A Co-evolutionary Cartesian Genetic Programming with Adaptive Knowledge Transfer

Jinghui Zhong*, Linhao Li*, Wei-Li Liu*, Liang Feng† and Xiao-Min Hu‡
*Guangdong Provincial Key Lab of Computational Intelligence and Cyberspace Information
School of Computer Science and Engineering, South China University of Technology, Guangzhou, China
Corresponding author email: jinghuizhong@gmail.com
†College of Computer Science, Chongqing University, Chongqing, China
‡School of Computers, Guangdong University of Technology, Guangzhou, China

***Abstract*—Cartesian Genetic Programming (CGP) is a powerful and popular tool for automatic generation of computer programs to solve user defined tasks. This paper proposes a Co-evolutionary CGP (named Co-CGP) which can automatically gain high-order knowledge to accelerate the search. In the Co-CGP, two modules are working in cooperation to solve a given problem. One module focuses on solving a series of small scale problems of the same type to generate the building blocks. Simultaneously, the second module focuses on combing the available building blocks to construct the final solution. Besides, an adaptive control strategy is introduced to automatically evaluate the effectiveness of the building blocks and adjust the search behaviour adaptively so as to improve search efficiency. The proposed Co-CGP is tested on eight problems with different complexities. Experimental results show that the Co-CGP can significantly improve the performance of CGP, in terms of both search efficiency and accuracy.**

***Index Terms*—Cartesian Genetic Programming, Co-evolutionary Algorithm, Transfer Learning, Evolutionary Computation**

I. Introduction

Genetic Programming (GP) is an evolutionary computation technique for automatic generation of computer programs to solve user-defined tasks [1], [2]. Due to its simple principle and powerful search ability, GP has undergone rapid developments in both theoretical and practical applications over the past decades. More recently, GP has gained increasing popularity as a tool for data mining and knowledge discovery, where a range of successful applications including classification problems [3], [4], symbolic regression problems [5], [6], automatic model construction [7], and descriptive analysis [8]have been reported.

However, GP suffers from the deficiency of slow search convergence, especially on complex and large scale problems. Besides the iterative process of population sampling during the GP searching which makes it slow, the other bottleneck may be attributed to the lack of knowledge reuse while solving problems with similar characteristics, i.e., GP usually starts its search from ground zero on a given new problem to solve. Hence, developing a more effective method to accelerate GP on complex and large scale problems is of great interest in the data mining and knowledge discovery community.

A common method for human to solve complex problems is to utilize high-order knowledge that is learned from past solved problems of similar type or small-scale problems of the same type. Taking the famous Tower of Hanoi problem for example, as shown in Fig. 1, it is challenging to directly find a solution to the Tower of Hanoi problem with $n(n >> 3)$ disks. However, if we were to obtain a solution to the trivial case for $n = 3$, one can construct a solution to the case of $n = 4$ simply by: 1) moving the upper 3 disks from column A to column B using the attained solution, 2) moving the last disk to column C, and 3) moving the three disks from column B to C based on the attained solution. In this manner, one can easily find solutions to large scale cases. However, for a given problem, identifying the high-order knowledge and deciding how to use them appropriately often requires domain knowledge and can be non-trivial.

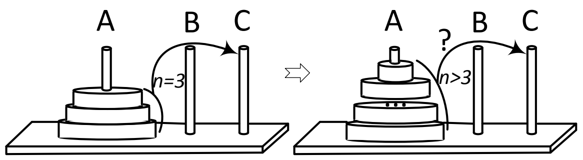

Fig. 1. The tower of hanio problem.

In the GP community, some methods have been developed to accelerate GP by incorporating high-order knowledge, such as the Model Acquisition (MA) technique [9], the Automatically Defined Function (ADFs) technique [10], and the meta-meme learning technique [11]. The MA technique tries to mine promising partial solutions from the historical search data, and then ensure their survival during the evolutionary process or utilize them to generate offspring [9], [12]. Meanwhile, the ADF technique considers the high order knowledge as sub-functions that are self-evolved by GP search dynamically along the search [10], [13], [14], [15]. Meuth et.al. [11] developed a Meta-learning architecture which introduces a memory component to archive generalized solutions learned from the past problems solved. These solutions are then used to improve the performance of the search. Zhong et al. [16] proposed a multifactorial genetic programming (MF-GP) which can solve multiple problems at the same time. In MF-GP, common knowledge learned from solving different problems are shared during the evolution so as to improve the search efficiency of solving all problems. Iqbal et.al. [17] developed an enhanced learning classifier system, where high-order knowledge that learned from smaller problems was utilized in learning more complex, large-scale problems. Similarly, Jackson and Gibbons proposed a two-layer GP (named LLGP) which gains high-order knowledge by solving small scale problems. The LLGP has been shown to perform better than the classical GP, but its performance has not been very satisfactory. Besides, if the knowledge learned from small scale problems is ineffective, the performance of the LLGP search may degrade significantly.

