What is Needed to Promote an Asynchronous Program Evolution in Genetic Programing?

Keiki Takadama[1], Tomohiro Harada[1,2], Hiroyuki Sato[1], and Kiyohiko Hattori[1]

[1] The University of Electro-Communications, Japan
[2] Research Fellow of the Japan Society for the Promotion of Science DC
{keiki@inf,harada@cas.hc,sato@hc,hattori@inf}.uec.ac.jp
http://www.cas.hc.uec.ac.jp

Abstract. This paper focuses on an *asynchronous* program evolution in evolutionary computation, which is hard to evolve individuals (*i.e.*, programs) effectively unlike a *synchronous* program evolution that evolves individuals effectively by selecting good parents after evaluations of *all* individuals in each generation. To tackle this problem, we explore the mechanism that can promote an asynchronous program evolution by selecting a *good* individual without waiting for evaluations of *all* individuals. For this purpose, this paper investigates the effectiveness of the proposed mechanism in genetic programing (GP) domain by evaluating it in the two types of problems (*i.e.*, the arithmetic and Boolean problems). Through the intensive experiments of the eight kinds of testbeds under the two types of problems, the following implications have been revealed: (1) the program *asynchronously* evolved *with* the proposed mechanism can be completed with the shorter execution steps than the program *asynchronously* evolved *without* the proposed mechanism; and (2) the program *asynchronously* evolved *with* the proposed mechanism can be completed with mostly the same or shorter execution steps than the program *synchronously* evolved by the conventional GP.

Keywords: genetic programming, asynchronous evolution, Tierra

1 Introduction

The *synchronous* evolution, which evolves individuals (*i.e.*, solutions) through a comparison with all of them, is the core approach in the conventional Evolutionary Algorithms (EAs) such as Genetic Algorithm (GA) [4] and Genetic Programming (GP) [7]. This approach needs the evaluations of *all* individuals to select the parent as the *good* individuals and to delete the *bad* individuals for the next population generation. This requires to wait for the slowest evaluation of a certain individual, which increases the computational time. For this issue, the *asynchronous* evolution is recently attracted much attention on [9] [3], which evolves individuals independently, *i.e.*, it should not wait for the evaluations of other individuals. For example, Differential Evolution (DE) [14] and

MOEA/D [15] creates a child independently from one or a few other individual(s) to generate the next population unlike the synchronous evolution that creates children after evaluating all individuals.

What should be noted here, however, is that the asynchronous evolution is indispensable in the *program evolution* because of the following reasons: (1) the asynchronous program evolution has a potential of continuously evolving individuals (*i.e.*, programs) even if individuals cannot complete their evaluation (*e.g.*, due to an infinite loop), because it is not required to wait for the evaluation of *all* individuals, and (2) the asynchronous program evolution gives a lot of selection chance for the quickly evaluated individuals by evolving them immediately after completing their evaluations. However, the conventional asynchronous EAs such as DE and MOEA/D cannot be easily applied to the program evolution because they are not designed for the program evolution. To tackle this problem, our previous researches proposed *Tierra-based Asynchronous Genetic Programming (TAGP)* [5,6,11] as the asynchronous-based GP, which can asynchronously evolves the *programs* having the same advantage of the other asynchronous EAs. Concretely, TAGP employs the idea of a biological evolution, *Tierra* [12], where individuals are asynchronously evolved through a reproduction of individuals.

However, the current version of TAGP is hard to evolve the programs effectively. This is because TAGP cannot guarantee to select the *good* individuals as the parents from a population due to a lack of a comparison with other individuals, *i.e.*, TAGP selects the individuals which complete to execute their instructions but does not select them from the viewpoint of evaluation (fitness). To overcome this problem, this paper proposes the mechanism that selects the *good* individual without waiting for evaluations of *all* individuals. Concretely, the winner of the current and previous parents and the winner of the previous and two times previous parents are mated in the crossover operation. Such a minimum set of the tournament selection contributes to selecting good individual as the parent (Hereafter, we call this mechanism as *the temporal difference (TD) tournament selection*). Note that this mechanism cannot guarantee to select *best* individual, but the selection of a *good* individual has a great potential of promoting an *asynchronous* program evolution. To investigate the asynchronous evolution ability of TAGP with the proposed mechanism, this paper firstly compares the results of TAGP *with* the temporal difference tournament selection (TAGP with the TD tournament selection) and TAGP *without* it, and secondly compares the results of TAGP with the TD tournament selection as the asynchronous-based GP and the simple GP (steady-state GP) [13] as the synchronous-based GP. In the experiment, we employ the assembly language program as the machine-code program and investigate its evolution in the two types of problems (*i.e.*, arithmetic and Boolean problems).

This paper is organized as follows. Section 2 explains a biological evolution simulator, Tierra, which idea is employed in TAGP, and Section 3 shows the algorithm of TAGP with/without the proposed mechanism. The testbed problems are explained in Section 4. Section 5 conducts the experiment, and Section 6 discusses their results. Finally, our conclusion is given in Section 7.

