Evolving Exact Integer Algorithms with Genetic Programming

Thomas Weise Mingxu Wan Ke Tang Xin Yao

***Abstract*— The synthesis of exact integer algorithms is a hard task for Genetic Programming (GP), as it exhibits epistasis and deceptiveness. Most existing studies in this domain only target few and simple problems or test a small set of different representations. In this paper, we present the (to the best of our knowledge) largest study on this domain to date. We first propose a novel benchmark suite of 20 non-trivial problems with a variety of different features. We then test two approaches to reduce the impact of the negative features: *(a)* a new nested form of Transactional Memory (`TM`) to reduce epistatic effects by allowing instructions in the program code to be permutated with less impact on the program behavior and *(b)* our recently published Frequency Fitness Assignment method (`FFA`) to reduce the chance of premature convergence on deceptive problems. In a full-factorial experiment with six different loop instructions, `TM`, and `FFA`, we find that GP is able to solve all benchmark problems, although not all of them with a high success rate. Several interesting algorithms are discovered. `FFA` has a tremendous positive impact while `TM` turns out not to be useful.**

I. Introduction

Genetic Programming (GP) is a family of Evolutionary Algorithms where the candidate solutions are programs, usually represented as tree data structures [2]. GP is most successful in areas such as Symbolic Regression, where the evolved programs x compute approximate results, i.e., where the difference between the output of a program for a training case and the expected result is a meaningful quality measure.

Synthesizing exact algorithms poses a more difficult problem. When computing the greatest common divisor of two numbers or performing a prime test, the results are either right or wrong. Additionally, correct solutions (programs) need to employ memory and loops or recursion and cannot be reduced to closed formulas. In such problems, the number of solved training cases is often used as objective function instead of an (meaningless) error sum. Two characteristic pathologies emerge:

- *Epistasis.* Adding, removing, or rearranging instructions has a huge (usually negative) impact on the behavior of a program.
- *Deceptiveness.* The objective functions are deceptive, since a program which can solve 80% of the training cases is not necessarily more similar to an exact solution which can only solve 10% [3].

As a result, the evolution of integer algorithms has not seen much progress. Existing studies often consider only few and simple problems, test very few different (sometimes problem-tailored) instruction sets, and do not target epistasis or deceptiveness, as we will show in Section II. Our own prior works [3–5], which build the foundations for this study, have similar shortcomings. Here, we follow a more holistic approach in order to improve on this situation:

- We introduce a benchmark suite for integer algorithm synthesis with 20 problems targeting several different aspects of the domain.
- We test two different approaches in order to improve the success probability of GP: the new Transactional Memory idea (`TM`) to reduce epistasis and the recently published Frequency Fitness Assignment (`FFA`, [3]) to find better solutions for deceptive problems.
- We conduct a large-scale experimental study where we apply the above two methods in conjunction with six different loop instructions from [4].

With this largest experimental study on evolutionary algorithm synthesis, we explore the current limits of GP. We also improve the capabilities of GP in this domain and with our new methods.

We find that GP can solve several hard algorithm synthesis tasks. Problems that require loops and many memory cells tend to be hardest. `FFA` significantly improves the success probability of GP in algorithm synthesis problems. Differently from our anticipation in the future work section of [5], the introduction of Transactional Memory into GP did not turn out to be helpful.

In the following section, we will discuss related works on loop structures, epistasis, and deceptiveness in GP. Then, in Section III, we define the approaches used in the present study, regarding the same three topics. A new suite of twenty benchmark problems is introduced in Section IV and then used in the experimental study described in Section V. We summarize our findings in Section VI.

II. Related Work

A. GP of Algorithms with Loops

Non-trivial algorithm synthesis problems require iterative computations. A program representation must thus provide control flow primitives such as recursion or loops. The latter is the subject of our study.

T. Weise and K. Tang are with the Joint USTC-Birmingham Research Institute in Intelligent Computation and Its Applications (UBRI), School of Computer Science and Technology, University of Science and Technology of China; Hefei, Anhui, China, 230027. email: {tweise|ketang}@ustc.edu.cn; X. Yao is with the UBRI and CERCIA in the School of Computer Science, The University of Birmingham; Birmingham, U.K.email: x.yao@cs.bham.ac.uk

Pioneering work in this area has been done by Koza [2]. Conditional loop instructions were used by his student Finkel [6] to solve the factoring problem. Lai [7] used GP with loops to solve the greatest common divisor problem, which is also present in our benchmark suite. Qi et al. [8] introduced a loop which always performs N iterations, where N is given beforehand by experience. Nesting of loops is forbidden (but allowed in our study). Ciesielski and Li [9] applied two forms of for-loop structures to solve a modified Santa Fe trail problem and a sorting task. The *counter loop* `CL` (see Section III-A) in our experiments works in the same way. Wijesinghe and Ciesielski [10] use `CL`-style loops to produce regular patterns in bit strings of different length. The main differences to our study are the problem types (exact integer algorithms vs. bit pattern generation) and our more general setup (multiple loop instruction, multiple different benchmarks, `FFA`, `TM`). Chen and Zhang [11] used a loop which is similar to our *while loop* to solve the factorial problem, which we discuss in Section IV. In this work, the instruction set used was very small so the achieved success rates may not be generalized.

In summary, we can state that most of the related works either focus on very few loop structures and very few problems, have an instruction set tailored to the problem, or consider static problems which can also be solved by overfitted, non-general programs.

B. Epistasis in Genetic Programming

Epistasis is defined as a form of interaction between different genes in biology. In GP, we can observe mutual dependencies between the instructions of a program, which is one of the main difficulties when synthesizing algorithms.

One way to reduce the epistasis in GP are *soft assignments* [12]. Traditionally, instructions like `x=y` overwrite the value of `x` with the value of `y`. In [12], this strict semantic is replaced with $\mathtt{x}_{t+1} = \gamma \mathtt{y}_t + (1-\gamma)\mathtt{x}_t$ where $\mathtt{x}_{t+1}$ is the value that the variable `x` will have after and $\mathtt{x}_t$ its value before the assignment. In the exact algorithm synthesis domain, where approximate results are useless, this method is not applicable.

In our previous work [5], we proposed the Rule-based Genetic Programming (`RBGP`) method in order to reduce the epistasis in GP. Here, a program is represented as a set of rules, each consisting of a condition and an action part. The action part of a rule may modify a variable and is executed if the condition evaluates to true. The new value of the variable is kept in a temporary storage `W`, which is committed to the memory `R` *after all* rules have been applied. The program is executed in a loop as long as variables change.

In `RBGP`, the order of rules plays almost no role. A random permutation of its rules is unlikely to change a program's behavior, since `R` changes only after all rules have been evaluated. The drawback of this method is that it does not allow for nested control scopes.

