DOI: http://dx.doi.org/10.7551/978-0-262-31709-2-ch102



# Analyzing Program Evolution in Genetic Programming using Asynchronous Evaluation

Tomohiro Harada[1,2] and Keiki Takadama[1]
[1]The University of Electro-Communications, Japan
[2]Research Fellow of the Japan Society for the Promotion of Science DC
harada@cas.hc.uec.ac.jp

**Abstract**

This paper investigates the evolution ability of Tierra-based Asynchronous Genetic Programming (TAGP) as GP using an *asynchronous* evaluation. We compare TAGP with two simple GP methods, steady-state GP and GP using $(\mu + \lambda)$-selection as GP using a *synchronous* evaluation. Three GP methods are compared in experiment to minimize the size of an actual assembly language program in several computational problems, two arithmetic and two boolean problems. The intensive comparisons have revealed the following implications: (1) TAGP has higher evolution ability than GP using synchronous evaluation, *i.e.*, TAGP can evolve smaller size programs which cannot be evolved by GPs using synchronous evaluation; and (2) the diversity of the programs evolved by TAGP can derive a high evolution ability in comparison with GP using synchronous evaluation.

## Introduction

*Evolutionary Algorithms (EAs)* like *Genetic Algorithm (GA)* (Goldberg, 1989) and *Genetic Programming (GP)* (Koza, 1992) requires the appropriate diversity of a population to evolve solutions efficiently. As approaches to such diversity, the conventional EAs focused on the genetic operators or selection strategies. For example, the adaptive parameter setting methods have been proposed to control a probability of the genetic operators depending on the fitness in the population (Subbu et al., 1998; Yun and Gen, 2003; Lin and Gen, 2009), or NSGA-II (Deb et al., 2002) which is well known multi-objective evolutionary algorithm (MOEA) employs an idea of the crowding distance which is based on the distance of solutions to maintain the diversity of solutions. Furthermore, EAs such as *Differential Evolution (DE)* (Storn and Price, 1997) and *MOEA/D* (Zhang and Li, 2007) which have recently attracted much attention on have a high evolution ability by evolving solutions independently, which contributes to maintaining appropriate diversity of solutions. Such an independent evolution is based on the *asynchronously* evaluation approach which evolves solutions asynchronously unlike the conventional EA approach which is based on the *synchronous* evaluation approach which evolves solutions *synchronously*, *i.e.*, solutions are evolved by genetic operators after all individuals are evaluated. One main advantage of such *asynchronous evaluation* in EAs is to be able to derive the diversity of a population without any special heuristic operation.

This paper aims at verifying the evolution ability of the asynchronous evaluation on the program evolution. Since the previous asynchronous approaches such as DE and MOEA/D cannot be easily applied to the program evolution, this paper employs a novel GP method using the asynchronous evaluation, named as *Tierra-based Asynchronous Genetic Programming (TAGP)* (Nonami and Takadama, 2007; Harada et al., 2010, 2011) the previous researches proposed. TAGP is based on the idea of a biological evolution simulator, Tierra (Ray, 1991) and asynchronously evaluates and evolves programs. Since TAGP has the same advantage of the other EAs using asynchronous evaluation such as DE and MOEA/D, TAGP has a potential of evolving programs efficiently by maintaining the appropriate diversity of a population. To investigate such as evolution ability of TAGP, this paper compares TAGP as GP using the asynchronous evaluation, with two simple GP methods, steady-state GP (SSGP) (Reynolds, 1993) and GP using $(\mu + \lambda)$-selection ($(\mu + \lambda)$-GP) as GP using the synchronous evaluation. The experiment applies these three GP methods to several computational problems to minimize the size of an actual assembly language program.

In the following section, we firstly explain a biological evolution simulator, Tierra, which ideas are employed in TAGP, and explains the algorithm of TAGP. Then we compares TAGP with SSGP and $(\mu + \lambda)$-GP in several computational problems, and gives its result and detailed analyses. Next section discusses the difference of three GP methods, and this paper finally gives conclusions and future works.

## Tierra

Tierra (Ray, 1991) proposed by T. S. Ray is a biological evolution simulator, where digital creatures are evolved through a cycle of a self-reproduction, deletion and genetic operators such as a crossover or a mutation. Digital creatures live

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/713/1901563/978-0-262-31709-2-ch102.pdf by guest on 29 September 2023

in a memory space corresponding to the nature land on the earth, and they are implemented by a linear structured computer program such as the assembly language to reproduce (copy) themselves to a vacant memory space. CPU time corresponding to energy like actual creatures is given to each creature, and they execute instructions of a self-reproduction program within allocated CPU time. Since given CPU time is shorter for execution time of programs, all programs are executed in parallel. Lifespan of a program is decided with a *reaper* mechanism. All programs are arranged in a queue, named as *reaper queue*, and a reproduced program is added to the end of the reaper queue. While program execution, a program that can correctly execute its instruction moves its position in the reaper queue to lower, while one that cannot correctly execute its instruction moves its position to upper. Then, when a memory space is filled, a program that is at a top of the reaper queue is deleted from the memory. Due to the reaper mechanism, programs that cannot reproduce themselves within allocated CPU time or include some incorrect instructions are deleted from the memory, while creatures that can reproduce themselves propagate in the memory.

