Genetic Programming with Memory for Approximate Data Reconstruction*

Lukas Sekanina and Tadeas Juza

Abstract This chapter addresses the computation-memorization trade-offs in the context of genetic programming (GP). We introduce genetic programming with memory (GPM) in which GP evolves not only the expression but also the content of a small local memory to better approximate the original data set. In particular, we evolved expression-memory pairs that can serve as weight generators and thus approximate the weights associated with convolutional layers of some convolutional neural networks (CNNs). This is potentially interesting for the efficient implementations of hardware accelerators of CNNs in which memory access is significantly more energy-demanding than arithmetic operations. In our approach, most of the weights are approximated using an evolved expression; only some fraction of them must be read from memory. For example, if memory contains 10% of the original weights, the weight generator evolved for a convolutional layer can approximate the original weights such that the CNN utilizing the generated weights shows less than a 1% drop in the classification accuracy on the MNIST data set. The memory requirements are reduced 3.1× or 12.6× for 8-bit or 32-bit weights, respectively. Additional experiments conducted for more complex CNNs and challenging image classification benchmarks show various impacts of weights' approximation on classification accuracy.

* This is a version created by the authors. The final version is available on the Springer Verlag website: SEKANINA Lukas and JUZA Tadeas. Genetic Programming with Memory for Approximate Data Reconstruction. Genetic Programming Theory and Practice XXI. Singapore: Springer Nature Singapore, 2025, pp. 199-218. ISBN 978-981-9600-76-2. Available from: https://link.springer.com/chapter/10.1007/978-981-96-0077-9_10

Lukas Sekanina
Brno University of Technology, Faculty of Information Technology, Czech Republic, e-mail: sekanina@fit.vutbr.cz

Tadeas Juza
Brno University of Technology, Faculty of Information Technology, Czech Republic, e-mail: xjuzat00@stud.fit.vutbr.cz

1 Introduction

Genetic programming (GP) is typically used to evolve an expression implementing an approximate mapping between two sets. Such an expression consists of input variables, operators, and constants. Since the constant values can be evolved, they can be seen as application-specific memory content needed to implement the mapping effectively. The concept of memory has been introduced into GP in different ways [8, 9, 17, 18] but it has never been dominating in major applications of GP. Considering not only symbolic regression but computing in general, there are many cases in which it is more beneficial to memorize some knowledge rather than to compute it. For example, in hardware dividers or CORDIC algorithm [19], introducing a lookup table enables the development of fast and low-power implementations. This research addresses the **computation-memorization trade-offs** in the context of GP.

In our previous paper [7], we introduced *genetic programming with associative memory* (GPAM) in which GP evolves not only the expression but also the content of a small local memory to better approximate the original data set. Particularly, GPAM evolves an expression $G(x)$ and content of associative memory (content addressable memory) $AM(x)$ with the goal of minimizing the approximation error w.r.t a chosen error metric. The resulting pair (G, AM) then produces the regression output y' such that if x is in AM then $y' = AM(x)$, otherwise $y' = G(x)$. GPAM was evaluated in two case studies.

Firstly, we considered symbolic regression for data sets containing many outliers. The goal is to memorize some of these outliers, as they could sometimes be useful. Unfortunately, they are almost always inexactly reproduced by any evolved expression in the standard GP approach. For this experiment, relevant synthetic data sets were obtained by sampling standard benchmark functions for symbolic regression in which some fraction of data points was replaced by randomly generated values.

Secondly, GPAM evolved an expression-memory pair to approximate a set of weights belonging to one layer of a convolutional neural network (CNN). The motivation was to replace energy-demanding reads of external weight memory (which a CNN hardware accelerator must conduct to perform every inference) with a cheap on-chip weight generator utilizing a small local memory. Promising results were obtained in a simplified setup that has not addressed some important aspects of the method, such as memory cost.

This work further analyzes and extends the GPAM approach in several directions. Since associative memory is sometimes unnecessary, we refer to the updated approach as *genetic programming with memory* (GPM). GPAM and GPM can employ any variant of GP. Like in [7], we will use Cartesian genetic programming (CGP) [12].

In the case of symbolic regression, we extend the number of benchmark problems to nine and evaluate them in a different setup. The original approach did not distinguish training and test data. Hence, we adopt a standard methodology of GP and perform the evolution with training data and the evaluation with test data; how-

ever, the randomly generated values (in the benchmark functions) remain identical in both sets.

In the case of the weight generator application, we propose CGP to work with 8-bit functions (in fixed-point representation) rather than 32-bit (floating-point) functions to allow for smaller hardware implementation of evolved expressions. Furthermore, we propose a new memory addressing scheme based on standard memory, enabling us to avoid expensive content-addressable memory. The newly proposed weight generator replaces the original set of weights of a CNN layer. On selected CNN layers, we analyze to what extent the proposed weight generator can produce such weights whose utilization during inference leads to an acceptable accuracy drop of CNN while memory content transferred from external memory is significantly reduced.

The rest of this chapter is organized as follows. Section 2 introduces the concept of symbolic regression utilizing genetic programming with memory. The idea of an on-chip weight generator as a replacement of weight reading from external memory is presented in Section 3. Both approaches are experimentally evaluated – the results are summarized in Section 4. Finally, conclusions are given in Section 5.

2 Symbolic Regression with GPAM

The concept of *memory* has been connected with GP in numerous ways, including individually named storage elements [8], a linear array of indexed memory elements [17], and associative memories similar to neural networks [18]. GP also automatically evolved simple abstract data structures such as stacks, queues, and lists [9].

In our previous approach, GPAM, we simultaneously evolved an expression and content of associative memory for purposes of symbolic regression. In particular, GP evolves expression G and content of a small associative memory (AM), which stores a subset D_{AM} of data points from the training data set D, where $(x_i, y_i) \in D$, $|D| = n$, $|D_{AM}| = k_{AM}$, and $k_{AM} \ll n$. The resulting value y' is computed using Eq. 1:

$$y'(x) = \begin{cases} AM(x) & \text{if } x \text{ is in } AM \\ G(x) & \text{otherwise} \end{cases} \tag{1}$$

Fig. 1 shows the overall scheme of the method. The Agg block aggregates the input data to form a key, which addresses a given value in AM. It is assumed that the condition "x is in AM" can be evaluated quickly. G is evolved with standard CGP [12] whose chromosome is extended with k_{AM} items referring to stored data points (x_i, y_i). An example of encoding used in CGP is given in Fig. 2.

