Christiansen Grammar Evolution: grammatical evolution with semantics

Ortega A., de la Cruz M., and Alfonseca M., *Escuela Politécnica Superior, Universidad Autónoma de Madrid*

***Abstract*— This paper describes Christiansen Grammar Evolution (CGE), a new Evolutionary Automatic Programming algorithm that extends standard Grammar Evolution (GE) by replacing context-free grammars by Christiansen grammars. GE only takes into account syntactic restrictions to generate valid individuals. CGE adds semantics to ensure that both semantically and syntactically valid individuals are generated. It is empirically shown that our approach improves GE performance and even allows the solution of some problems difficult to tackle by GE.**

***Index Terms*— Genetic algorithms, Languages, Formal languages, Automatic programming**

I. Introduction

THE design of programming languages, compilers and interpreters is one of the main topics in Theoretical Computer Science. Chomsky grammars [1] provide a powerful tool that allows the development of systematic algorithmic techniques. The Chomsky hierarchy shows the relationship between grammar types and their expressive power. For example, type 2 (or context free) and type 0 grammars are respectively related to pushdown automata and Turing machines. Type 0 languages can be found for any computable problem, while there is a proper subset of this set of problems that may be represented by type 2 languages, because pushdown automata cannot be considered universal computers.

Chomsky type 0 grammars are very difficult to handle (design and parse). Therefore, in spite of their greater power, they have been little used in computer science to formally specify high level programming languages. Context free grammars are used instead.

This approach poses a difficulty: since programming languages must be able to describe algorithms for any problem that a digital computer can solve (that is, any computable problem) it seems clear that something must be added to context free grammars to keep the expressiveness of Chomsky type 0 grammars. This difficulty is the reason for the rather artificial distinction between the syntax and the semantics of high level programming languages. The mandatory declarations of variables before their use, or the proper correspondence of type and number in the arguments of a function, are examples of constructions that depend on the context.

Several approaches to efficiently express every computable problem (that is, to reach the same expressive power as Chomsky type 0 grammars) by means of extended context-free grammars have been published since the sixties. They can be grouped according to their possession of the *adaptability property: (a grammar is said to be adaptable if it can be modified while it is being used)*. Attribute grammars [2] are the best-known non-adaptable grammars more complex than context free grammars. This paper uses Christiansen grammars, which are adaptable. More detailed classifications may be found in [3].

It is well known that all these formalisms have the same expressive power as type 0 Chomsky grammars. Thus, which one to use for a given problem depends only on the preferences of the programmer, and the ease of use of the formalism.

A. *Christiansen grammars*

Christiansen grammars [3, 4] are an extension of attribute grammars where the first attribute associated to every symbol is a Christiansen grammar.

The derivation relationship is redefined to make the model adaptable: the first attribute contains the rules applicable to the corresponding symbol. As with any other attribute, its value can be computed while the grammar is being used, thus the grammar may be changed on the fly.

Several formal notations have been used to describe Christiansen grammars. This paper follows that used in [3], which is very similar to typical attribute grammars and extended attribute grammars [5]. It is slightly more declarative, and explicitly specifies, for every attribute, whether it is inherited (↓) or synthesized (↑).

The full syntax is as follows:

-- Nonterminals are written in angled brackets and are followed by the parenthesized list of their attributes.

-- In the production rules, the names of the attributes are implicit. Their values are used instead. This syntax is similar

AKNOWLEDGEMENT: This work has been sponsored by the Spanish Ministry of Education and Science (MEC), project number TSI2005-08225-C07-06. The authors want to acknowledgement Manuel Cebrián’s help in the statistical analysis of the experiments.

Alfonso Ortega de la Puente, Marina de la Cruz Echeandía and Manuel Alfonseca are with the Departamento de Ingeniería Informática de la Escuela Politécnica Superior de la Universidad Autónoma de Madrid, España (e-mail: {alfonso.ortega,marina.cruz,manuel.alfonseca}@uam.es).

to that of variables in logic programming (Prolog, for example), where the names actually used stand for constraints between the variables. Semantic actions follow their corresponding production rule in brackets, where {} stands for no semantic action.

-- λ is the empty word.

-- As in logic programming, additional semantic actions, which cannot be expressed by the values of the attributes, follow the rule between brackets. These actions are usually written in pseudo code.

Neither attribute grammars nor Christiansen grammars restrict a priori the way in which their attributes may depend on other attributes. Nevertheless, attribute grammars where no inherited attribute depends on a synthesized attribute located at its right are well-known. They are expressive enough and the algorithms to handle them are easy to describe and to program. This paper refers only to Christiansen grammars which satisfy this condition.

The following example, borrowed from [3], describes a Christiansen grammar for a toy programming language. The following is a sample program in that language:

```
{
  int i;
  int j;
  i=j;
}
```

A program in this language has two sections (which may be empty) enclosed between brackets:

--The declaration of all the variables used in the second section. Only integer variables are allowed.

--The sequence of statements. Each statement ought to be an assignment between previously declared identifiers.

This context-dependent feature may be analyzed without any auxiliary symbol table.

The preceding program can be described by the following Christiansen grammar:

GC={
Σ_N={<program>(↓g),
<decl-list>(↓gi, ↑go),
<dcl>(↓gi, ↑go),
<alpha-list>(↓gi, ↑word),
<alpha>(↓gi, ↑word),
<stm-list>(↓g),
<stm>(↓g),
<id>(↓g)},
Σ_T={ "{", "}", int, a, ⋯, z},
<program>(↓g),
P={
<program>(↓g_0)→"{"<decl-list>(↓g_0, ↑g_1)
  <stmt-list>(↓g_1)"}" {}
<decl-list>(↓g, ↑g) → λ{}
<decl-list>(↓g_0, ↑g_2)→ <decl>(↓g_0, ↑g_1)
  <decl-list>(↓g_1, ↑g_2){}
<decl>(↓g, ↑g∪{<id ↓g_{id}>→w})→
  int <alpha-list>(↓g, ↑w){}
<alpha-list>(↓g, ↑w) → <alpha>(↓g, ↑w){}
<alpha-list>(↓g, ↑w_1w_2)→<alpha>(↓g, ↑w_1)
  <alpha-list>(↓g, ↑w_2){}
<alpha>(↓g, ↑a) → a{}
…
<alpha>(↓g, ↑z) → z{}
<stmt-list>(↓g) → λ{}
<stmt-list>(↓g) → <stmt>(↓g)<stmt-list>(↓g)
<stmt>(↓g) → <id>(↓g) = <id>(↓g){} }
}
where

--g_0 is the initial grammar for the list of declarations. Notice that g_0 has no rule for the nonterminal symbol <id>.

--g_1 holds the changes made to g_0 by the list of declarations (each declaration of the form int <identifier>; adds to g_0 the rule <id> → <identifier>) and becomes the initial grammar for the list of statements; for each declared variable, g_1 contains one rule for nonterminal <id>.

Notice that

--The expression g∪{<id ↓g_{id}>→w} is the way in which rules for the declared variables are added. In the example, the set of rules added is

{<id ↓g_{id}> → i,
<id ↓g_{id}> → j }

where g_{id} must be understood as a new attribute name (with no associated value).

--By means of the rules for <alpha-list>, the identifier name used in the declaration is synthesized as the value of its second attribute.

Figure 1 shows the semantically annotated parse tree for the program. It only shows the value of the grammar attributes. The *word* attribute for <alpha> and <alpha-list> does not appear in figure 1. Continuous arrows are used for synthesized attributes. Changes in the grammars are highlighted by means of italic and underlined fonts. The initial value of the grammar inherited by the axiom (GC) is the whole grammar itself.

Christiansen grammars vs. attribute grammars

Christiansen grammars, initially named Generative Grammars, were proposed to describe extensible languages (programming languages that allow the user to enrich them with new concepts in the form of new linguistic constructs). In [6] other formalisms to obtain this goal, including attribute grammars, are compared with Christiansen grammars, and the following conclusions are drawn:

--Christiansen grammars have enough expressive power to fully describe, in a natural and elegant way, this family of languages.

--Christiansen grammars are as suitable as attribute grammars for describing the context-sensitive aspects of traditional (non-extensible) programming languages.

--Christiansen grammars are better for describing extensible constructs: The context free rules of an attribute grammar are fixed, hence too general; thus, extensibility tends to be implemented in the form of very complex constrains on attributes whose semantics become too loaded, resulting in not very elegant descriptions. Christiansen grammars, on the other hand, describe extensible constructs in a natural way. For instance, new control structures can be coded as new syntactic rules, and added and removed from the current grammar when necessary; new types of expressions can be coded as new nonterminals whose production rules can be treated in a similar way. Christiansen grammars also handle in an elegant way some difficult questions, such as the possible ambiguity of the grammar after the user has extended it.

--There are some constructs difficult to represent by Christiansen grammars (such as forbidding the declaration of entities with the same name in the same block, polymorphism or recursive declarations); but they are also very difficult to represent with attribute grammars.

B. Evolutionary Automatic Programming

Evolutionary Automatic Programming (EAP) [7] is the term used to represent those systems that use evolutionary computation to automatically generate computer programs. EAP techniques can be grouped depending on the way programs are represented: tree-based systems, which handle the derivation trees of the programs, or string-based systems, which represent the individuals as strings of symbols.

Tree-based systems

The best known tree-based system is Genetic Programming (GP), proposed by Koza [8] in the nineties, to automatically generate LISP programs that would solve given tasks. A few GP extensions have used formal grammars in some way: cellular encoding [9], which evolves neural networks; Whigham's approach [10], which adds biases (domain dependent knowledge) to GP by means of context free grammars, to ensure the syntactic correctness of the individuals in the initial population; GPK (Genetic Programming Kernel, developed by Horner [11]), which uses a tree representation similar to that of Whigham.

Finally, we shall mention in more detail a few approaches to handle context-sensitive constrains:

[12] proposes a tree-adjunct grammar guided genetic programming (TAG3P). A tree-adjunct grammar is a grammar which handles trees rather than strings, by means of the adjunct operation, which takes two trees and generates a new tree that can be used as a derivation tree. TAG3P uses two kinds of related grammars: the context free grammar of the target language and its corresponding lexicalized tree-adjunct grammar (a tree-adjunct grammar with at least one terminal node in every tree). TAG3P shows a good performance when solving classical problems (symbolic regression, trigonometric identities), due to its efficiency preserving and combining building blocks.

In [13] Hussain et al. evolve artificial neural networks encoded by means of attribute grammars, to increase the expressive power and performance of the search, while preserving syntactic correctness. In this method, genes are translated into derivation trees; genetic operators also act directly on derivation trees.