As results of such evolution, Tierra generates, for example, programs, called *parasite*, that reproduce themselves by using other program's instructions, or ones, called *hyper-parasite*, that have immunity to the *parasites*. Note that this evolution is not pre-programmed in Tierra but is caused by *emergence* (Langton, 1989). As the final stage of Tierra, programs that have shorter program size or have efficient algorithm are generated, that require less CPU time than an initial program to reproduce themselves (ATR Evolutionary Systems Department, 1998).

## Tierra-based Asynchronous Genetic Programming

### Overview

The previous researches focus on the feature of Tierra which can evolve programs, *i.e.*, digital creatures, with asynchronous execution, and have proposed a novel GP based on Tierra mechanism, named as *Tierra-based Asynchronous Genetic Programming (TAGP)* (Nonami and Takadama, 2007; Harada et al., 2010, 2011). To apply Tierra to evolving programs with a given task, the previous research introduces *fitness* commonly used in EAs to evaluate programs, and also introduces reproduction and deletion mechanisms depending on *fitness* into Tierra. This is because it is impossible to give any purposes to programs in Tierra whose purpose is only to reproduce themselves.

Fig. 1 shows an image of TAGP. TAGP firstly starts from a program that completely accomplishes the given task. Programs that consist of a linear structured instructions and some registers are stored in a limited memory space. Each program executes a small number of instructions, which is

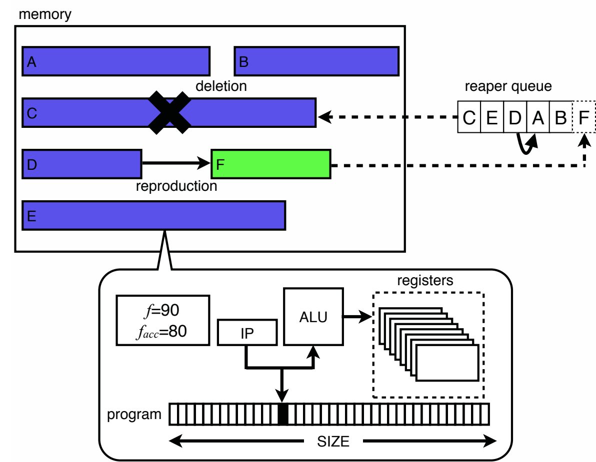



Figure 1: An image of TAGP

preconfigured, *e.g.*, three instructions, to simulate a parallel execution. All programs are arranged on *reaper queue* that controls lifespan of programs. When an execution of a program is finished, its fitness is evaluated depending on its execution result, and the reproduction and the reaper queue control are asynchronously conducted according to its fitness. When the memory is filled with programs, programs that are arranged at the upper of the reaper queue are removed from the memory.

### Algorithm

TAGP evolves programs through the following selection, reaper queue control, reproduction, and deletion algorithms. The algorithm of TAGP is shown in Algorithm 1 where 1, $prog.f_{acc}$ and $prog.f$ respectively indicate accumulated and evaluated fitness, and $rand(0,1)$ indicates random real value between 0 to 1. $prog_{prev}$ indicates a previously selected program, while $elite_{prev}$ indicates a previously selected *elite* program (detailed in below).

**Selection and reaper queue control** When an execution of one program is finished, its fitness is evaluated depending on its register value and the evaluated fitness is added to an *accumulated fitness* $prog.f_{acc}$ (the $1^{st}$ line in Algorithm 1). Based on $prog.f_{acc}$, whether a program is selected as a reproduction candidate or not is determined. Let represent the maximum fitness as $f_{max}$, if the accumulated fitness of a program exceeds $f_{max}$, it is selected as a reproduction candidate, and $f_{max}$ is subtracted from its accumulated fitness (the $2^{nd}$ and $3^{rd}$ lines). While if not, a program is not selected. Depending on this selection condition, a program that completely accomplishes the given task, *i.e.*, its fitness is equal to $f_{max}$, is invariably selected because the accumulated fitness always exceeds $f_{max}$. High fitness programs have a high potential to be selected because the accumulated

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/713/1901563/978-0-262-31709-2-ch102.pdf by guest on 29 September 2023

