Structure-Based Constants in Genetic Programming

Christian B. Veenhuis

Berlin University of Technology, Berlin, Germany
veenhuis@googlemail.com

Abstract. Evolving constants in Genetic Programming is still an open issue. As real values they cannot be integrated in GP trees in a direct manner, because the nodes represent discrete symbols. Present solutions are the concept of ephemeral random constants or hybrid approaches, which have additional computational costs. Furthermore, one has to change the GP algorithm for them. This paper proposes a concept, which does not change the GP algorithm or its components. Instead, it introduces structure-based constants realized as functions, which can be simply added to each function set while keeping the original GP approach. These constant functions derive their constant values from the tree structures of their child-trees (subtrees). That is, a constant is represented by a tree structure being this way under the influence of the typical genetic operators like subtree crossover or mutation. These structure-based constants were applied to symbolic regression problems. They outperformed the standard approach of ephemeral random constants. Their results together with their better properties make the structure-based constant concept a possible candidate for the replacement of the ephemeral random constants.

Keywords: Genetic Programming, Constant, Structure-based Constant, Constant Function, Subtree Relationship, Full Tree Normalization, Generic Benchmark, Polynomial Benchmark, Sum-of-Gaussians Benchmark.

1 Introduction

If one applies Genetic Programming to problem domains, whose solutions are represented by mathematical expressions, as for instance in symbolic regression, one possibly needs to enable the usage of constants. Since constants are typically real values and the tree nodes represent discrete symbols like SIN or ADD, the question arises how to integrate these real values into the trees, a problem which is still considered as an open issue [8].

In order to overcome GP's weakness in discovering numerical constants, Koza has introduced the concept of Ephemeral Random Constants (ERC) [6,7]. These constants are represented by the terminal symbol $\Re$, whereby each terminal node additionally keeps a real value with the numerical value of this constant. For this concept it is necessary to change two aspects of GP: Firstly, the initialization component of GP needs to be changed. Each time, the terminal symbol of constants shall be added to an initial tree, a random number uniformly drawn from a given interval $[c_{min}, c_{max}]$ is added to its node, too. It is hoped that GP is able to produce all needed constants later on based on these pre-created ones by combining them with the (mathematical) operations contained in the set of functions. Secondly, the data structure of a node needs to be extended to be able to held this additional real value. Thus, this concept can not be used with every GP implementation without changing it. Beside these disadvantages, the ERC concept has the advantage that the number of created constants is not restricted, since a constant here is just a terminal symbol being randomly selected by the initializer.

Although the ephemeral random constants can still be considered as standard, other methods have been developed as well to improve the numeric quality of constants. In [1,2] the authors introduced an operator they called "Numeric Mutation", which extends the ERC concept. This operator is applied to a subset of the population at each generation and replaces all constants in existence by new ones uniformly drawn from a given constant interval, which is defined around the current constant value to be changed. The bounds of this interval are controlled by a "temperature factor" adopted from Kirkpatrick's simulated annealing method [5]. This "temperature factor" is set in dependence to the objective value of the best individual. If the run converges to good solutions, the interval becomes narrower, which leads to smaller changes in constant values. Otherwise, the interval becomes wider to allow bigger changes. Since this concept uses ephemeral random constants, it inherits their disadvantages: the need to change the node structure as well as to change the initialization component. Furthermore, a GP can not be run with standard configurations, because the new mutation operator needs to be considered.

Another method that changes constant values is introduced in [9]. There, the authors borrow the idea from [4] who use a table (array) of constants, whereby the constants in the trees use indices into this table. This concept was extended by using a sorted table so a smaller index also means a smaller constant value. Before creating the initial population, this sorted table is filled up with random values. Then, while creating the initial trees, each time the terminal symbol of constants shall be added, an index into the sorted table is added to its node, too. Based on these indices, the authors introduced two mutation operators. The first one is called "Uniform" and replaces an index of a constant by a new index, which is uniformly drawn from the whole range of indices. The second operator is called "Creep" and chooses the new index directly above or below the old one. This way also the change of the constant value is relatively small. In order to use this concept, one needs to change the node structure to be able to store an index as well as to change the initialization component. Furthermore, the sorted table mechanism needs to be integrated into the GP approach. This sorted table is of fixed size. Thus, the number of constants needs to be pre-specified. Finally, GP can not be run with standard configurations, because the new mutation operators need to be considered, too.