The fitness function, evaluating a given (G, AM) pair on training data set D is based on minimizing a given error metric:

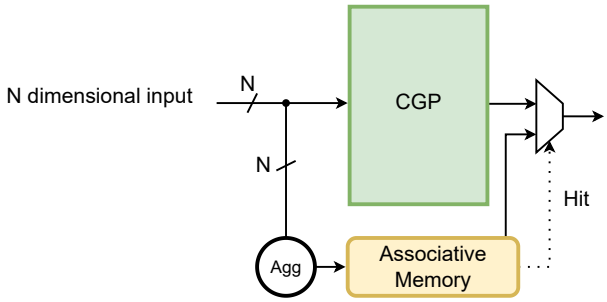

Fig. 1 GPAM: The regression output is obtained from either the expression evolved with CGP or associative memory.

$$Fitness(G,AM,D) = \sum_{i=1}^{n} (y_i - y'(x_i))^2. \quad (2)$$

The standard CGP, with a common mutation operator, modifies c_m integers of the chromosome representing expression G. Another mutation operator was introduced to replace a data point (x_i, y_i) randomly picked in AM with another data point randomly taken from D. The replacement is applied with the probability p_{mutmem}.

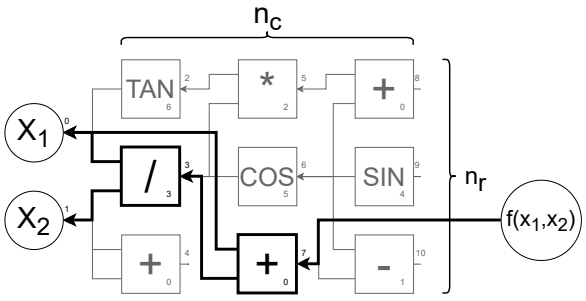

Fig. 2 Example of CGP representation for expression $f = x_1 + x_1/x_2$. CGP is defined using parameters: the number of primary inputs ($n_i = 2$, in our example), primary outputs ($n_o = 1$), columns ($n_c = 3$), and rows ($n_r = 3$). The function set contains 7 functions: + (encoded 0), − (1), ..., Tan (6). In the chromosome, a triplet (source$_1$, source$_2$, function) is devoted to encoding one node. The last integer specifies the output node. GPAM extends the chromosome with k_{AM} data points stored in AM. The resulting chromosome is (0, 0, 6)(0, 1, 3)(1, 1, 0)(2, 3, 2)(3, 3, 5)(0, 3, 0)(5, 6, 0)(6, 6, 4)(6, 7, 1):(7):(3, 0.1)(6, 2.1) for $k_{AM} = 2$.

Contrasted to [7], we will evaluate GPAM on a more comprehensive set of benchmark problems. In the experimental evaluation, we will employ test data points different from those used during the evolution. However, the random modifications of the data set (i.e., the introduced outliers) will remain preserved in the test data set. The reason for distinguishing between training and test data is to get GPAM closer

to the standard GP practice. We will investigate the impact of memory sizing on the regression error when data with many outliers are present. We will deal with neither alternative aggregation functions nor utilization of constants in CGP as these techniques were already discussed in [7].

3 Weight Generator with GPM

The second application is motivated by the current progressive development of energy-efficient hardware accelerators of convolutional neural networks. This chapter provides motivation for our work, introduces the GPAM-based approach [7], and proposes its improvement, the GPM-based approach eliminating the need for associative memory.

3.1 Efficient Processing of CNNs

Hardware accelerators of CNNs are often used for inference in mobile and other embedded devices and are heavily optimized for low energy, area on a chip, and latency. Their implementations employ many processing elements performing elementary operations such as "multiply and accumulate" (MAC) in a systolic parallel architecture. CNN is usually executed layer by layer in the accelerator. Even a single layer can not usually be executed on processing elements at once due to its size. The execution has to be scheduled in many steps. Since a common CNN can have millions of weights, keeping them in local memory is impossible. Hence, they are stored in external DRAM memory. Their subsets are moved to the accelerator's local memory buffer when scheduled by a controller. Many authors reported that memory accesses significantly contribute to the overall energy consumption of the accelerator [2, 16]. For example, while a single 8-bit multiplication only requires 0.2 pJ, reading 32 bits from local memory needs 5 pJ, and reading 32 bits from external DRAM memory consumes 640 pJ (for 45 nm technology [16]).

CNNs are highly error-resilient, so their implementation can be simplified without significantly dropping accuracy. The simplification (approximation) techniques include pruning, utilizing approximate implementations of arithmetic operations, and weight compression [1, 5]. For example, in the *weight sharing* method (also known as the *weight compression*), a group of similar weights is replaced by a single value determined using a clustering algorithm like K-means to reduce the number of weights in memory. NSGA-II was employed to find the best trade-off between the number of shared weights for each layer and the classification accuracy drop. The method introduces an error because not all weights are reconstructed perfectly. The authors reported $5\times$ compression rate for some CNN models with 32 bit weights when evaluated on the ImageNet image classification benchmark [3].

3.2 GPAM-based Method

In our previous work [7], we evolved expression-memory pairs to generate weights for selected layers of CNNs. We aimed at minimizing access to external memory and, thus energy needed for one inference. The method starts with a fully trained CNN-based classifier with the classification accuracy A_c on a test data set. Let S and S_j be the original set of weights and a subset of weights ($S_j \subset S$) belonging to layer j, respectively.

The objective is to evolve a weight generator (consisting of the expression G_j and associative memory AM_j) that can sequentially approximate S_j, one weight in one step. The generated weights and incoming activations are used as the inputs to the systolic array computing output feature maps. An open question is how to generate the input sequence for CGP. In [7], we utilized the following indexes and information relevant for the j-th CNN layer to create the sequence of inputs to CGP: input channel index, output channel index, weight window coordinate w_x, weight window coordinate w_y, and the average of the weights in the kernel (w_{avg}). These items were aggregated by means of concatenation to form a 5×32-bit input vector (a key) to associative memory. Fig. 3 shows that the output produced by (G_j, AM_j) was then added to the average w_{avg} and interpreted as the weight at position (w_x, w_y) in the corresponding channels. The fitness of (G_j, AM_j) was obtained using Eq. 2; i.e., the objective was to minimize the error between the original weights and approximate weights because it is much faster than measuring the classification accuracy (for a CNN utilizing the generated weights) directly on test data.