**Algorithm 1** The algorithm of TAGP

1: $prog.f_{acc} \leftarrow prog.f_{acc} + prog.f$
2: **if** $prog.f_{acc} \geq f_{max}$ **then**
3: $prog.f_{acc} \leftarrow prog.f_{acc} - f_{max}$
4: **repeat**
5: down reaper queue position
6: **until** $rand(0,1) < P_{down}(prog.f)$
7: reproduce better program of $prog$ and $prog_{prev}$ with genetic operators
8: $prog_{prev} \leftarrow prog$
9: **if** $prog.f = f_{max}$ **then**
10: **if** $prog$ is better than $elite_{prev}$ **then**
11: reproduce $prog$ without any genetic operators
12: **else**
13: reproduce $prog$ with genetic operators
14: **end if**
15: $elite_{prev} \leftarrow prog$
16: **end if**
17: **else**
18: **if** $rand(0,1) < rand(\frac{prog.f}{f_{max}}, 1)$ **then**
19: remove $prog$ from memory
20: **end if**
21: **repeat**
22: up reaper queue position
23: **until** $rand(0,1) > P_{up}(prog.f)$
24: **end if**

fitness frequently exceeds $f_{max}$, while low fitness ones are hard to satisfy this condition.

Then, a position in the reaper queue of a program that satisfies the selection condition becomes lower than the current one, *i.e.*, its deletion probability decreases (the $4^{th} \sim 6^{th}$ lines), while one that does not satisfies the condition becomes upper, *i.e.*, its deletion probability increases (the $21^{st} \sim 23^{rd}$ lines). The move distance is determined by the move rate represented as $P_{down}$ and $P_{up}$ which are calculated as the following equation based on fitness,

$$P_{down}(f) = \frac{f}{f_{max}} \times P_r, \; P_{up}(f) = \frac{f_{max} - f}{f_{max}} \times P_r, \quad (1)$$

where $P_r$ is the maximum probability of $P_{down}$ and $P_{down}$, which is preconfigured. Depending on these equations, higher fitness programs are arranged on lower position in the reaper queue, *i.e.*, survive long, while lower ones are arranged on upper, *i.e.*, are easily removed.

**Reproduction** Programs selected depending on the selection condition become a reproduction candidate. To reproduce better program asynchronously, TAGP only compares two programs, a currently selected program and a previously selected program described as $prog_{prev}$ in Algorithm 1 and better one is selected as a parent. A selected program generates an offspring with the genetic operator such as a crossover and a mutation, and an offspring is reproduced to a vacant memory space that is larger than its program size (the $7^{th} \sim 8^{th}$ lines). Additionally, the elite preserving strategy (Jong and Alan, 1975) is applied to preserve programs that can accomplish the given task (the $9^{th} \sim 16^{th}$ lines). If a current program is evaluated as $f_{max}$, it is compared with one that is previously evaluated as $f_{max}$ represented as $elite_{prev}$ in Algorithm 1. Then if the current one is better, it is reproduced as an elite program *without* the genetic operators to preserve better program, *i.e.*, generating a copy of the elite, while if not, it is reproduced *with* the genetic operators. TAGP employs four genetic operators, a crossover, a mutation, and an instruction insertion/deletion. The crossover operator combines a reproduced program with a previously selected parent. The mutation operator changes one random instruction in a reproduced program to other random instruction. The insertion operator inserts one random instruction into a reproduced program, while the deletion operator removes one instruction selected at random in a reproduced program.

**Deletion** TAGP conducts two deletion. One is a deletion based on the reaper queue that is conducted during the reproduction process. If a vacant memory space is not found when reproducing an offspring, programs that is arranged upper in the reaper queue are removed until a total vacant memory space becomes greater than a certain threshold, *e.g.*, usually set as 20% of the memory. This deletion remove elder and lower fitness program.

While another deletion is a *natural death* which is based on the idea of *sugarscape* (Epstein and Axtell, 1996). The natural death applied to programs that do not satisfied the selection condition according to the $18^{th} \sim 20^{th}$ line in Algorithm 1, where $rand(a, 1)$ indicates random real value between $a(\leq 1)$ to 1. This deletion removes lower fitness programs even if the memory is not filled.

## Experiment

To validate the evolution ability of TAGP, this paper compares TAGP with two simple GP methods, steady-state GP (SSGP) (Reynolds, 1993) and GP using $(\mu + \lambda)$-selection ($(\mu+\lambda)$-GP), with four computational problems. SSGP and $(\mu + \lambda)$-GP are hereinafter collectively called SGPs (synchronous GPs). SSGP selects two parents from the population and generates two offspring, then the worst two programs in the population are replaced with generated offspring. $(\mu + \lambda)$-GP generates $\lambda$ offspring from the population consisting of $\mu$ programs, and leaves better $\mu$ programs from $(\mu + \lambda)$ programs to next generation, where $\mu = \lambda$ in this experiment.