The above methods work by adding operators to GP or changing its initialization component. Another category of methods hybridize GP with other optimizers, whereby the other optimizers are used to adjust the constant values. These approaches are not GP methods as such, but hybrid concepts. In [10] GP is combined with a local optimizer based on gradient descent. There, at each generation all constant values of all trees are optimized by three iterations of a gradient descent approach. Since this concept uses ephemeral random constants, the node structure as well as the initialization component of a GP need to be changed, if one wants to use this concept. Furthermore, the local optimizer needs to be embedded as well.

In [3] another GP hybrid is presented, which uses a genetic algorithm as local optimizer for constants. Quite similar to the work of [4], the constants use indices into a table of constants. The difference is that each individual has its own local constant table. All these constant tables of the population are optimized by the employed genetic algorithm. If one wants to use this concept, a lot of GP aspects need to be changed: the node structure (for the index), the initialization component and the individual, which needs to keep its table of constants. Since this table is of fixed size, the number of constants per individual also needs to be pre-specified. Last but not least, the genetic algorithm needs to be integrated as well.

All in all, one can state that the methods developed so far have one "problem" in common: they all change either the GP algorithm as such by introducing new mechanisms like tables or by hybridizing GP with other algorithms, or they change components of GP like the initialization procedure to randomly produce constants or indices. If one wants to use constants, one can be certain to have to change his GP library for this. Furthermore, a hybrid could be considered to be not a GP anymore, but merely GP-like! A hybrid solves the constant problem just for the hybrid itself and not for GP in general. Thus, these approaches are not suitable to solve the constant problem in a sufficient manner. They are all "pragmatic" solutions. This raises the question, whether it is possible to replace the current concepts by another one, which *does not need to change GP or one of its components.* Further desirable properties would be that *as few parameters as possible* need to be specified and that the concept is *problem-independent.*

These are strong wishes, but can they be fulfilled? A main reason for the nature of current solutions to the constant problem is that intuitively everyone associates a constant with a value. And a value needs to be hold in a variable, it needs to be created and adjusted. This automatically leads to changes of data structures and of the procedures that deal with them. In order to get rid of this effect, one has to change the overall nature of solutions to the constant problem. But, if one wants to change the nature of current constant concepts, one needs to change the nature of constants, too.

Since the business of GP and its operators is to create and rearrange (sub)trees, a constant should also be a subtree to be under the influence of the typical genetic operators like subtree crossover or mutation. Therefore, in the proposed concept, a constant is not anymore a terminal, but a function. As a function it has subtrees as operands. The value of this constant function is derived from the tree structures of its operands. Thereby, the constant functions do not consider the content of the nodes. They are completely based on properties of the tree structures. Thus, they can be used in and added to each GP application being this way problem-independent.

The further paper is organized as follows. Section 2 introduces the proposed concept called *Structure-based Constants*. Two generic benchmark functions, which allow to specify the number of constants, are introduced in section 3. In section 4 the conducted experiments with their results are presented. Finally, in section 5 some conclusions are drawn.

2 Structure-Based Constants

Since the business of GP operators is to create and rearrange trees, a constant should also be represented in some way by a tree. This way, constants would also be under the influence of the subtree crossover and mutation operators. Therefore, in this section, a constant concept called *Structure-based Constants* (SC) is introduced, which replaces constant terminals by constant functions denoted by SC. A constant function has subtrees as operands, like all other functions, too. The value of this constant function is deterministically derived from the tree structures of its children (subtrees). This way, the constants are totally integrated into the GP trees without the need to pre-generate randomized ones or to determine them by additional algorithms in parallel or afterwards. This saves computation time and one can keep the *original GP algorithm*.

Let $\mathbb{T}$ denote the set of all trees

$$\mathbb{T} = \{T_1, \cdots, T_{|\mathbb{T}|}\}$$

and $|T|$ the number of nodes (or cardinality) of the given tree $T \in \mathbb{T}$. Each tree has at least a root node and is not empty.