LOGENPRO [14, 15] and DCTG-GP [16], which can be considered Prolog (logic) GP implementations. LOGENPRO (LOgic grammar based GENetic PROgramming system) combines GP and Inductive Logic Programming, and offers a tool mainly used for data mining applications. LOGENPRO and DCTG-GP (Definite Clause Translation Grammars Genetic Programming) use DCG (Definite Clause Grammars), the logic version of attribute grammars [17]; consequently, both allow the description of context-free and context-sensitive constraints. DCTG-GP is inspired by the LOGENPRO system, but is not related to Inductive Logic Programming or to machine learning and data mining. For the purposes of this paper, LOGENPRO and DCTG-GP share two main characteristics: both use a Prolog system and represent programs by means of their parse trees.

String-based systems

String-based systems were initially discarded, because GP gave better results. String-based systems try to take advantage of the potential benefits of separating genotype and phenotype. Tree-based systems do not distinguish both levels explicitly. GE (Grammatical Evolution [7]) is the latest, most promising string-based approach, which will be further described in more detail. Other string-based approaches that use context-free grammars have been proposed: Binary Genetic Programming [18] contains some of the features fully exploited later by GE. GADS [19] (Genetic Algorithm for Deriving Software) and CFG/GP [20] (Context Free Grammars Genetic Programming) both use context free grammars as their output language specification. Other systems handle string-based systems without grammars: Discipulus^{TM} (RML Technologies, 1998) [21] is a commercial tool which implements the AIM-GP system (Automatic Induction of Machine Code for Genetic Programming [22]), greatly improving the performance of GP by coding the individuals as low level machine programs.

This paper is not the first attempt to add semantics to GE. The same authors have described previously Attribute Grammar Evolution [23], a variant of GE that replaces context free grammars by Attribute Grammars, one of the better known non-adaptable extensions of context free grammars, that makes them equivalent to type 0 Chomsky grammars. There are also other works which use at the same time GE and attribute grammars ([7] for the GAuGE algorithm and [24-25] for the knapsack problem), GE and context-sensitive grammars [26], or GE and some adaptive forms of grammars [27].

Comments

Other approaches are difficult to classify in this way. In [28], for instance, Paterson applies a similar approach to GE to include a context free grammar in GP, to map the genotype into a phenotype. Phenotypes can be expressed as strings or as trees.

[29] [30] and [31] describe PRODIGY (Program distribution estimation with grammar model), a framework which extends estimations of distribution algorithms (EDA) to the Genetic Programming domain. The basic idea of EDA is to estimate the probability distribution of optimal solutions by means of evolutionary techniques: a first model of the distribution is proposed and sampled to produce the initial

population. The model is iteratively refined and sampled until satisfying a termination condition. Shan et al. use stochastic context free parametric Lindenmayer grammars to include the main properties of GP into the model: internal hierarchical structure, locality of dependence, position independence, modularity and non-fixed complexity. They use two parameters: depth and location in the tree where the rules can be applied. While evolving the model, the structure and the probabilities of the grammar are modified (learned) separately. PRODIGY has been successfully used with some classic GP and learning problems such as symbolic regression and time series prediction. This is not the only attempt to use stochastic context free grammars in GP. [32], [33] and [34] propose similar models that only modify the probabilities of applying the rules of the grammar.

Reference [7] describes and shows theoretical and empirical arguments indicating that GE may be more powerful, general and suitable than other string and tree based systems. It is worth noticing that approaches mixing logic and genetic engines to automatically generate programs share a few disadvantages: Christiansen [4] reports the large amount of backtracking and the potentially infinite loops inherent to the underlying resolution strategy of Prolog systems, and the theoretical semi-decidability of first order logic. This drawback could make Prolog incompatible with applications where performance is important, such as most Evolutionary Automatic Programming experiments. We are not comparing our approach to those based on logic, because we intend to keep our algorithm independent of the resolution engine. Other EAP algorithms previously introduced have been exhaustively compared to GE [7], thus we only have to compare our results with those of GE. However, we have proposed two different ways of adding semantics to GE (AGE [23] and CGE, the topic of this paper), both approaches having been compared with GE. The comparison between AGE and CGE, their performance, their expressive power, the kind of problems better tackled by each approach, and so forth, will be the subject of our future research.

The advantages of CG over AG described in previous paragraphs are inherited by CGE; so CGE provides the user with a more flexible, natural and elegant formalism than AGE to describe the candidate solutions. Our experiments suggest also some performance advantage; it seems that this depends on the way in which CGE moves across the search space. Further experiments will be necessary to confirm and explain these inklings.

C. Grammatical evolution (GE)

GE [7] is an EAP algorithm based on strings, independent of the language used. Genotypes are represented by strings of integers (each of which is named *codon*) and the context-free grammar of the target programming language is used to deterministically map each genotype into a syntactically correct phenotype (a program). In this way GE avoids one of the main difficulties in Evolutionary Automatic Programming [7]: the results of genetic operators are guaranteed to be syntactically correct, while allowing the inclusion of multiple types.

The following scheme shows the way in which GE combines traditional genetic algorithms with genotype-to-phenotype mapping:

1) Generate at random an initial population of genotypes.
2) Translate each member of this initial set into its phenotype.
3) Sort the genotype population by their fitness (computed from the phenotypes).
4) If the best individual is a solution, the process ends.
5) Create the next generation: the mating-pool is chosen by means of a fitness-proportional parent selection strategy. Their genetically modified offspring is generated, and the worst individuals are replaced by them.
6) Go to step 2.

II. Motivation

GE being a general purpose stochastic search technique that uses a context-free grammar to avoid syntactic mistakes, it should be possible to improve its performance by adding some *semantics*, so that only syntactically and semantically correct programs are generated. In artificial intelligence, informed search algorithms usually have more power than their blind counterpart. It seems reasonable to hope that some of the approaches proposed since the sixties to formally describe the “semantics” of high level programming languages may be useful to our purpose. Different approaches to expressing the semantics of high level programming languages have advantages and drawbacks, but provide the language designer with a set of complementary tools. In the future, we will perform a study as wide as possible, but this paper is only focused in the use of Christiansen grammars.

III. Christiansen Grammar Evolution (CGE)

Normal GE genotypes are deterministically translated by applying to each codon the following process:

1) Choose the leftmost nonterminal symbol in the sentential form being processed.
2) Number the n right hand sides of all the rules for this nonterminal symbol (from 0 to n-1) where the rules are in an arbitrary order which should be maintained during the whole process.
3) Select the right hand side of the rule whose number equals codon mod (number of right hand sides for this nonterminal).
4) Derive the next word by replacing the nonterminal by the selected right hand side.

Several GE variants try to make this mapping more flexible. πGE [35] is a position-independent variation on GE’s typical genotype-phenotype mapping process. The non terminal symbol changed is not necessarily the leftmost one, but is computed from the codon by applying the function *mod(codon, number of non terminals in the sentential form)*.

Position independence seems to be an important feature to increase the performance of evolutionary algorithms and has also been considered by the same authors in Chorus [36] and GauGE [37].

Christiansen grammar evolution makes the GE genotype-to-phenotype mapping adaptive, by using a Christiansen grammar in place of the context-free grammar normally used in GE. The Christiansen grammar is designed to express both the syntactic and the semantic conditions that a valid phenotype must comply with.

The mapping of GE genotypes to their corresponding phenotypes has two important properties:

-- The mapping implicitly builds the derivation tree in depth-first order (choosing each time the left deepest node).

-- The mapping is deterministic: a given genotype has to be translated into the same phenotype under all possible circumstances. This is accomplished by numbering the different right hand sides for the same nonterminal and computing the "codon mod number of right hand sides" operation.

A. Genotype-to-phenotype mapping

CGE adds the following tasks to the previous algorithm:

1.1) Evaluate the attributes.

1.2) Select the applicable rules from the first attribute in each nonterminal.

The attributes are evaluated by means of the derivation tree. Each time that a node of the tree is expanded, the values of all the attributes that can be evaluated are computed in the following way:

--Attributes inherited from the parent symbol are evaluated directly.

--If the node symbol is prefixed by other symbols in the right hand side where it appears, attributes inherited from the left siblings are also evaluated.

--After expanding the last child of a node, the parent synthesized attributes are evaluated.

Our algorithm borrows a few interesting theoretical results from syntactic analysis techniques. Reference [38] shows that syntactically driven left-to-right translation schemes guarantee the proper evaluation of the kind of attributes previously described. The same reference also shows that this kind of attributes can be considered complete (they can represent any kind of attributes) and are compatible with a left to right depth-first route across the derivation tree. Since the genotype-to-phenotype mapping builds trees to derive words, rather than to analyze them, backtracking is needed to ensure the proper conclusion of the translation.

Notice that the main feature of Christiansen Grammars is the modification of the set of rules applicable to each given nonterminal. This is done by removing and adding rules to the initial inherited grammar, keeping the initial order (new rules are added to the end). Rules are numbered after changing the grammar and before each derivation step, in this way ensuring a deterministic genotype-to-phenotype mapping.

B. Example

The proposed algorithm has been used successfully to solve the following sample problem: "given any logical function with a given number of input variables, find a logically equivalent symbolic expression that uses only the operators in one of the three following complete sets: {and, or, not}, {nand}, {nor}."

The set of logical operators: {and, or, not, nand, nor}, contains the following five complete subsets {and, or, not}, {nand}, {nor}, {and, not} and {or, not}, any of which is capable of representing all the possible logic functions. In this paper, we are interested in finding a symbolic expression that represents a target logic function that uses only the operators in one of the first three complete subsets. A more detailed study, taking into account all the complete subsets, will be performed in the future to better characterize the properties of the CGE approach.