### Computational problems

This paper applies three GP methods to the following four computational problems shown in Table 1.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/713/1901563/978-0-262-31709-2-ch102.pdf by guest on 29 September 2023

Table 1: Computational problems

| Problem type | Function | # of training data |
|---|---|---|
| Arithmetic1 | $x^4 + x^3 + x^2 + x$ | 16 |
| Arithmetic2 | $x^y$ | 25 |
| Boolean2 | 6-multiplexer | 64 |
| Boolean2 | 5bits digital adder | 32 |

Fitness is evaluated as

$$fitness = f_{max} - \frac{1}{n} \sum_{i=1}^{n} |\hat{y_i} - y_i|, \quad (2)$$

where $f_{max}$ indicates the maximum fitness, $n$ indicates the number of the training data, $\hat{y_i}$ indicates the function value calculated from the $i^{th}$ training data, while $y_i$ indicates the output of a program in respect to the $i^{th}$ input value. Note that when comparing the same fitness programs, the program size is firstly compared, and if is also equal to, the number of executed instructions to calculate all training data is finally compared.

This paper employs a program written by an actual assembly language embedded on PIC16 micro-controller unit (Microchip Technology Inc., 2007) developed by Microchip Technology Inc.. This is 12bits word assembly language, and has 33 simple instructions that consist of add-subtract, logical, bit, and branch instructions, but not contain a multiplication. One program can use 16 general 32-bits registers, named as $R0$ to $R15$, and one temporary 32-bits register, named as $W$, while its size is limited to 256 instructions. The input value is firstly set from $R1$ register, and other registers are initialized as 0. Note that since this instruction set does not include a multiplication instruction, programs have to combine some instructions and loop structures to calculate multiplication. Therefore, to calculate both of regression problems, programs have to include loop structures. Concretely, Arithmetic1 includes three multiplications ($x^2$, $x^3$, and $x^4$), while Arithmetic2 includes double multiplications loop to calculate $x^y$ in both initial programs.

The experiment starts from an initial program that can completely solve the given task, and compares evolution ability of three GP methods by observing how small program can be obtained in one hour.

## Parameter settings

Common parameter settings for all three GP methods is shown in Table 2, ones for only SGPs is shown in Table 3, and ones for only TAGP is shown in Table 4. Three GP methods employ the same genetic operators, crossover, mutation, and instruction insertion and deletion, and also employ the same parameters for the crossover, mutation, insertion, and deletion rate. Only one genetic operator is executed with the configured probability for each reproduction. Crossover method is two point crossover, and the maximum fitness ($f_{max}$) is set as 100. In SGPs, the binary tournament selection is employed, while the upper execution steps are restricted to 50,000 and if execution steps exceeds, its fitness becomes the minimum value. In TAGP, the deletion removes programs until the total vacant memory exceeds 20% of the memory size, while $P_r$ is set as 0.9, which is the maximum probability of $P_{down}$ and $P_{up}$.

Table 2: Common parameter settings

| Parameter | value |
|---|---|
| Crossover rate | 0.7 |
| Mutation rate | 0.1 |
| Insertion rate | 0.1 |
| Deletion rate | 0.1 |
| Crossover method | Two point crossover |
| $f_{max}$ | 100 |

Table 3: Parameter settings of SSGP and $(\mu + \lambda)$-GP

| Parameter | value |
|---|---|
| Selection | Binary tournament |
| Upper execution steps | 50000 |

Table 4: Parameter settings of TAGP

| Parameter | value |
|---|---|
| Removing threshold | 20% of memory |
| $P_r$ | 0.9 |

Table 5: Memory size of TAGP and population size of SSGP and $(\mu + \lambda)$-GP in each problem

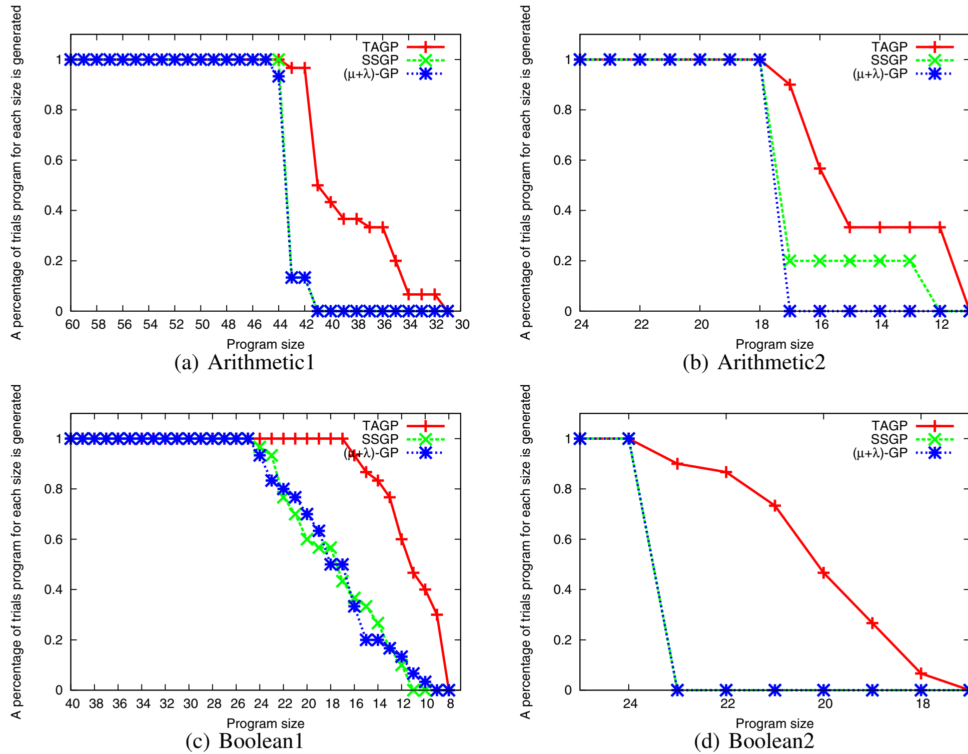
| Problem type | TAGP | SSGP | $(\mu + \lambda)$-GP |
|---|---|---|---|
| Arithmetic1 | 6400 | 100 | 100 |
| Arithmetic2 | 6400 | 200 | 50 |
| Boolean1 | 25600 | 200 | 200 |
| Boolean2 | 6400 | 50 | 200 |

