

Novel Loop Structures and the Evolution of Mathematical Algorithms

Mingxu Wan, Thomas Weise, and Ke Tang

University of Science and Technology of China, Hefei, Anhui 230026, China

Preview

This document is a preview version
and not necessarily identical with
the original.

<http://www.it-weise.de/>

Abstract In this paper, we analyze the capability of Genetic Programming (GP) to synthesize non-trivial, non-approximative, and deterministic mathematical algorithms with integer-valued results. Such algorithms usually involve loop structures. We raise the question which representation for loops would be most efficient. We define five tree-based program representations which realize the concept of loops in different ways, including two novel methods which use the convergence of variable values as implicit stopping criteria. Based on experiments on four problems under three fitness functions (error sum, hit rate, constant 1) we find that GP can statistically significantly outperform random walks. Still, evolving said algorithms seems to be hard for GP and the success rates are not high. Furthermore, we found that none of the program representations could consistently outperform the others, but the two novel methods with indirect stopping criteria are utilized to a much higher degree than the other three loop instructions.

1 Introduction

Genetic Programming (GP) [4, 15], today is used for many tasks such as symbolic regression, the synthesis of electronic circuits, or the design of distributed systems [16]. However, there exists little evidence that it is actually suitable to derive *programs* in the intuitive sense of the word, maybe in the form of C++ programs. More specifically, we consider the evolution of non-trivial, non-approximative, deterministic mathematical algorithms which compute distinct discrete results.

Such synthesis tasks are fundamentally different from the domains where GP excels. If an intermediate solution generates close-to-optimal outputs, this does not necessarily mean that its program structure is also similar to a good

solution. In [14, 18], we pointed out that traditional imperative program representations (such as those commonly used for Standard or Linear Genetic Programming) exhibit strong epistatic effects [19]: It is not possible to modify one part of a program without affecting the behavior of the other parts [14, 17]. As a consequence, the fitness landscapes of the problems under consideration are often rugged and likely contain large neutral areas. Hence, Genetic Programming utilizing imperative program representations similar to high-level programming structures may not be efficient.

We first provide a comprehensive related work study in Section 2. Many of the works presented there are positive about the ability to synthesize programs which effectively use loops with GP. However, their experimental setups are often specialized and their results cannot be generalized. With our work, we contribute evidence about the actual utility of loop instructions in Genetic Programming. We will show that some of the example problems we used in our experiments are hard for GP, although they are not too different from those listed in the related work.

We test five different loop representations which are specified in Section 3. Two novel loop representations, *conditional assignments* (CA) and *implicit loops* (IL) are defined. Both replace the traditional, explicit loop conditions with an *implicit* convergence condition based on side-effects. Whereas in CA, the whole programs are turned into single loops, IL allows the nesting of independent loops.

We conduct a large-scale experimental study in which we apply Genetic Programming using the five loop representations to four test problems (specified in Section 4) using different fitness functions (error sum, hit rate, and constant 1). In our experiments discussed in Section 5, we find that Genetic Programming is usually more efficient than random walks. However, its capability to solve the harder problems was not yet satisfying. We show that the new non-imperative and implicit loop structures CA and IL are utilized more efficiently by Genetic Programming than those presented in the related work. Although, from an absolute fitness point of view, the three other methods perform not significantly worse, Genetic Programming does more efficiently utilize our novel loop structures and with CA, even can solve one of the hardest benchmark tasks.

2 Related Work

The usage of loops in GP has a tradition of at least 15 years, but only few works considering it have actually been published. Teller [13] proved that (tree-based) Genetic Programming with indexed memory can be Turing complete and also proposed some methods to limit the runtime of programs [12]. These works do not contain any experimental validation and mainly introduce conceptual ideas.

Qi et al. [11] introduced a structure which uses a constant N as loop counter, where N is given *beforehand* by experience. Nesting of loops is forbidden. This method is used to solve the lawn mower problem for an 8-by-8 square and N is set to 8. Such high degree of incorporated a priori knowledge prevents drawing any general conclusions from this work.

Pioneering work in terms of loops in GP has been done by Koza [4]. In this paragraph, we outline the contributions of two of his students: A new syntax in which conditional loops and alternatives were used was established by Finkel [3]. She solved the factoring problem with this approach and applied a penalty in the fitness to keep the programs small. Lai [5] introduced a new method to solve the greatest common divisor (GCD) problem which we discuss in Section 4.3. 12 fitness cases were used and when number of hits was 12, additional test cases are used to test the success of the program. Lai [5] states that his results lack of generality because of the small number of test cases he used. Both, Finkel and Lai, use relatively large populations, up to 5000 individuals. With such large populations, high success rates could be achieved by sheer chance. Thus, in our experiments, we compare the performance of Genetic Programming also with random walks.

