Soft Comput (2016) 20:3537–3548
DOI 10.1007/s00500-015-1710-9

Model approach to grammatical evolution: theory and case study

Pei He[1,2,3] · Zelin Deng[3] · Houfeng Wang[4] · Zhusong Liu[5]

Published online: 23 May 2015

Abstract Many deficiencies with grammatical evolution (GE) such as inconvenience in solution derivations, modularity analysis, and semantic computing can partly be explained from the angle of genotypic representations. In this paper, we deepen some of our previous work in visualizing concept relationships, individual structures and total evolutionary process, contributing new ideas, perspectives, and methods in these aspects; reveal the principle hidden in early work so that to develop a practical methodology; provide formal proofs for issues of concern which will be helpful for understanding of mathematical essence of issues, establishing of an unified formal framework as well as practical implementation; exploit genotypic modularity like modular discovery systematically which for the lack of supporting mechanism, if not impossible, is done poorly in many existing systems, and finally demonstrate the possible gains through semantic analysis and modular reuse. As shown in this work, the search space and the number of nodes in the parser tree are reduced using concepts from building blocks, and concepts such as the codon-to-grammar mapping and the integer modulo arithmetic used in most existing GE can be abnegated.

Keywords Genetic programming · Grammatical evolution · Finite state automaton · Model

Communicated by V. Loia.

✉ Pei He
bk_he@126.com

Zelin Deng
zl_deng@sina.com

Houfeng Wang
wanghf@pku.edu.cn

Zhusong Liu
25421944@qq.com

1 School of Computer Science and Educational Software, Guangzhou University, Guangzhou 510006, People's Republic of China

2 Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, People's Republic of China

3 School of Computer and Communication Engineering, Changsha University of Science and Technology, Changsha 410114, People's Republic of China

4 Institute of Computational Linguistics, Peking University, Beijing 100080, People's Republic of China

5 School of Computer Science and Technology, Guangdong University of Technology, Guangzhou 510006, People's Republic of China

1 Introduction

Types (Pierce 2002) and recursion (Boolos et al. 2002) have long been two important concepts in programming theories. Many successful solutions to problems in the areas of language design and programming techniques are more or less a matter of them. Therefore, it would seem to be valuable to apply these broad ideas to improve algorithm development and analysis in newer areas of programming research, such as genetic programming (GP), which is the application of evolutionary computing techniques to the induction of program code.

For the work described in this paper, three important developments in genetic programming form the basis. The first of these is the work of Koza (1992), who carried out the first substantial studies of GP across many problem areas and created the canonical tree-based form of GP that is the foundation for many variants of and applications of GP (Montana 1995; O'Neill and Ryan 2001; Ferreira 2001; He et al. 2011a, b; Du et al. 2014; Alfonseca and Gil 2013; Burbidge and Wilson 2014; Fernandez-Blanco et al. 2013; Oltean et al. 2009; Howard et al. 2011; Harman et al. 2012; Langdon and Harman 2015; Mckay et al. 2010) even today. The notion of types was introduced into GP by Montana (1995) in the Strongly Typed Genetic Programming (STGP) system. Finally, the development of grammatical evolution (GE) by O'Neill and Ryan (2001) provided a way to create typed programs in a GP-style process in an arbitrary language defined by a grammar. With canonical GP, Koza (1992) deals with compositions of functions, variables and constants with little attention to type constraints, thus proposing the closure problem as necessary condition for GP to work. To overcome these issues and allow the use of a system of types in GP, Montana (1995) allowed the typing terminals and operator arguments in the second phase of the development process of GP, presenting the so-called STGP approach; however, no basic change was made to the program representations when compared to the approach of Koza. Finally, the work by O'Neill and Ryan (2001) on GE aroused a wide range of interest among GP researchers and practitioners (Burbidge and Wilson 2014; Hugosson et al. 2010; Oltean et al. 2009; Risco-Martin et al. 2014; Wilson and Kaur 2009). In fact, compared to other two kinds of GP methods discussed, grammar-related approaches can be best suited to type descriptions and recursive program generations.

Up to now, GE has been applied successfully in many areas, which include financial prediction, pattern recognition, symbolic regression, robot control, etc. (Oltean et al. 2009; O'Neill and Ryan 2001). For instance, Gavrilis et al. (2008) proposed a GE-based method for improving classification accuracy. This evolutionary method is among the most recent methods of constructing new features from the original set of primitive features. Dempsey et al. (2006) employ GE to evolve best rules in a dynamic environment for trading, thereby forming an adaptive trading system. This sets their work apart from traditional ones which often use rules statically over test data. Hugosson et al. (2010) have deeply explored genotype representations of GE and compared the performance advantage between binary and integer forms of GE. However, when taking into consideration all GE-related works, there are the following drawbacks raised among them. The questions are that convergence comes at a price of efficiency and readability and that the building block, a frequently used redundancy-related concept, is hard to use in GE.

Of course, these problems have been initially studied in our previous work (He et al. 2011b) based on model approach (He et al. 2008, 2011a). In the present paper, we will deepen them by contributing new ideas, perspectives, and methods in the following aspects: reveal the principle hidden in early work so that to develop a practical methodology; provide formal proofs for issues of concern which, to our knowledge, will be helpful for understanding of mathematical essence of issues, establishing of an unified formal framework as well as practical implementation; exploit genotypic modularity like modular discovery systematically; and demonstrate the possible gains through semantic analysis and modular reuse. We solve them by modeling the syntactically usable information of GE as an automaton (Aho et al. 2007; Hopcroft et al. 2008). The rest of this paper is organized as follows: In Sect. 2, we introduce GE as well as its related problems. Sects. 3, 4, 5 and 6 deal with modeling problems, experiments, modularity and discussions, respectively. Finally in Sect. 7, we conclude our work and sketch out future directions for this work.

2 Preliminaries

Grammatical evolution resembles canonical GP (Koza 1992) in the use of an evolutionary process to automatically generate computer programs, but differs in the use of linear genotypic binary strings that are transformed into functional phenotypic programs in light of a grammar. Since GE outperforms GP in many aspects, such as easy for delineation of both type and domain knowledge, it has been widely taken up both by researchers and practitioners. Before deeply investigating into GE from modeling perspective, we will summarize its characteristics and define necessary concepts as follows.

2.1 Grammar and programming language

Definition 1 (*Context-free Grammar*; Aho et al. 2007) A context-free grammar $G = (V_{\mathrm{N}}, V_{\mathrm{T}}, B, P)$ consists of four components: (a) a set V_{N} of non-terminal symbols or syntactical variables; (b) a set V_{T} of terminal symbols; (c) a designation of one of the non-terminals in V_{N} as the start symbol B; (d) a set P of productions applied for generation of programs. Each production is of the form $A \to \alpha$. When A has many alternatives, we shall treat them differently in the following discussion. Note that context-free grammar is also shortened to grammar in the later sections.

Definition 2 (*Derivations*) Let $G = (V_{\mathrm{N}}, V_{\mathrm{T}}, B, P)$ be a grammar with $A \to \gamma \in P$, and $\alpha A\beta \in (V_{\mathrm{N}} \cup V_{\mathrm{T}})^*$. A *direct derivation* of $\alpha\gamma\beta$ from $\alpha A\beta$, denoted $\alpha A\beta \Rightarrow \alpha\gamma\beta$, is a substitution of γ for some A in $\alpha A\beta$. Particularly, we call $\alpha A\beta \Rightarrow \alpha\gamma\beta$ the leftmost derivation, denoted $\alpha A\beta \xRightarrow[lm]{A\to\gamma} \alpha\gamma\beta$, if A does not appear in α. By the way, $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ and $\alpha_1 \xRightarrow[lm]{} \alpha_2 \xRightarrow[lm]{} \cdots \xRightarrow[lm]{} \alpha_n$ are abbreviated to $\alpha_1 \xRightarrow{*} \alpha_n$ and $\alpha_1 \xRightarrow[lm]{*} \alpha_n$, respectively. Derivations with no production involved are referred to as *zero derivations*.