The following Christiansen grammar will be used to represent logic expressions with six input variables in postfix notation (the axiom <fb> stands for ***f***unction of ***B***oolean values):

Gb= {{<fb>($\downarrow$gi, $\uparrow$go), <op>($\downarrow$gi, $\uparrow$go), <op1>($\downarrow$gi, $\uparrow$go)},
{and, or, not, nand, nor, v_0, v_1, v_2, v_3, v_4, v_5},
<fb>($\downarrow$gi, $\uparrow$go),
P={
<fb>($\downarrow g_0$, $\uparrow g_3$)$\rightarrow$
  <fb1>($\downarrow g_0$, $\uparrow g_1$)<fb2>($\downarrow g_1$,$\uparrow g_2$)<op>($\downarrow g_2$,$\uparrow g_3$){}
<fb>($\downarrow g_0$,$\uparrow g_2$) $\rightarrow$<fb1> ($\downarrow g_0$,$\uparrow g_1$)<op1>($\downarrow g_1$,$\uparrow g_2$) {}
<fb>($\downarrow$g, $\uparrow$g) $\rightarrow v_0${}
<fb>($\downarrow$g, $\uparrow$g) $\rightarrow v_1${}
<fb>($\downarrow$g, $\uparrow$g) $\rightarrow v_2${}
<fb>($\downarrow$g, $\uparrow$g) $\rightarrow v_3${}
<fb>($\downarrow$g, $\uparrow$g) $\rightarrow v_4${}
<fb>($\downarrow$g, $\uparrow$g) $\rightarrow v_5${}
<op>($\downarrow$g, $\uparrow g_m$) $\rightarrow$
  and{$\uparrow g_m$=***$\downarrow$g-{<op>$\rightarrow$nand, <op>$\rightarrow$nor}***}
<op>($\downarrow$g, $\uparrow g_m$) $\rightarrow$or{$\uparrow g_m$=***$\downarrow$g-{<op>$\rightarrow$nand, <op>$\rightarrow$nor }***}
<op>($\downarrow$g, $\uparrow g_m$) $\rightarrow$
  nand{$\uparrow g_m$=***$\downarrow$g-***
    {<op>$\rightarrow$and,
    <op>$\rightarrow$or,
    <fb>$\rightarrow$<fb><op1>,
    <op1>$\rightarrow$not,
    <op>$\rightarrow$nor}}
<op>($\downarrow$g, $\uparrow g_m$) $\rightarrow$
  nor{$\uparrow$gm=***$\downarrow$g-***
   {<op>$\rightarrow$and,
   <op>$\rightarrow$or,
   <fb>$\rightarrow$<fb><op1>,
   <op1>$\rightarrow$not,
   <op>$\rightarrow$nand}}
<op1>($\downarrow$g,$\uparrow g_m$)$\rightarrow$not{$\uparrow g_m$=***$\downarrow$g-{<op>$\rightarrow$nand, <op>$\rightarrow$nor}***}
}}

Notice that the only allowed modifications remove from the

TABLE I
PARAMETERS FOR THE EXPERIMENT

Parameter	Explanation
Input set	All the possible inputs for the function
Fitness function	Number of successes of the candidate solution over the whole input set
Initial population	Random
Population size	500 individuals
Replacement strategy	Generational
Parent selection strategy	Fitness-proportional probability.
Genotype size	Variable length, initially within the range [10,50]
Codon value	Within [0,256]
Crossover	One point crossover chosen at random within [0, number of used codons]
Crossover ratio	0,9
Mutation ratio (individual)	0,9
Maximum generation	400

grammar the operators that do not belong to the selected complete set. This is highlighted with a bold and italic font in the last five rules.

The first rule shows how the modifications in the grammar are propagated. This rule represents the transformation of a Boolean function into a binary operation.

--The initial grammar inherited by the left hand side of the rule $\downarrow g_0$ is also inherited by its first child as its initial grammar.

--The possible modifications made by the sub-expression are recorded in the synthesized grammar $\uparrow g_1$, which is inherited by the second sub-expression as its initial grammar.

--Possible changes made during the analysis of <fb2> are recorded in the synthesized grammar $\uparrow g_2$, which is inherited by the binary operator term as its initial grammar.

--The binary operator modifies its initial grammar by removing the operators that do not belong to the same complete set. These changes are recorded in the synthesized grammar $\uparrow g_3$, which is passed back to the parent rule.

Figures 2-5 show the genotype-to-phenotype mapping. The derived strings use an italic font for the terminal symbols and an italic and underlined font for the current nonterminal symbol. The tree shows attribute inheritance by means of dotted arrows, synthesis by solid ones, and uses an italic and underlined font to highlight synthesized attributes.

In the beginning (figure 2), no codon has been consumed, the derivation tree has only the root, with the axiom of the grammar, and the starting string is also the axiom.

Figure 3 shows the first derivation, where the rule numbered 0 (180 mod 6) is applied to the axiom. The root is expanded in the tree, three new children nodes are created, but only the left most one inherits its first grammar from its father.

Figure 4 shows the first change in the grammar: when the node that contains symbol <op1> is expanded, and the leaf with the terminal symbol ***not*** is added, the semantic actions that compute the value of its second attribute remove from the production rules the subset {<op>$\rightarrow$nand, <op>$\rightarrow$nor}. In this way, the rules for logical operators are reduced to the set {<op>$\rightarrow$and, <op>$\rightarrow$or, <op1>$\rightarrow$not}. This is the only effective change over the grammar, because the generation of any new valid operator (***and***, ***or*** and ***not***) will remove again the same subset of rules ({<op>$\rightarrow$nand, <op>$\rightarrow$nor}), changing nothing.

Figure 5 shows the end of the derivation: there are only terminal symbols in the current string, and all the nodes of the derivation tree have computed the value of all their attributes.

C. *Results*

Table 1 shows the parameters used in this experiment. Mutation is applied to every descendant. If the mutation ratio randomly determines that the genotype has to be mutated, a single codon (only one) is randomly chosen and replaced by another value. This is not the only possible implementation: in [7] mutation operates on bits, rather than individuals, which allows multiple mutations in the same genotype (even multiple mutations in the same codon). These differences must be taken into account when comparing the actual values for the mutation ratio: even the highest value in our experiments corresponds to low values in the alternative implementation described in [7].

This problem is increasingly difficult for a higher number of input variables.

Figure 6 shows the results after 200 runs of the algorithm for less than 400 generations. The number of generations is represented in the x-axis while the y-axis corresponds to the cumulative frequency of success.

--The curve with circular marks shows the result of the problem with four input variables. The target logic function used was *(v_0 or v_1) and (v_2 or v_3)*. 100% of the runs reached success before 324 generations.

--The curve with square marks shows the result of the problem with five input variables. The target logic function used was *(v_0 or v_1) and (v_2 or v_3) and v_4*. 13% of the runs reached success before 397 generations.

--The curve with triangular marks shows the result of the problem with six input variables. The target logic function used was *(v_0 or v_1) and (v_2 or v_3) and (v_4 or v_5)*. 1.5% of the runs reached success before 391 generations.

D. *Performance comparison*

As previously indicated, we only have compared empirically the performance of CGE with that of GE. Even in this case, it is difficult to choose the appropriate problem for the comparison, because:

-- Christiansen grammars are equivalent to type 0 Chomsky grammars, but we decided to choose the target class of phenotypes context free, since otherwise we would not be able to compare CGE with GE, showing that the size of the search space is not the only performance advantage of CGE over GE.

-- On the other hand, some syntactic features of the problem have to be described as context sensitive constraints, otherwise the Christiansen Grammar would have a trivially empty semantics.

The following decisions have been made:

-- We have used the Christiansen Grammar of the previous example. It is well known that its language is context free.

-- We have designed two different context free grammars to compare GE and CGE performance. One of them is the *context-free scheme* of the Christiansen grammar (the context free grammar one gets by removing the semantic actions and attributes from the Christiansen grammar). The other is a context free grammar for the same language: the union of the grammars for the three context free languages of logical expressions, each with a complete set of operands.

D.1 First experiment

The GE parameters used are the same as in the CGE experiments, except for the grammar. The following context free grammar has been used:

Gb= {{<fb>,<op>,<op1>},
{and, or, not, nand, nor, v_0, v_1, v_2, v_3, v_4, v_5},
<fb>,
P={<fb>$\rightarrow$<fb><fb><op>,
<fb>$\rightarrow$<fb><op1>,
<fb>$\rightarrow$$v_0$,
<fb>$\rightarrow$$v_1$,
<fb>$\rightarrow$$v_2$,
<fb>$\rightarrow$$v_3$,
<fb>$\rightarrow$$v_4$,
<fb>$\rightarrow$$v_5$,
<op>$\rightarrow$and,
<op>$\rightarrow$or,
<op>$\rightarrow$nand,
<op>$\rightarrow$nor,
<op1>$\rightarrow$not}
}

With this context-free grammar, there is no direct way to express the restriction that the logic functions in different complete sets cannot be mixed. Therefore, we have tried two approaches: in the first one, the worst fitness value is assigned to those individuals which merge operators in different complete sets; in the second, the fitness value is not punished, and solutions which merge operators are removed by hand after the experiment finishes.

Figure 7 compares the results of these approaches for the experiment with four variables. The circular marks represent the CGE case (also shown in figure 6). The triangular marks represent the GE results: only 16% of the runs reached success after 400 generations, compared to 100% with CGE. The square marks represent GE results without fitness penalties: only 24% of the runs reached success after 400 generations, compared to 100% with CGE.

It is worth noticing that the algorithm using standard GE was unable to find a solution for the function with six input variables in any of the runs we performed.

D.2 Second experiment

The context free grammar of this experiment is the following:

Gb=
{{<fb>,<fb1>,<fb2>,<fb3>,<op>,<op_2>,<op_3>,<op1>},
{and, or, not, nand, nor, v_0, v_1, v_2, v_3, v_4, v_5},
<fb>,
P={<fb>$\rightarrow$<fb1>,
<fb>$\rightarrow$<fb2>,
<fb>$\rightarrow$<fb3>,
<fb1>$\rightarrow$<fb1><fb1><op>,
<fb1>$\rightarrow$<fb1><op1>,
<op1>$\rightarrow$not
<op>$\rightarrow$or,
<op>$\rightarrow$and,
<fb1>$\rightarrow$$v_0$,
<fb1>$\rightarrow$$v_1$,
<fb1>$\rightarrow$$v_2$,
<fb1>$\rightarrow$$v_3$,
<fb1>$\rightarrow$$v_4$,
<fb1>$\rightarrow$$v_5$,
<fb2>$\rightarrow$<fb2><fb2><op_2>,
<fb2>$\rightarrow$$v_0$,
<fb2>$\rightarrow$$v_1$,
<fb2>$\rightarrow$$v_2$,
<fb2>$\rightarrow$$v_3$,
<fb2>$\rightarrow$$v_4$,
<fb2>$\rightarrow$$v_5$,
<op_2>$\rightarrow$nand,
<fb3>$\rightarrow$<fb3><fb3><op_3>,
<op_3>$\rightarrow$nor,
<fb3>$\rightarrow$$v_0$,
<fb3>$\rightarrow$$v_1$,
<fb3>$\rightarrow$$v_2$,
<fb3>$\rightarrow$$v_3$,
<fb3>$\rightarrow$$v_4$,
<fb3>$\rightarrow$$v_5$}
}

Figure 8 compares the results of both approaches for the experiment with four, five and six variables after 200 runs of the algorithm for less than 400 generations. Small marks are used for the CGE results; big marks are associated to GE. The circular marks represent the case with four variables. The figure shows that the performance of CGE is the same as that of GE in that case. Square marks represent the experiment with five variables. In this case, CGE improves greatly the performance of GE: many more runs find the solution, and the maximum cumulative success frequency is also higher: 0.03% of the runs reached success after 400 generations with GE, compared to 13% with CGE. Triangular marks are used for the case with six variables. Notice that the algorithm using standard GE was again unable to find a solution for this case in any of the runs we performed.

D.3 Third experiment

Actually, the previous experiment does not show a remarkable performance improvement of CGE over GE in the easiest case. Nevertheless, CGE seems to be better as the problem becomes harder (five and six variables).

If it does not matter which complete set of logical functions is used to solve this problem, GE could reduce its search space by using a context free grammar for each of the possible sets. This experiment is really made of three tests, each of which uses only one set of logical functions.