Furthermore, let $[c_{min}, c_{max}]$ be a pre-specified interval of constant values as it is also used for ephemeral random constants. Then, the structure-based constants (SC), as proposed and used in this paper, can be defined as in the following subsections.

2.1 Subtree Relationships

This category of structure-based constants uses the child-trees of the constant function and sets their tree properties into relationship. A simple property of a tree is the number of nodes it is composed of. These numbers of nodes of the child-trees are combined to deterministically compute constants:

Definition 1 (SC_{quot})**.** *The structure-based constant*

$$SC_{quot} : \mathbb{T} \times \mathbb{T} \to \mathbb{R}$$

is the quotient of the number of nodes of its left and right child-trees T_{left} and T_{right}:

$$SC_{quot}(T_{left}, T_{right}) := \frac{|T_{left}|}{|T_{right}|}$$

Definition 2 (SC_{mmquot})**.** *The structure-based constant*

$$SC_{mmquot} : \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{R}$$

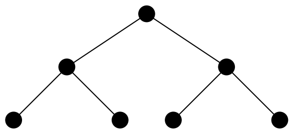
is the minimum-maximum quotient of the number of nodes of both child-trees T_{left} and T_{right} mapped into the constant interval $[c_{min}, c_{max}]$:

$$SC_{mmquot}(T_{left}, T_{right}) := c_{min} + \frac{\min(|T_{left}|, |T_{right}|)}{\max(|T_{left}|, |T_{right}|)}(c_{max} - c_{min})$$

2.2 Full Tree Normalization

This category of structure-based constants uses a child-tree of the constant function and normalizes one of its tree properties with respect to a full tree structure. A full tree is a structurally complete tree (and some authors call it also complete tree, perfect tree or perfect A-ary tree):

Definition 3 (Full Tree). *A full tree denoted by $T_{L,A}$ is a rooted A-ary tree with L levels, i.e., a tree with exactly one root node and every internal node has exactly A children. All nodes of the last level are in existence and no internal node is missing as depicted in the following for a full tree $T_{3,2}$ with 3 levels and an arity of 2:*



A simple property of a tree is the number of nodes it is composed of. Thus, the number of nodes of a child-tree is normalized by the number of nodes of a full tree to deterministically compute constants. The total number of nodes of a full tree $T_{L,A}$ can be computed by Eq. (1).

$$N_{nodes}(L, A) = \sum_{i=0}^{L-1} A^i \qquad (L \geq 1, A \geq 1) \tag{1}$$

Definition 4 (SC_{full})**.** *The structure-based constant*

$$SC_{full} : \mathbb{T} \rightarrow \mathbb{R}$$

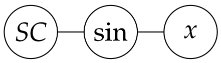
normalizes a child-tree T by its corresponding full tree based on the number of nodes and maps it into the constant interval $[c_{min}, c_{max}]$:

$$SC_{full}(T) := c_{min} + \frac{|T|}{N_{nodes}(level(T), arity(T))}(c_{max} - c_{min})$$

The function $level(T)$ delivers the maximum depth of subtree T and $arity(T)$ the maximum arity occurring in T.

The full tree represents the maximum structure of a tree. Thus, a subtree T can be either this full tree, which leads to a quotient of 1, or a structurally smaller version of the full tree, which produces a quotient < 1. In this sense, a tree is divided by a maximum tree performing this way a normalization.

The corresponding full tree used by SC_{full} is based on the arity and depth of the child-tree. Thus, a child-tree $[\sin[x]]$ as depicted in



is a full tree, because the maximum arity occurring is 1 and the depth is 2, which leads to $N_{nodes}(L, A) = N_{nodes}(2, 1) = 2 = |\ [\sin[x]]\ |$. But typically mathematical expressions also allow binary operations so the question arises whether it would not be better to use the global maximum arity from the whole function set. In this case the former example would not be anymore a full tree, because the corresponding full tree would have $N_{nodes}(L, A) = N_{nodes}(2, 2) = 3$ nodes. A child-tree that is a full tree represents c_{max}. Maybe using the global maximum arity produces more different constants, because the number of possible full trees is reduced. In order to examine this, a variation to SC_{full} is defined in the following, which uses the global maximum arity out of the function set denoted by A_{max}.