Inspired by the above works, this paper proposes a novel Co-evolutionary Cartesian Genetic Programming (Co-CGP), which mimics the divided-and-conquer search strategy to gain high-order knowledge and to accelerate the search. The proposed method differs from the existing approaches in that the proposed Co-CGP integrates two modules which are working co-operatively. The first module focuses on finding solutions to a series of decomposed sub problems with increasing problem dimensionality. The solutions obtained by this module are treated as high-order functions and are maintained in a building block pool. Meanwhile, the second module focuses on searching for effective strategies to combine the available building blocks and construct the final solution. The first module would stop when all sub problems have been solved or when the second module meets a termination condition. This co-evolutionary mechanism provides a flexible and efficient way to gain high-order knowledge from sub problems, because we can set an arbitrary number of sub problems for the first module, while the second module does not need to wait until all the sub problems have been solved. Besides, an adaptive controlling strategy is introduced to automatically evaluate the effectiveness of the building blocks and adjust the search behaviour accordingly. With the adaptive control strategy, building blocks are more likely to be used only when they are effective for constructing the promising solutions, thus benefitting search performance. The proposed Co-CGP was applied to eight test instances with different complexities, and the experimental results obtained are reported to demonstrate the effectiveness of the proposed method.

II. Preliminaries

Cartesian Genetic Programming (CGP) is a GP variant proposed by Miller and Thomson [18] for digital circuits design. The distinct feature of CGP is that it represents a program of directed graph instead of parse tree as in the traditional GPs. This feature enables it to reuse subgraphs that previously exist, which is useful for improving GP search efficiency. Due to its fast search efficiency and ease of implementation, CGP has attracted increasing attentions over the years and has recently become a popular choice for automatic programming [19].

In CGP, each chromosome consists of two parts: function nodes and output nodes. Each function node represents a particular function and it is encoded by a number of genes. The first gene encodes the function of the node (e.g., "+", and "sin"), and the remaining genes encode the input sources

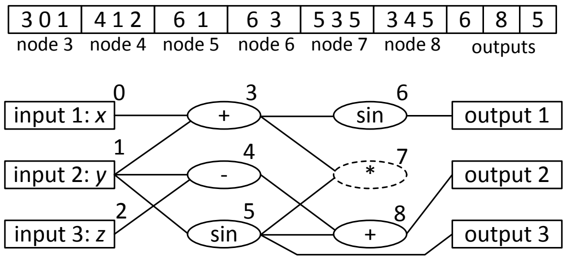

Fig. 2. Traditional chromosome representation in CGP.

of the function. The input sources of a function node can be a previous node or a program input (i.e., terminals such as x, y, and π). The number of input sources is determined by the function type of the node. For example, "+" has two input sources while "sin" has only one input source. In the original CGP, the program inputs and the function nodes are labelled by integers, where 0 to $n-1$ represent the n program inputs and n to $n+m-1$ represent the m function nodes in the chromosome. The values of the k output genes are also integers, each of which represents the source of the corresponding output.

Fig. 2 shows an example of the CGP chromosome which contains three program inputs and three functions. The first three nodes are terminals which are fixed, hence the first three node are not encoded in the chromosome. The terminals and functions are represented by: $x \rightarrow 0, y \rightarrow 1, z \rightarrow 2, + \rightarrow 3, - \rightarrow 4, * \rightarrow 5, sin \rightarrow 6$. The first node in the chromosome is node 3 which contains three genes, i.e., "3 0 1". Gene 3 means the function type is "+", 0 and 1 mean the input source of the function is node 0 (i.e., x) and node 1 (i.e., y) respectively. Hence, the node 3 is decoded as $x+y$. In this way, we can decode node 4 as $y-z$, etc. The three outputs are decoded as $f_1(x,y,z) = sin(x+y)$, $f_2(x,y,z) = sin(y) + (y-z)$, and $f_3(x,y,z) = sin(y)$ respectively. It should be noted that some function nodes (e.g., node 7) may be inactive in a search generation (i.e., they have no influence on the outputs), but they may turn active in future generations. This feature allows the encoded program to have a variable length, which is beneficial to the search efficiency [20].

With the above chromosome representation, the original CGP utilizes the (1+4) evolutionary strategy to evolve the chromosome. The second step is repeated until the termination condition is met. For the details on implementations, the readers are referred to [18].

III. The Proposed Co-CGP

A. General Search Mechanism

The proposed algorithm comprises of two components, namely the CGP-I and the CGP-II, as shown in Fig. 3. The CGP-I focuses on solving a set of small-scale sub problems with increasing dimensionality, using the traditional CGP.

Various schemes to configure the dimensionality of the sub-problems may be considered, including a simple random setting or varying the dimensionality in an ascending order at fixed interval until the problem dimensionality is reached. Denote the set of dimensions as $D = \{d_1, d_2, ..., d_m\}$, where d_m is the maximum dimension considered in CGP-I. The solutions found by CGP-I are then archived in a building block pool.

Meanwhile, the CGP-II aims to construct a solution to the given problem by combining the extracted building blocks and the basic components (i.e., the original functions and terminals). In CGP-II, the building blocks are treated as functions with input arities. The number of input arities is then set equal to the dimensionality of the corresponding small problem that the building block solves.