Ciesielski and Li [2] applied two forms of `for`-loop structures to solve a modified Santa Fe trail problem and a sorting task. We use a very similar loop structure, the *counter loop* `CL` (see Section 3), in our experiments. In [6], they used a similar approach to the visit-every-square problem and a modified Santa Fe trail task and showed that in these problems, loop structure are advantageous and lead to a reduction of program size. Both problems are, however, static by nature and cannot be compared with algorithm tasks with changing inputs (where issues such as overfitting arise).

Chen and Zhang [1] used a loop which is similar to our *while loop* (see Section 3) to solve the factorial problem which we discuss in Section 4.4 for the inputs from 1 to 15. Their GP method could find a perfect solution in 25 out of the 50 runs. However, the instruction set used was very small so such success rates maybe do not bear too much evidence and cannot be generalized.

Wijesinghe and Ciesielski [20] investigated how indexed loops can be implemented in Genetic Programming. He used them to train programs to produce regular patterns in bit strings but no results from this work are available.

In the past, various loop structures have been introduced and applied to several problems. Yet, no data is available on the comparison of different kinds of loop structures on different benchmark problems. Most works used much beforehand knowledge and utilized it to construct specialized loop structures. In our work, we use instruction sets which are not tailored for the problems we apply GP to, test them on different problems, and compare loop structures in a realistic way.

3 Loop Representations

For our experiments, we use tree-based Standard Genetic Programming with memory. The inner nodes of a (program) tree are expressions and the leaf nodes are terminals. We utilize the instruction set defined in Table 1 and combine it with one of the following five loop structures: the counter loop (`CL`), the memory loop (`ML`), the while loop (`wL`), conditional assignments (`CA`), and the implicit loop (`IL`).

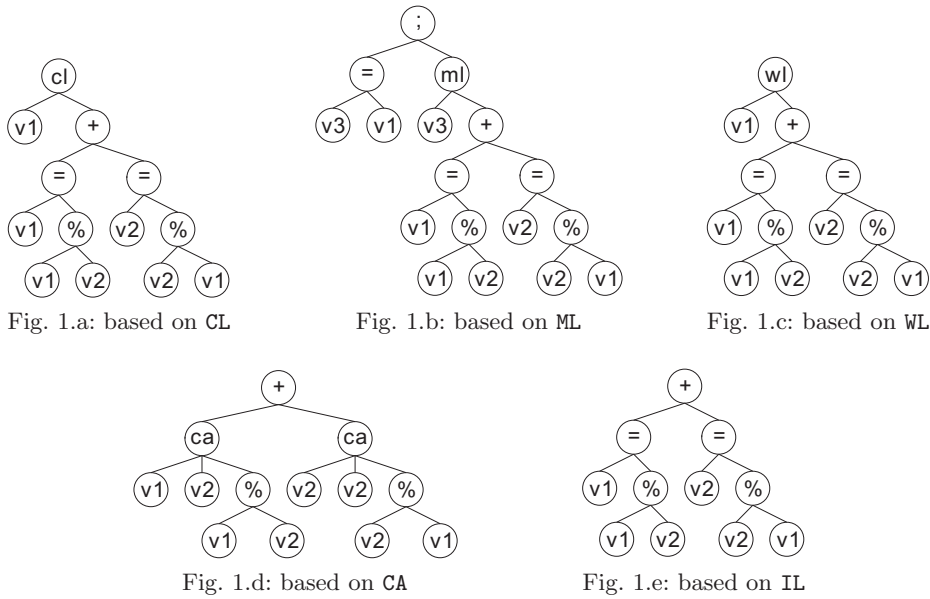


Figure 1: Ideal solutions to the GCD problem

The nodes which represent a *counter loop* (CL) have two children, say x and y , each of which is a subtree. x denotes an expression which, if evaluated, returns the number of times the loop body y should be executed. Fig. 1.a is an example for a program computing the GCD of two numbers in this representation.