Definition 3 (δ_A^γ) Let $G = (V_{\mathrm{N}}, V_{\mathrm{T}}, B, P)$ be a grammar with $A \to \gamma \in P$, and $\delta \in (V_{\mathrm{N}} \cup V_{\mathrm{T}})^*$. δ_A^γ is a substitution of γ for the leftmost occurrence of A in δ. Particularly, we define $\delta_A^\gamma = \delta$ for $A \notin \delta$ and regard $\delta_A^\gamma = \delta$ as a zero derivation (i.e. $\delta = \delta$).

Definition 4 (*Sentential forms*) Given a grammar $G = (V_N, V_T, B, P)$ and a string $\alpha \in (V_N \cup V_T)^*$, α is a sentential form of G, if $B \overset{*}{\Longrightarrow} \alpha$ (α is also called a sentence if $B \overset{*}{\Longrightarrow} \alpha \in V_T^*$). α is a LM sentential form, provided all direct derivations involved in $B \overset{*}{\Longrightarrow} \alpha$ are leftmost ones.

Definition 5 (*Language*) Let $G = (V_N, V_T, B, P)$ be a grammar. The language commonly used in compiler constructions is $LM(G)=\{\alpha \in V_T^* \mid \alpha$ is a LM sentential form of $G\}$.

As for $LM(G)$, one can refer to standard texts (Aho et al. 2007; Hopcroft et al. 2008) for the details.

2.2 Classical grammatical evolution

Classical grammatical evolution (CGE or GE for short in the following discussion) is a combination of genetic algorithm and context-free grammar. In principle, it uses a genotype-to-phenotype mapping to interpret a string of codons (integers in [0, 255], usually represented as 8 bits) as certain sentential forms given the context-free grammar. The important rule almost all existing GE obeyed in choosing production for the leftmost derivation is the use of natural transaction (i.e. natural binary encoding) and modulo translation (O'Neill and Ryan 2001), i.e. production applied is determined by this formula: (codon integer value) mod (number of rules of X), where X stands for some nonterminal. Once sentences are successfully evolved in this way, so are the functional phenotypic programs. Theoretically, this mapping mechanism works for evolving programs in any language. For the details of algorithmic structure of GE, one can refer to these papers (O'Neill and Ryan 2001; Wilson and Kaur 2009) or Sect. 3.4. In the following sections, we will address such GE issues as modeling issues, space reduction, semantic computing and effective evaluation, etc. Some of them have long been neglected in GE researches.

3 Modeling grammatical evolution

3.1 Principle

The ambition to establish a visualized formal GP can be traced back to our previous work (He et al. 2008, 2011a, b). As revealed in the published literature, most of existing GP methods employed in software engineering are mainly developed on the basis of software testing, thus providing few means to address such important programming issues as semantics, correctness, etc. GE, to some extent, can help with some conveniences by the aid of its embedded grammar system, but still it is not as easy as one might imagine. Occasionally, these approaches may get stuck in some predicaments like aiming to evaluate infinite cycles. In view of this, the original model approach (He et al. 2008, 2011a) was proposed to combine Hoare-logic-style assertion-based specifications and model checking within a GP framework (Harman et al. 2012). In fact, semantic-related genetic programming has received great interests in recent years, and gradually emerging into a popular research issue (Krawiec 2014; Vanneschi et al. 2014).

Model-based GP framework concerns the use of visualized techniques like finite state transition diagram to chart evolution processes. The modeling ideas are summarized from the following aspects. They offer many features most existing counterparts do not cover, such as easiness in genotype analysis, modular representation, functional reuse, visualized structure study, and effective implementation:

- Define search space as a set of generalized Hoare formulae [in the case of HGP (He et al. 2011a)] or a set of grammatical derivations [in the case of MGE (He et al. 2011b)];
- Model search space in the context of relations of subsets and transition diagram;
- Search under the concerned model for the desired gene or solution through using a well defined heuristic algorithm. For the case of HGP, gene is a program represented by certain path of the diagram; for the case of MGE, gene represented also by path is a sequence of productions from which the expected program can be grammatically derived.

3.2 Model and existence theory

Definition 6 (*Justification*) Let $G = (V_N, V_T, B, P)$ be a grammar, and $\alpha, \beta \in (V_N \cup V_T)^*$. A sequence $s = p_1 p_2 \ldots p_n$ of productions justifies the derivations $\alpha \overset{*}{\Longrightarrow} \beta$, if s, α, β can establish the series of derivations $\alpha \underset{lm}{\overset{p_1}{\Longrightarrow}} \alpha_1 \underset{lm}{\overset{p_2}{\Longrightarrow}} \cdots \underset{lm}{\overset{p_{n-1}}{\Longrightarrow}} \alpha_{n-1} \underset{lm}{\overset{p_n}{\Longrightarrow}} \beta$ or $\alpha \underset{lm}{\overset{s}{\Longrightarrow}} \beta$.

Definition 7 (*ε-Equivalent*) Let $G = (V_N, V_T, B, P)$ be a grammar, and $\alpha \in (V_N \cup V_T)^*$. Two justifications j_1, j_2 of α are ε-equivalent, if they are exactly the same except for usage of ε (empty word).

Definition 8 (*Left most grammar model*) Let $G = (V_N, V_T, B, P)$ be a grammar. A finite state transition graph Gph $= \langle V, E\rangle$ with edges in E labeled either by productions (or production names) or empty (or ε) words is a left most grammar model of G, denoted $LMGM(G)$, if for $\alpha \in (V_N \cup V_T)^*$, α is a leftmost sentential form of $G \Leftrightarrow$ there exists a path starting at the initial state in $LMGM(G)$ such that the sequence s concatenated from edge labels along it justifies α, i.e. $B \underset{lm}{\overset{s}{\Longrightarrow}} \alpha$.

Theorem 1 (Existence theorem) *Given a grammar $G = (V_N, V_T, B, P)$, there exists a leftmost grammar model $LMGM(G)$ constructed by Algorithm 1.*

Algorithm 1 (Construction of $LMGM(G)$)

Input: a grammar $G = (V_N, V_T, B, P)$.

Output: $LMGM(G)$. Here S_B, ε stand for the initial vertex and zero derivation, respectively.

(1) Draw two vertices S_N and S_N^α for each production $N \to \alpha \in P$. When N has many alternatives, we should treat them separately as different production.
(2) Draw an ε arrow from V to V for each vertex V of step 1;
(3) Draw an arrow from S_N to S_N^α for vertices of step 1 if $N \to \alpha \in P$; naming the arrow either with the production or the production name.
(4) Calculate Follow(X) for all X in V_N as follows:

   A. $Y \in$ Follow(X), if there exists a production $A \to \cdots X\alpha Y \cdots \in P$ with $\alpha \in V_T^*$, and $X, Y \in V_N$;
   B. Follow$(A) \subseteq$ Follow(X) , if $A \to \cdots X\alpha \in P$. Where $A, X \in V_N, \alpha \in V_T^*$.

(5) Calculate $LMC(S_M^\alpha)$ for all states of the form S_M^α as follows:

   A. $LMC(S_M^\alpha) = \{A\}$, if $\alpha \notin V_T^*$ and A is the leftmost non-terminal symbol of α;
   B. $LMC(S_M^\alpha) =$ Follow(M), if $\alpha \in V_T^*$.

(6) Draw an ε arrow from S_M^α to S_N , for every state S_N with subscript $N \in LMC(S_M^\alpha)$.