In practice, the sub-problems initialized in the CGP-I may or may not be effective for constructing the final solutions, due to the lack of enough priori knowledge. Hence, in the worst case, increasing the number of ineffective building blocks can lead to degradation on the search efficiency of CGP-II. To address the above issue, an adaptive control strategy is introduced in the CGP-II. The key idea is to generate offspring of CGP-II by two modes, namely mode-I and mode-II. In mode-I, an offspring is generated without the use of the explicit building blocks identified, as in the classical CGP. In mode-II, an offspring is generated by considering the building blocks as parts of the function set. The probability of generating offspring using mode-I or mode-II is determined by a parameter (α). When the building blocks obtained by the CGP-I proven to be useful (e.g., the building blocks frequently exist in the best-so-far solution), α is then reinforced (increased) adaptively. Otherwise, α is decremented. In this manner, the CGP-II can adaptively evaluate the effectiveness of the building blocks and control the production of suitable offspring along the search, and thus enhancing search efficiency.

B. Chromosome Representation

The Co-CGP contains two types of chromosomes: one for CGP-I and the other for CGP-II. Since the search mechanism of CGP-I is the same as the traditional CGP, the traditional graph representation in CGP can be used directly to encode the individuals of CGP-I. Meanwhile, the CGP-II differs from CGP in that it can make use of the extracted building blocks to construct solutions. By treating the building blocks as functions with variable input arity, the traditional representation in CGP can also be used to encode individuals of CGP-II.

Fig. 4, shows an example solution to a small-scale problem with three program inputs. This building block can be treated as a function with three input arities and one output. As considered in the traditional CGP, the new function can be represented by a function node with four genes. The first gene encodes the id of the building block, while the other three genes encode the assignment of the three arities, respectively. For the sake of brevity, in this paper, building blocks with only one output is considered. Fig. 5 shows an example solution of the CGP-II which combines both the building blocks and basic components. Clearly, with the use of high-order building blocks, the solution of CGP-II is noted to be more compact and readable than those without use of the building blocks.

C. Algorithm Framework

With the above chromosome representation, the CGP-I and CGP-II operate on the $(1 + \lambda)$ ES to search for the optimal solutions. The outline of the proposed framework is presented in Algorithm 1, which comprises the following three steps.

Algorithm 1: Co-CGP.

```
   /* Initialization.                                   */
1  Initialize $\{d_1, d_2, ..., d_m\}$.
2  $\alpha = 0.5; g = 0;$
3  Generate an initial population for CGP-I
4  $\Phi$ = the best solution in CGP-I (i.e., $Best_{small}$)
5  Generate an initial population for CGP-II.
6  while termination condition not met do
       /* Update CGP-I                                  */
7      if CGP-I is active then
8          Generate $\lambda$ offspring by mode-I.
9          Update $Best_{small}$ and $\Phi$
10         if solve the current problem then
11             if $d_i < d_m$ then
12                 $d_i = d_{i+1}$
13                 Generate an initial population for CGP-I
14             else
15                 set CGP-I to be inactive
       /* Update CGP-II                                 */
16     for $i = 1$ to $\lambda$ do
17         if $rand(0, 1) < \alpha$ then
18             Generate an offspring by mode-II
19         else
20             Generate an offspring by mode-I
21     Update $Best_{large1}$ and $Best_{large2}$
       /* Adaptive controlling.                         */
22     if $Best_{large2}$ is updated by one of the offspring
        then
23         if $Best_{large2}$ contains building blocks then
24             $\alpha = \alpha * \rho + (1 - \rho) * 1$
25         else
26             $\alpha = \alpha * \rho + (1 - \rho) * 0$
27     else
28         $\alpha = \alpha * \rho + (1 - \rho) * 0.5$
29     $g = g + 1$
30 return $Best_{large2}$
```

1) Initialization: This step initializes the parameter settings of the algorithm, including α, g, and the dimension set D. Denote the given problem as P_{large} and the small-scale problem with dimension d_i as P_{d_i}. The first problem to be

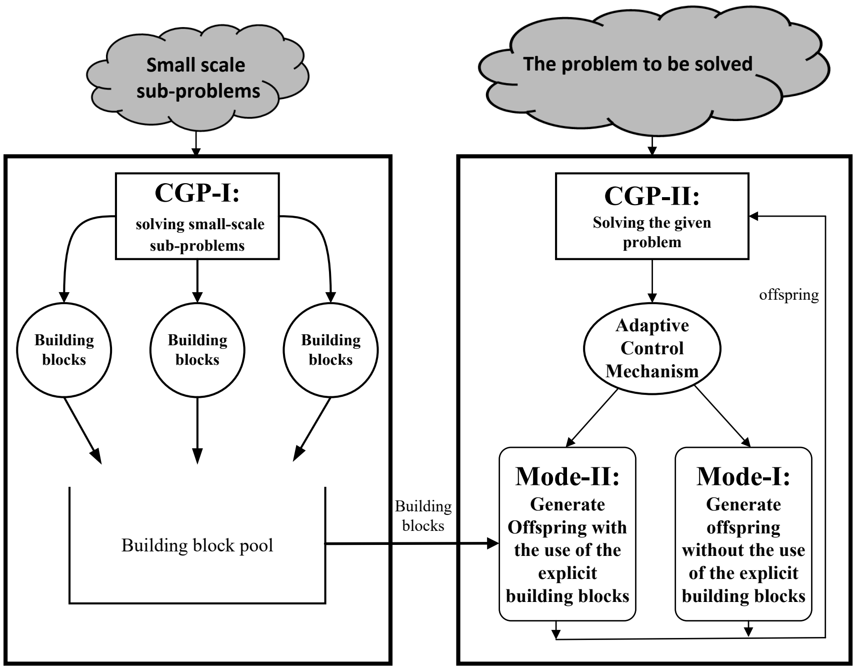
Small scale sub-problems
The problem to be solved
CGP-I: solving small-scale sub-problems
Building blocks
Building blocks
Building blocks
Building block pool
Building blocks
CGP-II: Solving the given problem
offspring
Adaptive Control Mechanism
Mode-II: Generate Offspring with the use of the explicit building blocks
Mode-I: Generate offspring without the use of the explicit building blocks