Memory loop nodes ML also have two children x and y . x is a terminal node which identifies a variable and y is a subtree representing the loop body. The loop body (which can access and modify the variable x) is executed until x becomes less or equal 0. The variable x is also decreased by one in each loop iteration. An example of this representation is given in Fig. 1.b.

The *while loop* WL has the same structure as CL. Here, however, the subtree x represents a condition. The loop body subtree y is repeatedly evaluated until this condition becomes 0. Fig. 1.c illustrates a WL program for computing the GCD.

In our fourth loop representation, the new *conditional assignment* operator CA replaces the standard variable assignment =. Different from the = nodes used in the other program representations, CA has three children, x , y , and z . x is a condition expression subtree. y is a single terminal node identifying the target variable. The value of the expression subtree z will only be computed and assigned to the variable y if the condition x evaluates to a non-zero value. Like IL, the *complete* program tree will be evaluated repeatedly until no variable *changes*

anymore. It thus takes on the function of a loop body. Again, we give an example for this notation in Fig. 1.d.

Conditional assignments are different from the *Soft Assignment* method developed by McPhee and Poli [9, 10] where the strength of the assignment operator is weakened in order to allow for a more smooth change of the program fitness during the evolution. In the CA representation, $y=z$ means that the variable y will actually take on the exact value of the expression z , but the assignment will only be performed if $x \neq 0$.

The *implicit loop* IL has only a single child x . x is a subtree and represents the loop body. The loop body may contain variable assignments and will be evaluated until there is no change in any variable. We sketch an IL program for computing the GCD in Fig. 1.e.

Operator	Function
$+, -, *$	well-known arithmetic operators.
$a \% b$	protected modulo division, returns $a \bmod b$ if $b \neq 0$, and a otherwise
$a = b$	set the value of variable a to the result of the expression b (which is also returned)
$a; b$	concatenation of two expressions a and b , return value of b
v_1, v_2, \dots, v_n	the value of the one of n memory variables
$0, 1$	the only two ephemeral random constants available

Table 1: Basic operators in our experiments.

Different from the other three loop instructions, the novel loop representations (CA and IL) do not use a certain explicit expression (such as $x \leq 0$) to decide whether they should stop. Instead, they run until all variables have converged to stationary values. CA still retains some kind of explicit condition for the loop body, though decomposed into conditions for each assignment. In IL, *only* the side-effects determine the number of loop iterations.

4 Test Problems

We try to synthesize four different mathematical expressions, three of which can only be computed using a loop structure. For each problem, we use several training cases t_i . After executing an evolved program P , we expect its result $P(t_i)$ to be stored in its last memory cell v_n . We generally allowed the programs to utilize $n = 2$ two memory cells.

4.1 Polynomial Problem ($pr = 1$)

First, we propose a *trivial* polynomial problem $\phi_1(t_i) = t_i^3 + t_i^2 + 2 * t_i$ which does not require a loop in the instruction set at all. This test problem was designed to test whether the Genetic Programming system works correctly, can solve basic symbolic regression problem, and whether it utilizes some of the loops

even in cases where they are not strictly necessary. We use $tc = 100$ training cases $t_i \in 1 \dots 100$ and, at the beginning of each program execution, initialize all variables with t_i .

4.2 Sum Problem ($pr = 2$)

The second problem is to find the sum $\phi_2(t_i) = \sum_{j=1}^{t_i} j$ of the first t_i natural numbers. If a division operator is present in the instruction set, this problem can be solved without using loops. We omit the division operation from the instruction set, thus forcing Genetic Programming to synthesize suitable loops ($n(n+1)/2$ cannot be discovered with the available instructions). We again use $tc = 100$ training cases $t_i \in 1 \dots 100$ and, at the beginning of each program execution, initialize all variables with t_i except for the last one which is initialized with 0.

4.3 GCD Problem ($pr = 3$)

In the GCD problem [5], we try to find an algorithm which can compute the greatest common divisor $\phi_3(t_{i,1}, t_{i,2}) = \text{gcd}(t_{i,1}, t_{i,2})$. A training case $t_i = (t_{i,1}, t_{i,2})$ this time consists of the two natural numbers $t_{i,1}$ and $t_{i,2}$. We randomly create $tc = 100$ training cases at the beginning of each generation. At the beginning of the program execution, the first two variables v_1 and v_2 are initialized with $t_{i,1}$ and $t_{i,2}$. In the *memory loop* representation ML, each possible solution requires at least three memory cells so we allowed this representation to utilize $n = 3$ variables (initially $v_3 = 0$). In Figure 1, we sketch manually derived optimal solutions for the GCD task in each of the five program representations.