C. Deceptiveness in Genetic Programming

Objective functions representing the successfully solved training cases in algorithm synthesis are likely to be deceptive. A program x_1 that can solve 30% of the training cases is not necessarily structurally closer to a correct solution than a program x_2 that can solve only 10% of them [3]. However, the selection schemes common in GP all will with high probability choose x_1 over x_2, regardless of how often such a choice has already been made in the past and whether or not it led to any further improvement. Thus, the optimization process may get trapped in a local optimum.

Two approaches that integrate with EAs to solve problems with such features are Fitness Uniform Selection Scheme (`FUSS`) [13] and Novelty Search (`NS`) [14]. `FUSS` selects individuals uniformly distributed in the *objective space* spanned by the current population and `NS` selects individuals which *behave* differently from previous solutions. `NS` is not applicable in this domain, since there are too many programs with different outputs (behaviors) for a given input. We apply `FFA` in our experiments, which is similar to `FUSS`, with the advantage that it allows GP to temporarily perform strong exploitation [3] (see Section III-C).

III. Approaches in this Study

A. Loop Structures

The loop structures used in our experiments are based on our prior work [4]. The first three loop instructions are similar to what we know from high-level programming languages, where the number of iterations is determined by an explicit expression. The second three iterate until some implicit condition defined over the environment is met, e.g., until no variable changes anymore. Figure 1 illustrates algorithms for computing the `gcd` (greatest common divisor) with each of these instructions.

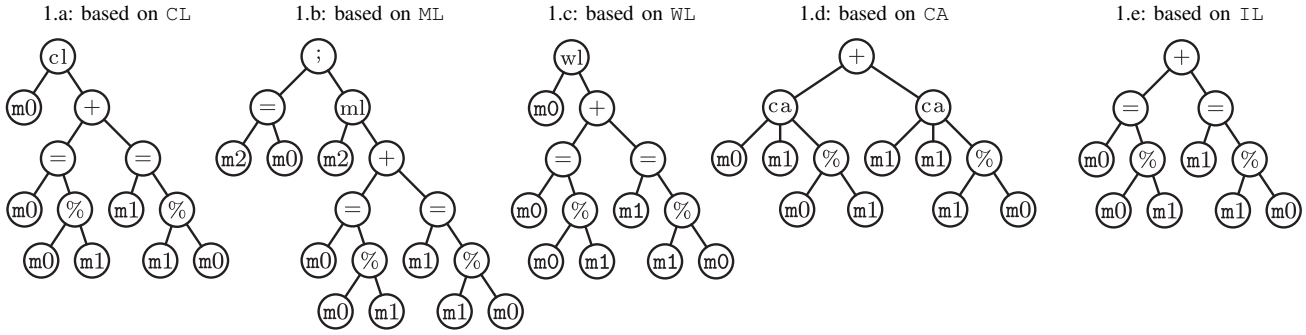
The nodes which represent a *counter loop* `CL` have subtrees, `x` and `y`. `x`, evaluated before entering the loop, is an expression returning the number of times the loop body `y` is to be executed.

The first subtree `x` of a *memory loop* is a terminal node identifying a variable. The second one, `y`, represents the loop body, which can access and modify `x`. `y` is executed until `x` becomes less or equal to 0. `x` is decreased by one in each loop iteration.

The *while loop* `WL` has the same structure as `CL` and is similar to the loop used in [11]. Here, the subtree `x` represents a condition. The loop body `y` is repeatedly executed until x evaluates to 0.

The new *conditional assignment* operator `CA` is similar to `RBGP` [5] with the extension that it also allows the nesting of conditions and assignments and does not necessarily utilize Transactional Memory. `CA` replaces the variable assignment instruction "=" and has three subtrees: `x`, `y`, and `z`. `x` is a single terminal node identifying the target variable. The value of the expression `z` will only be computed and assigned to the variable `x` (and returned) if the condition `y` evaluates to a non-zero value. Otherwise, the return value of the `CA` node is 0. In this representation, the complete programs will be evaluated repeatedly until no variable changes anymore.

Fig. 1: Algorithms that compute the GCD (greatest common divisor) of two variables $\mathtt{m}_0$ and $\mathtt{m}_1$ and return it in the last memory cell $\mathtt{m}_1$ ($\mathtt{m}_2$ for ML), i.e., that solve the gcd problem. (see Table II for the definition of the instructions)



The *implicit loop* IL has only a single subtree x representing the loop body, which will be evaluated until there is no change in any variable anymore.

Finally, we combine IL with CA and obtain the representation IC by replacing the traditional "=" operator with its conditional version and providing the implicit loop node in the instruction set.

B. Transactional Memory

A single piece of *transactional* memory, as used in RBGP, does not allow for more complex control structures. Loop conditions must be repeated in each instruction inside the loop, which makes the specification of nested loops complicated and their evolution less likely.

In order to integrate the beneficial transactional memory concept into standard GP and to enable the evolution of more complex control flows, we propose the *nesting* of memory structures, i.e., to provide one RBGP-style memory record for each (nested) control scope. We therefore use a stack M of these records in the program interpreter. Whenever a new control structure (a loop or a block of instructions) is entered, a new copy $\mathtt{M}_{i+1}$ of the *current* memory $\mathtt{M}_i$ is pushed on top of M. Like in RBGP, we distinguish variable values $\mathtt{M}_i.\mathtt{R}$ seen by read accesses and the uncommitted written values $\mathtt{M}_i.\mathtt{W}$.

When entering a new block or loop, both the memory $\mathtt{M}_{i+1}.\mathtt{R}$ and $\mathtt{M}_{i+1}.\mathtt{W}$ are copied from $\mathtt{M}_i.\mathtt{W}$. Transparent to the program in execution, its memory reading and writing operations always access the memory records on top of the stack. At the end of the control structures, this record is committed to its predecessor record ($\mathtt{M}_i \longleftarrow \mathtt{M}_{i+1}$), again invisible to the program. The process in execution has no knowledge about the stack of memories – only the *interpretation* of memory accesses has changed.

We illustrate the nested Transactional Memory approach using the example program given in Figure 2, which computes the factorial p of a natural number a. Both assignments are implicitly translated to changing the write memory $\mathtt{M}_1.\mathtt{W}$ with values computed from the read memory $\mathtt{M}_1.\mathtt{R}$ of the second memory record and current top-of-stack $\mathtt{M}_1$. Since $\mathtt{M}_1.\mathtt{W}$ is committed to $\mathtt{M}_1.\mathtt{R}$ only at the bottom of the loop, both instructions inside the loop see exactly the same values, regardless in which order they are executed. This principle

Fig. 2: A program using transactional memory for computing the factorial p of a natural number a.

```
1  // initially, the memory stack only holds record M0 (M0.R = M0.W)
2  // the input number a is located at m0 = M0.R[0] = M0.W[0]
3  // the output p is expected at m1 = M0.R[1] after the program finishes
4  m1 = 1;                  // actually, this means m1 = M0.W[1] = 1
5  // at the start of the loop, the new record M1 is pushed onto the stack
6  // and M1.R is set to M0.W and M1.W is set to M0.W
7  while (m0 > 0) {   // actually M1.R[0] > 0
8       m1 = m1 * m0  // actually M1.W[1] = M1.R[1] * M1.R[0]
9       m0 = m0 - 1   // actually M1.W[0] = M1.R[0] −1
10 } // commit M1.R = M1.W and M0.W = M1.W after each iteration
```

holds for arbitrarily nested loops and can be extended to function calls. We refer to this new nested memory setup as *Transactional Memory* (TM) and to GP setups with a traditional, single memory cell array as *simple memory* (SM).