2 Tierra

Tierra [12] is the biological evolution simulator, where the digital creatures are evolved through a cycle of the self-reproduction, deletion, and genetic operators such as a crossover or a mutation. The digital creatures live in a memory space corresponding to land on earth, and they are implemented by a linear structured computer program such as the assembly language. The aim of the digital creatures is to reproduce themselves to a vacant memory space, *i.e.*, all programs are designed to copy themselves. The CPU time corresponding to energy like actual creatures is given to each creature, and it allows the digital creatures to execute their instructions within the allocated CPU time. Since given CPU time (*e.g.*, a time for executing a few instructions) is generally designed to be shorter than the execution time of all instructions in the program, all programs can be executed in parallel. The lifespan of the program is decided (*i.e.*, the program is deleted) by the *reaper* mechanism, and all programs are inserted in a queue, named as *reaper queue*. As the brief algorithm of Tierra is summarized as follows:

1. The program that the CPU time is assigned executes a few instructions.
2. The program that can correctly execute its instruction moves to the lower (younger) position in the reaper queue, while one that cannot correctly execute its instruction moves to the upper (older) position.
3. The program is reproduced when executing all of its instructions, and the reproduced program is added to the lowest (youngest) in the reaper queue. The genetic operators are applied in a certain percentage when reproducing the program.
4. When a memory space is filled, the program located at the most upper (oldest) position in the reaper queue is deleted.
5. Return to 1.

By the above algorithm, the programs which cannot reproduce themselves within the allocated CPU time or include some incorrect instructions are deleted, while the ones which can reproduce themselves propagate in the memory. As results of such an evolution, the programs that have the short program size or have special features are generated as emergent phenomena [8].

3 Tierra-based Asynchronous Genetic Programming

3.1 Overview

As described in the previous section, Tierra can asynchronously evolve programs (*i.e.*, digital creatures). However, the aim of all programs in Tierra is to reproduce themselves, which means that any other aims cannot be assigned to the programs, *i.e.*, the programs in Tierra cannot solve any given problems from the engineering viewpoint. To overcome this issue, we have proposed the new GP based on Tierra mechanism, named as *Tierra-based Asynchronous Genetic Programming (TAGP)* [5,6,11], which introduces the *fitness* commonly used in EAs to evaluate the programs, *i.e.*, the programs are reproduced or deleted according to their *fitness* values.

Fig. 1 shows an image of TAGP. TAGP starts from the programs that completely solve the given task. These programs consist of some instructions with some registers, IP, and ALU, and they are stored in a limited memory space. All programs are inserted in the *reaper queue* that controls their lifespan and they execute a few instructions in turn for a parallel execution. When all instructions in one program are completely executed, its fitness is calculated according to its result, and the program is reproduced asynchronously according to its fitness value. When the memory is filled with the programs, the program located at the most upper in the reaper queue is removed from the memory.

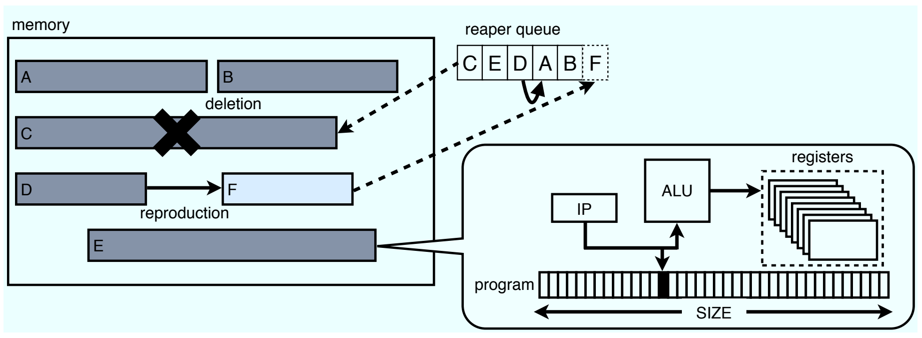


Fig. 1. An image of TAGP

3.2 Algorithm

The algorithm of TAGP is shown in Algorithm 1 which enables TAGP to evolve the programs for a given problem through the following (1) selection and reaper queue control, (2) reproduction, and (3) deletion procedures. In this algorithm, $prog$, $prog.f$, and $prog.f_{acc}$ respectively indicate the program, its fitness, and its accumulated fitness. *pre-prog* and *prepre-prog* respectively indicate the previous parent and two times previous parent. f_{max} indicates the maximum value of the fitness, while $rand(0, 1)$ indicates the random real value between 0 to 1. P_{down} and P_{up} are described later.