Fig. 3. The general framework of the proposed Co-CGP.

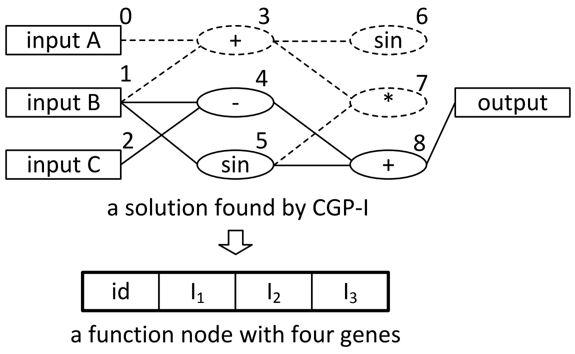

Fig. 4. Encoding a sub solution by a function node.

solved by CGP-I is set to be P_{d_1}. Then $(1 + \lambda)$ random offspring are generated as the initial population of CGP-I. The best individual is recorded as $Best_{small}$, which forms the first element of the building block pool Θ.

After that, $(1 + \lambda)$ random offspring are generated as the initial population of CGP-II. The initial individuals of CGP-II are generated by mode-I with a probability of $1 - \alpha$, and by mode-II with a probability of α. The best individual that are generated by mode-I is recorded as $Best_{large1}$, and the best among all individuals is recorded as $Best_{large2}$. Clearly, $Best_{large2}$ is always better or equal to $Best_{large1}$.

2) Update CGP-I: The CGP-I has an active state and an inactive state. This step first checks the state of CGP-I. If it is in inactive state, then this step will be omitted. Otherwise, λ new offspring will be generated by mutating $Best_{small}$. If the best offspring is better than the $Best_{small}$, it will be used to update $Best_{small}$ and the building block pool as well. Denoted the available building blocks as:

$$\Theta = \{\Phi_{d_1}, \Phi_{d_2}, ..., \Phi_{d_m}\} \tag{1}$$

where Φ_{d_i} is the best-so-far solution to P_{d_i}. Whenever a better solution Φ'_{d_i} is obtained, Φ_{d_i} will be updated as Φ'_{d_i} immediately. If the optimal solution of the current problem P_{d_i} is obtained, then CGP-I will increase the dimensionality of the problem and restart to solve $P_{d_{i+1}}$. If the currently solved problem is the last problem (i.e., $d_i = d_m$), CGP-I will set to inactive state.

3) Update CGP-II: After updating CGP-I and the building block pool, this step aims to generate λ offspring to update $Best_{large1}$ and $Best_{large2}$. The offspring are generated by mode-I with a probability of $(1 - \alpha)$, and by mode-II with a probability of α. In mode-I, the offspring is generated by mutating $Best_{large1}$, where a function node can only be mutated to a basic function (i.e., can not mutate to a building block). Meanwhile, in mode-II, the offspring is generated by mutating $Best_{large2}$, where the function node can be mutated to a basic function or any available building block. After generating λ offspring, $Best_{large1}$ and $Best_{large2}$ are updated accordingly.

Besides, this step also updates α by analyzing the best-so-far solution $Best_{large2}$. If none of the offspring is better or equal to $Best_{large2}$, α will be updated by:

$$\alpha = \alpha * \rho + (1 - \rho) * 0.5 \tag{2}$$

By moving α to 0.5, both mode-I and mode-II have equal chances to take effect, which can avoid the case where α

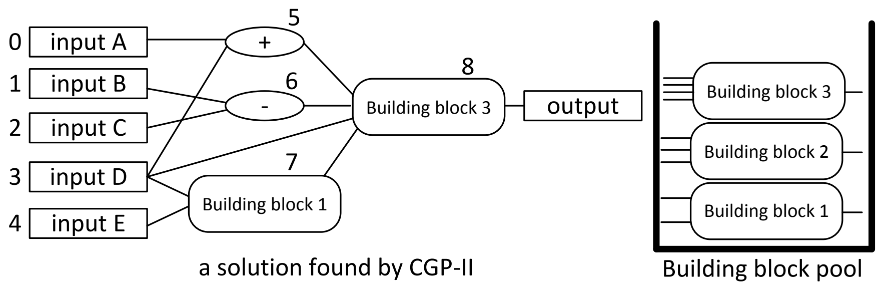

Fig. 5. An example solution provided by CGP-II.