This approach, in fact, minimizes the number of accesses to external weight memory by reducing the number of weights that have to be stored in memory. Most of the weights can be generated using the evolved expression due to weigh similarity and redundancy of CNNs in general. It is assumed that the expression can be represented by a small configuration bitstream stored in external memory, which is sent into a reconfigurable weight-generating circuit when a given layer is processed by the accelerator. Note that genetic algorithms (GA) were utilized for data compression in the past [4, 14]. For example, GA determined positions of pixels serving as input to a human-created pixel predictor used in lossless image compression in [14].

In this use case, training and test data sets are not distinguished during the evolution of the expression-memory pair because GPAM always works with one particular subset of weights (S_j). The resulting expression-memory pair has to be optimized for this subset and, in fact, performs compression of this subset. Hence, the method will be evaluated as a data compression method in Chapter 4.2.2 and could be helpful when analyzing properties of statistical modeling of data, e.g., in the context of the minimum description length [13].

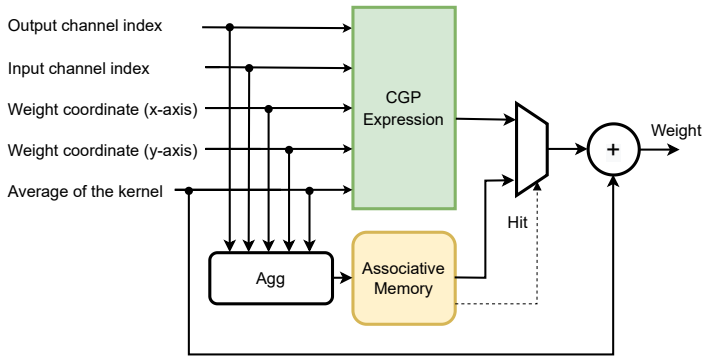

Fig. 3 GPAM-based weight generator for a single layer of a CNN according to [7].

3.3 GPM-based Method

The proposed GPM-based method improves the original GPAM-based approach to weight generation in the following aspects.

(i) *The 32-bit floating-point operations used by CGP are replaced with 8-bit fixed-point operations.* The simplified operations of the CGP's function set could negatively affect the accuracy of symbolic regression. However, they significantly contribute to reducing the complexity of the potential implementation of the method in the CNN accelerator. Note that, for example, an 8-bit fixed-point multiply consumes 15.5× less energy (12.4× less area) than a 32-bit fixed-point multiply, and 18.5× less energy (27.5× less area) than a 32-bit floating-point multiply [16]. Another benefit is the reduced bit width of all the connections on a chip.

(ii) *Associative memory is replaced with standard memory, and a new addressing scheme is introduced.* Associative memory is expensive because it has to store a key (obtained by the aggregation function) and the corresponding weight. Furthermore, its implementation is resource-demanding as the input key has to be compared in parallel with all the stored keys. Recall that associative memory holds a subset of h weights of the entire set of weights belonging to one layer.

The proposed solution exploits the fact that we can sort the subset of weights according to the weight's order and pre-calculate the distance d_j between any two neighboring weights. The distance and weight are stored at the same address in *Memory*. Note that *Memory* is addressed with addresses $0,1,\ldots,h-1$. Fig. 4 shows a new addressing scheme utilizing standard memory, where the data is selected based on the address (i.e., no content addressing is employed). The number *Generator* sequentially generates numbers $p = 0,1,\ldots$. The current distance d_j and weight w_j are prepared in two registers. If $p = d_j$, then the weight stored in the register represents the output of GPM, and the following steps are carried out: the address of *Memory* is increased by 1, a new distance-weight pair is read from *Memory*, stored in the registers, and the number *Generator* is reset. If $p \neq d_j$, the evolved expression generates the output weight. As the number of bits needed to represent a distance

is very small, significant memory savings are obtained compared with associative memory. The proposed solution utilizes neither the weight average as the CGP's input nor the output adder.

(iii) *Memory requirements are determined.* The memory requirements of the GPAM method can be determined as:

$$M_b = h * (k_b + w_b) + G_b, \tag{3}$$

where h, k_b, w_b, and G_b are the number of weights in memory, the number of bits of the key (i.e., the resulting bit width after the aggregation), the number of bits needed for one weight, and the number of bits to encode the G expression, respectively. If the GPM-based approach is considered, the key's bit width is replaced in Equation 3 by the number of bits needed to store the longest distance d between two weights. The compression rate (CR) delivered by the proposed method is

$$CR = \frac{n * w_b}{M_b}, \tag{4}$$

where n is the total number of weights of one layer.

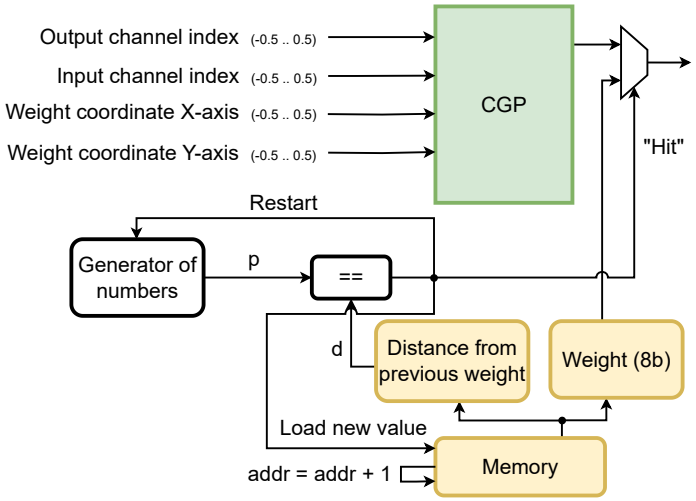

Fig. 4 Proposed weight generator for a single layer of a CNN. Associative memory is replaced with a common memory and a simple address-generating scheme.

4 Results

The experimental results are discussed in two sections – the symbolic regression on synthetic data sets and the weight generation in CNN accelerators.

4.1 Symbolic Regression

The goal is to show that GPAM can approximate data sets containing many random values (outliers) much better than a standard GP. We will first describe how the synthetic benchmark problems are created and then evaluate GPAM on them.