(1) Selection and reaper queue control
When one program is completely executed, its fitness is calculated and the calculated fitness is added to an accumulated fitness $prog.f_{acc}$ (line 1 in Algorithm 1). If the accumulated fitness of the program exceeds f_{max}, it is selected as a reproduction candidate, and f_{max} is subtracted from its accumulated fitness (line 2 and 3). If not, the program is not selected as a reproduction candidate. Note that the program having a high fitness has a high probability to be selected

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:   [ TAGP without the TD tournament selection ]
        generate an offspring by mating prog with pre-prog through genetic operators
                              or
      [ TAGP with the TD tournament selection ]
        generate an offspring by mating the winner of prog and pre-prog with the
        winner of pre-prog and prepre-prog through genetic operators
 8:   prepre-prog ← pre-prog, pre-prog ← prog
 9:   delete the program located the most upper in repair queue
10: else
11:   repeat
12:     up reaper queue position
13:   until $rand(0,1) > P_{up}(prog.f)$
14: end if
```

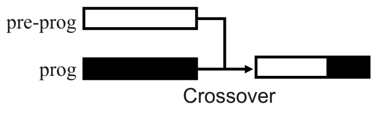
as a reproduction candidate because the accumulated fitness frequently exceeds f_{max}, while the program having a low fitness is hard to satisfy this condition.

After that, the position in the reaper queue of the program selected as the reproduction candidate becomes lower than the current one, *i.e.*, its deletion probability decreases (which means to survive long) (line 4∼6), while the position of the program not selected as the reproduction candidate becomes upper, *i.e.*, its deletion probability increases (which means to be easily removed) (line 11∼13). The lower/upper distance is determined by P_{down} and P_{up} which are calculated as the following equation based on fitness, where P_r is the maximum probability of P_{down} and P_{down}, which is predetermined.

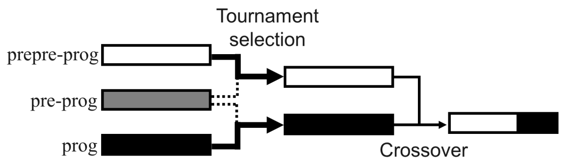
$$P_{down}(f) = \frac{f}{f_{max}} \times P_r, \qquad P_{up}(f) = \frac{f_{max} - f}{f_{max}} \times P_r \tag{1}$$

(2) Reproduction

To reproduce the program asynchronously, the previous TAGP (*i.e.*, TAGP *without* the TD tournament selection) generates an offspring by mating *prog* with *pre-prog* through the genetic operators such as a crossover, a mutation, an instruction insertion/deletion operations as shown in Fig. 2(a) (line 7). As mentioned in Section 1, however, the programs selected as the reproduction candidate does not always good ones. This is because the programs that quickly complete their evaluations have a high possibility to increase the accumulated fitness even though they have a low fitness (*i.e.*, they are not good ones). To overcome this problem, TAGP *with* the TD tournament selection generates an offspring by mating the winner of *prog* and *pre-prog* with the winner of *pre-prog* and *prepre-prog* through the genetic operators as shown in Fig. 2(b) (line 7). Such a minimum set of the tournament selection contributes to selecting *good* parent programs. The main difference between the conventional tournament selection and TD tournament selections is that the individuals are randomly selected in the former selection while they are determined as the current, previous, two times previous parents in the latter selection. As the end of the reproduction operation, the programs of *pre-prog* and *prepre-prog* are copied from those *prog* and *pre-prog* (line 8).



(a) TAGP without TD tournament selection (Previous TAGP)



(b) TAGP with TD tournament selection (Proposed TAGP)

Fig. 2. Without/with TD tournament selection