C. Frequency Fitness Assignment

The objective function $f(x) \in [0, 1]$ in our work returns the error rate, the fraction of training cases t for which a program x gave the *wrong* result. f is specified in Equation 1, where tc is the total number of training cases and ϕ is the correct result for a training case.

$$f(x) = \frac{|\{i : (i \in 1..tc) \wedge (x(t_i) \neq \phi(t_i))\}|}{tc} \qquad (1)$$

Frequency Fitness [3] is a fitness measure that can be substituted in place of the direct invocation of the objective function in an optimization process. It rewards the discovery of new and different behaviors (from the perspective of the original objective function) instead of just good objective values.

The Frequency Fitness $H[f(x)]$ represents the history of the search accumulated in a lookup table H. If tc training cases are used, there are $tc + 1$ possible objective values 0, $\frac{1}{tc}$, $\frac{2}{tc}$, ..., 1. For each of these objective values $y \in [0, 1]$, H holds the number of times it was discovered during the optimization process. We use this value as fitness measure in GP.

In our domain, most of the possible programs x cannot solve any training case and have objective value $y = f(x) = 1$. Under FFA, the fact that many programs have this bad objective value means that the frequency of $y = 1$, $H[1]$,

Fig. 3: The **GP−FFA process** as an extension of the original GP−DIR.

1) **Allocate a frequency table H of appropriate size to accommodate one integer for each of the $tc + 1$ possible objective values. Initialize all table slots with 0.**
2) Generate initial population with ramped-half-and-half.
3) Compute the objective value of each of the n programs x in the population Pop.
4) **FFA: update the frequency table H: $\forall x \in Pop$ do $H[f(x)] \longleftarrow H[f(x)] + 1$.**
5) Fill a mating pool of n individuals by choosing them from Pop with a selection algorithm (e.g., tournament selection). **Instead of using $f(\cdot)$ for comparing individuals, use the Frequency Fitness $H[f(\cdot)]$.**
6) Create a new population by deriving $mr * n$ individuals via sub-tree replacement mutation and $(1 - mr) * n$ individuals via sub-tree exchange crossover from the parent programs in the mating pool.
7) If the maximum number of generations has not been exhausted, go back to Step 3.
8) Return: The program with the best f-value found.

will quickly increase. As H is subject to minimization, these solutions will be considered as bad by the optimization process – exactly as if f was used directly.

All objective values $f(x')$ are treated the same by FFA when they are discovered first. At this point in time, they are local optima and $H[f(x')] = 1$ holds. Solutions with easy-to-discover features will be sampled often and become uninteresting as the frequency of their objective value increases. Thus, FFA allows GP to escape from local optima. GP−FFA remembers the solution with the *best objective value* and returns it even if the GP process abandons it later. We refer to GP setups with Frequency Fitness as FFA and to those without as DIR and illustrate both in Figure 3.

IV. Benchmark Problems

In order to get a broad overview of the ability of Genetic Programming to synthesize exact integer algorithms, we chose 20 benchmark problems with different features (as summarized in Table I). For most of these problems, we use $tc = 100$ training cases t_i to evaluate the error rate f of the candidate solutions x. Each training case t has one expected output value $\phi(t)$ and usually one input value, but there also are problems where with more inputs. The training cases are stored in the first memory cells and the remainder of the q memory cells are initialized with zero prior starting a program x. After the execution of x, we expect its result $x(t_i)$ to be stored in its last memory cell $\mathtt{m}_{q-1}$. Most of the problems ask for an integer result, but two are Boolean decision tasks and either 0 or 1 as output.

1) Polynomial Problem [po2]. In the polynomial problem po2 [4] with $\phi_1(t_i) = t_i^3 + t_i^2 + 2 * t_i$ the training cases are the natural numbers from 1 to 100.

2) Sum Problem [su2]. The goal of the sum problem su2 [4] is to find the sum $\phi_2(t_i) = \sum_{j=1}^{t_i} j$ of the first t_i natural numbers [4], where $t_i = i$. The division operation is not present in the instruction set, thus forcing GP to synthesize loops.

3) Factorial Problem [fac]. The factorial problem fac [4, 11] asks for a program which can compute $\phi_3(t_i) = t_i!$ of a natural number $t_i = i$. The $tc = 12$ training cases are the natural numbers from 1 to 12. The last memory cell is always initialized to 1 before program startup.

TABLE I: Characteristics and descriptions of the benchmarks.

***A)* Number q of Memory Cells**
Goal: Explore impact of number of memory cells
2: Problems 1 (po2) to 3 (fac)
3: Problems 4+5 (po3) to 17 (mod)
5: Problems 18 (mi5) to 20 (sm5)
***B)* Number tc of Training Cases**
Goal: Explore impact of number of possible objective values
12: Problem 3 (fac)
31: Problem 7 (exp)
40: Problem 8 (ℓ20)
100: all others
***C)* Number of Input Parameters for Program**
Goal: Explore impact of number of input parameters
1: Problems 1 (po2) to 5 (su3), 7 (exp) to 14 (lsb), and 16 (qad)
2: Problems 6 (gcd), 15 (mul), and 17 (mod)
5: Problems 18 (mi5) to 20 (sm5)
***D)* Result Type**
Goal: Explore influence of result type
1. Boolean decision problem (0/1): Problems 8 (ℓ20) and 9 (prm)
2. Computation with integer result: all others
***E)* Is a Loop Required?**
Goal: Are problems that require loops harder, i.e., are loops harder to synthesize than sequences with multiple copies of instructions?
1. *very likely*: Problems 2 (su2), 5 (su3), 7 (exp), 9 (prm), 10 (ssq), and 14 (lsb) to 17 (mod)
2. *only few iterations* (may be unwound): Problems 3 (fac), 6 (gcd), and 11 (sra) to 13 (ild)
3. *no*: Problems 1 (po2), 4 (po3), 8 (ℓ20), and 18 (mi5) to 20 (sm5)
***F)* Number of Symbols (Functions+Terminals) Available to GP**
Goal: Which impact have more symbols = larger search spaces?
7: Problems 15 (mul) to 17 (mod)
8: Problems 1 (po2) to 3 (fac)
9: Problems 4 (po3) to 6 (gcd)
10: Problems 7 (exp) to 14 (lsb)
12: Problems 18 (mi5) to 20 (sm5)
This is not strictly a feature of the benchmark suite. The two constants (0 and 1) and read access to each memory cell are counted, the loop structure is not counted, as CA does not have one.