Definition 5 **(SC_{full-g}).** *The structure-based constant*

$$SC_{full-g} : \mathbb{T} \to \mathbb{R}$$

normalizes a child-tree T by a corresponding full tree based on the number of nodes and maps it into the constant interval $[c_{min}, c_{max}]$:

$$SC_{full-g}(T) := c_{min} + \frac{|T|}{N_{nodes}(level(T), A_{max})}(c_{max} - c_{min})$$

The function $level(T)$ delivers the maximum depth of subtree T and A_{max} is the global maximum arity out of the function set.

3 Benchmark Functions

In order to evaluate the capabilities of the introduced structure-based constants concept, two generic benchmark functions (subsections 3.1 and 3.2) were specifically created for the work at hand and allow to specify the number of constants c_{num} as a sort of parameter of model complexity.

3.1 Polynomial Benchmark

This generic benchmark function allows to specify the number of constants c_{num} as a benchmark parameter. It is designed for the standard set of functions (see Table 1). Since the standard set is particularly able to build polynomials, this benchmark function is just a reduced polynomial, whose degree is used as the number of constants:

$$P_{c_{num}}(x) = \sum_{i=1}^{c_{num}} (c_{min} + \frac{i}{c_{num}}(c_{max} - c_{min})) \cdot x^i \qquad (2)$$

The interval $[c_{min}, c_{max}]$ is the allowed range of constant values. Note that the x^0 term is omitted. This way, the degree of the polynomial is the number of wished constants c_{num}. The coefficients, which are the searched constants, increase from the lowest to the highest exponent.

3.2 Sum-of-Gaussians Benchmark

Like the previous one, also this benchmark function allows to specify the number of constants c_{num} as a benchmark parameter. But this one is designed for the extended set of functions (see Table 1). The idea is to use a Gaussian for each constant, whereby the constant shifts the Gaussian's position and works this way as an offset. They are shifted in a way that all Gaussians are distributed over the given range $[X_{min}, X_{max}]$. All shifted Gaussians are summed up to build the Sum-of-Gaussians Benchmark:

$$G_{c_{num}}(x) = \sum_{i=1}^{c_{num}} e^{-\left(x + X_{min} + (i-0.5)\frac{X_{max}-X_{min}}{c_{num}} \right)^2} \qquad (3)$$

4 Experiments

The aim of the conducted experiments was to find out, whether the structure-based constants perform at least comparably to Koza's ERC concept. That they have better properties, because they do not change anything in the GP algorithm, is not enough. Thus, all four SCs $(SC_{quot}, SC_{mmquot}, SC_{full}, SC_{full-g})$ as well as Koza's ERC concept were applied to all benchmark functions as introduced in section 3. Each of the two generic benchmarks was used with $c_{num} = 1, \cdots, 10$ constants leading to 10 different benchmark functions per generic benchmark. The used $[c_{min}, c_{max}]$ interval for constants was set to $[-5, +5]$ for all structure-based constants as well as for ERC. According to the used function sets (see below), the global maximum arity A_{max} for SC_{full-g} was set to 2.

For each benchmark function and constant concept, 50 independent runs were performed. For all benchmark functions $P_{c_{num}}$ and $G_{c_{num}}$, samples of 100 points were used with all $x^{(k)}$ being uniformly distributed in $[X_{min}, X_{max}] = [-10, 10]$ and $y^{(k)} = P_{c_{num}}(x^{(k)})$ or $y^{(k)} = G_{c_{num}}(x^{(k)})$, respectively. For all experiments, a standard GP was used with the settings as given in Table 1.