The three context free grammars for this experiment are the following:

$G_{and\_or\_not}$= {{<fb1>,<op>,<op1>},
{and, or, not, v_0, v_1, v_2, v_3, v_4, v_5},
<fb1>,
P={<fb1>→<fb1><fb1><op>,
<fb1>→<fb1><op1>,
<op1>→not
<op>→or,
<op>→and,
<fb1>→v_0,
<fb1>→v_1,
<fb1>→v_2,
<fb1>→v_3,
<fb1>→v_4,
<fb1>→v_5}}
G_{nand}= {{<fb2>,<op_2>},
{nand, v_0, v_1, v_2, v_3, v_4, v_5},
<fb2>,
P={<fb2>→<fb2><fb2><op_2>,
<fb2>→v_0,
<fb2>→v_1,
<fb2>→v_2,
<fb2>→v_3,
<fb2>→v_4,
<fb2>→v_5,
<op_2>→nand}}
G_{nor}= {{<fb3>,<op_3>},
{nor, v_0, v_1, v_2, v_3, v_4, v_5},
<fb3>,
P={<fb3>→<fb3><fb3><op_3>,
<op_3>→nor,
<fb3>→v_0,
<fb3>→v_1,
<fb3>→v_2,
<fb3>→v_3,
<fb3>→v_4,
<fb3>→v_5}}

Figures 9, 10 and 11 compare the results of both CGE and GE approaches for the experiments with four, five and six variables after 200 runs of the algorithm for less than 400 generations. There are several important conclusions:

--GE performance strongly depends on the complete set of logical functions used. So, using a general grammar which considers the three cases at the same time seems advisable, even though the search space becomes greater. Therefore, CGE has shown to be a better choice in this case.

--CGE is actually better than GE in some cases. In fact, for 4 variables, GE is better only for the *nor* set and worse for the *and_or_not* set, while for the *nand* set GE never finds any solution. For 5 variables, GE is better for the *and_or_not* case, and never finds solutions for the *nor* and the *nand* cases. Finally, for 6 variables, the CGE approach is about as good as GE for the *and_or_not* case. Table 2 compares the mean and the variance of the number of generations needed to reach a solution using CGE, GE with fitness punishing, and GE without penalties, for the problem with 4 variables, and the first two for the problem with 5 variables. We have applied the Welch-test [39] to our distributions to estimate the confidence of the conclusion. The Welch-test is adequate for stochastic variables with dissimilar variances. The default hypothesis is that GE is better than CGE (CGE means would be greater or equal than GE means). This hypothesis can be discarded, because the Welch-test concludes that its confidence value is 0 in all the comparisons but one, where it is 1e-12. Therefore, we may conclude that CGE solutions are faster than GE with about a 100% confidence.

--In conclusion, CGE provides a much better general approach than GE, as it always finds a solution, while GE with a single complete set may never find one.

IV. CONCLUSIONS

This paper proposes CGE, a new Evolutionary Automatic Programming method that improves the expressive power of GE by adding a way to add semantics to the rules that an individual must comply with, before it can be generated.

The experiments we have performed provide an inkling that this procedure is better than standard GE (which only uses syntax), increasing the efficiency of the algorithms by orders of magnitude. Obviously, more experiments should be performed to confirm this inkling.

This paper shows that the performance of GE to solve context free problems is clearly improved by CGE, and suggests the way in which it is possible to find problems difficult to solve by GE, but which are tackled naturally and efficiently by CGE.

The performance comparison between GE and CGE also suggests that the improvement does not actually depend on the context dependent nature of constraints, but on the ease of the formalism used to express them. This has been made clear by us in a different publication [40], where we have attempted to find a solution of the well-known P-median problem by means of both GE and CGE.

In the future we plan to apply our approach to new problems difficult to solve without adding some semantics to the description of the candidate solutions. We shall also test other ways to specify semantics different from Christiansen grammars and attribute grammars. Our group also plans to perform both theoretical and empirical studies to characterize the properties of this technique, and to design a general methodology to automatically solve given tasks by means of variants of GE that include semantics.

One of the main questions that this methodology still has to answer is how to decide in advance which EAP is more

TABLE 2
STATISTICS FOR THE EXPERIMENT

	CGE 4 vars	GE punishing 4 vars	GE 4 vars	CGE 5 vars	GE 5 vars
Mean	73.045	200.85	168.68	371.55	399.52
Variance	4016.6	12396.01	14351.0	6893.4	45.125

suitable for a given problem. That is, if the problem under consideration should be tackled with or without semantics and, in the first case, if AGE or CGE should be used. At this point, choosing CGE rather than AGE is a matter of taste, comfort and ease of use.

REFERENCES

[1] A. N. Chomsky, "Formal properties of grammars," in *Handbook of Math. Psych.*, vol. 2, John Wiley and Sons, New York, pp. 323–418, 1963.

[2] D. E. Knuth, "Semantics of Context-Free Languages," in *Mathematical Systems Theory*, vol. 2, nº 2, pp. 127–145, 1968.

[3] J. N. Shutt, "Recursive Adaptable Grammars. A thesis submitted to the Faculty of the Worcester Polytechnic Institute in partial fulfillment of the requirements for the degree of Master of Science in Computer Science," August 10, 1993 (emended December 16, 2003)

[4] H. Christiansen, "A Survey of Adaptable Grammars," in *ACM SIGPLAN Notices*, vol. 25, nº11., pp. 35–44. November 1990.

[5] Watt and Madsen. "Extended Attribute Grammars" The Computer Journal. Online ISSN 1460-2067 - Print ISSN 0010-4620. 1983; 26: 142-153.

[6] Christiansen, H. "The syntax and semantics of extensible languages." Datalogiske Skrifter nº 14, 1988 (Technical report series) Computer Science Section. Roskilde University. Roskilde. Denmark.

[7] M. O'Neill & R. Conor, "Grammatical Evolution, evolutionary automatic programming in an arbitrary language," Kluwer Academic Publishers, 2003.

[8] J. R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection," MIT Press, Cambridge, Massachusetts. 1992.

[9] Gruau, F. "Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm" PhD thesis, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure de Lyon, France, 1994.

[10] Whigham, P.A., "Grammatical Bias for Evolutionary Learning." PhD thesis, School of Computer Science, University College, University of NewSouth Wales, Australian Defence Force Academy. B. Grammatical evolution, 1996.

[11] Horner, H., "A C++ class library for genetic programming: The vienna university of economics genetic programming kernel." Vienna University of Economics, 1996.

[12] Nguyen Xuan Hoai, R.I. McKay, and D. Essam "Genetic programming with context-sensitive grammars.", EuroGP 2002, LNCS 2278, pp. 228-237, 2002.

[13] Hussain, T.S. & Browse, R.A. "Network generating attribute grammar encoding", 1998 IEEE International Joint Conference on Neural Networks, May 4-9, 1998 in Anchorage, Alaska.

[14] Wong, M. L. and Leung, K. S., "Evolutionary program induction directed by logic grammars." *Evolutionary Computation*, 5(2): 143-180, 1997.

[15] Wong, M. L. and Leung, K. S. "Data Mining Using Grammar Based Genetic Programming and Applications." *Volume 3 of Genetic Programming*. Klower Academic Publishers, 2000.

[16] Brian J. Ross, "Logic-based genetic programming with definite clause translation grammars", *New Generation Computing*, vol. 19, n.4, pp. 313-337, 2001.

[17] Bratko, I., "Prolog programming for artificial intelligence" Addison-Wesley Publishers Company, Inc. 1990.

[18] Keller, R. E. and Banzhaf, W. "Genetic programming using genotype-phenotype maping from linear genomes into linear phenotypes." In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 116-122, Stanford University, CA, USA. MIT Press, 1996.

[19] Paterson, N. R. and Livesey, M., "Distinguishing genotype and phenotype in genetic programming." In Koza, J. R., editor, *Late Breaking Paers at the Genetic Programming 1996 Conference* Stanford University July 28-31, pages 141-150, Stanford University, CA, USA. Stanford Bookstore, 1996.

[20] Freeman, J. J. , "A linear representation for GP using context free grammars." In Koza, J. R., Banzhaf, W., Chellapilla, D., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 72-77, University of Wisconsin, Madison, Wisconsin, USA. Morgan Kaufmann, 1998.

[21] Register Machine Learning Technologies, Inc., "Discipulus Users Manual, Version 3.0.", 2002. Available from www.aimlearning.com.

[22] Nordin, P. "AIMGP: A formal description." In Koza, J. R., editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, Wisconsin, USA. Stanford University Bookstore, 1998.

[23] M. de la Cruz, A. Ortega, M. Alfonseca, "Attribute Grammar Evolution." *LNCS* 3562, pp. 182-191. J. Mira and J.R. Álvarez (Eds.) Springer-Verlag Berlin Heidelberg, 2005.

[24] O'Neill, Michael and Cleary, Robert and Nikolov, Nikola S. "Solving Knapsack Problems with Attribute Grammars." In Proceedings of the Grammatical Evolution Workshop, GECCO 2004.

[25] Cleary R. and O'Neill M. "An Attribute Grammar Decoder for the 0/1 Multiconstrained Knapsack Problem." In Proceedings of EvoCOP 2005.

[26] Keijzer, M., Babovic, V., Ryan, C., O'Neill, M., and Cattolico, M. "Adaptive logic programming." In Proceedings of GECCO 2001.

[27] Dempsey I., O'Neill M. and Brabazon A. "Meta-Grammar Constant Creation with Grammatical Evolution by Grammatical Evolution." In Proceedings of GECCO 2005

[28] Paterson, N. "Genetic programming with context-sensitive grammars." PhD thesis. 2002

[29] Shan, Y. McKay, R. I. Abbass, H. A. Essam, D.: "Program distribution estimation with grammar models."

[30] Shan, Y., McKay, R., Baxter, R., Abbass, H., Essam, D., and Nguyen., H. (2004). "Grammar model based program evolution." In Proceedings of The Congress on Evolutionary Computation, Portland, USA. IEEE.

[31] Shan, Y., McKay, R. I., Abbass, H. A., and Essam, D. (2003). "Program evolution with explicit learning: a new framework for program automatic synthesis." In Proceedings of 2003 Congress on Evolutionary Computation, Canberra, Australia. University College, University of New South Wales, Australia.

[32] Ratle, A. and Sebag, M. "Avoiding the bloat with probabilistic grammar guided genetic programming." In Collet, P., Fonlupt, C., Hao, J.K., Lutton, E., and Schoenauer, M., editors, Artificial Evolution 5th International Conference, Evolution Artificielle, EA 2001, volume 2310 of LNCS, pages 255–266, Creusot, France. Springer Verlag.

[33] Tanev, I. "Implications of incorporating learning probabilistic context sensitive grammar in genetic programming on evolvability of adaptive locomotion gaits of snakebot." In Proceedings of GECCO 2004, Seattle, Washington, USA.