4.1.1 Creating Synthetic Benchmark Problems

We consider nine well-known benchmark functions that are typically utilized to evaluate the GP-based symbolic regression methods [11]. Table 1 defines these benchmark functions, the intervals used for sampling, and the number of sampled points. We created training sets for GPAM by sampling these functions. In each data set, we replaced some percentage of data points (controlled by the parameter τ) with randomly generated values from interval $[-10,10]$ to generate a benchmark problem with "outliers". Then, we created test sets by (1) reusing the randomly generated data points from the training sets and (2) adding new data points by sampling the original functions again. It holds that training and test sets have the same size and contain the same randomly generated data points ("outliers"). It has to be emphasized that GPAM does not know where the randomly generated data points are located.

Table 1 Baseline functions used to generate synthetic benchmark problems for GPAM.

Name	Function	Size	Interval	Step
Koza-1	$f(x) = x^4 + x^3 + x^2 + x$	40	$[-1,1]$	0.05
Nguyen-7	$f(x) = \ln(x+1) + \ln(x^2+1)$	20	$[0,2]$	random
Nguyen-10	$f(x_0,x_1) = 2 * \sin(x_0) * \cos(x_1)$	100	$[-1,1]$	random
Korns-4	$f(x_0,x_1,x_2,x_3,x_4) = -2.3 + 0.13 * \sin(x_2)$	10 000	$[-50,50]$	random
Keijzer-1	$f(x) = 0.3 * x * \sin(2\pi x)$	20	$[-1,1]$	0.1
Keijzer-8	$f(x) = \sqrt{x}$	100	$[0,100]$	1
Vladislavleva-1	$f(x_0,x_1) = \frac{e^{-(x_0-1)^2}}{1.2+(x_1-2.5)^2}$	100	$[0.3,4]$	random
Vladislavleva-2	$f(x) = e^{-x}x^3(\cos(x)\sin(x))(\cos(x)\sin^2(x)-1)$	100	$[0.05,10]$	0.1
Vladislavleva-5	$f(x_0,x_1,x_2) = 30\frac{(x_0-1)(x_2-1)}{x_1^2(x_0-10)}$	300	$x_0,x_2 : [0.05,2]$ $x_1 : [1,2]$	random

4.1.2 Setup

Based on some trial runs and with respect to results of [7], the CGP parameters are defined as summarized in Table 2. CGP employs different function sets for different benchmarks; see Table 3. According to [12], CGP employs the $(1+\lambda)$ search strategy, where λ offspring individuals are generated from one parent. The fitness function is as defined by Eq. 2.

Table 2 Parameters of CGP when used in GPAM.

Parameter	Value
n_i	function dimension
n_o	1
$n_c \times n_r$ (columns $\times$ rows)	20×2
L-back	maximum
Search method	$1+\lambda$, $\lambda = 4$
Generations	5000
c_m	2 integers per chromosome
p_{mutmem}	0.2 (the mutation probability for memory)
Float constant probability	0.05 (0.10 if memory size = 0)
AM constant probability	0.05 (0.00 if memory size = 0)

Table 3 Function sets used by CGP for different benchmarks.

Function name	Function set
Koza, Nguyen	$+$, $-$, $*$, %, *sin*, *cos*, e^x, $log(\|x\|)$
Korns	$+$, $-$, $*$, %, *sin*, *cos*, e^x, $log(\|x\|)$, x^2, x^3, *sqrt*, *tan*, *tanh*
Keijzer	$+$, $*$, $\frac{1}{x}$, $-x$, *sqrt*
Vladislavleva-5	$+$, $-$, $*$, %, x^2
Vladislavleva-1	$+$, $-$, $*$, %, x^2, e^x, e^{-x}
Vladislavleva-2	$+$, $-$, $*$, %, x^2, e^x, e^{-x}, *sin*, *cos*

4.1.3 Example Result

To illustrate the behavior of the proposed approach, let us first analyze an example result evolved by GPAM – the approximation of the Nguyen-7 benchmark (with $\tau = 50\%$, i.e., 10 modified values) when GPAM has no memory (k_{AM} = 0%) and when it can use memory whose size is 60% of original data points. For k_{AM} = 0%, Fig. 5 shows that CGP evolved a very complex expression (Eq. 5) trying to fit as many data points as possible.

$$f(x) = \frac{1}{x}(x(-2x\cos(x) - \ln(\frac{|\frac{(2x\cos(x)-\cos(2x))\ln(2|x\cos(x)|)}{x\cos(x)}|}{2}) + \ln(|\cos(2x)|) -$$

$$\sin(\frac{\ln(|\cos(2x)|)}{x}) + \cos(2x)) - (e^{\cos(x)} * x - \ln(|\cos(2x)|)) \sin(\ln(|\cos(2x)|)x)) \tag{5}$$

However, in the case that associative memory is available, the evolved expression (Eq. 6) is simple:

$$f(x) = ln(|e^x + x^3|) \tag{6}$$

and most "outliers" are correctly stored in memory (see the green points in Fig 5).

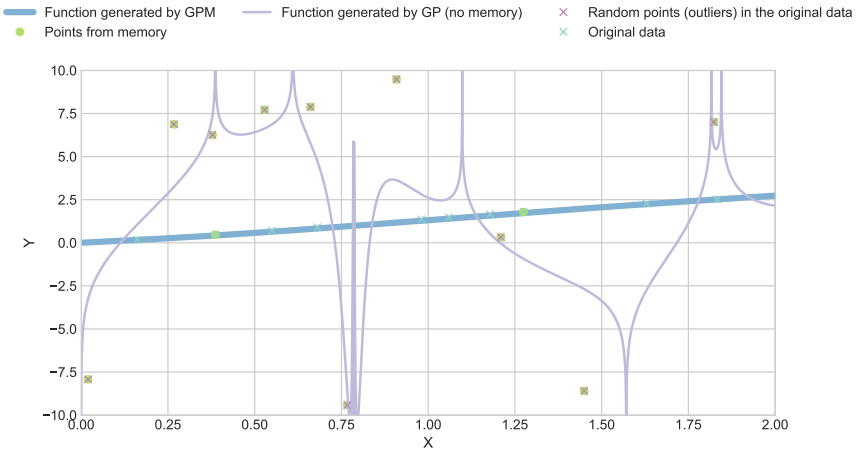

Fig. 5 The Nguyen-7 benchmark problem: original data points (blue ×) and randomly generated points (purple ×). One of the evolved solutions with k_{AM} = 60% is depicted in blue; green points (circles) are taken from *AM*. One of the evolved solutions with k_{AM} = 0% is depicted in purple.