Settings of population size of SGPs and memory size of TAGP is determined based on pre-experiment. SGPs compare results of 50, 100, and 200, while TAGP compares results of 6400, 12800, and 25600. Note that since the maximum program size is configured as 256 instructions, when the population size is set as 100, a used memory space of SGPs is equal to the memory size $25600 (= 256$ (instructions)$\times 100$ (individuals)) of TAGP.

## Result

The experiment conducts 30 trials for three GP methods, and we evaluate three GP methods based on a percentage of trials in which a program for each size can be generated in one hour. Fig. 2 shows the result of a percentage of trials in which a program for each size can be generated in one hour. In Fig. 2, the abscissa indicates the program size, while the ordinate indicates the percentage of trials which can generate a program for each size. The red lines show the result

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/713/1901563/978-0-262-31709-2-ch102.pdf by guest on 29 September 2023




(a) Arithmetic1

(b) Arithmetic2

(c) Boolean1

(d) Boolean2

Figure 2: A percentage of trials in which a program for each size is generated in one hour

of TAGP, the green lines show the result of SSGP, while the blue lines show the result of $(\mu + \lambda)$-GP.

From Fig. 2, it is revealed that, in all computational problems, TAGP can generate smaller size programs than both of SGPs. Mann-Whitney U test is used to compare the generated program size of all GPs in all problems. The level of significance is set at $\alpha = 0.05$, and significant differences between TAGP and SGPs are verified. Focusing on the program size finally generated with each GP, programs of a certain size are generated with all GPs, concretely 44, 18, and 24 in Arithmetic1, Arithmetic2, and both of boolean problems respectively. It is, however, hard for SSGP and $(\mu + \lambda)$-GP to generate programs that size is smaller than the size described above. While TAGP can generate such programs in most trials.

To clarify the reason why TAGP outperforms SGPs, in the following, we analyze evolution processes observed in experiments. The evolution processes falls into two main categories, *non-destructive evolution* and *destructive evolution*. *Non-destructive evolution* means that it does not affect calculation result, *i.e.*, does not decrease fitness of a program, during evolution process, and is easily achieved. While *destructive evolution* means that it is required to decrease fitness of a program or to increase program size during evolution process. Non-destructive evolution also falls into two categories, *single-step* and *multi-step*, the former means that it completes only single genetic operation, while the later means that it requires two or more genetic operations to complete its evolution process. Note that all destructive evolution is *multi-step* because it requires two or more genetic operations. The following sections describe details of these evolutions.

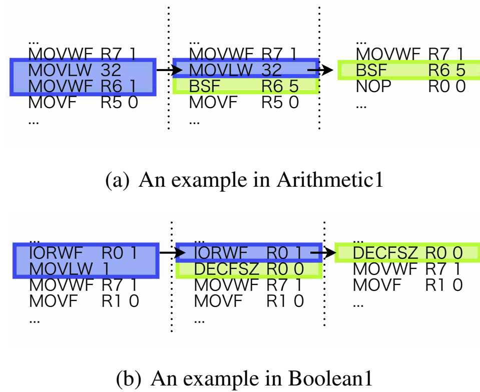
**Single-step non-destructive evolution** Single-step non-destructive evolution evolves programs by removing unnecessary instructions which do not affect the calculation result. As shown in Table 6, the problems except Boolean2 include some number of unnecessary instructions, and removing these instructions can decrease the program size. This evolution can be easily achieved because it can decreases the program size by not affecting the calculation result, *i.e.*, not decreasing fitness. Regarding Arithmetic1, Arithmetic2, and Boolean1, since programs that size is respectively 44, 18, and 36 are easily generated by only removing unnecessary instructions, all GPs accomplish 100% success rate, but one trial of $(\mu + \lambda)$-GP in Arithmetic1.