is quickly reduced to zero in the early state, and Mode-II can never be selected. Otherwise, if at least one offspring is better or equal to $Best_{large2}$, α will be increased or reduced accordingly, depending on whether $Best_{large2}$ contains building blocks or not. If $Best_{large2}$ contains at least one building block, that means the building blocks are useful for the construction of promising solutions. Thus α is updated by:

$$\alpha = \alpha * \rho + (1 - \rho) * 1 \tag{3}$$

so that CGP-II can be more likely to generate offspring by mode-II. Otherwise, α will be gradually reduced by:

$$\alpha = \alpha * \rho + (1 - \rho) * 0 \tag{4}$$

IV. Experimental Studies

A. Experimental Settings

This section investigates the proposed Co-CGP on two types of problems, namely the even-parity problem and the majority-on problem. We choose these problems as test cases not only because they are well-known and extensively studied in the GP community, but also because they become highly complex and difficult to solve as their dimensions increase. In addition, since there are strong linkages between the variables, it is not easy to combine the solutions identified from the small scale problems to construct the solutions of large scale problems. Hence they pose as suitable for studying the scalability of GPs.

The even-k-parity problem requires finding a boolean formula comprising k arguments. This boolean formula returns a 'true' value if and only if there are even number of arguments assigned with 'true' values. The even parity problems are well known to be hard-to-solve boolean optimization problems for blind random search and the traditional GP [21]. The majority-on problem requires finding a program that is capable of determining whether the majority of its boolean inputs are set to logic-one. For example, in the 5-input version, a solution will deliver a 'true' if three or more inputs are logic-one, and 'false' otherwise. As the majority-on problem consists of strongly overlapping classifiers, it is difficult for traditional learning methods to solve [22]. In this paper, four even parity problems (i.e., 5-parity to 8-parity problems) and four majority-on problems (i.e., 5-majority to 8-majority problems) are considered. For both types of problems, the function set used is $\{AND, OR, NAND, NOR\}$.

In the experimental study, the performance of Co-CGP is compared with four other state-of-the-art GP variants. The first one is the traditional CGP [18]. The second and the third ones are two GP variants that utilize ADFs, which are the GP with ADF [10] and the ECGP [15], respectively. The last is the LLGP which uses a layer learning strategy to improve the classical GP search. The results of GP with ADF, ECGP, and LLGP are directly obtained from that reported in the referenced papers, while those of the CGP are obtained using the standard source code provided by the authors. The CGP and Co-CGP are performed on a desktop computer with Intel(R) Xeon(R) CPU @ 3.2GHz and 16.0G RAM. The parameter settings of CGP and Co-CGP are set as follows: NP = 5;λ = 4 rows = 1; columns = 50; level-back = 50; mutation rate = 0.05. The other settings of the Co-CGP are listed as follows: $\rho = 0.1$, $D = \{2, 3, 4\}$. Further, the CGP and Co-CGP are configured to terminate when the CPU running time reach 50 seconds or a global optimal solution is obtained.

B. Comparison with CGP

Fig. 6 shows the search trends of the best fitness as obtained by the CGP and Co-CGP. For each test case, the average results of 30 independent runs for each algorithm are reported and analyzed. It can be observed that the CGP manages to solve the problems of smaller dimensions such as the 5-even parity problem and the 5-majority problem, relatively easily. However, as the dimension of the problem increases, the performances of the CGP is shown to degrade significantly. Meanwhile, owning to the proposed co-evolutionary search mechanism, the Co-CGP always converges to the global optimal solution (or a better solution) at a much faster rate on all the test cases considered. These results thus demonstrate the effectiveness of the proposed co-evolutionary search mechanism in improving the search efficiency of CGP.

Table I lists the average fitness of the final solution obtained by the CGP and Co-CGP over the 30 runs. For all eight test cases, Co-CGP either converges to the global optimal solutions or at improved solutions, thus once again demonstrating effectiveness of the proposed co-evolutionary search mechanism over the CGP.