[34] Bosman, P. A. N. and de Jong, E. D. "Grammar transformations in an eda for genetic programming." In Special session: OBUPM Optimization by Building and Using Probabilistic Models, GECCO, Seattle, Washington, USA.

[35] O'Neill1, M. Brabazon, A. Nicolau, M. Mc Garraghy, S. and Keenan, P. "πGrammatical Evolution." In K. Deb et al. (Eds.): GECCO 2004, LNCS 3103, pp. 617–629, 2004. Springer-Verlag Berlin Heidelberg 2004

[36] Ryan, C., Azad, A., Sheahan, A., O'Neill, M. "No Coercion and No Prohibition, A Position Independent Encoding Scheme for Evolutionary Algorithms—The Chorus System." Proc. of the 4th European Conference on Genetic Programming, EuroGP 2002, LNCS 2278, pp. 132-142. Springer-Verlag.

[37] Ryan, C., Nicolau, M., O'Neill, M. "Genetic Algorithms Using Grammatical Evolution." Proc. of the 4th European Conference on

Genetic Programming, EuroGP 2002, LNCS 2278, pp. 279-288. Springer-Verlag.

[38] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. “Compilers: Principles, Techniques, and Tools.” Addison-Wesley, Reading, MA, 1986.

[39] Welch, B. L.. “On the comparison of several mean values.” Biometrika 38, 330-336 (1951)

[40] A.L.Abu Dalhoum, M. Al Zoubi, M. de la Cruz, A.Ortega, M.Alfonseca: “A Genetic Algorithm for Solving the P-Median Problem”, Proceedings of the 2005 European Simulation and Modeling Conference (ESM2005), Oporto, Oct. 2005. In press.

Dr. Alfonso Ortega is currently a professor at the University. He formerly lectured at the Universidad Pontificia de Salamanca and worked at LAB2000 (an IBM subsidiary) as a software developer. He holds a doctorate in computer science from the Universidad Autónoma. Dr. Ortega has published about 15 technical papers on computer languages, complex systems, graphics, and theoretical computer science, and has collaborated in the development of several software products.

Marina de la Cruz is a lecturer at the University (Universidad Autónoma and Universidad Politécnica de Madrid). During about the last 15 years she has been a researcher in CIEMAT where she has contributed in about 10 international projects. Currently, she is doing Ph.D. research at the Universidad Autónoma, in genetic algorithms and formal complex systems and she is an author of several papers and communications to international journals and conferences on theoretical computer science.

Dr. Manuel Alfonseca is a professor at the University. He was formerly a Senior Technical Staff Member at IBM, having worked from 1972 to 1994 at the IBM Scientific Center in Madrid. Dr. Alfonseca was one of the developers of the APL/PC interpreter and related products; he has worked on computer languages, simulation, complex systems, graphics, artificial intelligence, object orientation, and theoretical computer science, and has published several books and about 180 technical papers, as well as 60 papers on popular science in a major Spanish newspaper. He is an award-winning author of 22 published books for children. Dr. Alfonseca holds a doctorate in electronics and an M.Sc. degree in computer science from the Universidad Politécnica de Madrid. He is an emeritus member of the IBM Technical Expert Council and a member of the Society for Computer Simulation (SCS), the New York Academy of Sciences, the IEEE Computer Society, the ACM, the British APL Association, and the Spanish Association of Scientific Journalism.

Figure 1: semantically annotated parse tree of the program

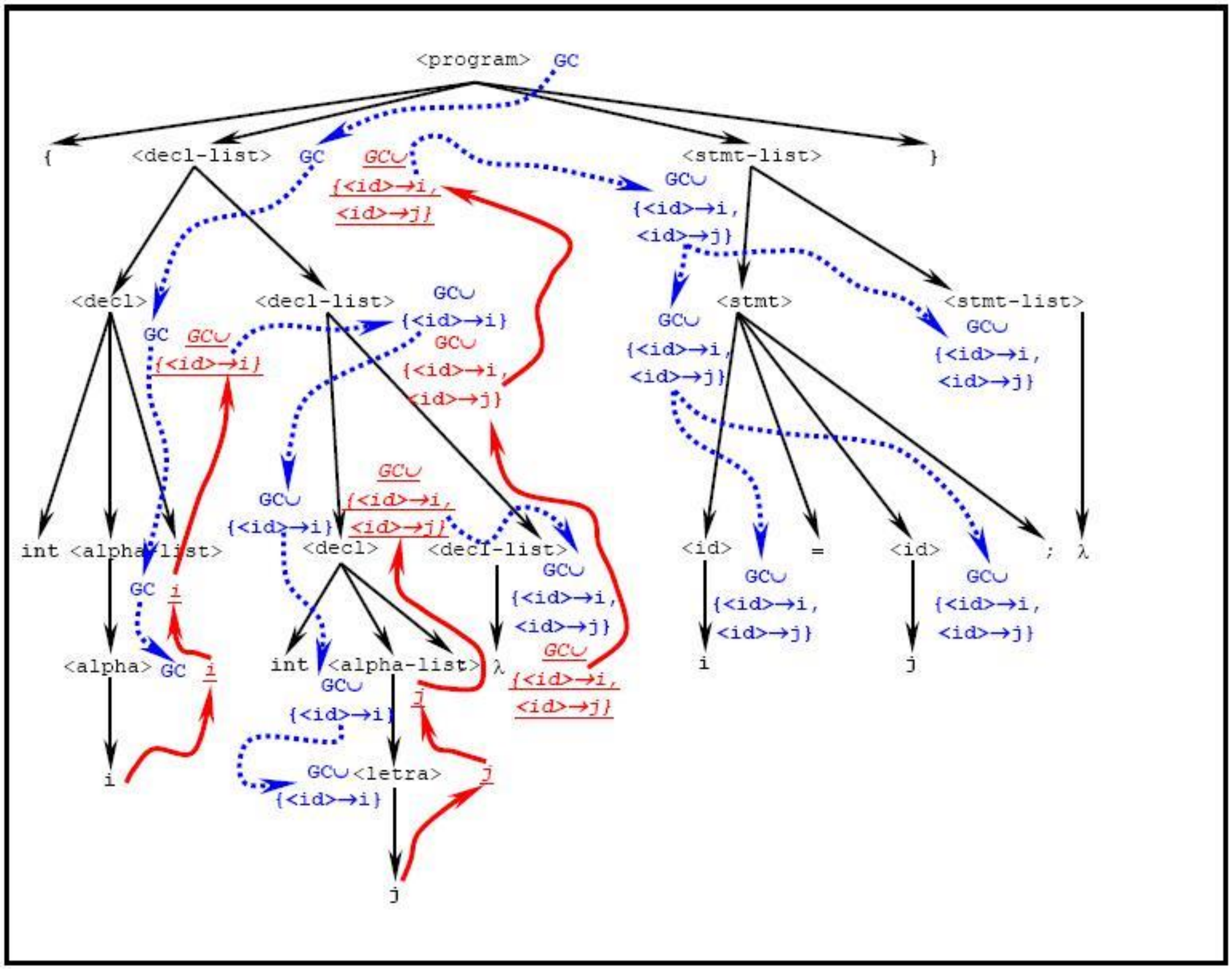
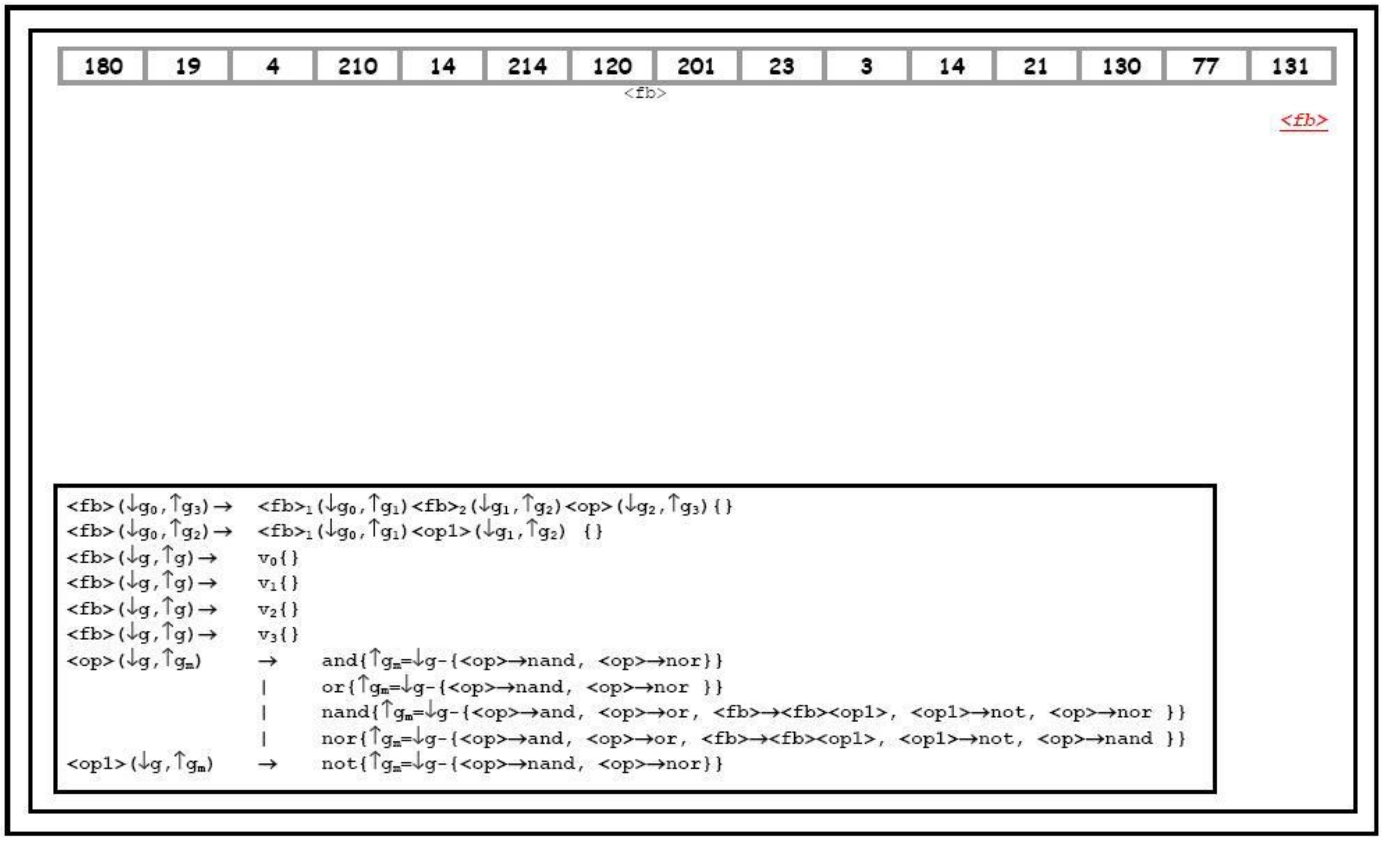
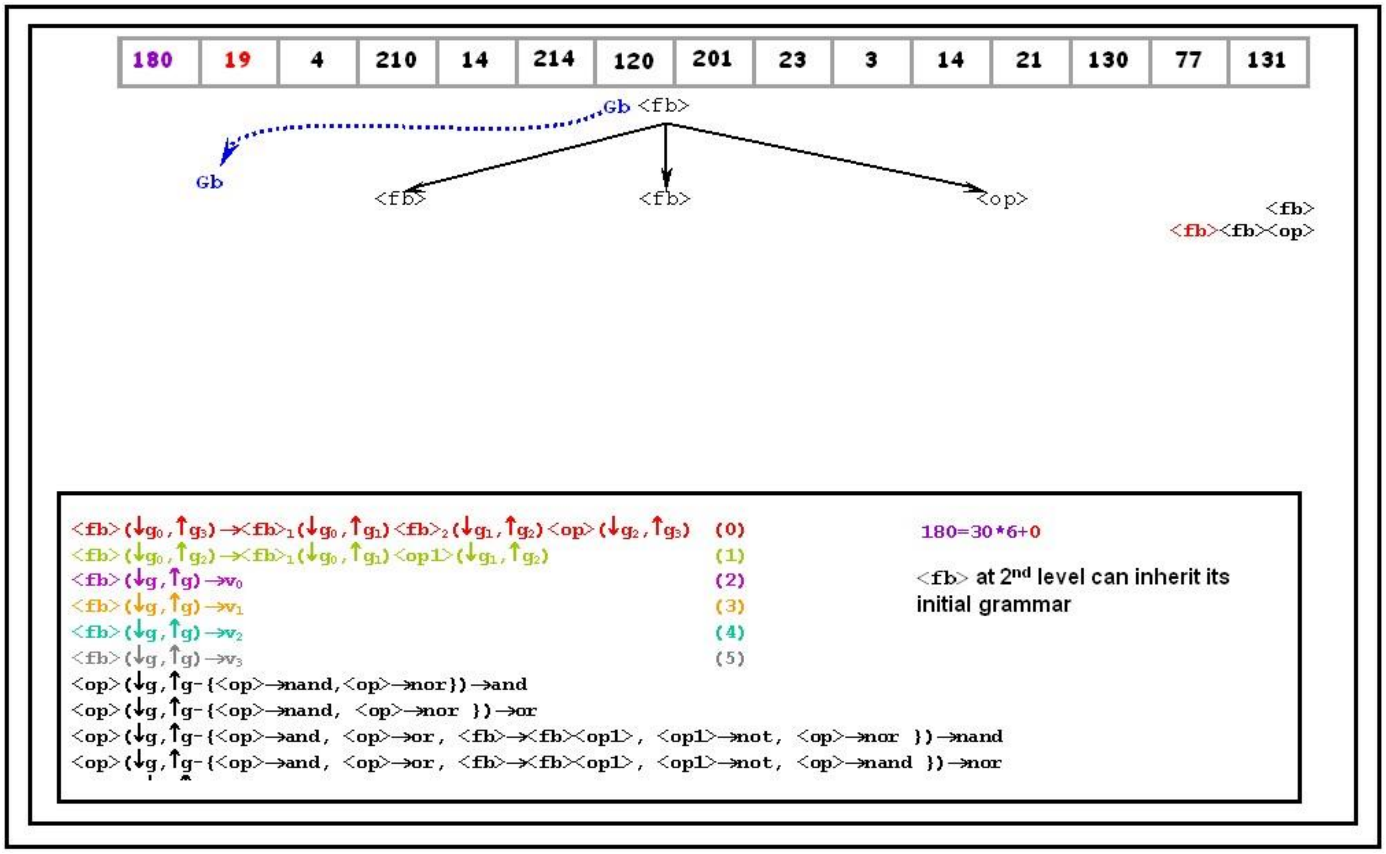