**Multi-step non-destructive evolution** Multi-step non-destructive evolution generates small size programs by replacing several instructions with one or a few instructions. Examples of observed evolutions are shown in Fig. 3. In

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/713/1901563/978-0-262-31709-2-ch102.pdf by guest on 29 September 2023

Table 6: The initial program size and the number of unnecessary instructions in each problem

| Problem type | initial program size | # of unnecessary instructions |
|---|---|---|
| Arithmetic1 | 62 | 18 |
| Arithmetic2 | 25 | 7 |
| Boolean1 | 41 | 5 |
| Boolean2 | 26 | 0 |




(a) An example in Arithmetic1

(b) An example in Boolean1

Figure 3: An example of multi-step non-destructive evolution

these figures, colored instructions are replaced with one instruction by using two genetic operations.

In Fig. 3, the left figure indicates a part of a program before evolutions, while the center and the right ones indicate the change of instructions. The colored instructions in the left figure substitutes a loop counter 32 to $R6$ register through the temporary register $W$. This process requires two instructions, $MOVLW$ which substitutes a literal to $W$ register and $MOVWF$ which substitutes the value of $W$ register to any register. These two instructions are replaced with $BSF$ which sets 1 to any one bit of any register. In this case $BSF$ sets 1 to $5^{th}$ bit of $R6$ register. This works just as well as the previous two instructions, *i.e.*, sets 32 to $R6$ register. While in Boolean1, a lot of examples are observed to replace logical instructions ($AND$, $OR$, and $XOR$) with conditional branch instructions. In example of Fig. 3(b), $OR$ instruction (represented as $IORWF$) is replaced with $DECFSZ$ instruction. $DECFSZ$ instruction decrements a register value and skip next instruction only if its result is equal to 0. Such replacement of logical instructions with branch instructions are often observed both of boolean problems, this is because most of boolean calculation can be calculated by using conditional branches.

Since multi-step non-destructive evolutions does not also affect the calculation result in their evolution process, it is possible to decrease the program size by sequential genetic operations. In particular, programs that size is smaller than 36 in Boolean1 and ones that size is 24 in Boolean2 are generated through this evolution.

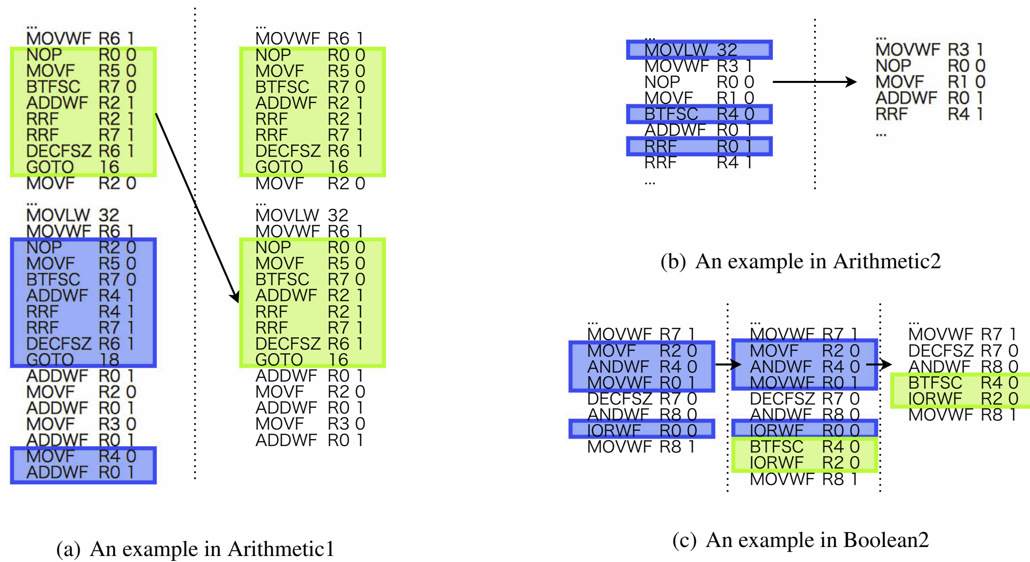
**Multi-step destructive evolution** Although multi-step destructive evolution also generates small size programs by replacing several instructions with one or a few instructions such as multi-step non-destructive evolution, the selection probability of the program decreases in the process of evolution, *i.e.*, fitness decreases or program size increases. It is difficult to achieve this evolution because programs that are in the process of evolution can be removed from the population before generating small size programs. Examples are shown in Fig. 4, where colored instructions have same meaning in previous figures.