4.4 Factorial Problem ($pr = 4$)

The factorial problem [1], i.e., synthesizing a program which can compute $\phi_4(t_i) = t_i!$ of a natural number t_i , also requires at least one effective loop. We use $tc = 10$ training cases $t_i \in 1 \dots 10$ and, at the beginning of each program execution, initialize the first variable with t_i and the other one with 1.

5 Experiments

5.1 Experimental Protocol

In our experiments, we used the Genetic Programming implementation of the ECJ framework [7] 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. For each configuration, we performed 100 independent runs in order to get statistically reliable results.

We used three different objective functions f_1 to f_3 . f_1 , defined in Equation 1, is the error sum and subject to minimization. For a given problem pr , it sums up the absolute difference of the problem-specific expected result $\phi_{pr}(t_i)$ for i^{th} training case (t_i) and the result produced by the evolved program P which, as stated in Section 4, is expected to occur in the last variable v_n after the execution of P .

$$f_1(P) = \sum_{i=1}^{tc} |\phi_{pr}(t_i) - P(t_i)| \quad (1)$$

$$f_2(P) = |\{(i \in 1..tc) \wedge (\phi_{pr}(t_i) = P(t_i))\}| \quad (2)$$

$$f_3(P) = 1 \quad (3)$$

Equation 2 specifies the hit rate f_2 , an alternative for f_1 which counts the number of correctly solved training cases (and is subject to maximization). We also tested summing up the logarithms or the arc tangents of the error values in order to alleviate the different scales of the errors in different training cases (especially in the factorial problem). However, this did not lead to significantly different results and thus, was omitted here. The purpose of f_3 is to turn the Genetic Programming process into a parallel random walk by completely randomizing its (tournament) selection procedure. All other procedures such as crossover and mutation are left intact and retain their associated probabilities. Hence, we can fairly compare these runs with those using f_1 and f_2 . We will prepend a **R** to the names of the random walk versions of the experiments.

5.2 Results in Terms of Fitness

As to be expected, in the polynomial problem, all the GP approaches found the solutions quickly. The parallel random walks were not able to discover even a single solution. A comparison shows that there is no statistically significant difference between the representations and all configurations using f_1 or f_2 are significantly better than those with f_3 . Hence, the well-known utility of GP on symbolic regression tasks has been verified for the presented approaches and we will focus on the other three experiments in our evaluation.

In each of the sub-figures of Figure 2, we illustrate the statistical evaluation of the performance of the different loop representations in 100 independent runs using either f_1 or f_2 on one problem. An arrow $a \leftarrow b$ from box b to box a means that approach b is statistically significantly worse than approach a according to a two-tailed Mann-Whitney U test [8] with a significance level of 2% based on the best f_1 (or f_2) value encountered in each run. The arrows are transitive, meaning that an arrow from $a \leftarrow b \leftarrow c$ also means $a \leftarrow c$. All approaches in a stack of boxes yield the same results in the tests with other methods whereas the tests amongst each others are inconclusive.

We find that for almost all program representations and problems, Genetic Programming is significantly more efficient than random walks. This adds evidence to the claim that GP may indeed be able to construct non-trivial non-

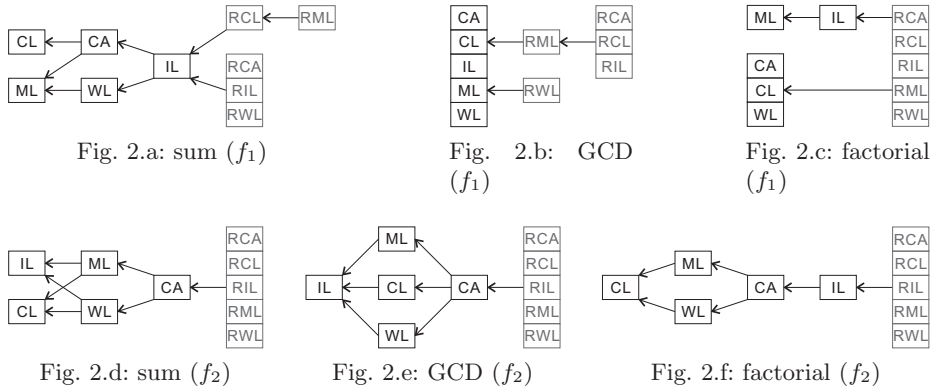


Figure 2: Statistical comparisons of the loop representations in different problems.

approximative algorithms. Apart from this basic fact, the loop representations do not exhibit any consistent significant differences values.