4+5) Three Memory Cell Versions [po3, su3]. We repeat the polynomial (po2) and sum (su2) problems, but provide $q = 3$ instead of 2 memory cells, as tasks po3 and su3. This gives information about how GP scales with the number of available memory cells.

6) GCD Problem [gcd]. In the gcd problem [4, 7], we try to find an algorithm which can compute the greatest common divisor $\phi_6(t_{i,1}, t_{i,2}) = \gcd(t_{i,1}, t_{i,2})$. A training case $t_i = (t_{i,1}, t_{i,2})$ this time consists of the two natural numbers $t_{i,1}$ and $t_{i,2}$.

For each of the $tc = 100$ training cases, we first generate a number $\xi \in 1..16$ and then, separately for $t_{i,1}$ and $t_{i,2}$, use ξ multiplied with uniform random numbers in 1..16 for a random number of repetitions. This way, we obtain a wide set of possible gcd results.

7) Binary Exponent Problem [exp]. Solving the binary exponent problem (exp) means to find an algorithm computing $\phi_7(t_i) = 2^{t_i}$. There are $tc = 31$ training cases $t_{i+1} = 2^i$ with $i \in 0 \ldots 30$. From this problem on (except for tasks mul to mod), the division operator is available to GP.

8) Less-Than-20 [ℓ20]. The goal of the "less-than-20" task

is to find a program that returns 0 if the training case t_i has a value of 20 or above and 1 otherwise, i.e., $\phi_8(t_i) = \begin{cases} 1 \text{ if } t_i < 20 \\ 0 \text{ otherwise} \end{cases}$. The difficulty here lies within producing an accurate result (0 or 1) with arithmetic operations only. This is made harder as only 0 and 1 are available as constants (and not 20). The training cases are the natural numbers from 0 to 39.

9) Prime Problem [prm]. In the prime problem, a function needs to be found that returns 1 if a number t_i is prime and 0 otherwise, i.e., $\phi_9(t_i) = \begin{cases} 1 \text{ if } t_i \text{ is prime} \\ 0 \text{ otherwise} \end{cases}$. This problem is harder than the $\ell 20$ task, as it requires the evolved algorithms to test all potential divisors except 1 and t_i itself. All of the $tc = 100$ unique training cases are from 2..499. Half of them are prime numbers, the others are uniformly distributed random non-prime numbers.

10) Sum-of-Squares Problem [ssq]. In the sum-of-squares task, the expected results is $\phi_{10}(t_i) = \sum_{i=1}^{t_i} i^2$. This is a harder version of the sum problem, as it requires squaring the elements before adding them up. The training cases the numbers from 0 to 99.

11) Square Root Problem [sra]. In the square root task, the goal is to find a program which can compute $\phi_{11}(t_i) = \sqrt{t_i}$ where the $tc = 100$ and $t_i = i^2$ with $i \in 0..99$.

12) Square Root Problem 2 [srb]. We repeat the sra experiment with the 100 numbers from 0..99 as training cases and set $\phi_{12}(t_i) = \lfloor \sqrt{t_i} \rfloor$. This should be harder, as there now are training cases which do not have an exact integer root.

13) Iterated Binary Logarithm Problem [ild]. The iterated version of the binary logarithm $y = 2^x \Rightarrow \mathrm{ld}(y) = \lfloor x \rfloor$ is defined as $\mathrm{ld}^\star(y) = \begin{cases} 0 \text{ if } t_i \leq 1 \\ 1 + \mathrm{ld}^\star(\mathrm{ld}(y)) \text{ otherwise} \end{cases}$ and $\phi_{13}(t_i) = \mathrm{ld}^\star(t_i)$. This problem is again hard, as it requires repeated division by 2 while counting up in another variable. As training cases, we use the natural numbers from 0 to 19 additional to 80 unique random numbers uniformly distributed in $20..(2^{31} - 1)$.

14) Least Significant Bit Problem [lsb]. The value of the least significant bit in the two's complement representation of an integer number t_i is the value $\phi_{14}(t_i) = \arg\max_{j \in \mathbb{N}_0} \left\{ 2^j | t_i \right\}$. Here we use $tc = 100$ unique random training cases which are approximately[1] uniformly distributed in $\phi_9(t_i)$. For each training case, this is achieved by shifting a bit mask full of 1 bits to the left by a random number uniformly distributed in 0..31 and computing *binary and* with a random number uniformly distributed in $1..(2^{31} - 1)$.

15) Multiplication Problem [mul]. In the mul problem, we want to find an algorithm that can multiply two natural numbers $\phi_{15}(t_i) = t_{i,1} * t_{i,2}$. The training cases are tuples $t_i = (t_{i,1}, t_{i,2})$ uniformly randomly distributed with $t_{i,1}, t_{i,2} \in 0..249$. For this and the following two problems, the multiplication, division, and modulo division operators have been excluded from the function set.

16) Quadratic Function Problem [qad]. Solving the qad problem means synthesizing an algorithm that can compute $\phi_{16}(t_i) = (t_i - 1)(t_i + 2)$, again without multiplication and division. The training cases are the numbers from 0 to 99.

[1]There are too few numbers with ϕ_{14} of 0, 1, etc. to achieve a fully uniform distribution.

TABLE II: Basic operators in our experiments.

Operator	Function
$+, -$	Addition, Subtraction.
$*$	multiplication, only available in problems 1–14 and 18–20
a/b	protected division, returns $\frac{a}{b}$ if $b \neq 0$, and a otherwise, only available in problems 7–14 and 18–20
$a\%b$	protected modulo division, returns $a \bmod b$ if $b \neq 0$, and a otherwise, only available in problems 1–14 and 18–20
$\mathtt{m_i} = \xi$	fixed-index memory assignment, set the value of variable $\mathtt{m_i}$ to the result of the expression ξ (which is also returned)
$a; b$	concatenation of two expressions a and b, return value of b
$\mathtt{m}_0, .., \mathtt{m}_{q-1}$	the value of the one of q memory variables
0, 1	the only two ephemeral random constants (ERCs) available

17) Modulo Division Problem [mod]. The modulo division problem mod aims at creating an algorithm that can compute $\phi_{17}(t_i) = t_{i,1} \bmod t_{i,2}$, i.e., the remainder of the division of $t_{i,1}$ by $t_{i,2}$. The training cases are $tc = 100$ tuples of two numbers, this time of the form $(1,1), (2,1), (2,2), (3,1), (3,2), (3,3), (4,1), \ldots$.

18) Minimum-of-Five Problem [mi5]. The last three problems all have five memory cells and the complete operator set (including multiplication, division, and modulo division) available. Their $tc = 100$ training cases are random numbers uniformly distributed in $0..(2^{31} - 1)$. In the mi5 task, the goal is to find the smallest of five numbers, i.e., $\phi_{18}(t_i) = \min \{t_{i,1}, t_{i,2}, t_{i,3}, t_{i,4}, t_{i,5}\}$.

The hardness of this task lies in the complex required code structure. No comparison operator or if-then-else construct is available. As memory is not indexed, each comparison must be synthesized separately and code replication will not work.