Table 1. The settings of the used GP approach

Objective:	Symbolic regression with constants evaluated by sum of deviations over all points $F(T) = \sum_k \lvert \mathcal{I}(T, x^{(k)}) - y^{(k)} \rvert$ with tree $T \in \mathbb{T}$ and interpreter $\mathcal{I}$
Standard Function Set:	NEG, ADD, SUB, MUL, DIV and X as terminal plus $\Re$ or the appropriate $SC_{...}$ constant
Extended Function Set:	NEG, ADD, SUB, MUL, DIV, POW, ABS, SQRT, SIN, COS, TAN, EXP, LN, LOG and X as terminal plus $\Re$ or the appropriate $SC_{...}$ constant
Initialization:	Ramped half-and-half, min = 2, max = 6
Crossover:	Standard crossover, probability = 0.9
Mutation:	Subtree mutation, probability = 0.1
Selection:	Tournament selection, 3 competitors
Fitness:	Raw (standard) fitness
Replacement:	Generational replacement scheme
Parameters:	Population size = 500, generations = 40, no elitists

4.1 Polynomial Benchmark

For this benchmark, the experiments were conducted for the first ten numbers of constants ($c_{num} \in \{1, \cdots, 10\}$). In Table 2 (page 134), the results are given as numerical values. The winners are printed in boldface. Inspecting this table reveals that the Polynomial benchmarks were best solved by all SC approaches. But among the SCs, there is no clear winner. Three of them win in four and five of the ten cases and one only in two cases (SC_{full-g}). Among the best three SCs, one (SC_{quot}) performed only well for the lower numbers of constants, whereas the other two (SC_{mmquot},SC_{full}) won cases over the whole range.

4.2 Sum-of-Gaussians Benchmark

Also for this benchmark, the experiments were conducted for the first ten numbers of constants ($c_{num} \in \{1, \cdots, 10\}$). According to Table 3 (page 135), the Sum-of-Gaussians benchmarks were best solved by SC_{mmquot}. It wins in 6 of the ten cases, outperforming this way all other methods (with respect to the number of won cases). The other three SCs win only in zero, one and two cases. Although very tight, the ERC method wins one case and outperformed SC_{quot}. The reached fitness values of all methods are close for all benchmarks.

4.3 Summary

In Table 4 (page 136) the final results over all benchmark functions are given in terms of the numbers of cases in which a constant concept performs as best. The last row gives the total numbers of won cases. From there it can be seen that the winner over all benchmarks is clearly SC_{mmquot}. The second bests are SC_{full}

Table 2. The obtained results for the **POLYNOMIAL** benchmark averaged over 50 independent runs. The columns 'Avg. Fitness' are the best fitness values reached on average and 'sd' are the appropriate standard deviations.

	ERC	SC_{quot}	SC_{mmquot}	SC_{full}	SC_{full-g}
c_{num}	Avg. Fitness	Avg. Fitness	Avg. Fitness	Avg. Fitness	Avg. Fitness
1	5.45922 (*sd*: 15.6871)	**0** (*sd*: 0)	**0** (*sd*: 0)	**0** (*sd*: 0)	**0** (*sd*: 0)
2	50.945 (*sd*: 92.6602)	0.000106878 (*sd*: 0.000748143)	**0** (*sd*: 0)	**0** (*sd*: 0)	**0** (*sd*: 0)
3	959.358 (*sd*: 1178.68)	**411.188** (*sd*: 365.07)	542.772 (*sd*: 399.006)	531.82 (*sd*: 467.849)	450.682 (*sd*: 307.877)
4	11401.3 (*sd*: 9706.61)	**2207.93** (*sd*: 4602.83)	4128.93 (*sd*: 3238.93)	3492.59 (*sd*: 3076.52)	4441.4 (*sd*: 3156.95)
5	102873 (*sd*: 113909)	**19747.8** (*sd*: 65657.6)	30664.9 (*sd*: 91940.9)	64953.9 (*sd*: 336065)	47246.9 (*sd*: 116539)
6	$1.12797e+006$ (*sd*: 1.14452e+006)	569027 (*sd*: 895749)	296845 (*sd*: 562651)	**284145** (*sd*: 314191)	436775 (*sd*: 640290)
7	$2.63987e+007$ (*sd*: 3.32783e+007)	$1.05336e+007$ (*sd*: 2.0743e+007)	**8.3599e + 006** (*sd*: 1.38128e+007)	$8.57059e+006$ (*sd*: 9.45898e+006)	$1.11821e+007$ (*sd*: 2.49654e+007)
8	$1.14364e+008$ (*sd*: 1.77641e+008)	$5.03722e+007$ (*sd*: 5.97271e+007)	$3.87227e+007$ (*sd*: 6.56279e+007)	**2.35991e + 007** (*sd*: 2.33994e+007)	$2.61194e+007$ (*sd*: 3.1674e+007)
9	$3.12286e+009$ (*sd*: 3.11849e+009)	$1.08845e+009$ (*sd*: 1.94807e+009)	**7.09581e + 008** (*sd*: 8.15087e+008)	$1.48731e+009$ (*sd*: 2.2571e+009)	$1.24917e+009$ (*sd*: 2.32232e+009)
10	$1.63328e+010$ (*sd*: 1.74503e+010)	$7.36193e+009$ (*sd*: 9.44926e+009)	$3.70365e+009$ (*sd*: 4.755e+009)	**3.0351e + 009** (*sd*: 4.06186e+009)	$4.18589e+009$ (*sd*: 9.61241e+009)