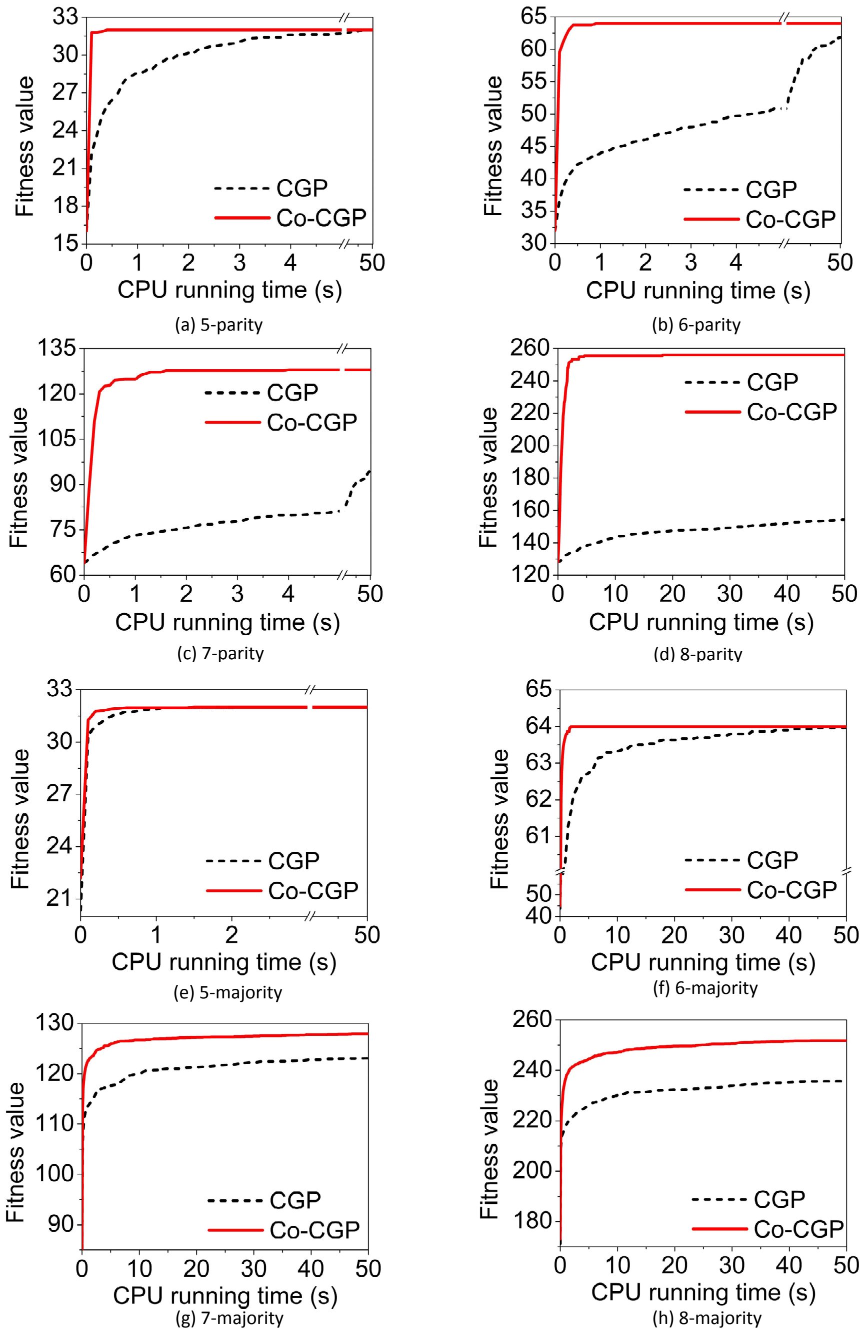

Fig. 6. Evolution curves of the best fitness values found by CGP and Co-CGP.

C. Comparison with Other GPs

In this subsection, we compare the Co-CGP to the relevant GP variants. First, we evaluate the results of our proposed approach against the published results of the GP with ADFs and ECGP [15], in terms of the the computational efforts (denoted as CE) [1]. The CE can be computed by

$$R(z) = ceil(\frac{log(1-z)}{log(1-P(NP,i))}) \tag{5}$$

$$CE = min_i\{NP \cdot R(z) \cdot (i+1) | i = 1, 2, ...\} \tag{6}$$

where $P(NP, i)$ is the proportion of runs that achieve the global optimal solution by generation i, and z is a probability value. The statistic CE is used to estimate how many fitness evaluations must be performed to solve a problem with a probability of z. As in [15], $z = 0.99$ is used in this study.

Table. II shows the CE of the three algorithms on the four parity problems. The value "N/A" means that the algorithm fail

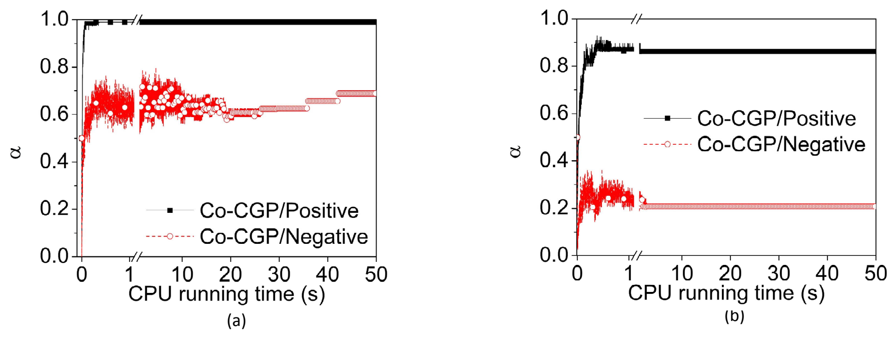

Fig. 7. α changes with the number of generations.