4.1.4 The Role of Memory Sizing

We performed 30 independent runs of GPAM with various settings of k_{AM} and τ for all benchmark data sets. Figure 6 shows in detail on the Keijzer-8 benchmark that the fitness values (i.e., the approximation error on training as well as test data) decrease with lowering the number of randomly generated data points (τ) in the data set and adding more memory (k_{AM}). If $\tau = 0$ and $k_{AM} = 0$, then GPAM becomes a standard CGP, which almost always delivers a solution showing a close to zero error. Note that the fitness values are normalized with respect to the number of data points in all figures showing box plots in this paper.

Fig. 7 shows the same plot for the remaining benchmark problems. Contrasted to [7], all these results are presented for test sets, i.e., the data unseen during the evolution (except the randomly generated data points). The exact obtained values are less important because they can slightly differ when another GP version is utilized.

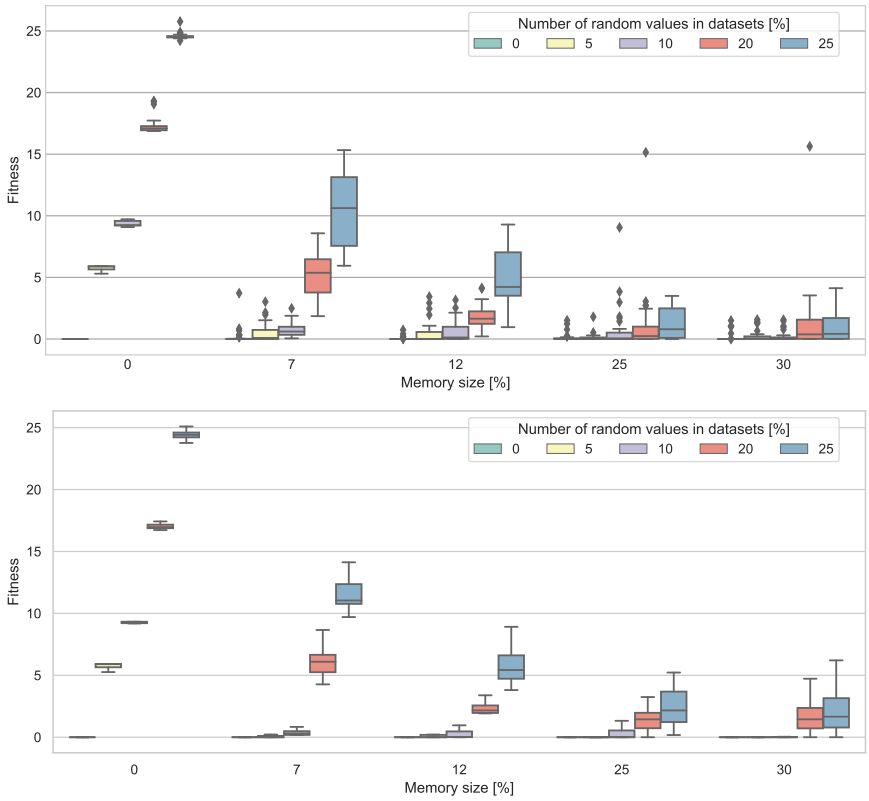

Fig. 6 Fitness values obtained by GPAM on Keijzer-8 benchmark problem for various associative memory sizes (k_{AM}) and numbers of randomly modified data points in data sets (τ). Box plots are constructed using training data (top) and test data (bottom); for the latter case, outliers in the box plots are omitted.

However, the following trend is significant: GPAM almost always provides better solutions for less contaminated data, and when the size of AM is increasing. We can conclude that GPAM can evolve a suitable expression and choose appropriate data points to be stored in associative memory to approximate non-trivial data sets.

The experiments were performed on the Intel Xeon CPUs E5-2630 v4 running at 2.20 GHz. For the most demanding setup, the median execution time is 280 s on a single core.

4.2 Weight Generation

The weight generation method based on GPM is evaluated on several layers of selected CNNs trained as image classifiers. We start with a simple CNN classifying the