Proof The proof is divided into three parts:

Part 1: proving that Follow$(X) = \{Y \in V_N | B \overset{*}{\Longrightarrow} \cdots X\alpha Y \ldots, \alpha \in V_T^*\}$, which means step 4 of Algorithm 1 finds out all possible nonterminals appearing on the right side of X in some sentential form of G with no nonterminal lying between them. By the meaning of Follow(X), we have, for X in V_N, that

$Z \in$ Follow$(X) \Leftrightarrow Z \in \{Y \in V_N \mid B \overset{*}{\Longrightarrow} \cdots X\alpha Y \cdots , \alpha \in V_T^*\}$

$\Leftrightarrow$ (a) $\exists A \to \cdots X\alpha Z \cdots$ in P with $\alpha \in V_T^*$ or (b) $\exists A \to \cdots X\alpha \in P$ with $\alpha \in V_T^*$ such that Z appears on the right side of A in some sentential form of G with no nonterminal lying between them, i.e. Z in Follow(A) (recalling the meaning of Follow(A)).

Part 2: proving that $LMC(S_M^\alpha) = \{X \in V_N \mid \exists \gamma, \delta \in V_T^*$,such that $B \underset{lm}{\overset{*}{\Longrightarrow}} \gamma M \cdots \underset{lm}{\overset{M\to\alpha}{\Longrightarrow}} \delta X \cdots$ hold$\}$, which means step 5 of Algorithm 1 finds out all possible nonterminals whose derivation, as far as the leftmost direct derivation is concerned, follows immediately the using of $M \to \alpha$.

By the meaning of $LMC(S_M^\alpha)$, and for M in V_N we have that Z in $LMC(S_M^\alpha) \Leftrightarrow Z \in \{X \in V_N \mid \exists \gamma, \delta \in V_T^*$ such that $B \underset{lm}{\overset{*}{\Longrightarrow}} \gamma M \cdots \underset{lm}{\overset{M\to\alpha}{\Longrightarrow}} \delta X \cdots$ hold$\} \Leftrightarrow$ (a) Z appears in α as the leftmost nonterminal, when $\alpha \notin V_T^*$, or (b) $\exists \gamma, \delta \in V_T^*$ such that $B \underset{lm}{\overset{*}{\Longrightarrow}} \gamma M \delta Z \ldots$, when $\alpha \in V_T^*$,i.e. $Z \in$ Follow(M) (recalling the meaning of Follow(M)).

By the proofs of part 1 and part 2, it follows that Follow(X), $LMC(S_M^\alpha)$ of Algorithm 1 define all the vertex pairs possibly to be connected.

Part 3: Let $L(LMGM(G))$ be the language accepted by $LMGM(G)$, and $J = \{j \mid j$ is a sequence of productions justifying some sentential form $\alpha \in (V_N \cup V_T)^*$ in $G\}$. To prove that $J_k \in J \Leftrightarrow$ there exists an ε-equivalent J_k' with J_k' in $L(LMGM(G))$. The proof goes by induction on the structures of derivations. By the way, treat all vertices as final states when in need, and S_B as the start state.

(i) Induction base: for zero derivation, and each production $p \in \{B \to \alpha \mid B \to \alpha \in P\}$, the proof is trivial.
(ii) Induction step: let $J_k p \in J$. Then p is either an ε or some production, say $X \to \delta$ in P. By induction hypothesis, we have $J_k \in J \Leftrightarrow$ there exits its ε-equivalent $J_k' \in L(LMGM(G))$. This means there exists a path in $LMGM(G)$ starting at the initial state, and ending at some of its states, say Q, to justify what J_k justifies. According to Algorithm 1, we can easily prove that $J_k'\varepsilon \in L(LMGM(G))$ for the case of $p = \varepsilon$. For the case of $p = X \to \delta \in P$, again by Algorithm 1, there must exist a path in $LMGM(G)$ of the form $S_X \overset{p}{\Longrightarrow} S_X^\delta$. Since $J_k p \in J \Leftrightarrow$ (a) Q is of the form S_Y with $S_Y = S_X$; or (b) Q is of the form S_M^α with $X \in LMC(S_M^\alpha)$. Obviously, the path constructed in $LMGM(G)$ from each of these two cases is actually $J_k p$, i.e. ε-equivalent to $J_k p$.

Note that some vertices like S_M^α, S_M^β in $LMGM(G)$ can be combined via the connection function into $\{S_M^\alpha, S_M^\beta\}$, if $LMC(S_M^\alpha) = LMC(S_M^\beta)$. This is reflected in Sect. 4. □

3.3 Complete mapping principle and automaton

A noteworthy problem raised in GE is the so-called incomplete mapping (see O'Neill and Ryan 2001 or Sect. 4.1). Although O'Neill and Ryan (2001) use the wrapping technique to overcome this problem, their method cannot guarantee that a (complete) sentence comprised of only terminals of some grammar can always be obtained.

In this subsection, the *complete mapping principle* is presented, which uses some default transformation or predefined rules to translate non-terminals of incomplete sentence into some syntactically valid strings of terminals of the concerned grammar directly. This thought can also be systematically stated in terms of automaton. It is reflected in Theorem 2.

Let $G = (V_N, V_T, B, P)$ be some context-free grammar where $V_N = \{B, N_1, N_2, \ldots, N_m\}$, $LMGM(G)$ be one of its leftmost grammar models. Again let I be a function or a interpretation $V_N \to V_T^*$ mapping non-terminal of V_N to some element of V_T^* that can be deduced from that non-terminal in G. Then the combination can be formed using the following procedure:

(i) Draw two vertices $S_{N\cup T}$ and S_T to represent the set of all sentential forms of G and the language $LM(G)$, respectively;
(ii) Draw the arrow I from vertex $S_{N\cup T}$ to vertex S_T, implying that I can recursively be used to transform any sentential form of $(V_N \cup V_T)^*$ into some sentence of V_T^*;
(iii) Draw ε arrows from $S_{N\cup T}$ to $S_{N\cup T}$ and S_T to S_T for vertices $S_{N\cup T}$ and S_T of step (i);
(iv) Draw arrows from each vertex of $LMGM(G)$ to vertex $S_{N\cup T}$, obtaining a finite state automaton FSA(G) with S_B (a special vertex of $LMGM(G)$, refer to Sect. 4) as its start state and vertex S_T its final state.

Theorem 2 *Let $G = (V_N, V_T, B, P)$ be a context-free grammar, $LMGM(G)$ and FSA(G) be its grammar model and corresponding finite state automaton obtained above. Then $\alpha \in LM(G) \Leftrightarrow$ there exists a path β I in FSA(G) from S_B to S_T with: (a) β is a path starting from S_B in $LMGM(G)$; (b) conducting derivations from start symbol B of G based on β production by production followed by the application of the default mapping principle to non-terminals of the obtained result will lead to α.*

3.4 Model based grammatical evolution

Unlike CGE which regards genotypes as sequences of codons, Model-based grammatical evolution, (MGE), generates computer programs directly from genotypes comprised of production names,say 1a1d4a2c1c3b. In principle, MGE generates computer programs in any language as follows (He et al. 2011b):

(1) Constructing the grammar model, say $LMGM(G)$, as described above for some given grammar G;
(2) Initializing both the developing program (a sentential form of G) and genotypes as, respectively, the start symbol B, and sequences of productions consistent with the concerned grammar model;
(3) Evolving genotype and executing the following steps for evolved genotype (a sequence of productions) repeatedly unless the developing program becomes a string comprised of only terminal symbols in the concerned grammar or some terminal condition is satisfied:
    (a) Reading a production or a rule name from the genotype;
    (b) Making the leftmost derivation for the developing program using the rule obtained in (a);
    (c) Back to (a) or executing (d) for some condition, say, whether the end of the concerned genotype has been arrived at by the rule interpretation procedure;
    (d) Applying the complete mapping procedure which replaces all non-terminal symbols in the developing program with predefined strings of terminal symbols.

Here, codon and modulo arithmetic calculations which play critical roles in CGE are omitted. MGE is practically implemented in Sect. 4 in building blocks, called Building block based GE (BGE).

