RA Leiden
The Netherlands
{jeggermo,joost,kosters}@liacs.nl

Abstract. We show how the understandability and speed of genetic programming classification algorithms can be improved, without affecting the classification accuracy. By analyzing the decision trees evolved we can remove the unessential parts, called *introns*, from the discovered decision trees. Since the resulting trees contain only useful information they are smaller and easier to understand. Moreover, by using these pruned decision trees in a fitness cache we can significantly reduce the number of unnecessary fitness calculations.

1 Introduction

Algorithms for data classification are generally assessed on how well they can classify one or more data sets. However, good classification performance alone is not always enough. Almost equally important can be the understandability of the results and the time it takes to learn to classify a data set. In this paper we focus on ways to improve the speed of our genetic programming (GP) algorithms as well as the understandability of the evolved decision trees.

Evolutionary algorithms generally spend a lot of time on calculating the fitness of the individuals. However, if one looks at the individuals during an evolutionary run, one often finds that some of the genotypes occur more than once. The main reason for this is that generally the diversity in a population decreases over time when a static fitness function is used. We can use these genotypical reoccurrences to speed-up the fitness calculations by storing each evaluated individual and its fitness in a *fitness cache*. If an individual's genotype is already stored in the cache then its fitness can simply be retrieved from the cache instead of the time consuming calculation which would otherwise be needed.

One of the problems of variable length evolutionary algorithms, such as tree-based genetic programming, is that the genotypes of the individuals tend to increase in size until they reach the maximum allowed size. This phenomenon is, in genetic programming, commonly referred to as *bloat* [2] and is caused by GP *introns* [2, 12, 11]. The term *introns* was first introduced in the field of genetic programming, and evolutionary computation in general, by Angeline [1] who compared the emergence of extraneous code in variable length GP structures to biological introns. In biology the term introns is used for parts of the DNA, sometimes referred to as junk DNA, which do not have any apparent function as they are not transcribed to RNA. In genetic programming, the term *introns* is used to indicate parts of an evolved solution which do not influence the result produced by, and thus fitness of, the solution (other than increasing its size).

The occurrence of *introns* in evolutionary algorithms has both positive and negative effects. A positive effect is that they might protect against the destructive effects of crossover operators [2]. However, earlier studies [12, 11] show that more than 40% of the code in a population can be caused by *introns*.

GP classifiers that use variable length (decision) tree structures are subject to *bloat* and they will thus also contain *introns*. In the case of our decision tree representations we can distinguish between two types of *introns*, *intron subtrees*: subtrees which are never traversed, and *intron nodes*: nodes which do not influence the classification outcome of the decision tree.

The negative effects of *introns* are two-fold. The decision trees found by GP algorithms can contain *introns* which makes them less understandable than semantically equivalent trees without *introns*. The second problem of *introns* is that while they do not influence the classification outcome of a decision tree, their evaluation does take time. More important, *introns* reduce the effectiveness of our fitness cache since it does not recognize semantically equivalent but syntactically different trees.

In this paper we present techniques to detect and prune the *introns* in decision trees. The pruned intron-free decision trees will generally be smaller, and thus easier to understand. By using the pruned intron-free trees for our fitness cache we improve its effectiveness.

The overview of the rest of the paper is as follows. In Section 2 we introduce the representations used by our GP algorithms. Then in Section 3 we show how *introns* can be detected and pruned. In Section 4 we describe the experiments followed by the results. Finally, in Section 5 we present the conclusions.

2 Full Atomic Representations

We use *full atomic* representations. A full atomic representation has atoms in the internal and leaf nodes. Each atom has a predicate of the form (*attribute operator value*(*s*)), where *operator* is a function returning a Boolean value (e.g., $<$ or $=$). In the leaf nodes we have *class assignment* atoms of the form ($class := C$), where C is a category selected from the domain of the attribute to be predicted. A small example tree is shown in Figure 1. A full atomic tree classifies an instance I by traversing the tree from root to leaf node. In each non-leaf node an atom is evaluated. If the result is true the right branch is traversed, else the left branch is taken. This process is repeated until a leaf node containing a *class assignment* node is reached, resulting in the classification of the instance.