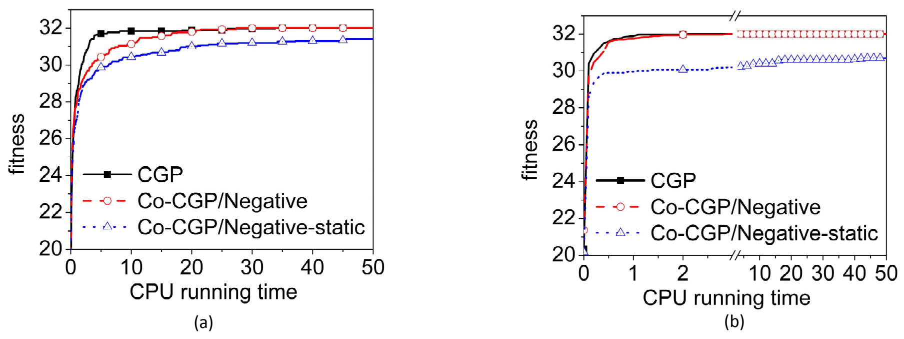

Fig. 8. The evolution curves of the best fitness of Co-CGP on the two new cases.

TABLE I
THE BEST FITNESS VALUES FOUND BY CGP AND CO-CGP.

problem	CGP-mean	CGP-Std.	Co-CGP-mean	Co-CGP-Std.
5-parity	32	0	32	0
6-parity	61.8	3.76	64	0
7-parity	94.8	5.78	128	0
8-parity	154.2	5.83	256	0
5-majority	32	0	32	0
6-majority	63.97	0.18	64	0
7-majority	123.1	1.53	127.9	0.40
8-majority	235.8	3.2	251.8	3.50

TABLE II
THE BEST FITNESS VALUES FOUND BY GP WITH ADFS, ECGP, AND CO-CGP ON FOUR EVEN PARITY PROBLEMS.

problem	GP with ADFs	ECGP	Co-CGP
5-parity	464,000	512002	34,488
6-parity	1,344,000	978,882	73,544
7-parity	N/A	1,923,842	116,792
8-parity	N/A	4,032,002	326,340

to find a solution in all the 30 runs. It can be observed that the GP with ADFs perform the worst. The GP with ADFs even fail to solve the last two even parity problems. Meanwhile, both ECGP and the Co-CGP achieve 100% successful rate on all four problems, but the Co-CGP requires much fewer CE on all problems, especially on the large scale cases.

Next, we compare Co-CGP with LLGP. In [23], the LLGP was applied to solve four-even parity problems, i.e., 4-parity, 5-parity, 6-parity, and 8-parity problem. For the same computational budgets, the Co-CGP is performed on each of the four problems for 100 independent runs. In each run, Co-CGP terminates when the number of fitness evaluations reaches 100,000. Other parameters of Co-CGP are set to be consistent to the above experiments. Table III lists the successful rate of the two algorithms in finding the global optimal solutions. It can be observed that the Co-CGP can achieve improved successful rate on all of the four even-parity problems.

TABLE III
THE SUCCESSFUL RATES OF LLGP AND CO-CGP FINDING A GLOBAL OPTIMAL SOLUTION.

problem	LLGP	Co-CGP
4-parity	95 %	100%
5-parity	92%	100%
6-parity	70%	99%
8-parity	25%	69%

D. *Algorithm Analysis*

This subsection analyses the effectiveness of the proposed adaptive controlling strategy. To this end, two new test cases are designed. In the first test case, the CGP-II focuses on solving the 5-parity problem, while the CGP-I focuses on solving the small scale majority-on problem. Similarly, in the second test case, the CGP-II focuses on solving the 5-majority problem, while the CGP-I focuses on solving the small scale even-parity problem. Since the CGP-I focuses on solving a different kind of problem, the building blocks provided by it is likely to be ineffective. Hence, we design these two test cases to study whether the adaptive control strategy can appropriately evaluate the effectiveness of building blocks and whether the performance of Co-CGP degrade in such cases.

Fig. 7 shows the search trends of α, where the Co-CGP using the right building blocks are labelled as "Co-CGP/Positive", while those using the inappropriate building blocks are labelled as "Co-CGP/Negative". It can be observed that the α of "Co-CGP/Positive" quickly increases to 1, which indicates that the building blocks are useful to generate the promising offspring. Meanwhile,the α of "Co-CGP/Negative" quickly converges to a small value, indicating that the building blocks are ineffective to generate promising offspring in the search. The α of "Co-CGP/Negative" in the first cases converges to a larger value than the second case since the building blocks in the first case has less negative effects than in the second case. Note that the algorithm terminates when a global optimal solution is attained, and the α of subsequent generations is set equal to the value in the current generation. Hence the α becomes stable in the final stage of the evolution process. Generally, the above results demonstrate that the proposed adaptive controlling strategy has been effective for evaluating the effectiveness of the identified building blocks.

Fig. 8 illustrates the trends of the best fitness along the search, where "Co-CGP/Negative-static" are results of Co-CGP without the adaptive control strategy (i.e., $\alpha = 1$). It can be observed that, on both test cases, "Co-CGP/Negative" is slightly worse than "CGP" in the early generations, but are the same in the later generation. Meanwhile, without the adaptive control strategy,"Co-CGP/Negative-static" is significantly worse than "CGP". These results demonstrate that, owning to the adaptive control strategy, the performance of Co-CGP did not degrade too much when using ineffective building blocks.