19) Maximum-of-Five Problem [ma5]. In the ma5 task, the goal is to find the largest of five numbers, i.e., $\phi_{19}(t_i) = \max \{t_{i,1}, t_{i,2}, t_{i,3}, t_{i,4}, t_{i,5}\}$.

20) Sum-of-Five Problem [sm5]. In the sm5 task, the goal is to find the sum of five numbers, i.e., $\phi_{20}(t_i) = t_{i,1} + t_{i,2} + t_{i,3} + t_{i,4} + t_{i,5}$. This should be harder than the mi5 and ma5 task, as here no partially correct program exists (a program only computing $\max \{t_{i,1}, t_{i,2}\}$ should have a hit rate of 40% in ma5).

V. Experiments

A. Setup

We use the Genetic Programming implementation of the ECJ framework [15] with a population size of 1000, a generation limit of 100, tournament selection with 7 contestants, 10% point mutation, 90% subtree exchange crossover, and a maximum tree depth of 17, as these settings performed well in previous experiments [4]. The instruction set is given in Table II. In the experiment, each of the six loop instructions is combined with either FFA or DIR and TM or SM. For each setting we performed 100 runs.

During the execution of a program, all of its loops together are allowed to perform at most 1000 iterations before it is forcefully terminated in order to prevent endless loops. The return value of a loop expression is the result of the sub-expression evaluated last.

B. Results

In total, we performed 48 000 runs, out of which only 5422 (11%) were successful, i.e., found a solution satisfying all training cases. This shows how hard the evolutionary synthesis of algorithms is, but also that it is not impossible.

1) Success Rate: In Table III, we print the fractions of successful runs for the different configurations and benchmark problems, i.e., the average number of runs that find the solution. The table presents the success rates of specific configurations of memory, fitness assignment, and loop structure in the lower and the mean rates aggregated over multiple setups in the upper part. The worst value per column and division is printed in gray color and the best in **bold face**.

a) Performance of Frequency Fitness: GP−FFA outperforms plain GP−DIR in 17 out of 20 problems and has a 40% higher mean success rate. There are 240 configuration pairs which only differ in whether GP−FFA or GP−DIR was applied. In 124 of these, the success rate achieved with GP−FFA is higher and only in 31 it is lower. In total, 3378 runs of GP−FFA were successful, whereas GP−DIR only discovered 2320 correct programs. Several task/loop structure combinations were only successful in conjunction with GP−FFA, e.g., IL on sm5 or CL on sra.

b) Performance of Transactional Memory: Our second new approach, Transactional Memory, does not have an overall good impact: its mean success rate is worse in 17 out of 20 experiments. Looking at "TM vs. SM" configuration pairs, we find 53 situations where TM improved the success rate and 112 where decreased it. GP with TM found 2215 solutions in total, while SM found 3483.

c) Performance of Loop Structures: The best loop structures, memory loop ML and conditional assignment CA, were both successful in 16% of all cases, with FFA even in 18%. The most efficient setup is SM-FFA-ML which finds correct solutions in about one fourth of all runs. The worst mean success rates have configurations with implicit loop IL and the while loop WL.

d) Influence of Problem Features: The easiest problem was the less-than-20 task ℓ20, followed by the greatest common divisor (gcd) and maximum-of-five (ma5) problems. Interestingly, they differ in almost all features. The only thing they have in common is that they can be solved either without loop structure or with an unwound loop.

Amongst the hardest tasks we find the prime number detection prm, the least significant bit task lsb, the square root task srb, and the sum-of-squares problem ssq. They all require a loop structure and for none of them a loop over a simple arithmetic operation (as, e.g., in the su2 task) is sufficient. In the prime problem, this feature is combined with a Boolean decision task.

Whether synthesizing a loop structure is necessary to solve a task seems to be the problem feature with the largest impact on the success rate. Still, for each problem there is at least one configuration achieving a success rate of at least 4% and each problem was solved at least 16 times, which definitely is significant.

Increasing the number q of memory cells from two to three (po2 and su2 vs. po3 and su3) basically halves the success rates. Although they even require five memory cells, the problems mi5 and ma5 can still be solved at fairly high rates, likely because they do not require loop structures.

2) Objective Function: Since the problems may be deceptive [3, 14], analyzing the success rate should be preferred over analyzing the objective values. Nevertheless, if we compare the runs using FFA with those using f directly, we find a much faster decline of the objective values (not illustrated here). The runs with TM converge slower than those with SM. Over all problems and setups, ML, CA, and the primitive CL perform best with respect to f.

3) Runtime: In Table IV, we provide the median required time per run measured in seconds for single-thread executions on an Intel Core i7 2.0GHz CPU, and 4GiB RAM, Java 1.7.0_01, and Windows 7 64bit.

FFA increases the median runtime because close-to-correct solutions are discovered more often. These often employ loop structures and run longer. TM increases the runtime as well, as it requires more complex interpretation.

The setup SM-FFA-ML, which needs about 40s per run at a success rate of 24%, seems to be a very effective choice. Although the conditional assignment CA often finds good solutions too, its runtime is about 20% higher, as it requires variable convergence detection.

4) Evolved Programs: Besides the statistical evaluation of the experiments, it is also interesting to see what kind of programs actually evolved. We cannot conduct a complete analysis of all 5700 evolved correct programs. We selected one program for each problem and translated it to Java. We list the most interesting and short ones in Figure 4.

The counter-loop program given in Figure 4.c looks confusing but indeed computes $\lfloor\sqrt{t}\rfloor$ for any positive integer t. Inside the loop, this program contains two identical copies of two lines of code. If one dares to remove one of these two copies, the program still works correctly ... except for the single input (training case) 5, for which it will then calculate 1 (instead of 2).

The most interesting result among the selection may be the CA-based algorithm 4.f using Transactional Memory for computing the lsb. It needs many iterations for small inputs and becomes *faster for larger inputs*: It needs more than 42s to compute $\texttt{lsb}(1) = 1$, 13s to get $\texttt{lsb}(3) = 1$, 0.5s for $\texttt{lsb}(105) = 1$, and 0.1s to get $\texttt{lsb}(532) = 4$ on the aforementioned system. This algorithm seemingly utilizes an overflow of the 32bit arithmetic.

All in all, we can confirm that GP is able to synthesize non-trivial algorithms and even sometimes discovers entirely new and potentially odd approaches for solving hard problems.

TABLE III: The success rates, i.e., the number of runs which found algorithms that can solve all training cases correctly, for different settings.