5.3 Results in Terms of Loop Effectiveness

Besides a comparison in terms of fitness, we are interested in how often Genetic Programming could solve a given problem under the viewpoint of effective utilization of the loop instructions. In this respect, each evolved program can be categorized in exactly one of the following groups:

1. **Category A.** There is no loop instruction in the program that influences the result in any way. An example of this case is sketched in Fig. 3.a.
2. **Category B.** At least one loop is present and is an effective loop (sketched in Fig. 3.b), i. e., its body is executed more than once and the result of the program depends on the number of executions. This can be verified by artificially limiting the maximum number of iterations to 1.
3. **Category C.** Otherwise, we check whether there is at least one loop which has degenerated to an alternative in the style of an `if-then` rule as illustrated in Fig. 3.c. This can be verified by always executing the loop exactly once and checking whether the results differ from the normal execution.
4. **Category D.** One endless loop is present which exceeds maximum number of allowed iterations (1000 in our experiments), as sketched in Fig. 3.d.

Notice that a loop may be effective or degenerated to an alternative but still can be endless at the same time. Categories **A** to **C** are exclusive, but programs of category **D** may additionally belong to any of the classes **A**, **B**, and **C**.

We now analyze the effectiveness of the loop representations in the different problems by categorizing the final results of each run into (at least) one of these four categories. In Table 2, we provide two numbers in the format “ s/c ” of each

```
v1∈1..10; //input
v2=5; //output (init)
while (v1>3) {
  v2=v2;
  v1--;
}
```

Fig. 3.a: Cat. A

```
v1∈1..10; //input
v2=5; //output (init)
while (v1<5) {
  v2++;
  v1++;
}
```

Fig. 3.b: Cat. B

```
v1∈1..10; //input
v2=5; //output (init)
while (v1<5) {
  v2=3;
  v1++;
}
```

Fig. 3.c: Cat. C

```
v1∈1..10; //input
v2=5; //output (init)
while (v1>0) {
  v2=3;
  v1++;
}
```

Fig. 3.d: Cat. D

Figure 3: Examples for different degrees of loop effectivity.

configuration and category, where c is the total number of programs of the given category returned by GP and s is the number of evolved programs of the category that actually solved the task perfectly.