Table 3. The obtained results for the **SUM-OF-GAUSSIANS** benchmark averaged over 50 independent runs. The columns 'Avg. Fitness' are the best fitness values reached on average and 'sd' are the appropriate standard deviations.

	ERC	SC_{quot}	SC_{mmquot}	SC_{full}	SC_{full-g}
c_{num}	Avg. Fitness	Avg. Fitness	Avg. Fitness	Avg. Fitness	Avg. Fitness
1	7.92484 (*sd*: 2.45881)	7.06104 (*sd*: 3.36676)	**6.86777** (*sd*: 3.5924)	7.81503 (*sd*: 2.66586)	7.6386 (*sd*: 2.8158)
2	**17.5234** (*sd*: 0.0690329)	17.5466 (*sd*: 0.002927)	17.5473 (*sd*: 1.06581*e*−014)	17.5458 (*sd*: 0.00873155)	17.5473 (*sd*: 1.06581*e*−014)
3	22.7068 (*sd*: 4.75156)	24.0421 (*sd*: 3.74773)	**21.9589** (*sd*: 5.61472)	24.3213 (*sd*: 4.34347)	24.4268 (*sd*: 3.72864)
4	29.8323 (*sd*: 0.344268)	29.6072 (*sd*: 1.40193)	**27.1038** (*sd*: 4.65437)	27.5833 (*sd*: 3.84845)	27.7314 (*sd*: 2.86172)
5	30.2103 (*sd*: 0.707123)	30.04 (*sd*: 1.22604)	24.1434 (*sd*: 6.01809)	25.2328 (*sd*: 6.67785)	**23.6527** (*sd*: 6.88129)
6	20.7481 (*sd*: 5.78303)	19.5825 (*sd*: 6.47734)	16.0718 (*sd*: 2.42268)	**15.1753** (*sd*: 3.70877)	15.7694 (*sd*: 3.41595)
7	23.6415 (*sd*: 1.02372)	19.2354 (*sd*: 5.1414)	**17.5072** (*sd*: 4.74239)	18.3589 (*sd*: 4.79823)	17.8244 (*sd*: 5.47548)
8	19.4204 (*sd*: 0.208013)	14.377 (*sd*: 4.89492)	**11.6531** (*sd*: 5.85684)	13.708 (*sd*: 6.13544)	12.6103 (*sd*: 6.13404)
9	14.7881 (*sd*: 0.177806)	12.4597 (*sd*: 3.8964)	**8.36133** (*sd*: 5.28631)	10.0999 (*sd*: 5.31754)	9.5832 (*sd*: 4.95699)
10	10.8935 (*sd*: 0.0634859)	10.5945 (*sd*: 0.853805)	10.4401 (*sd*: 1.31411)	10.2638 (*sd*: 1.32116)	**9.99404** (*sd*: 1.56859)

and SC_{full-g}. The worst of the structure-based constants is SC_{quot}, which only performs well for the Polynomial benchmark $P_{c_{num}}$ with fewer constants. Note that SC_{quot} is the only SC, which neither specifies a constant interval nor represents negative constants. All other methods (including ERC) allow for negative constants by using appropriate c_{min} and c_{max} bounds. It seems that forcing negative constants by an interval is easier for GP than to build negative constants by applying the NEG function to a positive constant.