Setup	po2	su2	po3	su3	fac	gcd	exp	ℓ20	prm	ssq	sra	srb	ild	lsb	mul	qad	mod	mi5	ma5	sm5	All
Memory: normal memory access (SM), transactional memory (TM), see Section III-B; aggregated over all settings																					
SM	**0.16**	**0.12**	**0.09**	**0.07**	0.07	**0.48**	**0.05**	**0.87**	**0.01**	**0.02**	**0.08**	**0.02**	**0.03**	0.01	**0.23**	**0.07**	**0.07**	**0.20**	**0.23**	0.04	**0.15**
TM	0.12	0.09	0.05	0.05	**0.13**	0.22	0.00	0.65	0.00	0.01	0.06	0.00	0.02	**0.01**	0.06	0.04	0.01	0.09	0.14	**0.09**	0.09
Fitness: error rate f used directly (DIR), Frequency Fitness (of error rate) (FFA), see Section III-C; aggregated over all settings																					
DIR	0.10	0.10	0.05	0.05	0.07	0.29	0.03	0.66	0.00	**0.02**	0.06	0.00	**0.03**	**0.01**	0.12	0.01	0.03	0.10	0.13	0.06	0.10
FFA	**0.18**	**0.11**	**0.09**	**0.07**	**0.13**	**0.41**	**0.03**	**0.86**	**0.01**	0.01	**0.08**	**0.02**	0.02	**0.01**	**0.17**	**0.09**	**0.05**	**0.20**	**0.24**	**0.07**	**0.14**
The full factorial resolution, i.e., results for each specific setup of memory, fitness assignment, and loop structure.																					
SM-DIR-CL	0.14	0.03	0.04	0.01	0.01	0.37	**0.20**	0.93	0.00	0.00	0.01	0.00	0.00	0.00	0.44	0.07	0.22	0.04	0.07	0.11	0.13
SM-DIR-ML	0.18	**0.66**	0.08	0.32	0.27	0.36	0.11	0.70	0.00	**0.21**	0.03	0.00	0.01	0.00	0.63	0.01	0.13	0.00	0.00	0.04	0.19
SM-DIR-WL	0.11	0.00	0.08	0.00	0.00	0.42	0.00	0.51	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.07	0.06
SM-DIR-CA	0.11	0.00	0.08	0.00	0.00	0.55	0.00	0.93	0.01	0.00	0.14	0.01	**0.13**	0.03	0.00	0.00	0.00	0.40	0.50	0.00	0.14
SM-DIR-IL	0.09	0.01	0.11	0.01	0.00	0.10	0.00	0.81	0.00	0.00	0.04	0.00	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.06
SM-DIR-IC	0.04	0.03	0.00	0.00	0.00	0.68	0.00	0.88	0.01	0.00	0.13	0.02	0.05	0.02	0.01	0.00	0.00	0.43	0.55	0.02	0.14
SM-FFA-CL	**0.32**	0.03	0.16	0.00	0.03	0.41	0.19	**1.00**	0.00	0.00	0.01	**0.07**	0.00	0.00	0.72	**0.41**	**0.31**	0.19	0.17	0.05	0.20
SM-FFA-ML	0.14	0.57	0.15	**0.46**	0.45	0.47	0.12	0.92	0.02	0.05	0.05	0.00	0.00	0.00	**0.89**	0.30	0.15	0.00	0.00	0.06	**0.24**
SM-FFA-WL	**0.32**	0.00	**0.17**	0.02	0.04	0.46	0.00	0.90	0.01	0.00	0.01	0.01	0.05	0.01	0.00	0.00	0.00	0.00	0.02	0.06	0.10
SM-FFA-CA	0.17	0.00	0.05	0.00	0.02	0.80	0.00	0.97	0.02	0.00	**0.22**	0.06	0.02	0.01	0.00	0.00	0.00	0.64	**0.74**	0.00	0.19
SM-FFA-IL	0.21	0.03	0.11	0.00	0.00	0.42	0.01	0.93	0.01	0.00	0.15	0.01	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.10
SM-FFA-IC	0.12	0.02	0.01	0.03	0.00	0.76	0.00	0.90	**0.05**	0.00	0.16	0.03	0.01	0.01	0.01	0.00	0.00	**0.72**	**0.74**	0.04	0.18
TM-DIR-CL	0.12	0.07	0.07	0.03	0.01	0.00	0.00	0.81	0.00	0.00	0.00	0.00	0.00	0.00	0.26	0.05	0.04	0.00	0.00	0.13	0.08
TM-DIR-ML	0.17	0.27	0.04	0.14	0.43	0.01	0.00	0.30	0.00	0.01	0.18	0.00	0.03	0.00	0.03	0.00	0.01	0.00	0.00	0.14	0.09
TM-DIR-WL	0.15	0.00	0.07	0.00	0.00	0.11	0.00	0.24	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.01	0.03	0.08	0.04
TM-DIR-CA	0.04	0.05	0.04	0.08	0.05	0.83	0.00	0.66	0.00	0.00	0.11	0.01	0.06	0.01	0.04	0.01	0.00	0.24	0.30	0.10	0.13
TM-DIR-IL	0.03	0.01	0.02	0.00	0.00	0.04	0.00	0.75	0.00	0.00	0.02	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.01	0.00	0.05
TM-DIR-IC	0.04	0.03	0.00	0.01	0.09	0.05	0.00	0.45	0.00	0.00	0.04	0.00	0.03	0.02	0.00	0.00	0.00	0.05	0.15	0.06	0.05
TM-FFA-CL	0.20	0.04	0.12	0.06	0.01	0.00	0.02	**1.00**	0.00	0.00	0.00	0.00	0.01	0.00	0.34	0.32	0.07	0.00	0.00	0.09	0.11
TM-FFA-ML	0.24	0.28	0.09	0.13	**0.58**	0.00	0.00	0.72	0.00	0.03	0.05	0.00	0.01	0.00	0.07	0.09	0.04	0.00	0.00	0.12	0.12
TM-FFA-WL	0.17	0.00	0.15	0.00	0.00	0.17	0.00	0.74	0.01	0.00	0.01	0.00	0.02	0.00	0.00	0.00	0.00	0.08	0.16	**0.16**	0.08
TM-FFA-CA	0.15	0.14	0.01	0.06	0.21	**0.87**	0.00	0.76	0.01	0.02	0.12	0.01	0.00	**0.04**	0.03	0.00	0.00	0.45	0.63	0.12	0.18
TM-FFA-IL	0.03	0.06	0.03	0.02	0.02	0.42	0.00	0.76	0.00	0.00	0.09	0.00	0.03	**0.04**	0.00	0.00	0.00	0.00	0.01	0.04	0.08
TM-FFA-IC	0.05	0.14	0.00	0.04	0.15	0.09	0.01	0.66	0.01	0.00	0.05	0.01	0.00	0.01	0.00	0.00	0.00	0.29	0.44	0.01	0.10
All	0.14	0.10	0.07	0.06	0.10	0.35	0.03	0.76	0.01	0.01	0.07	0.01	0.02	0.01	0.14	0.05	0.04	0.15	0.19	0.06	0.12

TABLE IV: The median time per run in seconds, aggregated over all setups of the groups given in the left-most column.