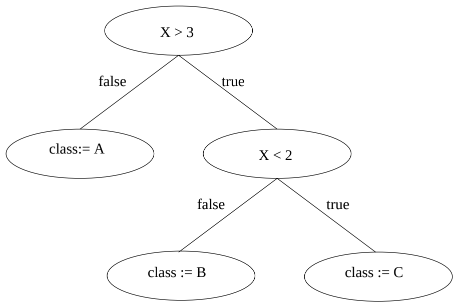

Fig. 1. A full atomic tree.

Table 1. Example data set.

X	**Y**	**class**
1	a	A
2	b	A
3	a	B
4	b	B
5	a	A
6	b	A

We next define the precise decision tree representation by specifying what atoms are to be used. In this paper we will use two representations:

– *Simple* GP [5], uses atoms based on each possible *attribute-value* combination found in the data set. For non-numerical attributes we use the equality operator (=) and for numerical attributes we use the less-than operator (<). The idea of this approach is to give the GP algorithm the most flexibility and let it decide on the best *attribute-value* combination at a given point in a tree.
– The second representation, *cluster* GP [4], uses unsupervised K-means clustering [13] to partition the domain of numerical valued attributes into a fixed number of clusters. The advantage of the clustering representation is that it leads to smaller search space sizes and better classification performance.

Example Consider Table 1. In the case of the *simple* representation we get the following atoms:

– Since attribute **X** has six possible values and is numerical valued we get the following atoms: $(\mathbf{X} < 1)$, $(\mathbf{X} < 2)$, $(\mathbf{X} < 3)$, $(\mathbf{X} < 4)$, $(\mathbf{X} < 5)$ and $(\mathbf{X} < 6)$.
– Attribute **Y** is non-numerical resulting in two atoms: $(\mathbf{Y} = a)$ and $(\mathbf{Y} = b)$.
– The two classes result in two terminal nodes: $(class := A)$ and $(class := B)$.

K-means clustering with $k = 3$ results in three clusters for attribute **X**. Thus, in case of the *cluster* GP representation the following atoms are used for attribute **X**: $(\mathbf{X} \in [1, 2])$, $(\mathbf{X} \in [3, 4])$ and $(\mathbf{X} \in [5, 6])$.

3 Intron Detection and Pruning

In [7] Johnson proposes to replace the standard fitness measures by static analysis methods [10]. Through these techniques an individual's behaviour can be evaluated across the entire input space instead of a limited number of test cases. However, for data classification replacing the fitness measure with static analysis techniques does not seem feasible due to the high dimensional nature of the search space. Instead of replacing the fitness function, Keijzer [8] showed how static analysis can be used as a pre-processor for fitness evaluations. By using interval arithmetic to calculate the bounds of symbolic regression trees, functions containing undefined values are either deleted or assigned the worst possible performance value.

We will use a combination of static analysis techniques and pruning to detect and remove *introns* from decision trees in order to address the problems they cause. Each individual T that is to be evaluated is scanned for *introns*. These *introns* are marked and a pruned copy T' of T is made in which the *introns* are removed. If T', which is semantically the same as T, matches a tree T^c found in the *fitness cache* the individual T is assigned the fitness of T^c. If T' does not match any tree in the cache it is evaluated using the fitness function and stored in the cache. In this case T is assigned the fitness of T'.

Since several syntactically different decision trees can have the same *pruned* form this should optimize the effectiveness of the cache. By using the original trees for the evolutionary process (e.g., crossover and mutation) as well as the tree size fitness measure, there is no influence on the classification performance of our algorithms.

3.1 Intron Subtrees

Intron subtrees are subtrees which can and will never be traversed because of the outcome of nodes higher up in the tree. An example of an *intron subtree* is shown in Figure 2(a).

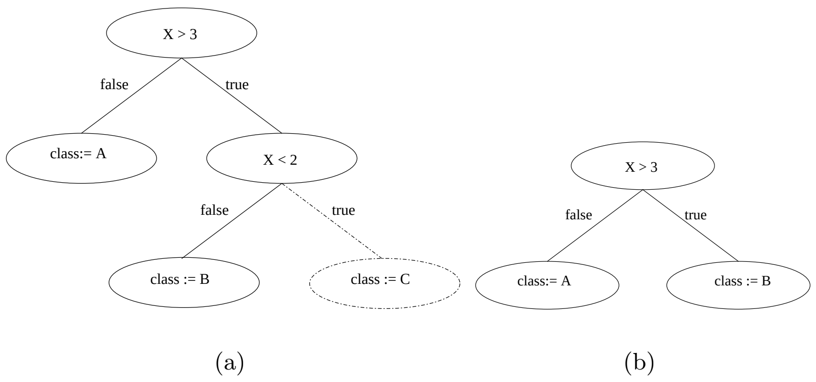

Fig. 2. Two syntactically different trees which are semantically the same. The left tree contains an *intron subtree* indicated by the dotted lines.