In an example of Arithmetic1, shown in Fig 4(a), a loop structure that calculates $x^4$ with $R4$ register is overwritten by a loop to calculate $x^2$ with $R2$ register. This loop overwriting enables a program to simultaneously calculate $x^2 + x^4$ with $R2$ register, and two instructions, $MOVF$ and $ADDWF$ which calculate $R0 \leftarrow R2 + R4 (= x^2 + x^4)$, at the end of the program becomes unnecessary which can be removed from the program. An example of Arithmetic2 removes three instructions colored in Fig 4(b), which gets same calculation result in different calculation process. Concretely, although a program calculates $x^y$ with combination of bit shifting and adding before removing instructions, one after evolution calculates it with only adding. In an example of Boolean2, a program into which two instructions, $BTFSC$ and $IORWF$, is added has same calculation result as before adding them, and four instructions, $MOVF$, $ANDWF$, $MOVWF$, and $IORWF$, can be removed because they become unnecessary instructions due to added two instructions.

The common feature of these evolutions is that programs in the process of evolution has either incorrect calculation result or large program size. In Arithmetic1, since a program that only overwrites a loop structure includes two adding process, it does not correctly calculate result and has low fitness. In Arithmetic2, it is necessary to remove all of three instructions simultaneously, and if at least one instruction remains in the program, it cannot also calculate correct results. In Boolean2, if both of required two instructions are added, the program can correctly calculate result, however, its size increases. This feature results to decreases the selection probability of programs that are in the evolution process, and it becomes difficult to preserve such programs in a population. Therefore, it is indispensable to maintain the diversity of programs to achieve destructive evolution.

Multi-step destructive evolution is necessary to generate programs that size is smaller than 43, 17, 23 in Arithmetic1, Arithmetic2, and Boolean2, respectively. As results of Fig 2, although such evolution can be achieved in TAGP, SGPs can achieve in few trials. Particularly, SGPs never achieve

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/713/1901563/978-0-262-31709-2-ch102.pdf by guest on 29 September 2023




(a) An example in Arithmetic1

(b) An example in Arithmetic2

(c) An example in Boolean2

Figure 4: An example of multi-step destructive evolution

Table 7: Summary of program evolution analyses

| | non-destructive | destructive |
|---|---|---|
| single-step | all GPs | - |
| multi-step | all GPs<br>TAGP>SGPs | TAGP |

this evolution in Boolean2, this is because Boolean2 requires more than three steps to generate programs that size is smaller than 23.

These results summarized in Table 7 reveal that SGPs achieve single-step non-destructive evolution which is easier than other evolutions. They, additionally, achieve multi-step non-destructive evolution in some trials. In contrast, it is revealed that TAGP cannot only achieve multi-step non-destructive evolution at a higher rate than SGPs (particularly in Boolean1), but also achieve destructive evolution which cannot be achieved SGPs. This result indicates that TAGP has higher evolution ability than SSGP and $(\mu + \lambda)$-GP.

## Discussion: diversity of programs

From the analyses of the program evolution, it is revealed that the diversity of programs is required to achieve complex program evolution. To confirm the diversity of programs in three GP methods, we verify the relation between the average fitness and the standard deviation of the program size in the memory/population of Arithmetic1 when a program that size is 44 and does not include unnecessary instructions is generated. Fig. 5 shows the scatter plot of the average fitness and the standard deviation of the program size for all

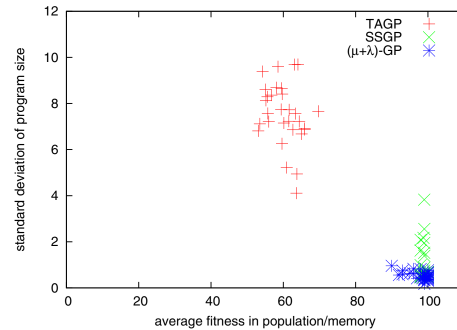



Figure 5: A scatter plot of the average fitness and the standard deviation of the program size in the memory/population of Arithmetic1

trials in Arithmetic1. In Fig. 5, the abscissa indicates the average fitness, while the ordinate indicates the standard deviation of the program size. The red points show the result of TAGP, the green points show the result of SSGP, while the blue points show the result of $(\mu+\lambda)$-GP. Note that the result of Arithmetic1 is only shown, but same trends are verified in other problems.

As shown in Fig. 5, it is indicated that both of SGPs have high average fitness and low standard deviation of the program size. This means that all programs in the population has very high fitness near the maximum, and also has similar program size. This result indicates that all programs are very similar to each other and the diversity of programs is very low. As mentioned in previous section, although the diversity of programs is required to generate a program that

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/713/1901563/978-0-262-31709-2-ch102.pdf by guest on 29 September 2023

size is less than 44 with either evolution cases, the diversities of SGPs are very low. From this result, it is revealed that it is difficult for SGPs to achieve multi-step destructive evolution because enough diversity of programs cannot be maintained.