For the genetic operators in both TAGPs, the crossover operator combines two programs at two different crossover point, while the mutation operator randomly changes one instruction in the program. The instruction insertion operator inserts one instruction into a random point of the program, while the instruction deletion operator removes one instruction from the program.

(3) Deletion
Both TAGPs conduct a deletion operator when an offspring is generated (9 line). Concretely, the program located at the most upper in the reaper queue (*i.e.*, the program with the low fitness) is removed.

4 Problem description

4.1 Testbed problem

The testbed problems in this experiment are shown in Table 1, which are classified into the following two types: (1) the arithmetic problems that requires the numeric calculations and (2) the Boolean problem that requires the logical calculation. We employ these different types of the problems to investigate an applicability of the proposed method in a wide range of the problem types (from numeric to logical calculation). Note that # data in Table 1 indicates the number of the input data, *e.g.*, 16 data (x_1, $\cdots$, x_{16}) are the input value for A1 testbed.

Table 1. Testbed problems in this experiment

Arithmetic		#data	Boolean		#data
A1	$f(x) = x^4 + x^3 + x^2 + x$	16	B1	8bit-Parity	256
A2	$f(x) = x^5 - 2x^3 + x$	16	B2	7bit-DigitalAdder	128
A3	$f(x) = x^6 - 2x^4 + x^2$	16	B3	6bit-Multiplexer	64
A4	$f(x, y) = x^y$	25	B4	7bit-Majority	128

4.2 Evaluation criteria

As the evaluation criteria, the individuals are evaluated from the viewpoint of (1) the fitness (*i.e.*, the correct parcentage of the given problems) and (2) the execution step. In the above problems, the aim of the GPs is to evolve the programs that have the 100% correct of given tasks while minimizing their execution steps. Regarding the fitness, the following fitness functions (f_{arith} and f_{bool}) are respectively employed for the arithmetic and Boolean problems, where $\hat{y_i}$ indicates the i^{th} output value of a program, while y_i^* indicates the i^{th} target value.

$$f_{arith} = f_{max} - \frac{1}{n}\sum_{i=1}^{n} |\hat{y_i} - y_i^*| \tag{2}$$

$$f_{bool} = f_{max} - \frac{2}{n}\sum_{i=1}^{n} \delta(\hat{y_i}, y_i^*), \qquad \delta(x, y) = \begin{cases} 0 & x = y \\ 1 & x \neq y \end{cases} \tag{3}$$

5 Experiment

5.1 Cases

To investigate the effectiveness of TAGP with the TD tournament selection, we employ Linear GP (LGP) [1] [2] using an actual machine code, and conduct the the following experiments:

- **Case 1:** A comparison of TAGP *with* the TD tournament selection (TAGP-TD) and TAGP *without* it (TAGP-NoTD)
- **Case 2:** A comparison of TAGP *with* the TD tournament selection (TAGP-TD) and the steady-state GP (SSGP) as the simple synchronous-based GP

The reason why we employ LGP is summarized as follows: (1) an individual in LGP has the variable length chromosome, which is a natural feature of the actual machine-code program; and (2) the individuals including the infinite loop can be generated, which should be solved by the proposed mechanism in the asynchronous program evolution.

As an actual machine-code program, we employ an instruction set embedded on PIC10 [10] developed by Microchip Technology Inc., which consists of 12 bits 33 instructions. This instruction set includes adder, subtracter, logic operations, bit operations, and branch instructions, while does not include the multiplier instruction because the multiplier instruction can be achieved by repeating adder and bit operations in the loop structures. One program has 16 general purpose registers and one working register, and each register consists of 32bits.

5.2 Parameter settings