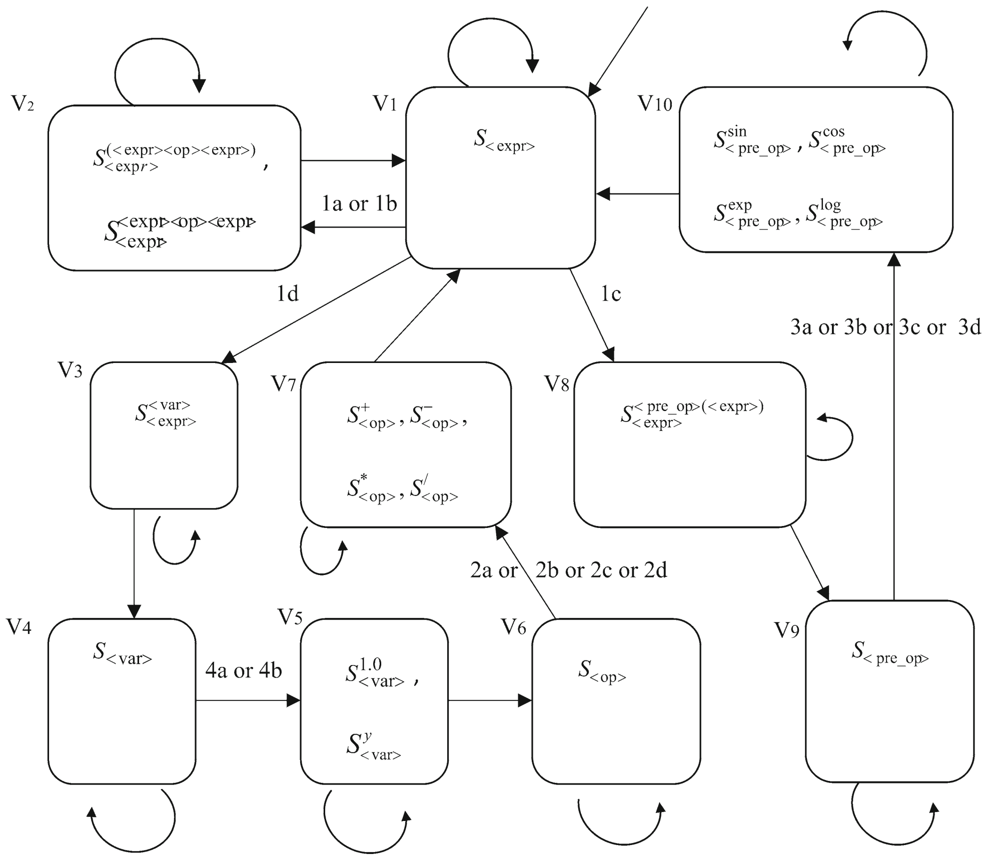
4 Experiments

We first provide some of our previous experiment results (He et al. 2011b) and then make deep analysis in light of modularity. The questions examined are the symbolic regression problems given in the work of O'Neill and Ryan (2001) and that of Oltean and Grosan (2003). To make sure that this method is a competitive approach to grammatical evolution, an analytical comparison with CGE, IGE (Hugosson et al. 2010), and PIGE (Fagan et al. 2010) is also conducted for effectiveness. The particular functions examined with 20 input values −1, −0.9, −0.8, −0.76, −0.72, −0.68, −0.64, −0.4, −0.2, 0, 0.2, 0.4, 0.63, 0.72, 0.81, 0.90, 0.93, 0.96, 0.99, 1 in the range [−1..1] are

$$f(y) = y^4 + y^3 + y^2 + y \tag{1}$$
$$g(y) = \sin(y^4 + y^2) \tag{2}$$
$$h(y) = \sin(\exp(\sin(\exp(\sin(y))))) \tag{3}$$

The grammar used in this problem is $G = (V_N, V_T, B, P)$. Where $V_N = \{\text{expr, var, op, pre\_op}\}$, $V_T = \{\sin, \cos, \exp, \log, +, -, *, /, y, 1.0, (,)\}$, $B = \langle\text{expr}\rangle$, and P can be represented as

(1)	⟨expr⟩	::= ⟨expr⟩⟨op⟩⟨expr⟩	(1a)
		| (⟨expr⟩⟨op⟩⟨expr⟩)	(1b)
		| ⟨pre_op⟩(⟨expr⟩)	(1c)
		| ⟨var⟩	(1d)
(2)	⟨op⟩	::= +	(2a)
		| −	(2b)
		| ∗	(2c)
		| /	(2d)



V2
$S_{<expr>}^{(<expr><op><expr>)}$,
$S_{<expr>}^{<expr><op><expr>}$
V1
$S_{<expr>}$
V10
$S_{<pre\_op>}^{sin}$, $S_{<pre\_op>}^{cos}$
$S_{<pre\_op>}^{exp}$, $S_{<pre\_op>}^{log}$
1a or 1b
1d
1c
3a or 3b or 3c or 3d
V3
$S_{<expr>}^{<var>}$
V7
$S_{<op>}^{+}$, $S_{<op>}^{-}$,
$S_{<op>}^{*}$, $S_{<op>}^{/}$
V8
$S_{<expr>}^{<pre\_op>(<expr>)}$
2a or 2b or 2c or 2d
V4
$S_{<var>}$
4a or 4b
V5
$S_{<var>}^{1.0}$,
$S_{<var>}^{y}$
V6
$S_{<op>}$
V9
$S_{<pre\_op>}$

Fig. 1 The leftmost grammar model $LMGM(G)$ of $G = (V_N, V_T, B, P)$. Where *arrows without labels* stand for ε *arrows*. Labels represented by logic disjunctions can also be expressed as sets of rule names. For example, we can denote the rules 1a or 1b by {1a, 1b}. Node V_1 is the start state of the automaton; the final state is technically omitted here (He et al. 2011b)

(3)	⟨pre_op⟩	::= sin	(3a)
		| cos	(3b)
		| exp	(3c)
		| log	(3d)
(4)	⟨var⟩	::= y	(4a)
		| 1.0	(4b)

4.1 Overview of the method

The method solving the above-mentioned problems is divided into four steps:

(a) Model construction: according to the existence theorem, we obtain a leftmost grammar model as shown in Fig. 1 (He et al. 2011b). It can be clearly seen that the edges of this model two have at most four choices. So for any n, the search space for a genotype of length n has the upper bound $O(4^{\frac{n}{m}})$. Where $m(\geq 1)$ is some number such that the value of $\frac{n}{m}$ represents the total of genotypic components or productions which may have multiple alternatives in the concerned genotype.

(b) System implementation: since this system is implemented in building blocks, it can essentially be referred to as BGE, meaning building block based grammatical evolution. To facilitate the comparison with CGE, IGE, and PIGE, certain corresponding GE systems are implemented for this service too. The involved pre-defined complete mapping function $I : V_N \rightarrow V_T^*$ is:

$$I(X) = \begin{cases} y & X \text{ is } \langle \text{expr} \rangle \\ y & X \text{ is } \langle \text{var} \rangle \\ + & X \text{ is } \langle \text{op} \rangle \\ \text{sin} & X \text{ is } \langle \text{pre\_op} \rangle \end{cases}$$

(c) Parameter settings: The major parameters employed are as follows: The parameter Runs (=100) defines the number of runs that will be conducted for each experiment on a particular problem. *Generation size*100; *Probability of crossover* 0.9; *Crossover model* two-point; *Population size* 50; *Probability of mutation* 0.15; *Mutation mode* block mutation; *Selection strategy* tournament; *Runs* 100; *Fitness evaluation* the least square error.

(d) Solving the problem: by running all the involved four GEs with the above parameters on the given sample dataset, we get the results shown in Figs. 2, 3, 4, 5, 6 and 7 (He et al. 2011b).