Problem	A	B	C	D	Problem	A	B	C	D	
Polynomial ($pr = 1$)	CL f_1	60/60	0/0	40/40	0/0	CL f_1	0/2	0/97	0/1	0/96
	CL f_2	33/72	1/28	0/0	1/25	CL f_2	0/53	1/47	0/0	0/42
	RCL f_3	0/100	0/0	0/0	0/0	RCL f_3	0/99	0/1	0/0	0/0
	ML f_1	90/90	10/10	0/0	0/0	ML f_1	0/4	26/96	0/0	0/64
	ML f_2	22/76	3/24	0/0	0/9	ML f_2	0/46	27/54	0/0	0/14
	RML f_3	0/91	0/9	0/0	0/0	RML f_3	0/95	0/5	0/0	0/0
	WL f_1	92/90	0/1	0/0	0/1	WL f_1	0/77	0/22	0/1	0/23
	WL f_2	91/91	0/9	0/0	0/9	WL f_2	0/85	0/15	0/0	0/15
	RWL f_3	0/99	0/1	0/0	0/1	RWL f_3	0/98	0/2	0/0	0/2
	CA f_1	0/0	100/100	0/0	0/0	CA f_1	0/0	0/100	0/0	0/0
	CA f_2	59/59	41/41	0/0	0/0	CA f_2	0/16	1/84	0/0	0/0
	RCA f_3	0/69	0/31	0/0	0/1	RCA f_3	0/70	0/30	0/0	0/5
	IL f_1	56/56	44/44	0/0	13/13	IL f_1	0/5	0/95	0/0	0/0
	IL f_2	100/100	0/0	0/0	0/0	IL f_2	0/26	0/74	0/0	0/1
	RIL f_3	0/62	0/38	0/0	0/5	RIL f_3	0/70	0/30	0/0	0/5
GCD ($pr = 3$)	CL f_1	0/29	0/46	0/25	0/46	CL f_1	0/2	0/92	0/6	0/90
	CL f_2	29/83	0/17	0/0	0/17	CL f_2	0/35	0/45	0/0	0/39
	RCL f_3	0/83	0/17	0/0	0/17	RCL f_3	0/99	0/1	0/0	0/0
	ML f_1	0/26	0/74	0/0	0/74	ML f_1	0/25	14/75	0/0	0/45
	ML f_2	29/89	1/11	0/0	1/11	ML f_2	0/37	0/63	0/0	0/40
	RML f_3	0/93	0/7	0/0	0/7	RML f_3	0/92	0/8	0/0	0/0
	WL f_1	0/31	0/69	0/0	0/69	WL f_1	0/77	0/23	0/0	0/23
	WL f_2	30/85	5/15	0/0	0/10	WL f_2	0/94	0/6	0/0	0/6
	RWL f_3	0/97	0/3	0/0	0/3	RWL f_3	0/100	0/0	0/0	0/0
	CA f_1	0/0	9/100	0/0	0/7	CA f_1	0/0	0/100	0/0	0/1
	CA f_2	0/0	7/100	0/0	0/7	CA f_2	0/14	0/86	0/0	0/1
	RCA f_3	0/62	0/38	0/0	0/2	RCA f_3	0/63	0/37	0/0	0/3
	IL f_1	0/0	1/100	0/0	0/13	IL f_1	0/13	0/87	0/0	0/0
	IL f_2	0/1	86/99	0/0	0/0	IL f_2	0/72	0/28	0/0	0/1
	RIL f_3	0/54	0/46	0/0	0/2	RIL f_3	0/59	0/41	0/0	0/4
Factorial ($pr = 4$)	CL f_1	0/29	0/46	0/25	0/46	CL f_1	0/2	0/92	0/6	0/90
	CL f_2	29/83	0/17	0/0	0/17	CL f_2	0/35	0/45	0/0	0/39
	RCL f_3	0/83	0/17	0/0	0/17	RCL f_3	0/99	0/1	0/0	0/0
	ML f_1	0/26	0/74	0/0	0/74	ML f_1	0/25	14/75	0/0	0/45
	ML f_2	29/89	1/11	0/0	1/11	ML f_2	0/37	0/63	0/0	0/40
	RML f_3	0/93	0/7	0/0	0/7	RML f_3	0/92	0/8	0/0	0/0
	WL f_1	0/31	0/69	0/0	0/69	WL f_1	0/77	0/23	0/0	0/23
	WL f_2	30/85	5/15	0/0	0/10	WL f_2	0/94	0/6	0/0	0/6
	RWL f_3	0/97	0/3	0/0	0/3	RWL f_3	0/100	0/0	0/0	0/0
	CA f_1	0/0	9/100	0/0	0/7	CA f_1	0/0	0/100	0/0	0/1
	CA f_2	0/0	7/100	0/0	0/7	CA f_2	0/14	0/86	0/0	0/1
	RCA f_3	0/62	0/38	0/0	0/2	RCA f_3	0/63	0/37	0/0	0/3
	IL f_1	0/0	1/100	0/0	0/13	IL f_1	0/13	0/87	0/0	0/0
	IL f_2	0/1	86/99	0/0	0/0	IL f_2	0/72	0/28	0/0	0/1
	RIL f_3	0/54	0/46	0/0	0/2	RIL f_3	0/59	0/41	0/0	0/4

Table 2: Evolved programs according to categories

What we can find in Table 2 is that the three non-trivial problems *cannot* be solved efficiently by Genetic Programming by utilizing the given population size and generation limit. Only the *memory loop* ML can solve the *sum* and *factorial* problem to some extent. This may be due to the fact that this representation is especially suitable for the underlying structure of these two problems.

The *GCD* problem was repeatedly solved especially by IL under f_2 , but also by CA, and the WL approach using effective loops (cat. **B**). Most approaches here were able to find solutions without loops (cat. **A**) by creating sufficiently many modulo divisions and variable assignments in one program.