While in TAGP, it is indicated that the average fitness is not maximum but is higher than 50,which is the half of the maximum fitness, and the standard deviation of the program size is also high. This indicates that TAGP can maintain lower fitness or larger size programs in the population, *i.e.*, several kind of programs are maintained. From this result, it is revealed that the high diversity of programs in TAGP contributes to achieve multi-step destructive evolution. This result advocates that TAGP have same feature of EAs using the asynchronous evaluation to maintain proper diversity of programs and to have high evolution ability.

## Conclusion

To investigate the evolution ability of TAGP as GP using the asynchronous evaluation, this paper compared TAGP with two simple GP methods, steady-state GP (SSGP) and GP using $(\mu + \lambda)$-selection ($(\mu + \lambda)$-GP) as GP using the synchronous evaluation. Intensive comparisons among three GP methods were conducted in four computational problems to minimize the size of an actual assembly language program.

We classify the evolution processes to two categories, *non-destructive evolution* and *destructive evolution* depending on whether the evolution process affects calculation results or not. The experimental result has revealed that the following implications: (1) TAGP has higher evolution ability than SSGP and $(\mu + \lambda)$-GP, *i.e.*, TAGP cannot only achieve *non-destructive evolution* which is easy to be accomplished, but also achieve *destructive evolution* which cannot be achieved by SSGP and $(\mu+\lambda)$-GP; and (2) the diversity of the programs in TAGP can derive a high evolution ability in comparison with SSGP and $(\mu + \lambda)$-GP. In detail, such diversity is indispensable to *destructive evolution*.

The following issues should be pursued in the near future: (1) experiments on other problems such as classification, (2) a comparison with other GP methods, (3) an improvement of evolution ability of TAGP, and (4) a parallelization of TAGP.

## Acknowledgements

This work was supported by Grant-in-Aid for JSPS Fellows Grant Number 249376.

## References

ATR Evolutionary Systems Department (1998). *Artificial Life and Evolutional System*. Tokyo Denki University Press.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on*, 6(2):182 –197.

Epstein, J. M. and Axtell, R. L. (1996). *Growing Artificial Societies: Social Science from the Bottom Up (Complex Adaptive Systems)*. The MIT Press, 1st printing edition.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1 edition.

Harada, T., Otani, M., Matsushima, H., Hattori, K., Sato, H., and Takadama, K. (2011). Robustness to Bit Inversion in Registers and Acceleration of Program Evolution in On-Board Computer. *Journal of Advanced Computational Intelligence and Interlligent Informatics (JACIII)*, 15(8):1175–1185.

Harada, T., Otani, M., Matsushima, H., Hattori, K., and Takadama, K. (2010). Evolving Complex Programs in Tierra-based On-Board Computer on UNITEC-1. In *International Astronautical Congress (IAC), 2010 61st World Congress on*.

Jong, D. and Alan, K. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, Department of Computer and Communications Sciences, University of Michigan.

Koza, J. (1992). *Genetic Programming On the Programming of Computers by Means of Natural Selection*. MIT Press.

Langton, C. G. (1989). *Artificial Life*. Addison-Wesley.

Lin, L. and Gen, M. (2009). Auto-tuning strategy for evolutionary algorithms: balancing between exploration and exploitation. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 13(2):157–168.

Microchip Technology Inc. (2007). *PIC10F200/202/204/206 Data Sheet 6-Pin, 8-bit Flash Microcontrollers*. Microchip Technology Inc.

Nonami, K. and Takadama, K. (2007). Tierra-based Space System for Robustness of Bit Inversion and Program Evolution. In *SICE, 2007 Annual Conference*, pages 1155–1160.

Ray, T. S. (1991). An approach to the synthesis of life. *Artificial Life II*, XI:371–408.

Reynolds, C. W. (1993). An evolved, vision-based behavioral model of coordinated group motion. In *Proc. 2nd International Conf. on Simulation of Adaptive Behavior*, pages 384–392. MIT Press.

Storn, R. and Price, K. (1997). Differential Evolution - A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *J. of Global Optimization*, 11(4):341–359.

Subbu, R., Sanderson, A. C., and Bonissone, P. P. (1998). Fuzzy Logic Controlled Genetic Algorithms versus Tuned Genetic Algorithms: An Agile Manufacturing Application. *Proceeding of the 1998 IEEE international symposium on intelligent control (ISIC)*, pages 434–440.

Yun, Y. and Gen, M. (2003). Performance Analysis of Adaptive Genetic Algorithms with Fuzzy Logic and Heuristics. *Fuzzy Optimization and Decision Making*, 2(2):161–175.

Zhang, Q. and Li, H. (2007). MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Trans. Evolutionary Computation*, 11(6):712–731.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/713/1901563/978-0-262-31709-2-ch102.pdf by guest on 29 September 2023