The common parameters in TAGP with/without the TD tournament selection and SSGP are shown in Table 2. In SSGP, in particular, the maximum execution step is set to $50,000$ because the program evolved by SSGP has the possibility of not completing its program due to an infinite loop. If a program does not complete in this maximum execution step, its fitness is evaluated as $-\infty$.

All experiments start from filling the population with an initial program that completely accomplishes the given problem shown in Table 1. Each experiment is conducted 30 independent trials. As mentioned in Section 4.2, we evaluate the effectiveness of TAGP with/without the TD tournament selection and SSGP from the viewpoint of the execution steps averaged from 30 trials. But, we do not evaluate the fitness (*i.e.*, the correct parcentage of the given problems) in this time because the best program evolved by these GPs has 100% correctness.

Table 2. Parameters

Parameter	value	Parameter	value
Number of evaluations	10^6	Crossover rate	0.7
Max. program size	256	Mutation rate	0.1
Pop. size	100	Insertion rate	0.1
f_{max}	100	Deletion rate	0.1

5.3 Results

•Case 1: Comparison of with/without TD tournament selection
Fig. 3 shows the execution steps averaged from 30 trials in TAGP-NoTD and TAGP-TD in A1 testbed. In this figure, the horizontal and vertical axes indicate the number of the evaluations and the average execution steps of the maximum fitness program, respectively. The dotted and solid lines respectively show the result of TAGP-NoTD and TAGP-TD, and the bars in the line indicate the standard deviation of the execution steps. As shown in Fig. 3, the average execution steps in TAGP-TD is smaller than TAGP-NoTD. Furthermore, the upper execution steps ($average + std$) in TAGP-TD is shorter than the lower execution steps ($average - std$) in TAGP-NoTD. This result indicates that TAGP-TD has better evolution ability than TAGP-NoTD in asynchronous program evolution. Note that this tendency is also found in other testbeds from A2 to A4 and from B1 to B4.

•Case 2: Comparison of TAGP with TD tournament selection & SSGP
Table 3 shows the average execution steps (30 trials) after the maximum number of evaluations in SSGP and TAGP-TD. In this table, the shorter execution

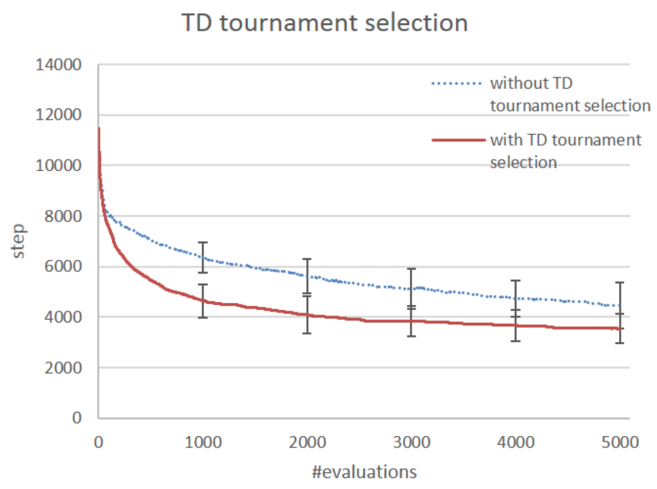


Fig. 3. The average execution steps (30 trials) in TAGP with/without the TD tournament selection: A1 problem

Table 3. The average execution step after the maximum evaluations (30 trials) in SSGP and TAGP with the TD tournament selection

Problem	SSGP	TAGP with TD tournament selection	reduction rate
A1	4659 (524)	**3293** (753)	29% down
A2	4734 (498)	**3314** (557)	29% down
A3	4928 (494)	**3369** (757)	32% down
A4	**4780** (1705)	4900 (2298)	3% up
B1	4873 (187)	**4838** (159)	1% down
B2	4774 (112)	**4661** (156)	2% down
B3	1762 (221)	**1114** (275)	37% down
B4	18492 (3752)	**15757** (3056)	15% down

steps in each testbed are indicated as **bold** style and a value in the parentheses indicates the standard deviation (std) of the execution steps. From these results, TAGP-TD outperforms SSGP except for the A4 testbed (Although the standard deviation of the execution steps of TAGP-TD is a little larger than that of SSGP, the upper execution steps ($average + std$) in TAGP-TD is shorter than the lower execution steps ($average - std$) in SSGP. For example, see Fig. 4.). Concretely, in the arithmetic problems, TAGP-TD reduces the execution steps around 30% (A1, A2, A3 testbeds) or derives mostly the same execution steps (A4 testbed) in comparison with SSGP. In Boolean problem, on the other hand, TAGP-TD reduces the execution steps from 15% (B4 testbed) to 37% (B3 testbed) or derives mostly the same execution steps (B1, B2 testbeds) in comparison with SSGP.

To investigate these results in detail, Figs. 4 and 5 show the average execution steps (30 trials) over the generation in A1, A4, B2, and B3 testbeds. We choose