Figure 2: CGE genotype-to-phenotype mapping example, initial step



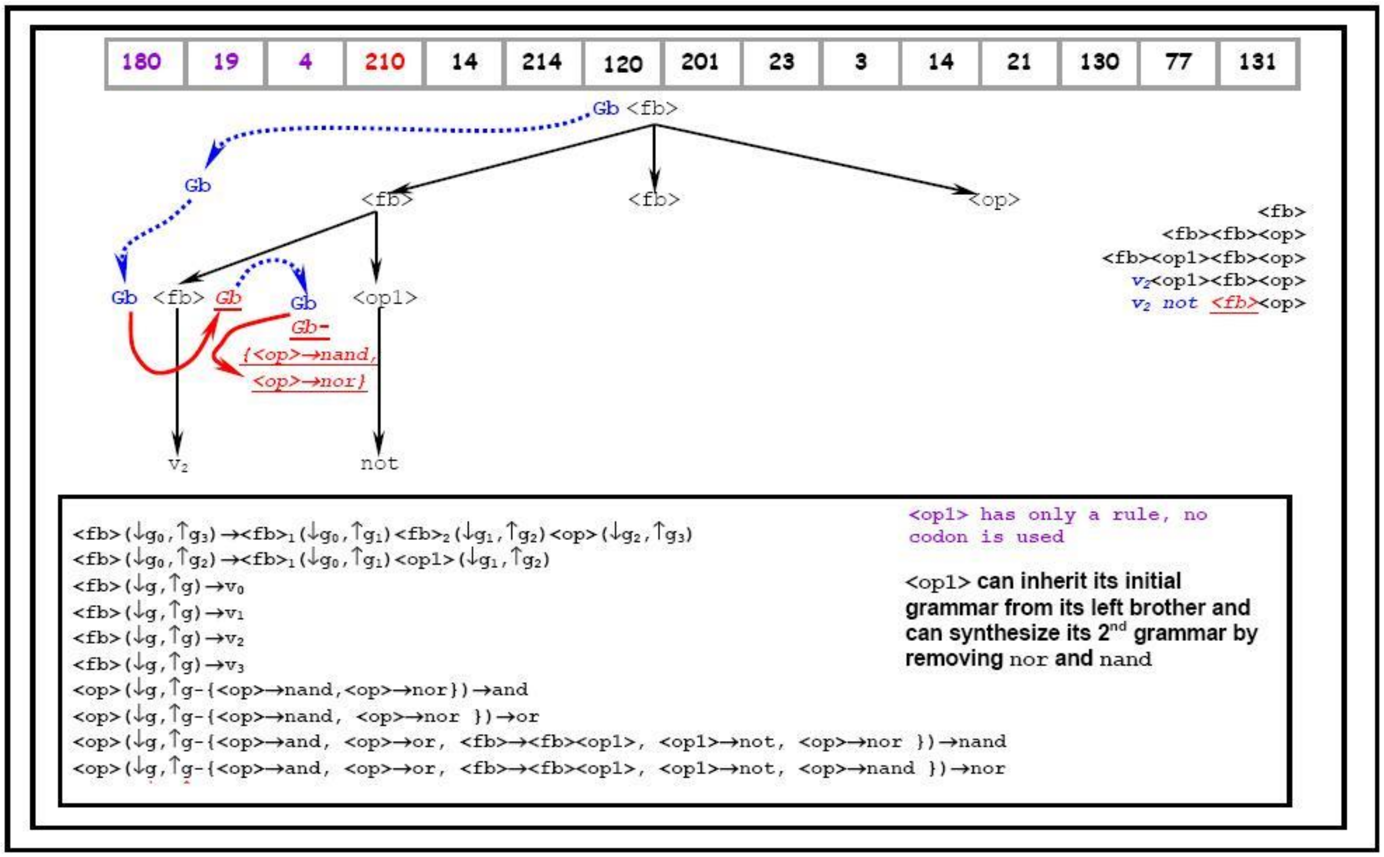
```
<fb>(↓g_0,↑g_3)→   <fb>_1(↓g_0,↑g_1)<fb>_2(↓g_1,↑g_2)<op>(↓g_2,↑g_3){}
<fb>(↓g_0,↑g_2)→   <fb>_1(↓g_0,↑g_1)<op1>(↓g_1,↑g_2)  {}
<fb>(↓g,↑g)→       v_0{}
<fb>(↓g,↑g)→       v_1{}
<fb>(↓g,↑g)→       v_2{}
<fb>(↓g,↑g)→       v_3{}
<op>(↓g,↑g_m)      →     and{↑g_m=↓g-{<op>→nand, <op>→nor}}
                   |     or{↑g_m=↓g-{<op>→nand, <op>→nor }}
                   |     nand{↑g_m=↓g-{<op>→and, <op>→or, <fb>→<fb><op1>, <op1>→not, <op>→nor }}
                   |     nor{↑g_m=↓g-{<op>→and, <op>→or, <fb>→<fb><op1>, <op1>→not, <op>→nand }}
<op1>(↓g,↑g_m)     →     not{↑g_m=↓g-{<op>→nand, <op>→nor}}
```

Figure 3: CGE genotype-to-phenotype mapping example, second step



```
<fb>(↓g₀,↑g₃)→<fb>₁(↓g₀,↑g₁)<fb>₂(↓g₁,↑g₂)<op>(↓g₂,↑g₃)   (0)
<fb>(↓g₀,↑g₂)→<fb>₁(↓g₀,↑g₁)<op1>(↓g₁,↑g₂)                (1)
<fb>(↓g,↑g)→v₀                                            (2)
<fb>(↓g,↑g)→v₁                                            (3)
<fb>(↓g,↑g)→v₂                                            (4)
<fb>(↓g,↑g)→v₃                                            (5)
<op>(↓g,↑g-{<op>→nand,<op>→nor})→and
<op>(↓g,↑g-{<op>→nand, <op>→nor })→or
<op>(↓g,↑g-{<op>→and, <op>→or, <fb>→<fb><op1>, <op1>→not, <op>→nor })→nand
<op>(↓g,↑g-{<op>→and, <op>→or, <fb>→<fb><op1>, <op1>→not, <op>→nand })→nor
```

Figure 4: CGE genotype-to-phenotype mapping example, third step



$\langle fb\rangle(\downarrow g_0,\uparrow g_3)\to\langle fb\rangle_1(\downarrow g_0,\uparrow g_1)\langle fb\rangle_2(\downarrow g_1,\uparrow g_2)\langle op\rangle(\downarrow g_2,\uparrow g_3)$

$\langle fb\rangle(\downarrow g_0,\uparrow g_2)\to\langle fb\rangle_1(\downarrow g_0,\uparrow g_1)\langle op1\rangle(\downarrow g_1,\uparrow g_2)$