One interesting finding is Genetic Programming using f_1 and f_2 utilized the loop structures (created programs of category **B**) much more frequently than the random walks. We conclude that the reason for the better performance of

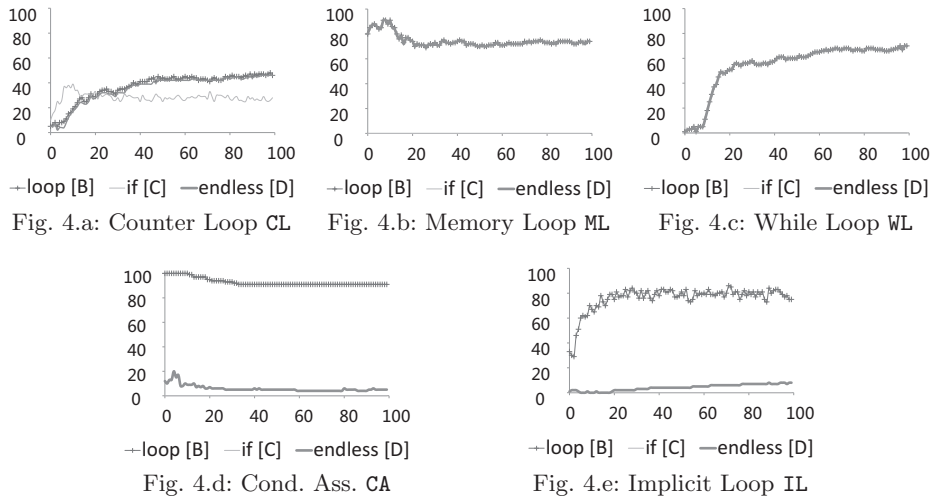


Figure 4: The development of loop utilization under f_2 for the GCD problem.

GP in comparison with random walks is that it is able to discover the benefits of loops.

Even more interesting is that the new loop instructions **CA** and **IL**, which both replace explicit stopping criteria by side-effects, were utilized much more frequently than the traditional loop instructions. Also, the number of endless loops they produce is much lower. Although we could not find a consistent better performance, we find that it seems to be easier to discover and adapt the **CA** and **IL** structures for Genetic Programming. Together with the finding that the better performance of GP compared to random walks is rooted in its utilization of loops and that **IL** and **CA** can most effectively solve the *GCD* problem, this gives a clear indication that the convergence-based termination has a high utility. Yet, the simple memory loop **ML**, too, is very efficient.

5.4 The Evolution of the Loop Utilization

In Figure 4, we illustrate the fractions of the programs in the population that belong to the different categories for the GCD problem. Similar diagrams for the other two non-trivial problems look basically the same and have been omitted here. It can be seen that in **CL**, **ML**, and **WL**, the fraction of programs with endless loops (category **D**) is almost identical with the programs that have effective loops (category **B**). In other words, these approaches are able to find loops that somehow compute good results, but they are not able to discover suitable loop stopping criteria. In **CA** and **IL**, on the other hand, the fraction of category-**B** programs is always much higher than the proportion of category-**D** programs throughout the evolution (see also Table 2). It should be noted that runs that

discover a solution before the 100 generations are elapsed will be terminated earlier, hence the number of programs with effective loops in the CA diagram goes down a bit after a few generations.

6 Conclusions

The goal of the research presented here was to (1) evaluate the capability of Genetic Programming approaches for synthesizing non-approximative, non-trivial, and deterministic mathematical algorithms with integer-valued results which involve loops and to (2) test whether novel loop structures with implicit stopping criteria can lead to better performance. We utilized five different program representations, two of which (CA and IL) possess features different from traditional loop instructions, and applied them to one basic and three non-trivial tasks.

We find that, on one hand, GP makes efficient use of the information provided by the objective functions, i. e., usually outperforms random walks. On the other hand, for non-trivial algorithm synthesis problems, Genetic Programming rarely finds suitable solutions. We thus have verified the “GP hardness” of the *sum*, *GCD*, and *factorial* problem on the presented instruction set and under the documented conditions.

We thus found both, evidence for the utility of GP and also evidence for the necessity of further research. It can be considered as a given fact that imperative Standard Genetic Programming in its current form does not perform well in the evolution of non-trivial and non-approximative algorithms. The positive impression promoted by the related work section could not be verified in our work here (under the parameters of experiments and validation).

With our experiments, we showed that implicit stopping criteria for loops, based on convergence of variables rather than explicit expressions, can much more easily be utilized by Genetic Programming. These representations are likely to have less epistatic effects since the stopping criteria do not “have to access” any variable. Hence, our future research will concentrate on further verifying low-epistatic control structures.

Acknowledgements. This work has been supported by China Postdoctoral Science Foundation Grant Number 20100470843.