A1 and B3 testbeds for the case where TAGP-TD derives better result than SSGP, while we choose A2 and B2 testbeds for the case where TAGP-TD and SSGP derive similar results In these figures, the horizontal and vertical axes and the bars in the line have the same meaning of the previous figure. The solid and dotted lines respectively show the result of TAGP-TD and that of SSGP.

As shown in Fig. 4, the average execution steps in TAGP-TD are shorter than SSGP. Like Fig. 3, the upper execution steps ($average + std$) in TAGP-TD is shorter than the lower execution steps ($average - std$) in SSGP. These results indicate that TAGP-TD shows the high evolution ability in comparison with SSGP. As shown in Fig. 5, on the other hand, the std ranges in TAGP-TD and SSGP are mostly the same, but the average execution steps in TAGP-TD becomes short quickly in comparison with those in SSGP in A4 testbed and those in TAGP-TD are shorter than those in SPSS in B2 testbed, even though both results in A4 and B2 testbeds are similar from Table 3. This result indicates that TAGP-TD has a potential of the quick search ability and high evolution ability in comparison with SSGP. Regarding testbed B2, in particular, Boolean problems are harder than the arithmetic problems from the viewpoint of evolving the problems, which means that even small reduction of the execution steps shows the high evolution ability of the TAGP-TD.

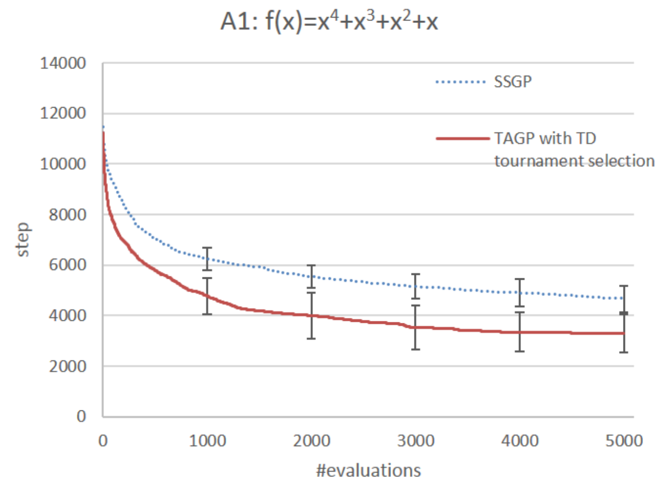
In total, TAGP-TD shows the high ability in both the arithmetic and Boolean problems.

6 Discussion

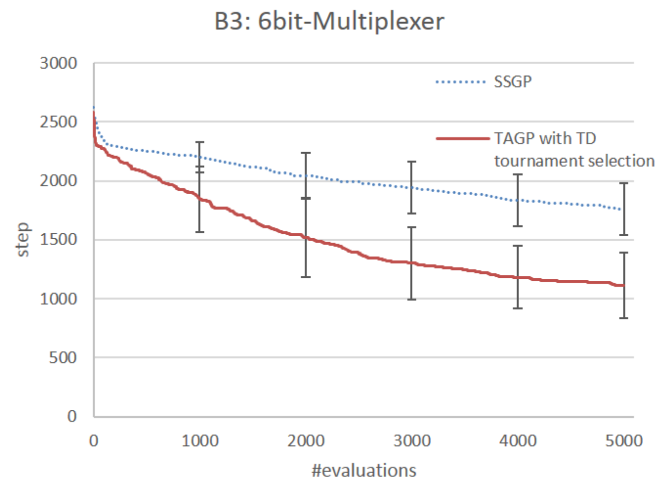
To investigate the reason why TAGP-TD outperforms SSGP from the viewpoint of the averaged execution steps, Fig. 6 show a part of their evolved programs in A1 testbed. This part of the evolved programs calculates x^2 in $f(x) = x^4 + x^3 + x^2 + x$ (x is implemented by four bits). In the evolved programs, R1, R2, R5, R6, and R7 indicate the general purpose register, while W indicates the working register. The input variable is set to R1 register, while the output result is set to R2 register. "$< -$" indicates substitution and "$>>$ 1" indicates 1 bit shift right.

From the evolved program in SSGP as shown in Fig. 6(a), the registers are initially set in line 1∼6, the calculations on the lowest, the second lowest, the third lowest, and fourth lowest bits are respectively conducted in line 7∼11, line 12∼16, line 17∼21, and line 22∼25 (note that four bits calculations are needed because x is implemented by four bits). The output result is calculated by repeating four instructions as the loop program in line 26∼31. From the evolved program in TAGP-TD as shown in Fig. 6(b), on the other hand, the registers are initially set in line 1∼3, the calculations on the lowest, the second lowest, the third lowest, and fourth lowest bits are respectively conducted in line 4∼7, line 8∼11, line 12∼15, and line 16∼18. The output result is calculated by repeating the same shift instruction in line 19∼46.

The main difference of the evolved programs between SSGP and TAGP-TD is summarized as follows: (1) the program size of the program evolved by TAGP-

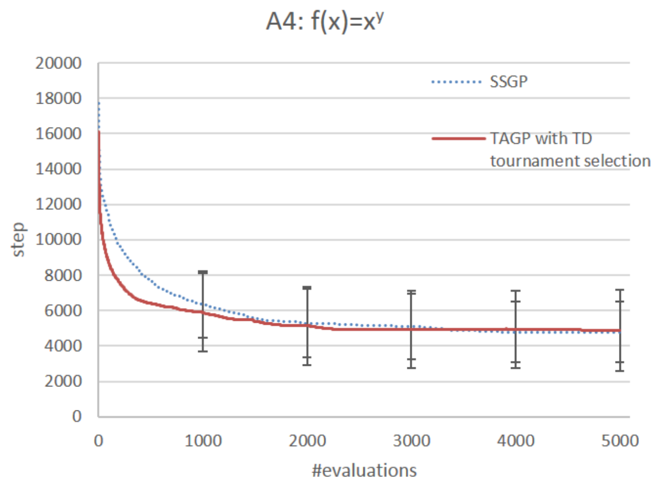


(a) A1 Testbed: $f(x) = x4 + x^3 + x^2 + x$

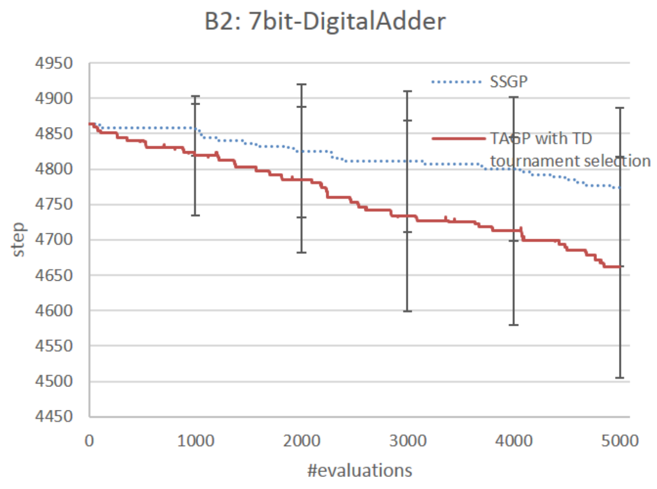


(b) B3 Testbed: 6bit-Multiplexer

Fig. 4. The average execution step after the maximum evaluations (30 trials) in SSGP and TAGP with the TD tournament selection: A1 and B3 problems



(a) A4 Testbed: $f(x, y) = x^y$



(b) B2 Testbed: 7bit-DigitalAdder

Fig. 5. The average execution step after the maximum evaluations (30 trials) in SSGP and TAGP with the TD tournament selection: A4 and B2 problems