In order to detect *intron subtrees* we recursively propagate the possible domains of the attributes through the decision trees in a top-down manner. Given a data set T we can determine for each attribute X_i the domain $D(X_i)$ of possible values. By propagating the domain of each attribute X_i recursively through the trees we can identify situations in which the domain of an attribute becomes empty ($\emptyset$), indicating the presence of a *intron subtree.*

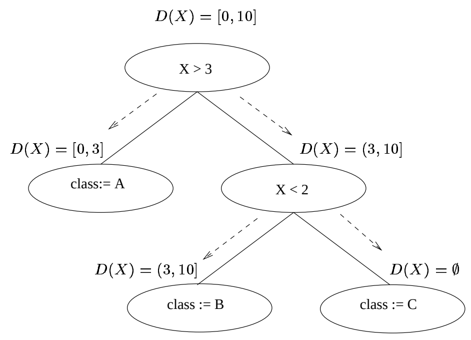

Fig. 3. A full atomic decision tree containing an intron subtree with the domain of attribute X displayed at each point in the tree.

Observe the decision tree T in Figure 3. Let X be a continuous valued attribute in the range $[0, 10]$. Before evaluation of the root node ($X > 3$) the domain of X is $[0, 10]$. Just as the node splits a data set into two parts, the domain of possible values of X is split into two. In the left subtree the domain of X is limited to $[0, 3]$ and in the right subtree the domain of possible values for X is $(3, 10]$. After the evaluation of node ($X < 2$) the possible domain of X for the left tree is the same as before the atom was evaluated. However, the possible domain of X for the right subtree is reduced to $\emptyset$, and this subtree is therefore marked as an *intron subtree* (see Figure 2 (a)).

After *intron subtrees* have been detected and marked the tree can be pruned. During the pruning phase the marked *intron subtrees* are removed and their originating root node ($X < 2$ in our example) is replaced by the remaining *valid* subtree. The resulting pruned tree for the example can be seen in Figure 2(b). Note that we assume that all attributes in the data sets are independent. In reality there may be relations between attributes (e.g., attribute X is always larger than attribute Y) in which case some *intron subtrees* are not detected.

3.2 Intron Nodes

Intron nodes are root nodes of subtrees of which each leaf node contains the same *class assignment* atom (e.g., $class := A$). The internal nodes in such a subtree do not influence the classification outcome. While the negative effects of *intron subtrees* are mostly related to size and thus understandability of our decision trees, *intron nodes* have a much more negative influence on the computation times. An example is shown in Figure 4(a).

In order to detect *intron nodes* we recursively propagate the set of possible class outcomes through the tree in a bottom-up manner. A leaf node always returns a set containing a single class. In each internal node the sets of possible classes of its children are joined. If the set of possible class outcomes for an

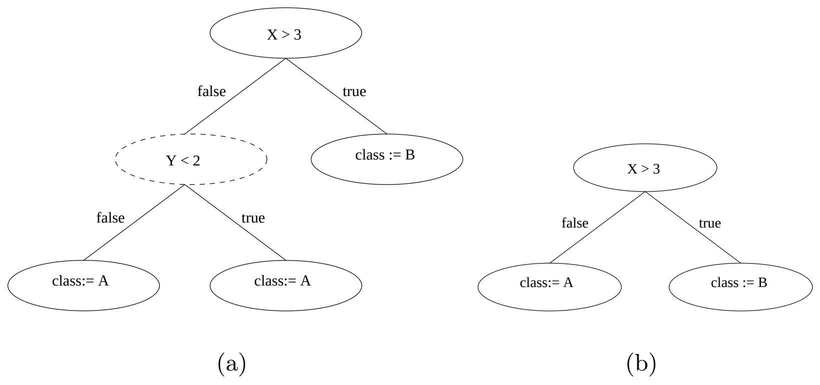

Fig. 4. Two syntactically different trees which are semantically the same. The left tree contains an *intron node* indicated by the dotted lines.

internal node contains only a single class the node is marked as an *intron node*. Once all the *intron nodes* have been detected the tree can be pruned. During the pruning phase the tree is traversed in a top-down manner and subtrees with an *intron node* as the root node are replaced by class assignment nodes corresponding to their possible class outcome detected earlier.