References

- [1] G. Chen and M. Zhang. Evolving while-loop structures in genetic programming for factorial and ant problems. In *18th Australian Joint Conference on Artificial Intelligence*, pages 1079–1085, Sydney, Australia, 2005.
- [2] V. Ciesielski and X. Li. Experiments with explicit for-loops in genetic programming. In *IEEE Congress on Evolutionary Computation*, volume 1, pages 494–501, Portland, OR, USA, 2004.

- [3] J. R. Finkel. Using genetic programming to evolve an algorithm for factoring numbers. In *Genetic Algorithms and Genetic Programming at Stanford*, pages 52–60. Stanford Bookstore, Stanford, USA, 2003.
- [4] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
- [5] T. Lai. Discovery of understandable math formulas using genetic programming. In *Genetic Algorithms and Genetic Programming at Stanford*, pages 118–127. Stanford Bookstore, Stanford, CA, USA, 2003.
- [6] X. Li and V. Ciesielski. An analysis of explicit loops in genetic programming. In *IEEE Congress on Evolutionary Computation*, pages 2522–2529, Edinburgh, UK, 2005.
- [7] S. Luke et al. *ECJ: A Java-based Evolutionary Computation Research System*. George Mason University, Fairfax, VA, USA, 2006.
- [8] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.
- [9] N. F. McPhee and R. Poli. Memory with memory: Soft assignment in genetic programming. In *Genetic and Evolutionary Computation Conference*, pages 1235–1242, Atlanta, GA, USA, 2008.
- [10] R. Poli, N. F. McPhee, L. Citi, and E. Crane. Memory with memory in genetic programming. *Journal of Artificial Evolution and Applications*, (570606), 2009.
- [11] Y. Qi, B. Wang, and L. Kang. Genetic programming with simple loops. *Journal of Computer Science and Technology*, 14(4):429–433, 1999.
- [12] A. Teller. Genetic programming, indexed memory, the halting problem, and other curiosities. In *7th Florida Artificial Intelligence Research Symposium*, pages 270–274, Pensacola Beach, FL, USA, 1994.
- [13] A. Teller. Turing completeness in the language of genetic programming with indexed memory. In *1st IEEE Conference on Evolutionary Computation*, pages 136–141, Orlando, FL, USA, 1994.
- [14] T. Weise. *Evolving Distributed Algorithms with Genetic Programming*. PhD thesis, University of Kassel, Kassel, Germany, 2009.
- [15] T. Weise. *Global Optimization Algorithms – Theory and Application*. 2009. URL <http://www.it-weise.de/>.
- [16] T. Weise and R. Chiong. Evolutionary approaches and their applications to distributed systems. In *Intelligent Systems for Automated Learning and Adaptation: Emerging Trends and Applications*, pages 114–149. 2009.
- [17] T. Weise and K. Tang. Evolving distributed algorithms with genetic programming. *IEEE Transactions on Evolutionary Computation*, 2010. accepted for publication.
- [18] T. Weise, M. Zapf, and K. Geihs. Rule-based genetic programming. In *2nd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems*, pages 8–15, Budapest, Hungary, 2007.
- [19] T. Weise, M. Zapf, R. Chiong, and A. J. Nebro Urbaneja. Why is optimization difficult? In *Nature-Inspired Algorithms for Optimisation*, pages 1–50. Springer, Berlin, Germany, 2009.
- [20] G. Wijesinghe and V. Ciesielski. Experiments with indexed for-loops in genetic programming. In *Genetic and Evolutionary Computation Conference*, pages 1347–1348, Atlanta, GA, USA, 2008.

Preview

This document is a preview version
and not necessarily identical with
the original.

<http://www.it-weise.de/>

```
@inproceedings{WWT2011NLSATEOMA,  
  author    = {Mingxu Wan and Thomas Weise and Ke Tang},  
  title     = {Novel Loop Structures and the Evolution of Mathematical Algorithms},  
  booktitle = {Proceedings of the 14th European Conference on Genetic Programming  
              (EuroGP'11)},  
  month     = apr # {27--29},  
  year      = {2011},  
  location  = {Torino, Italy},  
  publisher = {Springer-Verlag GmbH: Berlin, Germany},  
  series    = {Lecture Notes in Computer Science (LNCS)},  
  volume    = {6621},  
  pages     = {49--60},  
  doi       = {10.1007/978-3-642-20407-4_5},  
  url       = {http://www.it-weise.de/documents/files/WWT2011NLSATEOMA.pdf},  
}
```