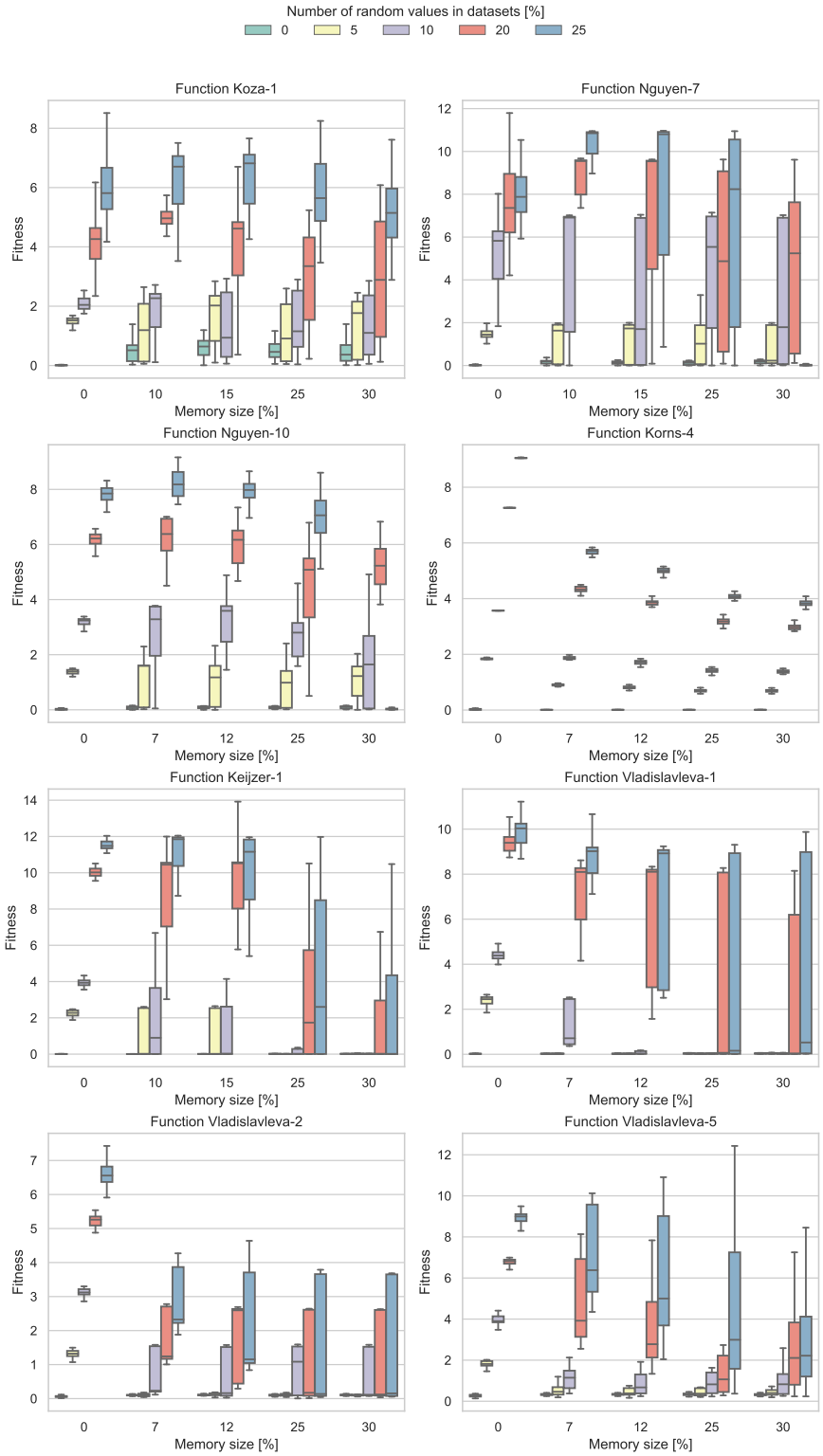

Fig. 7 Fitness values obtained by GPAM on eight benchmark problems for various associative memory sizes (k_{AM}) and numbers of randomly modified data points in data sets (τ).

MNIST data set, where GPAM and GPM are compared. Then, the same approach is adopted for more complex CNNs.

4.2.1 Simple CNN 1

The first CNN, denoted Simple CNN 1, consists of six layers; see Fig. 8. This CNN was trained using PyTorch to classify the MNIST data set (a digit classification task from a 28×28-pixel image [10]) and achieved 97.4% accuracy on test images. When the 8-bit precision is applied, the classification accuracy is 97.3%. We created two data sets; S1 contains 250 weights of the first convolutional layer, and S2 contains 5000 weights of the second convolutional layer; see their values plotted in Fig. 9.

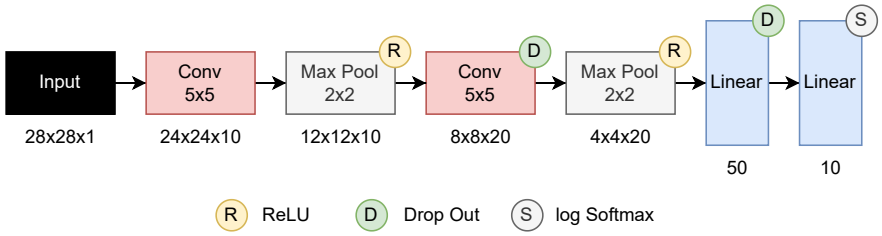

Fig. 8 CNN used as a benchmark problem for GPAM.

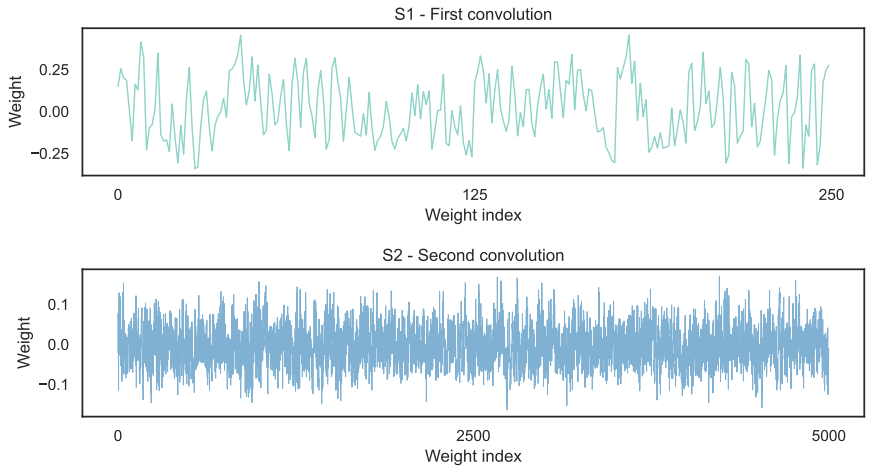

Fig. 9 Data sets created using the weights belonging to the first (S1) and second (S2) convolutional layer of Simple CNN 1.

Table 4 summarizes the setup of CGP parameters for the evolutionary design of weight generators. The setup is based on the results of the previous study [7] and a few additional test runs. GPAM utilizes floating point data representation, operates with the following function set $\Gamma_{32b} = \{+, -, *, \%, sin(x), cos(x), e^x, log(x), x^2, x^3, sqrt(x), tan(x), tanh(x)\}$, and constants can randomly be generated. The eight-bit function set used by GPM is $\Gamma_{8b} = \{x, not(x), rshift1, rshift2, lshift1, lshift2, or, and, xor, +, -, *, const00, constFF\}$. It includes two constants, 00 and FF. A weight is represented on 8 bits so that 1 bit is reserved for the sign, and the remaining (7) bits represent the interval [0; 0,5].

Table 4 Parameters of CGP when evolving the weight generator.