```
MOVF    R1 0 // W <- R1
MOVWF   R5 1 // R5 <- W
MOVWF   R7 1 // R7 <- W
MOVLW   32   // W <- 32
MOVWF   R6 1 // R6 <- W
MOVF    R5 0 // W <- R5
BTFSC   R7 0 // if R7[0] == 0 then skip
ADDWF   R2 1 // R2 <- R2 + W
RRF     R2 1 // R2 <- R2 >> 1
RRF     R7 1 // R7 <- R7 >> 1
DECFSZ  R6 1 // R6 <- R6 - 1 and if R6 = 0 then skip
BTFSC   R7 0 // if R7[0] == 0 then skip
ADDWF   R2 1 // R2 <- R2 + W
RRF     R2 1 // R2 <- R2 >> 1
RRF     R7 1 // R7 <- R7 >> 1
DECFSZ  R6 1 // R6 <- R6 - 1 and if R6 = 0 then skip
BTFSC   R7 0 // if R7[0] == 0 then skip
ADDWF   R2 1 // R2 <- R2 + W
RRF     R2 1 // R2 <- R2 >> 1
RRF     R7 1 // R7 <- R7 >> 1
DECFSZ  R6 1 // R6 <- R6 - 1 and if R6 = 0 then skip
BTFSC   R7 0 // if R7[0] == 0 then skip
ADDWF   R2 1 // R2 <- R2 + W
RRF     R2 1 // R2 <- R2 >> 1
DECFSZ  R6 1 // R6 <- R6 - 1 and if R6 = 0 then skip
NOP     16 0 // label(0)
RRF     R2 1 // R2 <- R2 >> 1
DECFSZ  R6 1 // R6 <- R6 - 1 and if R6 = 0 then skip
RRF     R2 1 // R2 <- R2 >> 1
DECFSZ  R6 1 // R6 <- R6 - 1 and if R6 = 0 then skip
GOTO    16   // goto label(0)
```

(a)SSGP