$\langle fb\rangle(\downarrow g,\uparrow g)\to v_0$

$\langle fb\rangle(\downarrow g,\uparrow g)\to v_1$

$\langle fb\rangle(\downarrow g,\uparrow g)\to v_2$

$\langle fb\rangle(\downarrow g,\uparrow g)\to v_3$

$\langle op\rangle(\downarrow g,\uparrow g-\{\langle op\rangle\to nand,\langle op\rangle\to nor\})\to and$

$\langle op\rangle(\downarrow g,\uparrow g-\{\langle op\rangle\to nand,\ \langle op\rangle\to nor\ \})\to or$

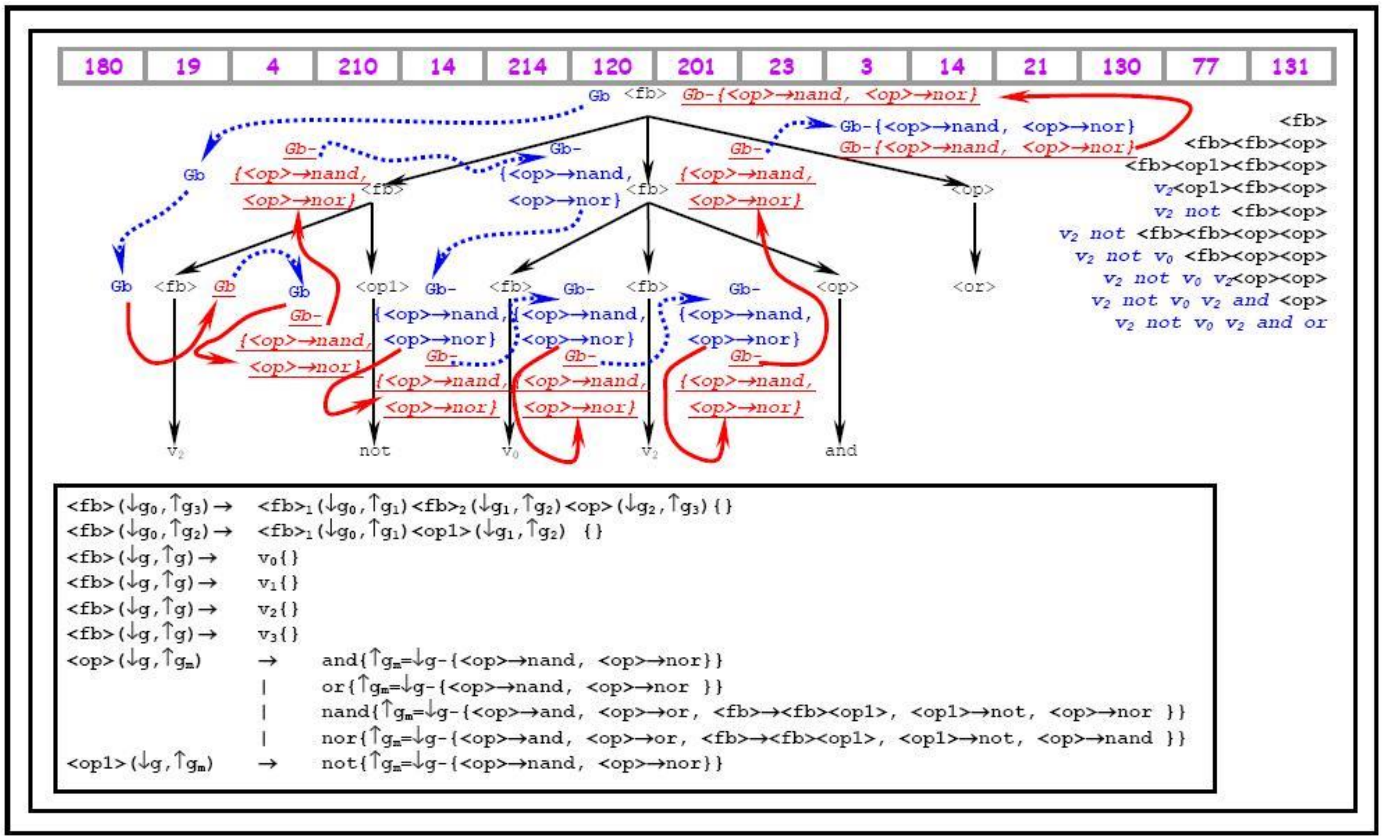
$\langle op\rangle(\downarrow g,\uparrow g-\{\langle op\rangle\to and,\ \langle op\rangle\to or,\ \langle fb\rangle\to\langle fb\rangle\langle op1\rangle,\ \langle op1\rangle\to not,\ \langle op\rangle\to nor\ \})\to nand$

$\langle op\rangle(\downarrow g,\uparrow g-\{\langle op\rangle\to and,\ \langle op\rangle\to or,\ \langle fb\rangle\to\langle fb\rangle\langle op1\rangle,\ \langle op1\rangle\to not,\ \langle op\rangle\to nand\ \})\to nor$

<op1> has only a rule, no codon is used

<op1> **can inherit its initial grammar from its left brother and can synthesize its 2nd grammar by removing** nor **and** nand

Figure 5: CGE genotype-to-phenotype mapping example, result



```
<fb>(↓g0,↑g3)→   <fb>1(↓g0,↑g1)<fb>2(↓g1,↑g2)<op>(↓g2,↑g3){}
<fb>(↓g0,↑g2)→   <fb>1(↓g0,↑g1)<op1>(↓g1,↑g2)  {}
<fb>(↓g,↑g)→     v0{}
<fb>(↓g,↑g)→     v1{}
<fb>(↓g,↑g)→     v2{}
<fb>(↓g,↑g)→     v3{}
<op>(↓g,↑gm)     →     and{↑gm=↓g-{<op>→nand, <op>→nor}}
                 |     or{↑gm=↓g-{<op>→nand, <op>→nor }}
                 |     nand{↑gm=↓g-{<op>→and, <op>→or, <fb>→<fb><op1>, <op1>→not, <op>→nor }}
                 |     nor{↑gm=↓g-{<op>→and, <op>→or, <fb>→<fb><op1>, <op1>→not, <op>→nand }}
<op1>(↓g,↑gm)    →     not{↑gm=↓g-{<op>→nand, <op>→nor}}
```

Figure 6: CGE performance for different instances of the function with boolean values problem