Parameter	Value
n_i	4
n_o	1
$n_c \times n_r$ (columns × rows)	20×10
L-back	maximum
Search method	$1+\lambda$, $\lambda = 4$
Generations	5000
c_m	2 (the integers to mutate in CGP)
p_{mutmem}	0.2 (mutation probability for memory)
Probability of creating a Constant	0.05 (GPAM)

Fig. 10 shows box plots with the fitness values obtained for S1 (left) and S2 (right) from 30 independent runs of GPAM (green) and GPM (yellow) utilizing various sizing of memory. We observe the same trend as in the previous experiments; the classification accuracy is improved if a bigger memory is available. If no memory is considered, the classification accuracy is not competitive at all. For memory containing 10% of the weights (selected by the method automatically), the classification accuracy is close to the baseline values. The best accuracy obtained by GPM (the 8-bit setup) is close to the 32-bit GPAM. The statistical evaluation expressed by the box plots reveals that GPAM consistently provides higher medians of accuracy for S2. Interestingly, GPM leads to higher medians for S1.

Eq. 7 gives one of the best-evolved 8-bit expressions for S1. Surprisingly, it uses only two inputs (x_1 and x_2). For S2, GPM delivered an expression (Eq. 8), which is a constant. It seems that for simple CNNs and classification problems, it is enough to use a simple expression, even a constant expression, and 10%-20% weights to approximate the original weight set with a small accuracy drop.

$$G_{S1}(x_0,x_1,x_2,x_3) = ((\text{FF} \gg 2) * (((((x_1 \text{ xor } x_2) - 00) \text{ or } ((x_1 \text{ xor } x_2) - 00)) \text{ or } (x_1 \text{ xor } x_2)) \text{ and } ((\sim (x_2 \text{ and } (x_1 \text{ xor } x_2))) \text{ or } (00 \ll 2)))) \quad (7)$$

$$G_{S2}(x_0,x_1,x_2,x_3) = \sim (x_2 \text{ xor } x_2) = -0.00390625 \quad (8)$$

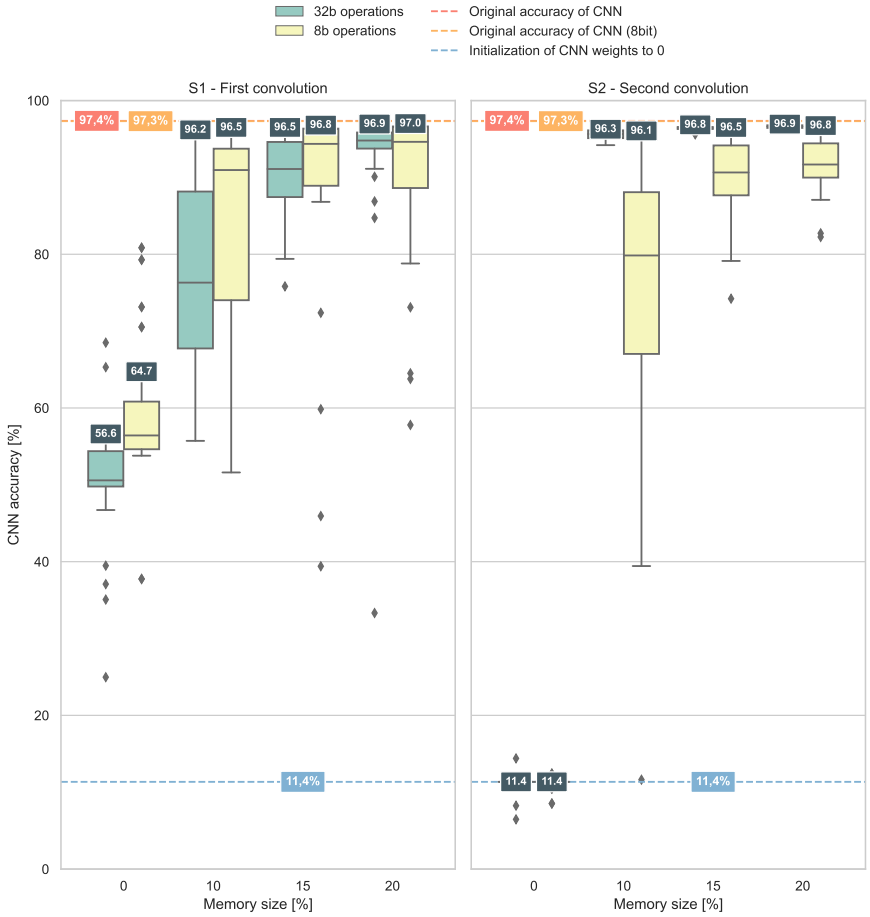

Fig. 10 Accuracy (on the MNIST test data) obtained from 30 runs of GPAM (green) and GPM (yellow) utilizing various sizing of memory on S1 and S2 data sets. The numbers in rectangles represent the best-obtained values under a given setup.

4.2.2 Memory Requirements

A common memory subsystem of a CNN accelerator would require external memory capacity $w_b \times n$ bits for n weights represented on w_b bits. Table 5 summarizes the number of bits obtained with several compression methods to represent the weights of convolutional layers captured by data sets S1 and S2. We included the lossless Huffman code with and without the symbol table, the ZIP compression algorithm, and the theoretical optimum computed using entropy (note that the frequency of occurrence of the weight values is known for S1 and S2). Then, the best-evolved solutions obtained by GPM and GPAM are reported. The solutions evolved by GPM led to the smallest memory requirements; however, an accuracy drop is introduced

as the weights are not reconstructed perfectly. For GPM and GPAM, the number of bits was obtained using Equation 3, which considers the number of weights in local memory, their bit width, the maximum distance between two weights (which is typically encoded on 6 or 7 bits), and the size of CGP chromosome needed to encode the evolved expression (100-300 bits).

Table 5 The number of bits obtained with several methods to represent the weights of convolutional layers captured by data sets S1 and S2.

Method / Data Set	S1 ($n = 250$) [bit]	S2 ($n = 5\,000$) [bit]
Original size (32 bit)	8 000	160 000
Original size (8 bit)	2 000	40 000
Huffman coding	1 737	28 183
Huffman coding (incl. the table)	3 867	29 355
ZIP	3 344	39 464
Theoretical optimum (entropy)	1 732	28 067
GPM / GPAM, $k_{AM} = 10\%$ (acc. drop 0.8%)	636 / 3 698	–
GPM / GPAM, $k_{AM} = 10\%$ (acc. drop 1.0%)	448 / 3 484	–
GPM / GPAM, $k_{AM} = 10\%$ (acc. drop 1.2%)	–	6 765 / 68 044

Regarding the computational requirements, for the most demanding setup (30% weights in memory), the median single-core execution time of GPM is 850 s on the Intel Xeon CPU E5-2630 v4 running at 2.20 GHz.