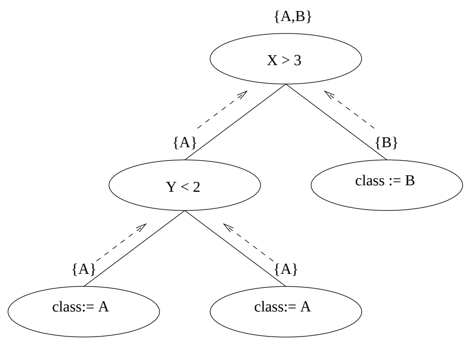

Fig. 5. A full atomic tree with the set of possible target classes for each node.

Consider the tree in Figure 5. The set of possible class outcomes for each leaf node consists of a single class, namely the target class. In the case of node ($Y < 2$), the sets of possible class outcomes of its subtrees are the same and contain only a single value A. Thus, the set of possible class outcomes for ($Y < 2$) also contains only a single value and it is therefore marked as an *intron node* (see Figure 4(a)). In the case of node ($X > 3$) the set of possible class outcomes consists of two classes and this node is therefore not an *intron node*. The resulting tree for the example can be seen in Figure 4(b).

Note that the pruned decision trees in Figures 2(b) and 4(b) are both semantically and syntactically the same although they are derived from syntactically different trees (Figures 2(a) and 4(a)). When *intron node* detection and pruning is used in conjunction with *intron subtree* detection it is important to apply both detection and pruning strategies in the right order. *Intron nodes* should be detected and pruned after *intron subtrees* to assure that all *introns* are found as the pruning of *intron subtrees* can influence the detection of *intron nodes*. Since *intron subtree* detection works top-down and *intron node* detection works bottom-up the two *intron* detection methods can be performed in a single tree traversal.

4 Experiments and Results

In order to determine the effects of *introns* on our GP algorithms we have performed experiments on six data sets from the UCI data set repository [3]. An overview of the data sets as well as the misclassification performance of our GP algorithms and C4.5 (as reported by Freund and Shapire [6]) is given in Table 2. For three data sets (Australian credit, Heart disease and German credit) we applied C4.5 ourselves since no results were reported. Each algorithm is evaluated using n-fold cross-validation and the performance is the average misclassification error over n folds. In n-fold cross-validation the total data set is divided into n parts. Each part is chosen once as the test set while the other $n-1$ parts form the training set. In all our experiments we use $n = 10$.

Table 2. The data sets used in the experiments.

data set				**misclassification rates**		
name	**records**	**attributes**	**classes**	**simple** GP	**cluster** GP	**C4.5**
Australian credit (statlog)	690	14	2	22.0	14.8	15.9
German credit (statlog)	1000	23	2	27.1	28.0	27.2
Pima Indians diabetes	768	8	2	26.3	26.3	28.4
Heart disease (statlog)	270	13	2	25.2	21.3	22.2
Ionosphere	351	34	2	12.4	10.5	8.9
Iris	150	4	3	5.6	2.1	5.9

A single GP implementation was used for both representations. It was programmed using the *Evolving Objects* library (EOlib) which is available from `http://eodev.sourceforge.net`.

In our GP system we use the standard GP mutation and recombination operators for trees. The mutation operator replaces a subtree with a randomly created subtree and the crossover operator exchanges subtrees between two individuals. Both the mutation rate and crossover rate are set to 0.9, optimizing the size of the search space that will be explored. The population was initialized using the ramped half-and-half initialization [2, 9] method to create a combination of full and non-full trees with a maximum tree depth of 6. We used a generational model (comma strategy) with population size of 100, an offspring size of 200 and a maximum of 99 generations, resulting in a maximum of 19.900 individuals. Parents were chosen by using 5-tournament selection. We did not use elitism as the best individual was stored outside the population. Each newly created individual, whether through initialization or recombination, was automatically pruned to a maximum number of 63 nodes. The results are computed over a 100 runs. The cache memory size did not cause any problems.