```
MOVF    R1 0 // W <- R1
MOVWF   R5 1 // R5 <- W
MOVWF   R7 1 // R7 <- W
BTFSC   R7 0 // if R7[0] == 0 then skip
ADDWF   R2 1 // R2 <- R2 + W
RRF     R2 1 // R2 <- R2 >> 1
RRF     R7 1 // R7 <- R7 >> 1
BTFSC   R7 0 // if R7[0] == 0 then skip
ADDWF   R2 1 // R2 <- R2 + W
RRF     R2 1 // R2 <- R2 >> 1
RRF     R7 1 // R7 <- R7 >> 1
BTFSC   R7 0 // if R7[0] == 0 then skip
ADDWF   R2 1 // R2 <- R2 + W
RRF     R2 1 // R2 <- R2 >> 1
RRF     R7 1 // R7 <- R7 >> 1
BTFSC   R7 0 // if R7[0] == 0 then skip
ADDWF   R2 1 // R2 <- R2 + W
RRF     R2 1 // R2 <- R2 >> 1
RRF     R2 1 // R2 <- R2 >> 1
RRF     R2 1 // R2 <- R2 >> 1
...[RRF     R2 1] x 17 ...
RRF     R2 1 // R2 <- R2 >> 1
RRF     R2 1 // R2 <- R2 >> 1
RRF     R2 1 // R2 <- R2 >> 1
RRF     R2 1 // R2 <- R2 >> 1
RRF     R2 1 // R2 <- R2 >> 1
RRF     R2 1 // R2 <- R2 >> 1
RRF     R2 1 // R2 <- R2 >> 1
RRF     R2 1 // R2 <- R2 >> 1
RRF     R2 1 // R2 <- R2 >> 1
RRF     R2 1 // R2 <- R2 >> 1
```

(b)TAGP with the TD tournament selection

Fig. 6. The evolved programs in SSGP and TAGP with the TD tournament selection in A1 testbed

TD (46 size) is larger than the one evolved by SSGP (31 size), but this is not a problem because the execution steps of TAGP-TD (3293 steps) is shorter than those of SSGP (4659 steps) as shown in Table 3. Concretely, the loop program in SSGP is executed many time until 4659 steps, while no loop program in TAGP-TD contributes to reducing the execution steps (3293 steps); (2) the program evolved by SSGP requires the R6 resister for a counter of the bit shift (line 5, 11, 16, 21, 25, 28, and 30) and the loop instructions (line 26 and 31), while the one evolved by TAGP-TD does not have such a register and instructions, which contributes to reducing the execution steps. These different features suggest that TAGP-TD has a great potential of reducing the execution steps.

Finally, the program structure of SSGP is more smart than that of TAGP-TD from the viewpoint of the loop instruction, but it is dangerous to evolve the programs with the loop instructions because such programs are easily to become the infinite loop programs or the fitness values of such programs drastically change by mutating the counter for the loop instruction. From this viewpoint, TAGP-TD avoids such a dangerous program evolution.

7 Conclusion

This paper focused on an *asynchronous* program evolution in evolutionary computation, which is hard to evolve individuals (*i.e.*, programs) effectively unlike a *synchronous* program evolution that evolves individuals effectively by selecting good parents after evaluations of *all* individuals in each generation. To tackle this problem, we explored the mechanism that can promote an asynchronous program evolution by selecting a good individual without waiting for evaluations of *all* individuals. Concretely, this paper proposed the *temporal difference (TD) tournament selection*, where the winner of the current and previous parents and the winner of the previous and two times previous parents are mated in the crossover operation. Such a minimum set of the tournament selection contributes to selecting good individual as the parent.

To investigate the effectiveness of the proposed mechanism, this paper evaluate it in the two types of problems (*i.e.*, the arithmetic and Boolean problems) in the GP domain. The intensive experiments of the eight kinds of testbeds under the two types of problems have revealed the following implications: (1) the program *asynchronously* evolved *with* the TD tournament selection can be completed with the shorter execution steps than the program *asynchronously* evolved *without* it; and (2) the program *asynchronously* evolved *with* the TD tournament selection can be completed with mostly the same or shorter execution steps than the program *synchronously* evolved by the simple steady-state GP (SSGP).

What should be noticed here is that these results have only been obtained from two types of problem, *i.e.*, arithmetic and Boolean problems. Therefore, further careful qualifications and justification, such as an analysis of results using other problems such as symbolic regression or classification problem, are needed to generalize the effectiveness of the proposed mechanism. As the other issue, the parents selected by the TD tournament selection become the same when the fitness of the previous parents is higher than that of the current and two times previous parents. Since this weak point of the proposed mechanism decreases its evolution ability, this issue should be solved. These important directions must be pursued in the near future in addition to the following future research: (1) an improvement of the proposed mechanism in A4 testbed; and (2) an extension of TAGP with the proposed mechanism not to set the parameter f_{max}.

References

1. Banzhaf, W., Francone, F.D., Keller, R.E., Nordin, P.: Genetic programming: an introduction: on the automatic evolution of computer programs and its applications. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1998)
2. Brameier, M.F., Banzhaf, W.: Linear Genetic Programming. Volume 117. Springer (2007)
3. Glasmachers, T.: A natural evolution strategy with asynchronous strategy updates. The fifteenth annual conference on Genetic and evolutionary computation conference (GECCO 2013), ACM, pp. 431–438 (2013)
4. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc. (1989)
5. Harada, T., Otani, M., Matsushima, H., Hattori, K., Sato, H., Takadama, K.: Robustness to Bit Inversion in Registers and Acceleration of Program Evolution in On-Board Computer. Journal of Advanced Computational Intelligence and Intelligent Informatics (JACIII) 15(8), 1175–1185 (10 2011)
6. Harada, T., Otani, M., Matsushima, H., Hattori, K., Takadama, K.: Evolving Complex Programs in Tierra-based On-Board Computer on UNITEC-1. In: International Astronautical Congress (IAC), 2010 61st World Congress on (2010)
7. Koza, J.: Genetic Programming On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
8. Langton, C.G.: Artificial Life. Addison-Wesley (1989)
9. Lewis, A., Mostaghim, S., Scriven, I.: Asynchronous multi-objective optimisation in unreliable distributed environments. Biologically-Inspired Optimisation Methods. Vol. 210, Studies in Computational Intelligence, Springer Berlin Heidelberg, pp. 51-78 (2009)
10. Microchip Technology Inc.: PIC10F200/202/204/206 Data Sheet 6-Pin, 8-bit Flash Microcontrollers. Microchip Technology Inc. (2007), `http://ww1.microchip.com/downloads/en/DeviceDoc/41239D.pdf`
11. Nonami, K., Takadama, K.: Tierra-based Space System for Robustness of Bit Inversion and Program Evolution. In: SICE, 2007 Annual Conference. pp. 1155–1160 (2007)
12. Ray, T.S.: An approach to the synthesis of life. Artificial Life II XI, 371–408 (1991)
13. Reynolds, C.W.: An evolved, vision-based behavioral model of coordinated group motion. In: 2nd International Conference on Simulation of Adaptive Behavior. pp. 384–392. MIT Press (1993)
14. Storn, R., Price, K.: Differential Evolution - A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. J. of Global Optimization 11(4), 341–359 (Dec 1997), `http://dx.doi.org/10.1023/A:1008202821328`
15. Zhang, Q., Li, H.: MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. IEEE Trans. Evolutionary Computation 11(6), 712–731 (2007)