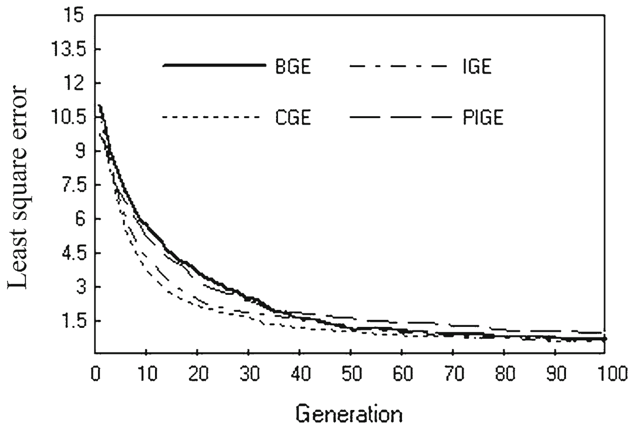


Fig. 2 Average fitness of 100 runs of the four GEs in Eq. (1)

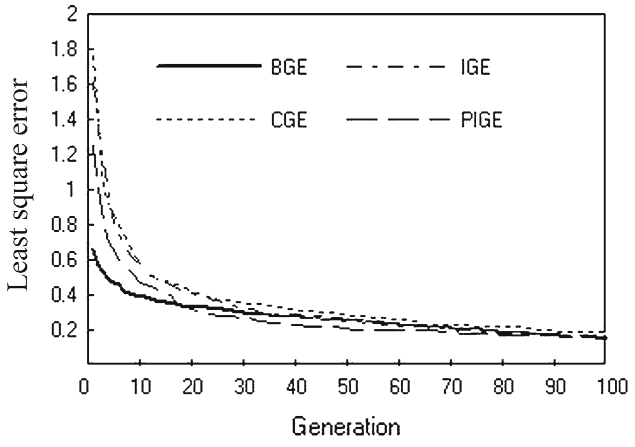


Fig. 3 Average fitness of 100 runs of the four GEs in Eq. (2)

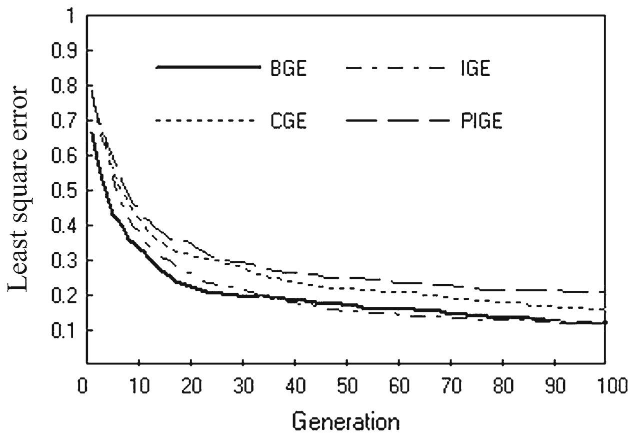


Fig. 4 Average fitness of 100 runs of the four GEs in Eq. (3)

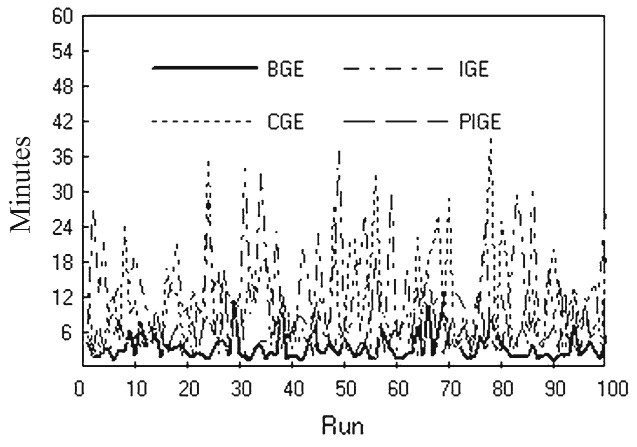


Fig. 5 Time used of 100 individual runs of the four GEs in Eq. (1)

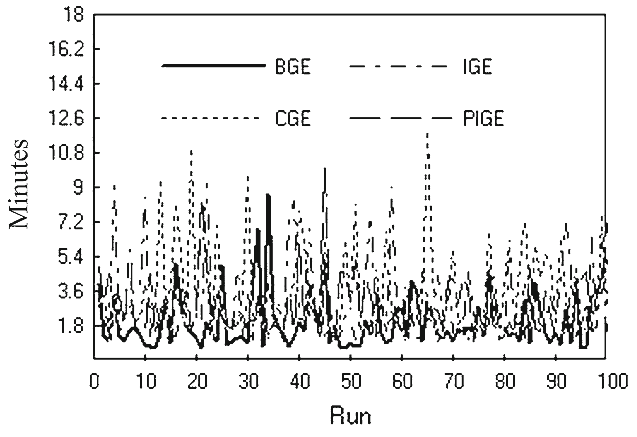


Fig. 6 Time used of 100 individual runs of the four GEs in Eq. (2)

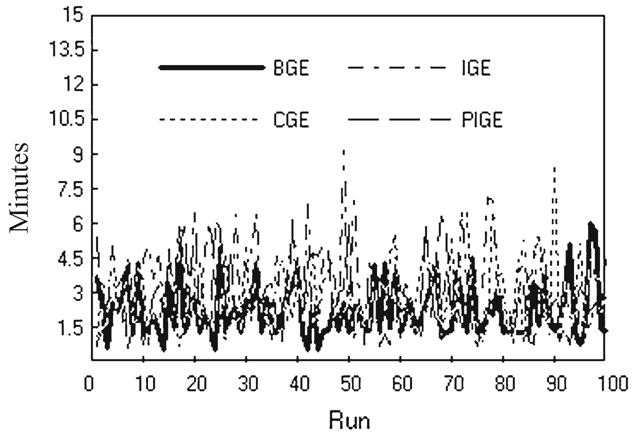


Fig. 7 Time used of 100 individual runs of the four GEs in Eq. (3)

4.2 Experimental results

As what the parameter Runs of step c indicates, we have made experiments on each particular problem for 100 runs. Figures 2, 3, and 4 illustrate the average fitness of 100 runs of BGE, CGE, IGE, and PIGE in the experiments. Figures

5, 6, and 7 compare these four methods with respect to time complexity. These figures, on the one hand, demonstrate that BGE has almost the same ability (refer to the shape of fitness profiles and the ultimate approximate solutions) as the other three GEs to generate the desired result, and the advantage in efficiency on the other. We will present further analysis and explanations in the following subsections.

4.3 Explanation

Building blocks (Swafford et al. 2011) are of great concern in the analysis of genetic programming. To guarantee the convergence of solutions, many approaches choose to prevent damage to building blocks of the chromosome as far as possible. In this subsection and Sect. 5, we not only define building blocks for our experiments, but also employ them to accelerate BGE, and explore why BGE is more effective than CGE. But what is also worth noticing is that the analysis result with CGE applies, due to the similarity of CGE, IGE, and PIGE in their mapping structures, to all of the involved GE variants.

4.3.1 Characteristics of method

To better understand BGE, the following topics to be further discussed in Sect. 4.3.2 should be clearly introduced.

Building blocks: By building blocks, we mean repeated genotypic patterns from which sub-expressions can be explicitly decoded. BGE is implemented in building blocks. Because there are three kinds of cycles starting and ending at vertex $S_{\langle\text{expr}\rangle}$ in Fig. 1, we naturally regard them as building blocks, named e, v, p, respectively. Here e, v, p are named after the first English letters of $[(\,]\,\langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle[\,)]$, $\langle\text{var}\rangle$ and $\langle\text{pre}-\text{op}\rangle(\langle\text{expr}\rangle)$, thus semantically telling these cycles apart. For simplicity, we will identify each vertex of Fig. 1 with a name, say $V_i(0 < i < 11)$, and interpret these denotations as follows.

e: the cycle $V_1V_2V_1$; v: the cycle $V_1V_3V_4V_5V_6V_7V_1$; p: the cycle $V_1V_8V_9V_{10}V_1$.