Whenever the fitness function has to evaluate a decision tree it first checks if the tree is already in the fitness cache. If it is, a *cache hit* occurs and the fitness of the tree is retrieved from the cache. If it is not in the cache the (pruned) decision tree is evaluated by the fitness function and the result is stored in the cache. We will count the number of cache hits to determine the effects of the *intron* detection and pruning strategies on the fitness cache.

Observe the cache hit results for the data sets in Tables 3(a) through (f). When we look at the cache hit percentages of both algorithms it is clear that detecting and pruning *intron subtrees* results in a larger increase in cache hits than detecting and pruning *intron nodes*. As expected the combination of detecting and pruning both *intron subtrees* and *intron nodes* offers the highest number of cache hits. Since the difference between detecting and pruning both types of *introns* on the one hand and detecting and pruning only *intron subtrees* on the other hand is in most cases larger than the difference between no *intron* detection and detecting and pruning *intron nodes* it is clear that first removing the *intron subtrees* allows for a better detection of *intron nodes*. If we look at the average runtimes we see that except for the Ionosphere data, *intron* detection and pruning results in a speed increase of up to 50%. The computation time increase on the Ionosphere data set is probably caused by the relatively large number of attributes combined with a small number of data records.

To determine the effect of intron detection and pruning we have measured the average size of all trees after pruning during an evolutionary run. We also measured the size of trees found to be the best, based on the classification performance on the training set. When we look at the results we see that there is virtually no effect of the representation on the sizes of the evolved decision trees. If we look at the average size of the pruned trees we see that pruning both *intron subtrees* and *nodes* reduces the size of the trees by 30 to 50%. If we look at the average size of the best found trees we see that pruning both the types of *introns* reduces the size by approximately 20%.

5 Conclusions

When we consider the effect of detecting and pruning *introns* on the size of the decision trees, it is clear that the pruned trees will be easier to understand although in some cases they are still quite large. The detection and pruning of *introns* also enables us to identify syntactically different trees which are semantically the same. By comparing and storing pruned decision trees in a fitness cache, rather than the original unpruned decision trees, we can greatly improve the effectiveness of the cache. The increase in cache hits means that less individuals have to be evaluated resulting in reduced computation times.

If we compare the computation times of the algorithms we note that the combination of both *intron* detection and pruning methods has a noticeable effect on the computation times. The decrease in computation time is different

Table 3. The average, minimum and maximum number of cache hits and average runtimes (relative to no *intron* detection and pruning).

Australian Credit

algorithm	intron	% cache hits			run-
	detection	avg	min	max	time
simple GP	none	16.9	10.3	31.7	1.0
simple GP	nodes	21.5	12.9	41.0	0.9
simple GP	subtrees	39.0	24.7	53.8	1.0
simple GP	both	46.8	28.9	64.7	0.9
cluster GP	none	18.9	10.9	40.1	1.0
cluster GP	nodes	23.5	13.5	49.1	0.9
cluster GP	subtrees	39.9	25.0	60.9	0.9
cluster GP	both	47.0	29.4	68.4	0.8

(*a*)

German Credit

algorithm	intron	# cache hits			run-
	detection	avg	min	max	time
simple GP	none	15.4	9.5	24.7	1.0
simple GP	nodes	19.0	11.8	29.2	0.9
simple GP	subtrees	44.1	25.4	56.9	0.7
simple GP	both	51.1	30.1	65.2	0.6
cluster GP	none	17.3	10.2	28.9	1.0
cluster GP	nodes	20.6	12.5	31.5	0.9
cluster GP	subtrees	30.3	15.5	48.4	0.9
cluster GP	both	35.6	19.7	57.6	0.8

(*b*)

Pima Indians Diabetes

algorithm	intron	# cache hits			run-
	detection	avg	min	max	time
simple GP	none	15.2	9.0	24.7	1.0
simple GP	nodes	19.1	11.7	29.4	0.9
simple GP	subtrees	38.6	23.3	57.9	1.1
simple GP	both	45.7	27.6	65.9	1.0
cluster GP	none	16.5	10.6	29.0	1.0
cluster GP	nodes	19.7	13.0	33.2	0.9
cluster GP	subtrees	42.1	23.3	61.0	1.0
cluster GP	both	48.5	27.2	68.8	0.9

(*c*)