Setup	po2	su2	po3	su3	fac	gcd	exp	ℓ20	prm	ssq	sra	srb	ild	lsb	mul	qad	mod	mi5	ma5	sm5	All
Memory: normal memory access (SM), transactional memory (TM), see Section III-B; aggregated over all settings																					
SM	**38.8**	54.1	27.1	54.5	7.8	**35.8**	**22.6**	**12.0**	**99.7**	**44.2**	**65.3**	**65.7**	**57.3**	**57.4**	**16.4**	**38.8**	**42.4**	**38.4**	**39.2**	**27.7**	**40.1**
TM	39.8	**50.3**	**22.0**	**50.3**	**7.2**	55.2	23.5	25.6	116.8	45.8	73.4	73.8	70.5	62.3	33.3	41.7	64.3	58.1	58.8	32.9	48.4
Fitness: error rate f used directly (DIR), Frequency Fitness (of error rate) (FFA), see Section III-C; aggregated over all settings																					
DIR	**35.7**	54.4	**21.9**	55.1	9.6	44.6	24.6	22.9	**52.5**	48.6	70.3	**66.9**	**56.2**	**56.2**	28.9	**18.7**	**45.9**	**41.5**	**41.9**	30.8	**41.0**
FFA	44.0	**50.6**	25.4	**50.1**	**5.4**	**44.5**	**21.3**	**11.8**	155.2	**40.4**	**69.2**	72.5	70.9	64.1	**19.9**	59.6	64.7	54.5	55.5	**29.0**	47.7
Loops: counter loop (CL), memory loop (ML), while loop(WL), implicit loop(IL), conditional assignment (CA), see Section III-A; aggregated																					
CL	61.7	91.6	36.7	84.1	8.7	73.0	30.2	**4.9**	201.6	74.3	92.5	132.9	74.7	61.5	49.0	30.8	85.0	56.6	59.0	48.3	62.9
ML	41.0	**40.0**	26.7	51.9	**3.9**	74.9	**18.1**	9.2	92.2	39.8	80.4	77.8	59.4	64.6	23.9	**19.3**	42.1	51.4	52.4	45.1	44.4
WL	37.9	44.0	26.3	45.4	10.8	72.8	25.8	20.1	**63.6**	36.3	59.6	72.0	74.8	69.5	22.5	30.4	50.9	56.9	56.3	35.3	43.5
CA	34.8	54.5	13.2	63.0	8.6	**5.4**	25.1	34.8	239.1	57.0	89.3	70.0	72.1	66.3	31.2	97.7	66.8	44.9	44.5	25.3	48.6
IL	42.6	45.1	32.5	**41.4**	6.7	34.3	19.5	20.0	93.7	**28.3**	**40.1**	51.5	49.3	**44.2**	**17.8**	42.4	**41.0**	40.5	39.8	18.6	**35.5**
IC	**18.5**	50.3	**12.8**	47.6	7.4	28.1	23.0	37.6	116.6	46.4	64.0	**49.3**	**49.1**	45.8	19.1	53.5	44.7	**35.4**	**35.2**	**16.5**	36.5
All	39.3	52.8	24.1	52.5	7.5	44.6	23.2	17.9	108.7	45.1	69.7	69.7	63.3	59.7	22.8	40.3	53.3	48.3	48.9	29.9	44.1

VI. Conclusions

The findings and key contributions of this paper can be summarized as follows: First, we contribute a benchmark suite with 20 test problems. This suite covers many different aspects that make an algorithm synthesis task hard.

Second, to the best of our knowledge, this work is the largest and most complete experimental study of integer algorithm synthesis. It provides a good overview over what GP can do and where its limits are in this domain.

Third, although the evolutionary synthesis of exact integer algorithms is hard, even truly non-trivial programs were discovered. Even though we were able to investigate only a small subset of the evolved programs manually, we found that GP discovers unusual, interesting, but still efficient

Fig. 4: Manual translation to Java for *selected* short and interesting programs solving the benchmark problems.

4.a. Translated `CL-TM` program for `qad`.

```
static int qad(int m0) {
  int m1 = 0, t = (m0 - 1), tm1 = 0;
  for (int i = t; i > 0; i--) {
    tm1 = (m1 + m0);
    t   = m0 + m0 + m1 - 1;
    m1  = tm1;
  }
  return (m0 - (1 - t));
}
```

4.b. Translated `WL-SM` program for `ℓ20`.

```
static int ℓ20(int m0) {
  int t = ((m0 - 2) / 3) / 3; // integer division
  if (t < 2) return 1;
  else       return 0;
}
```

4.c. Translated `CL-SM` program for `srb`.

```
static int srb(int m0) {
  int m2 = 0, t = 0;
  for (int i = m0; i > 0; i--) {
    t  = m0 / (1 + m2);
    m2 = (1 + m2 - ((t != 0) ? (m2 / t) : m2));
    t  = m0 / (1 + m2);
    m2 = (1 + m2 - ((t != 0) ? (m2 / t) : m2));
  }
  return m2;
}
```

4.d. Translated `CA-TM` program for `gcd`.

```
static int gcd(int m0, int m1) {
  int m2 = 0, tm0 = 0, tm1 = 0, tm2 = 0;
  for (;;) {
    if (m1 != 0) tm1 = m0;
    tm0 = (m0 != 0) ? (m1 % m0) : m1;
    if ((m0 != 0) && (m1 != 0) &&
        ((m1 % m0) == 0)) tm2 = m0;
    if ((m0 == tm0) && (m1 == tm1) &&
                       (m2 == tm2)) return m2;
    m0 = tm0; m1 = tm1; m2 = tm2;
  }
}
```

4.e. Translated `IC-SM` program for `mi5`.

```
static int mi5(int m0, int m1, int m2,
                       int m3, int m4) {
  int t;
  for(;;) {
    t = m4;
    if(m4 > m0) m4 = m0;
    if(m4 > m2) m4 = m2;
    if((m4 == t) && (m2 == m1) &&
                    (m1 == m3)) return m4;
    m2 = m1;
    m1 = m3;
  }
}
```

4.f. Translated `CA-TM` program for `lsb`.

```
static int lsb(int m0) {
  int m1 = 0, m2 = 0, tm0 = m0, tm2 = m2;
  for (;;) { // small m0 ⇒ many iterations!
    if (m0 != 0) {
      tm2 = (m1 - m0);
      m1  = 1;
      if (m2 != 0) {
        tm2 = (m2 - m0);
        if (tm2 != 0) tm0 = (m0 % tm2);
      }
    }
    if ((tm0 == m0) && (tm2 == m2)) return m2;
    m0 = tm0; m2 = tm2;
  }
}
```

algorithms.

Fourth, we were able to confirm with great significance that `FFA` has a tremendous positive impact on the performance of GP, as it can increase the success rate by 40%.

Fifth, we proposed the new Transactional Memory idea `TM` to reduce epistasis in GP. However, it turned out to be a bad idea. It works best together with the conditional assignment-based program `CA`, but even there decreases the mean success rate by 1%. Interestingly, `CA-TM` is most similar to our `eRBGP` method introduced in [5], from which we aimed to transfer the idea of transactional memory to standard GP.

Sixth, in terms of loop structure, both the traditional *memory loop* `ML` and the *conditional assignment* `CA` performed best, in average over all settings and experiments.