4.2.3 Other CNNs

We repeated the experiments from Chapter 4.2.1 on selected convolutional layers of some CNNs trained to classify other image data sets. Particularly, we dealt with another simple CNN trained on Fashion MNIST (Simple CNN 2 with an accuracy of 0.91), MobileNetV2 [15] on CIFAR-10, and ResNet-34 [6] on Imagenette. The composed sets of weights are denoted S3 - S10. Table 6 summarizes the basic setup and the obtained trade-offs between the accuracy drop and memory size needed by GPM. One can observe that the weights of some layers can be compressed so that only a small accuracy drop is present; however, if CNN is more complex, the accuracy drop becomes unacceptable.

5 Conclusions

We proposed GPM as an extension of our previous method GPAM, both dealing with the automated design of programs that can utilize small memory to memorize some important information from a training data set. We showed that the method can generalize on common data sets (infected with randomly generated values), which has not been demonstrated in [7].

Table 6 The basic setup of GPM and the obtained trade-offs between the accuracy drop and memory size (k_{AM})

Weight set	CNN	Data set	Layer	#weights	bits	acc. drop [%]	k_{AM} [%]
S1	Simple CNN 1	MNIST	1.	250	8	0.8	10
S1	Simple CNN 1	MNIST	1.	250	8	0.3	20
S2	Simple CNN 1	MNIST	2.	5 000	8	1.2	10
S2	Simple CNN 1	MNIST	2.	5 000	32	0.5	20
S3	Simple CNN 2	FMNIST	1.	400	32	7.4	20
S4	Simple CNN 2	FMNIST	2.	12 800	32	11.3	15
S5	MobileNetV2	CIFAR-10	1.	864	32	6.7	15
S6	MobileNetV2	CIFAR-10	3.	288	32	1.3	15
S7	MobileNetV2	CIFAR-10	5.	864	32	8.9	15
S8	ResNet–34	Imagenette	1.	9 408	32	37.5	30
S9	ResNet–34	Imagenette	2.	36 864	32	3.4	30
S10	ResNet–34	Imagenette	2. & 3.	73 728	32	19.8	10

We evolved expression-memory pairs that can serve as weight generators and thus approximate the weights associated with convolutional layers of selected CNNs. For example, if memory contains 10% of the original weights, the weight generator evolved for a convolutional layer can approximate the original weights such that the CNN utilizing the generated weights shows less than a 1% drop in the classification accuracy on the MNIST data set. The memory requirements are reduced 3.1× or 12.6× for 8-bit or 32-bit weights, respectively. The same approach was adopted for more complex CNNs and challenging image data sets. However, a detailed evaluation of the implementation on a particular hardware accelerator or its simulator has to be performed to obtain all important hardware characteristics, such as latency and energy of inference.

The proposed method can be extended in several directions, for example, by considering the importance of particular weights for the CNN accuracy in the fitness function, employing different GP variants, or creating the key for associative memory in alternative ways. Its utilization in other applications (such as logic circuits for machine learning or image filtering) is our future task.

Acknowledgements This work was supported by the Czech Science Foundation project GA24-10990S.

References

1. Armeniakos, G., Zervakis, G., Soudris, D., Henkel, J.: Hardware approximate techniques for deep neural network accelerators: A survey. ACM Comput. Surv. **55**(4), 83:1–83:36 (2023)
2. Capra, M., Bussolino, B., Marchisio, A., Shafique, M., Masera, G., Martina, M.: An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks. Future Internet **12**(7), 113 (2020)
3. Dupuis, E., Novo, D., O'Connor, I., Bosio, A.: A heuristic exploration of retraining-free weight-sharing for CNN compression. In: 27th Asia and South Pacific Design Automation

Conference, ASP-DAC, pp. 134–139. IEEE (2022)
4. Grasemann, U., Miikkulainen, R.: Effective image compression using evolved wavelets. In: H. Beyer, U. O'Reilly (eds.) Genetic and Evolutionary Computation Conference, pp. 1961–1968. ACM (2005)
5. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding. In: 4th International Conference on Learning Representations, ICLR (2016)
6. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016)
7. Juza, T., Sekanina, L.: GPAM: genetic programming with associative memory. In: Genetic Programming - 26th European Conference, EuroGP 2023, *LNCS*, vol. 13986, pp. 68–83. Springer (2023)
8. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge: Bradford Book, London : MIT Press (1992)
9. Langdon, W.B.: Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming! Springer (1998)
10. LeCun, Y., Cortes, C., Burges, C.: MNIST handwritten digit database. ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist **2** (2010)
11. McDermott, J., White, D., Luke, S., Manzoni, L., Castelli, M., Vanneschi, L., Jaśkowski, W., Krawiec, K., Harper, R., De Jong, K., O'Reilly, U.M.: Genetic programming needs better benchmarks. In: Proc. of the 14th International Conference on Genetic and Evolutionary Computation, pp. 791–798. ACM (2012)
12. Miller, J.F.: Cartesian genetic programming. Springer Berlin Heidelberg (2011)
13. Rissanen, J.: Information and Complexity in Statistical Modeling. Springer New York, NY (2007)
14. Sakanashi, H., Iwata, M., Higuchi, T.: Ehw applied to image data compression. In: T. Higuchi, Y. Liu, X. Yao (eds.) Evolvable Hardware, pp. 19–40. Springer (2006)
15. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.: Mobilenetv2: Inverted residuals and linear bottlenecks. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4510–4520. IEEE Computer Society (2018)
16. Sze, V., Chen, Y., Yang, T., Emer, J.S.: Efficient Processing of Deep Neural Networks. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers (2020)
17. Teller, A.: The evolution of mental models. In: K.E. Kinnear, Jr. (ed.) Advances in Genetic Programming, pp. 199–219. MIT Press (1994)
18. Villegas-Cortéz, J., Olague, G., Avilés-Cruz, C., Sossa, H., Ferreyra, A.: Automatic synthesis of associative memories through genetic programming: A first co-evolutionary approach. In: Applications of Evolutionary Computation, *LNCS*, vol. 6024, pp. 344–351. Springer (2010)
19. Volder, J.E.: The CORDIC trigonometric computing technique. IRE Transactions on Electronic Computers **EC-8**, 330–334 (1959)