Heart Disease

algorithm	intron	# cache hits			run-
	detection	avg	min	max	time
simple GP	none	13.7	8.9	22.9	1.0
simple GP	nodes	17.3	11.2	27.8	0.9
simple GP	subtrees	35.3	22.4	49.7	1.0
simple GP	both	42.1	26.4	58.5	0.9
cluster GP	none	13.3	10.0	23.8	1.0
cluster GP	nodes	16.1	12.3	26.7	1.0
cluster GP	subtrees	31.0	18.1	48.4	0.9
cluster GP	both	36.7	22.2	55.8	0.9

(*d*)

Ionosphere

algorithm	intron	# cache hits			run-
	detection	avg	min	max	time
simple GP	none	12.4	8.8	21.6	1.0
simple GP	nodes	16.1	11.4	28.7	1.0
simple GP	subtrees	22.7	12.0	45.9	1.5
simple GP	both	28.1	15.2	54.6	1.4
cluster GP	none	14.4	9.2	22.7	1.0
cluster GP	nodes	18.1	11.5	27.8	0.9
cluster GP	subtrees	27.7	14.0	51.0	1.4
cluster GP	both	33.6	17.0	60.0	1.3

(*e*)

Iris

algorithm	intron	# cache hits			run-
	detection	avg	min	max	time
simple GP	none	18.4	9.1	30.8	1.0
simple GP	nodes	21.5	10.4	36.3	1.0
simple GP	subtrees	53.0	33.2	65.0	0.7
simple GP	both	58.0	36.1	70.4	0.7
cluster GP	none	29.2	13.2	38.0	1.0
cluster GP	nodes	32.6	14.8	41.9	1.0
cluster GP	subtrees	71.0	57.7	77.5	0.6
cluster GP	both	74.5	61.9	80.0	0.5

(*f*)

from what we would expect when looking at the increase in cache hits and the reduction in tree sizes achieved by the detection and pruning strategies. This difference can be explained by the time spent by our algorithms on detecting and pruning the *introns*, initialization, mutation, recombination and selection as well as choices made during the implementation of the algorithms. Nevertheless, on the smallest data set (Iris) detecting and pruning both *intron subtrees* and *intron nodes* reduces the computation times of the *cluster* GP and *simple* GP algorithms by approximately 50% and 30%, respectively. On the German Credit data set, which contains the most records, the detection and pruning of both types of *introns* reduces the average computation time by over 35% for the *simple* GP and over 20% for the *cluster* GP algorithm.

For future research we are planning to test the *intron* detection and pruning technique on larger data sets. As the number of records in a data set increases the advantage of *intron* removal should become more apparent. We therefore think that the *intron* detection and pruning method will allow GP classifiers to scale better with larger data sets.

References

1. P.J. Angeline. Genetic programming and emergent intelligence. In K.E. Kinnear, Jr., editor, *Advances in Genetic Programming*, pages 75–98. MIT Press, 1994.
2. W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone. *Genetic Programming: An Introduction*. Morgan Kaufmann, 1998.
3. C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998. http://www.ics.uci.edu/∼mlearn/MLRepository.html.
4. J. Eggermont. Evolving fuzzy decision trees with genetic programming and clustering. In J.A. Foster et al., editor, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 71–82. Springer-Verlag, 2002.
5. J. Eggermont, J.N. Kok, and W.A. Kosters. Genetic programming for data classification: Partitioning the search space. In *Proceedings of the 2004 Symposium on applied computing (ACM SAC'04)*, pages 1001–1005. ACM, 2004.
6. Y. Freund and R.E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the 13th International Conference on Machine Learning*, pages 148–146. Morgan Kaufmann, 1996.
7. C. Johnson. Deriving genetic programming fitness properties by static analysis. In J.A. Foster et al., editor, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 298–307. Springer-Verlag, 2002.
8. M. Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In C. Ryan et al., editor, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 71–83. Springer-Verlag, 2003.
9. J.R. Koza. *Genetic Programming*. MIT Press, 1992.
10. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
11. T. Soule and J. A. Foster. Code size and depth flows in genetic programming. In J.R. Koza et al., editor, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 313–320. Morgan Kaufmann, 1997.
12. T. Soule, J. A. Foster, and J. Dickinson. Code growth in genetic programming. In J.R. Koza et al., editor, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223. MIT Press, 1996.
13. I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 2000.