Acknowledgements. National Natural Science Foundation of China under Grants 61175065, 61329302, and 61150110488, the Technological Fund of Anhui Province for Outstanding Youth under Grant 1108085J16, the Special Financial Grant 201104329 from the China Postdoctoral Science Foundation, and the Chinese Academy of Sciences (CAS) Fellowship for Young International Scientists 2011Y1GB01.

REFERENCES

[1] T. Weise, M. Wan, K. Tang, and X. Yao, "Evolving Exact Integer Algorithms with Genetic Programming," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'14), Proceedings of the 2014 World Congress on Computational Intelligence (WCCI'14)*. Beijing, China: Beijing International Convention Center (BICC): Los Alamitos, CA, USA: IEEE Computer Society Press, July 6–11, 2014, pp. 1816–1823, doi:10.1109/CEC.2014.6900292, isbn:978-1-4799-1488-3.

[2] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, ser. Bradford Books. Cambridge, MA, USA: MIT Press, December 1992, 1992 first edition, 1993 second edition. [Online]. Available: http://books.google.de/books?id=Bhtxo60BV0EC

[3] T. Weise, M. Wan, K. Tang, P. Wang, A. Devert, and X. Yao, "Frequency fitness assignment," *IEEE Transactions on Evolutionary Computation (IEEE-EC)*, vol. 18, no. 2, pp. 226–243, April 2014.

[4] M. Wan, T. Weise, and K. Tang, "Novel loop structures and the evolution of mathematical algorithms," in *Proceedings of the 14th European Conference on Genetic Programming (EuroGP'11)*, ser. Lecture Notes in Computer Science (LNCS), S. Silva, J. A. Foster, M. Nicolau, P. Machado, and M. Giacobini, Eds., vol. 6621/2011. Torino, Italy: Berlin, Germany: Springer-Verlag GmbH, April 27–29, 2011, pp. 49–60.

[5] T. Weise and K. Tang, "Evolving distributed algorithms with genetic programming," *IEEE Transactions on Evolutionary Computation (IEEE-EC)*, vol. 16, no. 2, pp. 242–265, April 2012.

[6] J. R. Finkel, "Using genetic programming to evolve an algorithm for factoring numbers," in *Genetic Algorithms and Genetic Programming at Stanford*. Stanford, CA, USA: Stanford University Bookstore, Stanford University, Fall 2003, pp. 52–60. [Online]. Available: http://www.genetic-programming.org/sp2003/Finkel.pdf

[7] T. Lai, "Discovery of understandable math formulas using genetic programming," in *Genetic Algorithms and Genetic Programming at Stanford*. Stanford, CA, USA: Stanford University Bookstore, Stanford University, Fall 2003, pp. 118–127. [Online]. Available: http://www.genetic-programming.org/sp2003/Lai.pdf

[8] Y. Qi, B. Wang, and L. Kang, "Genetic programming with simple loops," *Journal of Computer Science and Technology (JCST)*, vol. 14, no. 4, pp. 429–433, July 1999. [Online]. Available: http://www.zemris.fer.hr/~yeti/studenti/izvori/Genetic%20Programming%20with%20simple%20loops.pdf

[9] V. Ciesielski and X. Li, "Experiments with explicit for-loops in genetic programming," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'04)*, vol. 1. Portland, OR, USA: Los Alamitos, CA, USA: IEEE Computer Society Press, June 20–23, 2004, pp. 494–501. [Online]. Available: http://researchbank.rmit.edu.au/eserv/rmit:1498/n2004000338.pdf

[10] G. Wijesinghe and V. Ciesielski, "Evolving programs with parameters and loops," in *11th IEEE Congress on Evolutionary Computation (CEC'10), 2010 IEEE World Congress on Computation Intelligence (WCCI'10)*. Barcelona, Catalonia, Spain: Centre de Convencions Internacional de Barcelona: Piscataway, NJ, USA: IEEE Computer Society, July 18–23, 2010, pp. 1–8.

[11] G. Chen and M. Zhang, "Evolving while-loop structures in genetic programming for factorial and ant problems," in *Advances in Artificial Intelligence. Proceedings of the 18th Australian Joint Conference on Artificial Intelligence (AI'05)*, ser. Lecture Notes in Computer Science (LNCS), S. Zhang and R. Jarvis, Eds., vol. 3809/2005. Sydney, NSW, Australia: University of Technology, Sydney (UTS): Berlin, Germany: Springer-Verlag GmbH, December 5–9, 2005, pp. 1079–1085.

[12] N. F. McPhee and R. Poli, "Memory with memory: Soft assignment in genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'08)*, M. Keijzer, G. Antoniol, C. B. Congdon, K. Deb, B. Doerr, N. Hansen, J. H. Holmes, G. S. Hornby, D. Howard, J. Kennedy, S. P. Kumar, F. G. Lobo, J. F. Miller, J. H. Moore, F. Neumann, M. Pelikan, J. B. Pollack, K. Sastry, K. O. Stanley, A. Stoica, E. Talbi, and I. Wegener, Eds. Atlanta, GA, USA: Renaissance Atlanta Hotel Downtown: New York, NY, USA: ACM Press, July 12–16, 2008, pp. 1235–1242. [Online]. Available: http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/McPhee_2008_gecco.html

[13] M. Hutter and S. Legg, "Fitness uniform optimization," *IEEE Transactions on Evolutionary Computation (IEEE-EC)*, vol. 10, no. 5, pp. 568–589, October 2006. [Online]. Available: http://arxiv.org/abs/cs/0610126v1

[14] J. Lehman and K. O. Stanley, "Abandoning objectives: Evolution through the search for novelty alone," *Evolutionary Computation*, vol. 19, no. 2, pp. 189–223, Summer 2011. [Online]. Available: http://eplex.cs.ucf.edu/papers/lehman_ecj10.pdf

[15] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, J. Bassett, R. Hubley, and A. Chircop, *ECJ: A Java-based Evolutionary Computation Research System*. Fairfax, VA, USA: George Mason University (GMU), Evolutionary Computation Laboratory (EClab), 2006. [Online]. Available: http://cs.gmu.edu/~eclab/projects/ecj/


```
@inproceedings{WWTY2014EEIAWGP,
  author    = {Thomas Weise and Mingxu Wan and Ke Tang and Xin Yao},
  title     = {{Evolving Exact Integer Algorithms with Genetic
               Programming}},
  booktitle = {Proceedings of the IEEE Congress on Evolutionary
               Computation (CEC'14), Proceedings of the 2014 World
               Congress on Computational Intelligence (WCCI'14)},
  publisher = {Los Alamitos, CA, USA: IEEE Computer Society Press},
  address   = {Beijing, China: Beijing International Convention
               Center (BICC)},
  pages     = {1816--1823},
  year      = {2014},
  month     = jul # {~6--11, },
  doi       = {10.1109/CEC.2014.6900292},
  isbn      = {978-1-4799-1488-3},
},
```