V. Conclusion

This paper proposed a novel Co-evolutionary Cartesian Genetic Programming (Co-CGP), which can automatically gain high-order knowledge to accelerate the search. The proposed Co-CGP has been validated on eight test instances with different complexities. The experimental results demonstrate that the proposed method can significantly improve CGP in terms of search efficiency and accuracy. In this paper, the adaptive co-evolutionary mechanism was implemented based on the CGP. One future research direction is to apply the proposed mechanism to improve other GP variants such as the Gene Expression Programming (GEP). Besides, in this study, the problem to be solved and the sub problems are exactly of the same type, which limits the practical application of the proposed framework. In future, we plan to extend our framework to evolve multiple problems of different types.

Acknowledgment

This work is supported under the National Natural Science Foundation of China (Grant Nos. 61602181, and 61772142), Program for Guangdong Introducing Innovative and Enterpreneurial Teams (Grant No. 2017ZT07X183), and Pearl River S&T Nova Program of Guangzhou (Grant No. 201806010059).

References

[1] J. R. Koza, *Genetic Programming: vol. 1, On the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.

[2] J. Zhong, L. Feng, and Y.-S. Ong, "Gene expression programming: a survey," *IEEE Computational Intelligence Magazine*, vol. 12, no. 3, pp. 54–72, 2017.

[3] C. Zhou, W. Xiao, T. M. Tirpak, and P. C. Nelson, "Evolving accurate and compact classification rules with gene expression programming," *IEEE Trans. Evol. Comput.*, vol. 7, no. 6, pp. 519–531, 2003.

[4] P. G. Espejo, S. Ventura, and F. Herrera, "A survey on the application of genetic programming to classification," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 40, no. 2, pp. 121–144, 2010.

[5] M. Schmidt and H. Lipson, "Distilling free-form natural laws from experimental data," *science*, vol. 324, no. 5923, pp. 81–85, 2009.

[6] J. Zhong, Y. Ong, and W. Cai, "Self-learning gene expression programming," *Evolutionary Computation, IEEE Transactions on*, vol. 20, no. 1, pp. 65–80, Feb 2016.

[7] J. Zhong, W. Cai, M. Lees, and L. Luo, "Automatic model construction for the behavior of human crowds," *Applied Soft Computing*, vol. 56, pp. 368–378, 2017.

[8] S. Ventura and J. M. Luna, *Pattern Mining with Genetic Algorithms*. Springer, 2016.

[9] P. J. Angeline and J. Pollack, "Evolutionary module acquisition," in *Proceedings of the second annual conference on evolutionary programming*. Citeseer, 1993, pp. 154–163.

[10] J. R. Koza, *Genetic programming II: automatic discovery of reusable programs*. MIT press, 1994.

[11] R. Meuth, M.-H. Lim, Y.-S. Ong, and D. C. Wunsch II, "A proposition on memes and meta-memes in computing for higher-order learning," *Memetic Computing*, vol. 1, no. 2, pp. 85–100, 2009.

[12] Y. Kameya, J. Kumagai, and Y. Kurata, "Accelerating genetic programming by frequent subtree mining," in *Proc. Genetic Evol. Comput. Conf.* ACM, 2008, pp. 1203–1210.

[13] K. E. Kinnear Jr, "Alternatives in automatic function definition: A comparison of performance," *Advances in Genetic Programming*, pp. 119–141, 1994.

[14] T. Van Belle and D. H. Ackley, "Code factoring and the evolution of evolvability." in *Proc. Genetic Evol. Comput. Conf.*, vol. 2, 2002, pp. 1383–1390.

[15] J. A. Walker and J. F. Miller, "The automatic acquisition, evolution and reuse of modules in cartesian genetic programming," *IEEE Trans. Evol. Comput.*, vol. 12, no. 4, pp. 397–417, 2008.

[16] J. Zhong, L. Feng, W. Cai, and Y. Ong, "Multifactorial genetic programming for symbolic regression problems," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. In press, 2018.

[17] M. Iqbal, W. N. Browne, and M. Zhang, "Reusing building blocks of extracted knowledge to solve complex, large-scale boolean problems," *IEEE Trans. Evol. Comput.*, 2013 (in pressing).

[18] J. F. Miller and P. Thomson, "Cartesian genetic programming," in *Genetic Programming*. Springer, 2000, pp. 121–132.

[19] J. A. Walker, J. F. Miller, P. Kaufmann, and M. Platzner, *Problem Decomposition in Cartesian Genetic Programming*. Springer, 2011.

[20] T. Yu and J. Miller, "Neutrality and the evolvability of boolean function landscape," in *Genetic programming*. Springer, 2001, pp. 204–217.

[21] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong *et al.*, "Genetic programming needs better benchmarks," in *Proc. Genetic Evol. Comput. Conf.* ACM, 2012, pp. 791–798.

[22] M. Iqbal, W. N. Browne, and M. Zhang, "Extending learning classifier system with cyclic graphs for scalability on complex, large-scale boolean problems," in *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*. ACM, 2013, pp. 1045–1052.

[23] D. Jackson and A. P. Gibbons, "Layered learning in boolean gp problems," in *Genetic Programming*. Springer, 2007, pp. 148–159.