Since Fig. 1 delineates the relationships among productions of the grammar, these cycles stipulate which production can be followed by which other one and answer for our case whether a sequence of productions forms a justification (see Definition 6) of some derivation $\langle\text{expr}\rangle \overset{*}{\underset{lm}{\Longrightarrow}} \alpha(\alpha \in LM(G))$. For example, any derivation involved in a cycle p cannot be used in either e or v. However, elements of $\{e, v, p\}^*$ are allowed, summarizing all valid justifications of derivations.

Genotypic representation: One of the major differences between CGE and BGE is the representation. In view of the above fact, the genotypic representation is very simple in the case of BGE. In fact, elements of $\{e, v, p\}^*$ are all we need. So there is no necessity for validity checking.

Genetic operations: Two commonly used genetic operations such as crossover and mutation in BGE were implemented. Since there is no extra expense in validity checking in this case, their implementations are actually quite simple.Now, let us algorithmically depict the regular expression-based genetic operations. Note that all components, say f_i and h_i, involved in the following individuals are from $\{e, v, p\}$.

Crossover:

(1) Let $P_1 = f_1f_2\ldots\ f_m$, $P_2 = h_1h_2\ldots h_n$ be two individuals to cross over.
(2) Randomly choose two blocks, say $P_1[i..j]$ and $P_2[u..v]$, from P_1 and P_2 for swapping.
(3) Conduct two-point crossover on P_1, P_2 through constructing such individuals as $f_1f_2\ldots f_{i-1}h_u\ldots h_v\ f_{j+1}\ldots f_m$ and $h_1h_2\ldots h_{u-1}f_if_{i+1}\ldots f_jh_{v+1}h_{v+2}\ldots h_n$ for further use (the principle of single-point crossover is similar).

Mutation:

(1) Let $p = f_1f_2\ldots f_m$ be an individual to be mutated.
(2) Choose a block, say $p[i..j]$, from the sequence p for mutation.
(3) Replace f_k of p with some randomly chosen element, say x, in $\{e, v, p\}$ with $f_k \neq x$ for $i \le k \le j$.

4.3.2 Explanation of results

Why is BGE superior to CGE in these experiments, particularly with respect to time complexity? As is well known, CGE solves problems using the following steps: generating sequences of bits; executing some conversion algorithm to translate bits to codons; and deriving program in micro-steps within which numerous integer modulo operations are exploited to search through the search space (see Sect. 2). So if both the micro-step deducing sentential form from one single production and the search space can faithfully be expressed in building blocks and reduced, respectively, the search performance will naturally be improved.

Considering the leftmost grammar model $LMGM(G)$ of Fig. 1, it follows no matter what sequence of productions are employed in CGE for the program derivation, the sequence consists of three kinds of cycles in Fig. 1: e, v, p. These are the building blocks that are explored by BGE. As such, the genotype of each randomly generated individual of BGE differs from that of CGE, consisting of an element of $\{e, v, p\}^*$. As far as the phenotypic meaning of some q in $\{e, v, p\}^*$ is concerned, we will solve it through consultation of a dictionary for the semantics of a building block b of $\{e, v, p\}$. For instance, whenever p of some individual in $\{e, v, p\}^*$ is employed as a translation rule, $\langle\text{expr}\rangle$ can directly be interpreted as $\sin(\langle\text{exp}\rangle)$. However, CGE solves the same problem by successively applying derivations on such non-terminals as ⟨expr⟩ and ⟨pre_op⟩. So BGE, unlike CGE which takes into consideration all micro-inference steps in sentential form generations, works only on building blocks, simplifying parser tree constructions. Moreover, because genetic operations in BGE can represent those of CGE, they have no significant negative effect on the ultimate solution precision. Theorem 3 reflects this property. Then, when trying to solve the same problem, i.e. deriving the same sentence of some grammar, many calculations once necessary for CGE become unnecessary for the case of BGE. Finally, a more efficient evaluation algorithm can further be obtained from what is discussed in Sect. 5.

Theorem 3 *Given the grammar of Sect. 4 and a sentence S, there is a sequence c of productions for deriving S in CGE*$\Leftrightarrow$ *there exists a sequence b of building blocks with* $|b| \le |c|$ *for deriving S in BGE, where the symbol* $|x|$ *stands for the length of the sequence* x.

5 Case study on modularity of BGE

In this section, we will discuss some modularity issues related to efficient fitness evaluation. As we know, fitness evaluation is a costly part of real-world applications of GP and GE, typically making up the majority of total computational resources. Since the concept codon (a string of bits) of CGE has no meaning out of context, the same strings of codons often represent different kinds of semantic objects. So it is difficult to effectively realize fitness evaluation through just consulting codons and avoiding duplicated computations. However, BGE improves this.

As is known, each program (or phenotype or expression) can either be represented by a parse tree (or concrete syntax tree) or an abstract syntax tree. The major difference between these data structures is as follows: in the syntax tree, interior nodes represent programming constructs, while in the parse tree the interior nodes represent non-terminals. Thus to solve $(\sin(\cos(y))+\exp(y))+\log(\sin(\cos(y))+\exp(y))$, an effective approach to the evaluation of this expression is to avoid resolving the subexpression $\sin(\cos(y)) + \exp(y)$ or evaluating the same subtree repeatedly. So it is critical to find an effective approach for grouping similar operators to reduce the number of cases and subclasses of nodes in an evaluation of programs or expressions. This is our way forward. Because both kinds of trees can structurally be constructed from sequence of building blocks, we use Algorithm 2 (Fig. 8) to gather all the functional sequences of building blocks. Based on these sequences, the effective evaluation algorithm (Algorithm 3) can be obtained. Genotypes of CGE certainly do not offer such convenience. Figure 10 compares number of nodes of using or not using the node reduction technique in Eq. (1).

```
var s: string; r: set of string; x, y: integer;
// s is a genotype of {e1, e2, v1, v2, …, v8, p1, p2, p3, p4}*
procedure FindingSubexpressions(var x, y :integer; s: string);
  var   m, n: integer;
begin
  if x<=y then case s[x] of
              'v':  begin   y:= x +1; r:= r ∪ { copy(s, x, 2) } end ;
              'e':  begin
                       m:= x+2;   n:= y;
                       FindingSubexpressions(m, n, s);
                       n:= n+1;
                       FindingSubexpressions(n, y, s);
                       r:= r ∪ {copy(s, x, y-x+1)}
            //copy(s, k, j) stands for the substring s[k]… s[ j-k+1] of s
                     end;
              'p':  begin
                       m:= x+2;   FindingSubexpressions(m, y, s);
                       r:= r  ∪  {copy(s, x, y-x+1)}
                     end
            end
end;
begin
  let s in {e1, e2, v1, v2, …, v8, p1, p2, p3, p4}*, r be {}, x =1,
  and y be |s| (the length of the string s);
  call FindingSubexpressions(x, y, s)
  //obtaining generation rules of all subexpressions of s
end;
```

Fig. 8 Algorithm finding out generation rules of all subexpressions of some given expression

Before introducing the algorithms, some convention should be adopted for simplifying our description. Because a string from $\{e, v, p\}^*$ can be used as a generation rule for a program or expression, each component of the string must have a well-defined meaning. So, for simplicity, we represent the two cycles of e by $e1$ and $e2$; the eight cycles of v by $v1$, $v2, \ldots, v8$; and the four cycles of p by $p1$, $p2$, $p3$, and $p4$.

Algorithm 2 Let s be a string (genotype) of $\{e1, e2, v1, v2, \ldots, v8, p1, p2, p3, p4\}^*$. Solving all its substrings (or valid generation rules) for subexpression constructions as shown in Fig. 8.

Algorithm 3 Let s, say $e1e2p1p2v1p3v1p4e1p1p2v1p3v1$, be a string (genotype) of $\{e1, e2, v1, v2, \ldots, v8, p1, p2, p3, p4\}^*$ for generating some expression, say $(\sin(\cos(y)) + \exp(y)) + \log(\sin(\cos(y)) + \exp(y))$. Grouping all its substrings for effectively evaluation of that expression as shown in Fig. 9.