Considered over all 20 benchmark functions, the ERC concept only won 1 case. In all other 19 cases it was outperformed by most of the structure-based constants.

Table 4. The numbers of won cases over all benchmarks

Benchmark	ERC	SC_{quot}	SC_{mmquot}	SC_{full}	SC_{full-g}
# Polynomial	0	4	4	5	2
# Sum-of-Gaussians	1	0	6	1	2
$\sum$	1	4	10	6	4

5 Conclusions

In this paper a new constant concept called Structure-based Constants (SC) was introduced. It represents a constant by a tree so it is under the influence of the subtree crossover and mutation operators. Opposed to the common procedure, such a structure-based constant is not a terminal, but a function. The value of this constant function is derived from the tree structures of its child-trees. This new concept has a number of advantages compared to ERC and other solutions to the constant problem:

- Since a constant is a tree itself, it totally integrates into the GP trees. Thus, it is evolved with subtree crossover and mutation and **no additional optimizer is needed** anymore. This saves computation time.
- An SC only needs to be added to the function set the same way as all other functions, too.
- Neither the GP algorithm, nor one of its components need to be changed or extended. Thus, **one can keep the original GP** and use standard configurations.
- **Only few parameters** need to be specified. In fact, it is the interval of constant values as already used by the ERC approach. (For SC_{full-g} also A_{max} needs to be set. But this parameter cannot be freely adjusted by the user – it must be set to the global maximum arity of the used function set. Thus, it can not be considered to be a real parameter in the same sense as the interval bounds.)
- The SCs do not consider the content of tree nodes, because they are based on properties of tree structures. Thus, they are **problem-independent** and can be used in each GP application.

- Since the original GP is not changed, the SCs can be used with each GP implementation already in existence.

In 19 of 20 cases the SC concept outperformed ERC. With this result the demand that it must be at least comparable to ERC is fulfilled. It seems that the structure-based constants could be a suitable approach to replace Koza's ERC concept.

References

1. Evett, M., Fernandez, T.: Numeric Mutation Improves the Discovery of Numeric Constants in Genetic Programming. In: Proc. 3rd Annual Conference on Genetic Programming, pp. 66–71. Morgan Kaufmann (1998)
2. Fernandez, T., Evett, M.: Numeric Mutation as an Improvement to Symbolic Regression in Genetic Programming. In: Porto, V.W., Waagen, D. (eds.) EP 1998. LNCS, vol. 1447, pp. 251–260. Springer, Heidelberg (1998)
3. Howard, L.M., D'Angelo, D.J.: The GA-P: A Genetic Algorithm and Genetic Programming Hybrid. IEEE Intelligent Systems 10(3), 11–15 (1995), doi:10.1109/64.393137
4. Keith, M.J., Martin, M.C.: Genetic Programming in C++: Implementation Issues. In: Kinnear Jr., K.E. (ed.) Advances in Genetic Programming (1994)
5. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220, 671–680 (1983)
6. Koza, J.R.: Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems, Stanford University, Computer Science Department. Technical Report STAN-CS-90-1314 (June 1990)
7. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
8. O'Neill, M., Vanneschi, L., Gustafson, S., Banzhaf, W.: Open issues in genetic programming. Genetic Programming and Evolvable Machines 11(3-4), 339–363 (2010), doi:10.1007/s10710-010-9113-2
9. Ryan, C., Keijzer, M.: An Analysis of Diversity of Constants of Genetic Programming. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E.P.K., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 404–413. Springer, Heidelberg (2003)
10. Topchy, A., Punch, W.F.: Faster Genetic Programming based on Local Gradient Search of Numeric Leaf Values. In: Proc. of the Genetic and Evolutionary Computation Conference (GECCO 2001), pp. 155–162. Morgan Kaufmann (2001)