The algorithmic thought includes the following three steps:

(1) Apply Algorithm 2 to s; we can get generation rules of all subexpressions. For the case of $e1e2p1p2v1p3v1p4e1$

Fig. 9 Substrings obtained from Algorithm 2

The string *s*	the meaning	the list of step 1	corresponding subexpression	the list of step 2
*e*1	<expr>+<expr>	*v*1	y	*v*1
*e*2	(<expr>+<expr>)	*p*2*v*1	cos(y)	*p*2*v*1
*p*1	sin(<expr>)	*p*1*p*2*v*1	sin(cos(y))	*p*1*p*2*v*1
*p*2	cos(<expr>)	*v*1	y	
*v*1	y	*p*3*v*1	exp(y)	*p*3*v*1
*p*3	exp(<expr>)	*e*2*p*1*p*2*v*1*p*3*v*1	(sin(cos(y))+exp(y))	*e*2*p*1*p*2*v*1*p*3*v*1
*v*1	y	*v*1	y	
*p*4	log(<expr>)	*p*2*v*1	cos(y)	
*e*1	<expr>+<expr>	*p*1*p*2*v*1	sin(cos(y))	
*p*1	sin(<expr>)	*v*1	y	
*p*2	cos(<expr>)	*p*3*v*1	exp(y)	
*v*1	y	*e*1 *p*1*p*2*v*1*p*3*v*1	sin(cos(y)) + exp(y)	*e*1*p*1*p*2*v*1*p*3*v*1
*p*3	exp(<expr>)	*p*4*e*1*p*1*p*2*v*1*p*3*v*1	log(sin(cos(y)) + exp(y))	*p*4*e*1*p*1*p*2*v*1*p*3*v*1
*v*1	y	*s*	the desired expression	*s*

$p1p2v1p3v1$, we have a list of generation rules of subexpressions: $[v1, p2v1, p1p2v1, v1, p3v1, e2p1p2v1p3v1, v1, p2v1, p1p2v1, v1, p3v1, e1p1p2v1p3v1, p4e1p1p2v1p3v1, e1e2p1p2v1p3v1p4e1p1p2v1p3v1]$.

(2) Repeatedly delete rules from the list of step 1 until none of the rules appears more than once in the list. This can concretely be illustrated in Fig. 9 through the use of our example.

(3) Effectively evaluate the concerned expression based on the result, precisely the list, of step 2. For our case, only eight generation rules of the list must be evaluated and shared in the evaluation of $(\sin(\cos(y)) + \exp(y)) + \log(\sin(\cos(y)) + \exp(y))$.

Step 2 of Algorithm 3 is designed in terms of only one individual. When applying similar principles to all individuals involved in the history of evolution, we will get a significant performance improvement in fitness evaluations. This is measured in our experiment in Fig. 10. This comparison indicates that for each run of BGE in Eq. (1), there are only a few nodes or subexpressions that are truly worth tediously processing from scratch.

6 Discussion

A modeling approach has been extensively used by software engineering community to solve the different problems of software development including system representation, performance analysis, and solution description, etc. It benefits us in many ways, for example, presenting a human understandable description of some aspect of a system or presentation in a form that can be mechanically analyzed. Owing to that BGE is a software development approach and that it is not for purpose of the direct improvement on classical GE, but an extension to the earlier result, we will compare BGE and CGE in terms of model properties such as readability, transformation, and so on. Of course, similar comparisons can be conducted on IGE and PIGE, but they are not the essence of the issues. So, for brevity, the discussions center around the most important question, being developed only on BGE and CGE.

- Readability
  As we know, both BGE and CGE describe individuals in light of genotype and phenotype. Their major differences come from genotypic formation and genotypic decoding. Regarding formations, CGE relies on codons of 8 bits, and BGE (see Fig. 1) on building blocks or building block names such as the aforementioned *e, v, p*. Regarding decoding methods, CGE is more complex than BGE, using the integer modulo based mapping (codon integer value) mod (number of rules of *X*) (O'Neill and Ryan 2001) to search the vast phenotypic space, determine which production to employ at which time. This means a codon can convey nothing to us provided that the concerned leftmost non-terminal *X* is unknown at this moment. However, for any genotype of BGE, say *eepvvv* (refer to Sect. 4.3), we can read such meanings into it as using double *e*, one *p* and triple *v* blocks to derive a meaningful explanation. So as far as genotype is concerned, BGE has an advantage over CGE in readability. The benefits of this can be seen in analyses such as those shown in Sect. 5.
- Mechanical transformation
  Abstract representation and semantic transformation are key problems encountered in model approaches to software development. Readability gives BGE an advantage over CGE, and it is also the first step towards mechanical understanding or analysis. How can we effectively

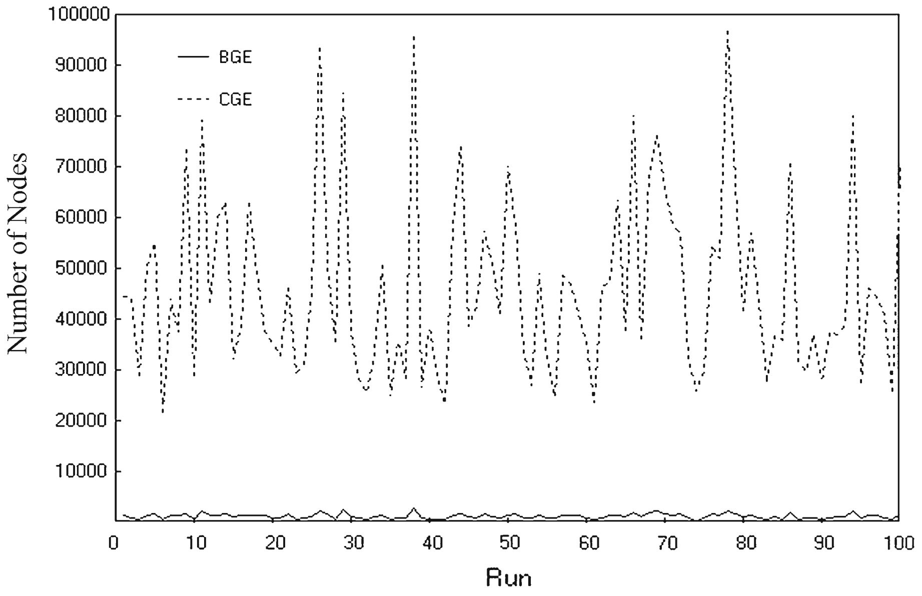


Fig. 10 Comparison of BGE with CGE with respect to node complexity

transform abstract representations to concrete implementation? When engaging in software reengineering and reconstruction, the Object Management Group (OMG) has once coined the unique concept Model-Driven Architecture (MDA) (France and Rumpe 2007) to depict the particular role of visual modeling tools for this activity. Another advantage of BGE lies in providing visual means for effective understanding of what message the grammar system conveys. For example, from Fig. 1, not only is it feasible to use such abstract representation as regular expression to express genotype, but also to technically implement efficient transformers for quickly obtaining phenotypic objects. Take the translation of $\langle \text{expr} \rangle$, when the cycle p is applied for this task, BGE looks the item $(p, \langle \text{expr} \rangle)$ up in a semantic dictionary, returning the concerned result (an element of $\{\sin(\langle \text{expr} \rangle), \cos(\langle \text{expr} \rangle), \log(\langle \text{expr} \rangle), \exp(\langle \text{expr} \rangle)\}$) directly. Instead, CGE achieves this through two grammatical derivation steps. Furthermore, the use of semantic dictionary in BGE opens out room for both application and imagination. For instance, either manually obtained semantics (or concise, simplified result) or some derivation-oriented bias information could be employed in dictionaries.

In short, BGE helps reach a new stage of understanding of GE. It has not only the potential to introduce new kinds of human–computer interactions into traditional CGE, but also the power to reveal the relationship between building block and CGEs micro-derivation step, to delineate the relations of the grammar both visually and logically. As shown in Fig. 1 of Sect. 4, it can also be seen that the hardness of problems depends precisely on two factors, i.e. the model and the problem itself. If problems can be depicted by the same model, their difficulties rely on themselves; otherwise, their models should also be taken into consideration. Nonetheless, we believe that this work will lead to more than just an extension of GE. It serves as a case study of how other concepts from software engineering and compiler construction can be brought into GE and GP. We will make further experiments with other extensive optimization problems such as security-related problems and some other real world problems like scheduling, services computing,cluster analysis in ambient network, etc. (Li et al. 2010, 2014; D'Apiec et al. 2014; Mokryani et al. 2013; Castiglione et al. 2015; Esposito et al. 2013; Habib and Marimuthu 2011) to certify the feasibility of this approach.

7 Conclusion

Grammatical evolution is an important automatic programming system consisting of a GA and a genotype-to-phenotype mapping. In this paper, we model GE by incorporating syntactical information into an automaton, focusing only on possibly valid compositions of productions. As such search space can be reduced, and parse tree construction and fitness evaluation can be simplified significantly in the case of the present approach, but functionality and expressiveness are still the same as before. Our future works will center on real-world applications, automatic detection and use of patterns or cycles,further comparisons of this approach with other GEs, and unifications of various GP variants.

Acknowledgments The research work was supported by National Natural Science Foundation of China (Grant No. 61170199, 61370117), the Scientific Research Fund of Education Department of Hunan Province, China (Grant No. 11A004),and the Guangzhou Zhujiang Science and Technology Future Fellow Fund (Grant No. 2012J2200094). Besides, He Pei would like to give special thanks to the late Prof. Kang Lishan and Prof. Tang Zhisong for introducing him to the area of evolutionary computation and Formal Methods.

Conflict of interest There are no conflicts of interest.

References

Aho AV, Lam MS, Sethi R, Ullman JD (2007) Compilers: principles, techniques, and tools, 2nd edn. Pearson Education, New York

Alfonseca M, Gil FJS (2013) Evolving an ecology of mathematical expressions with grammatical evolution. BioSystems 111(1):111–119

Boolos GS, Burgess JP, Jeffrey RC (2002) Computability and logic, 4th edn. Cambridge University Press, Cambridge

Burbidge R, Wilson MS (2014) Vector-valued function estimation by grammatical evolution. Inf Sci 258(1):182–199

Castiglione A, Pizzolante R, De Santis A, Carpentieri B, Castiglione A, Palmieri F (2015) Cloud-based adaptive compression and secure management services for 3D healthcare data. Future Gener Comput Syst 43C44(1):120–134

D'Apiec C, Nicola CD, Manzo R, Moccia V (2014) Optimal scheduling for aircraft departure. J Ambient Intell Human Comput 5(1):799–807

Dempsey I, O'Neill M, Brabazon A (2006) Adaptive trading with grammatical evolution. In: Proceedings of the 2006 IEEE congress on evolutionary computation, vol 1, pp 2587–2592

Du X, Ni YC, Xie DT, Yao X, Ye P, Xiao RL (2014) The time complexity analysis of a class of gene expression programming. Soft Comput

Esposito C, Ficco M, Palmieri F, Castiglione A (2013) Interconnecting federated clouds by using publish-subscribe service. Cluster Comput 16(4):887–903

Fagan D, O'Neill M, Galvan-Lopez E, Brabazon A, McGarraghy S (2010) An analysis of genotype–phenotype maps in grammatical evolution. In: EuroGP 2010, LNCS, vol 6021, pp 62–73

Fernandez-Blanco E, Rivero D, Gestal M, Dorado J (2013) Classification of signals by means of genetic programming. Soft Comput 17(1):1929–1937

Ferreira C (2001) Gene expression programming: a new adaptive algorithm for solving problems. Complex Syst 13(2):87–129

France R, Rumpe B (2007) Model-driven development of complex software: a research roadmap. In: Future of software engineering (FOSE2007) in international conference on software engineering (ICSE), vol 1, pp 37–54

Gavrilis D, Tsoulos IG, Dermatas E (2008) Selecting and constructing features using grammatical evolution. Pattern Recognit Lett 29(1):1358–1365

Habib SJ, Marimuthu PN (2011) Self-organization in ambient networks through molecular assembly. J Ambient Intell Hum Comput 2(3):165–173

Harman M, Mansouri SA, Zhang Y (2012) Search-based software engineering: trends, techniques and applications. ACM Comput Surv 45(1):11:1–11:61

He P, Kang LS, Fu M (2008) Formality based genetic programming. In: IEEE congress on evolutionary computation

He P, Kang LS, Johnson CG, Ying S (2011a) Hoare logic-based genetic programming. Sci China Ser F Inf Sci 54(3):623–637

He P, Johnson CG, Wang HF (2011b) Modeling grammatical evolution by automaton. Sci China Inf Sci 54(12):2544–2553

Hopcroft JE, Motwani R, Ullman JD (2008) Automata theory, languages, and computation, 3rd edn. Pearson Education, New York

Howard D, Brezulianu A, Kolibal J (2011) Genetic programming of the stochastic interpolation framework: convection diffusion equation. Soft Comput 15(1):71–78

Hugosson J, Hemberg E, Brabazon A, O'Neill M (2010) Genotype representation in grammatical evolution. Appl Soft Comput 10(1):36–43

Koza JR (1992) Genetic programming. MIT Press, Cambridge

Krawiec K (2014) Genetic programming: where meaning emerges from program code. Genet Program Evolvable Mach 15(1):75–77

Langdon WB, Harman M (2015) Optimizing existing software with genetic programming. IEEE Trans Evol Comput 19(1):118–135

Li J, Wang Q, Wang C, Cao N, Ren K, Lou WJ (2010) Fuzzy keyword search over encrypted data in cloud computing. In: Proceeding of the 29th IEEE international conference on computer communications (INFOCOM 2010), pp 441–445

Li J, Huang XY, Li JW, Chen XF, Xiang Y (2014) Securely outsourcing attribute-based encryption with checkability. IEEE Trans Parallel Distrib Syst 25(8):2201–2210

Mckay RI, Hoai NX, Whigham PA, Shan Y, O'Neill M (2010) Grammar-based genetic programming: a survey. Genet Program Evolvable Mach 11(3/4):365–396

Mokryani G, Siano P, Piccolo A (2013) Optimal allocation of wind turbines in microgrids by using genetic algorithm. J Ambient Intell Hum Comput 4(1):613–619

Montana DJ (1995) Strongly typed genetic programming. Evol Comput 3(2):199–230

Oltean M, Grosan C (2003) A comparison of several linear genetic programming techniques. Complex Syst 14(1):285–313

Oltean M, Grosan C, Diosan L, Mihaila C (2009) Genetic programming with linear representation: a survey. Int J Artif Intell Tools 19(2):197–239

O'Neill M, Ryan C (2001) Grammatical evolution. IEEE Trans Evol Comput 5(4):349–358

Pierce BC (2002) Types and programming languages. MIT Press, Cambridge

Risco-Martin JL, Colmenar JM, Hidalgo JI, Lanchares J, Diaz J (2014) A methodology to automatically optimize dynamic memory managers applying grammatical evolution. J Syst Softw 91(1):109–123

Swafford JM, O'Neill M, Nicolau M, Brabazon A (2011) Exploring grammatical modification with modules in grammatical evolution. In: EuroGP 2011, LNCS, vol 6621, pp 310–321

Vanneschi L, Castelli M, Silva S (2014) A survey of semantic methods in genetic programming. Genet Program Evolvable Mach 15(2):195–214

Wilson D, Kaur D (2009) Search, neutral evolution, and mapping in evolutionary computing: a case study of grammatical evolution. IEEE Trans Evol Comput 13(3):566–590