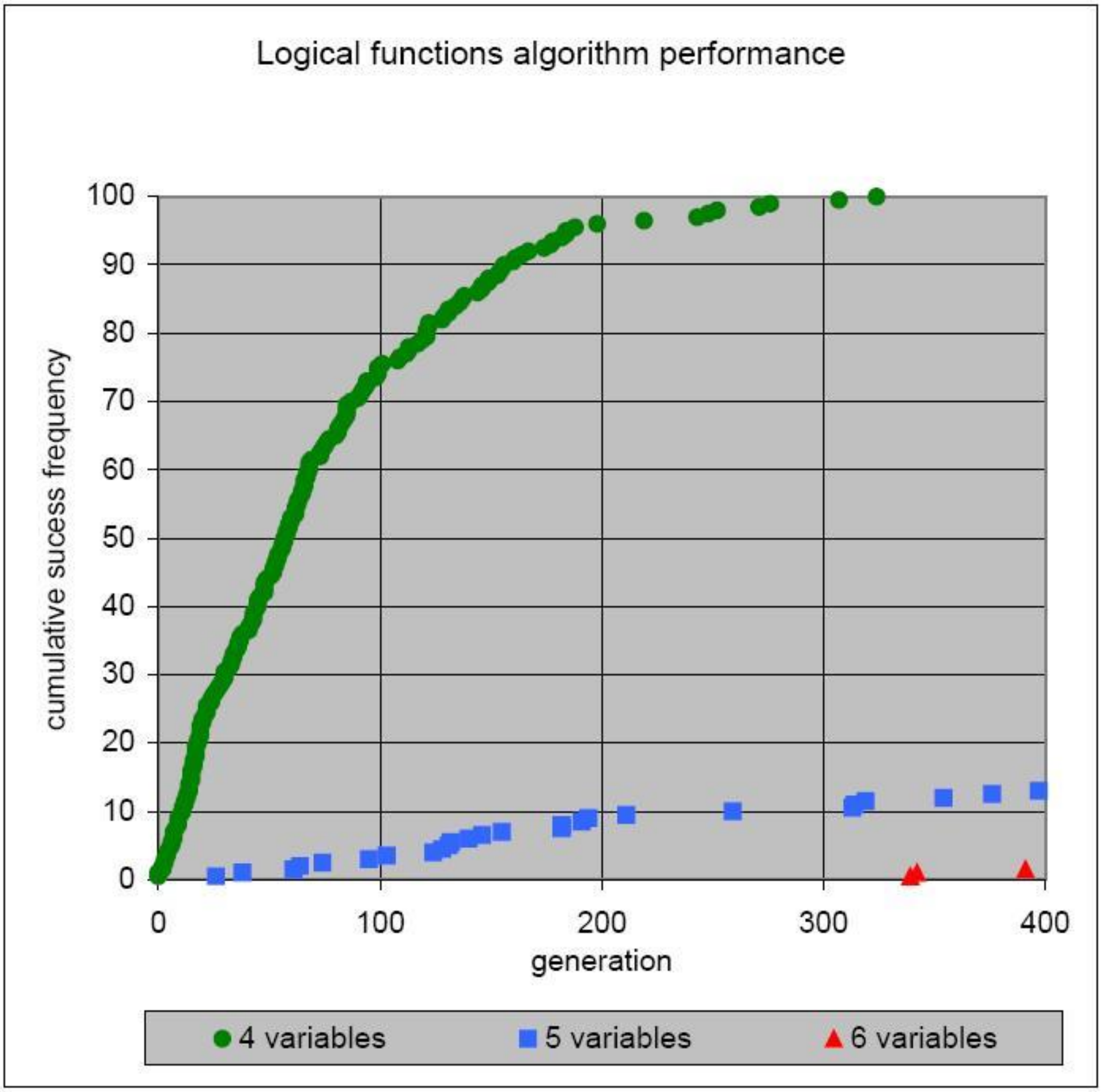


Figure 7: CGE vs. GE performance comparison for the first experiment

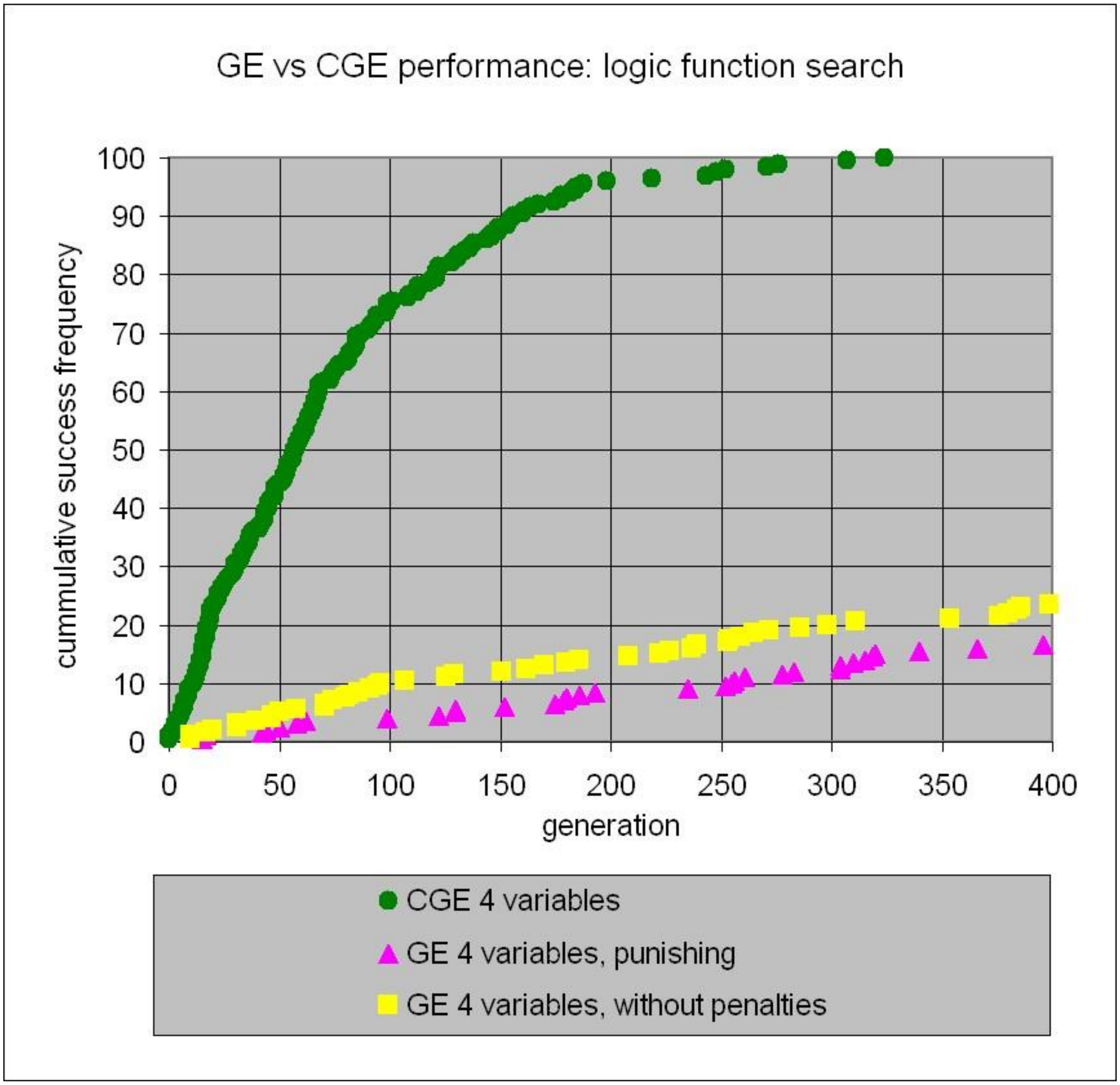


Figure 8: CGE vs. GE performance comparison for the second experiment

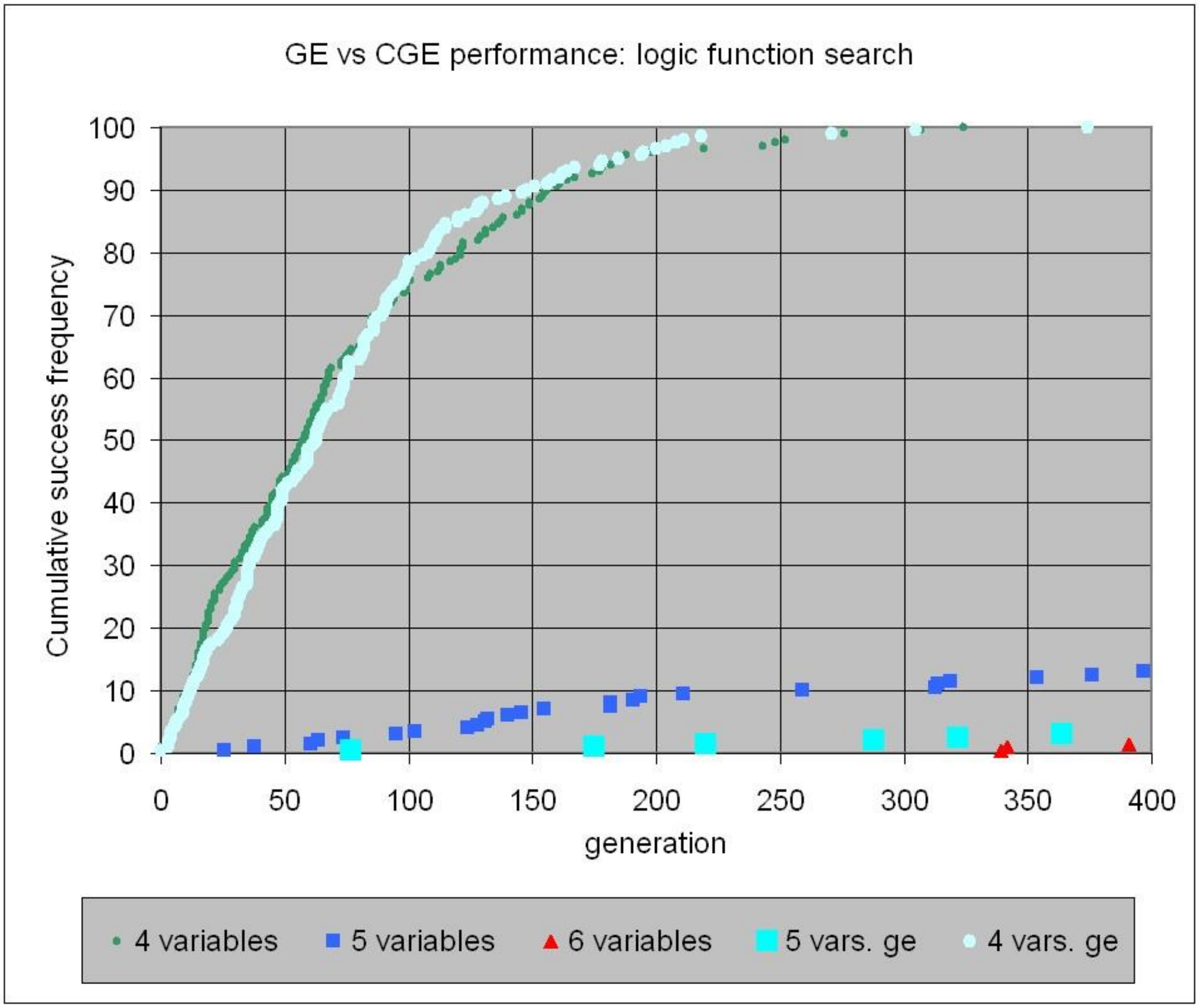


Figure 9: CGE vs. GE performance comparison for the third experiment: 4 variables

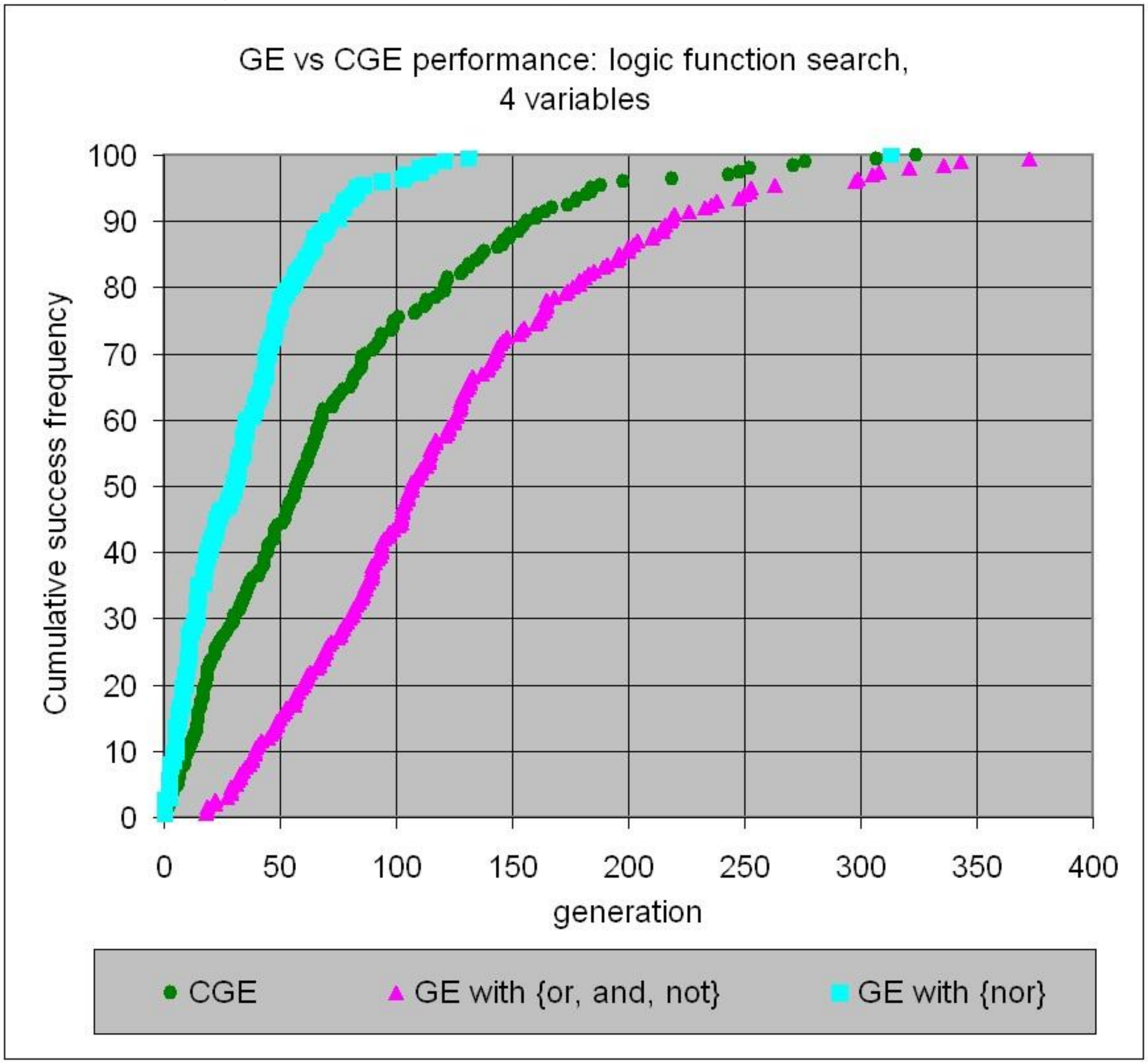


Figure 10: CGE vs. GE performance comparison for the third experiment: 5 variables

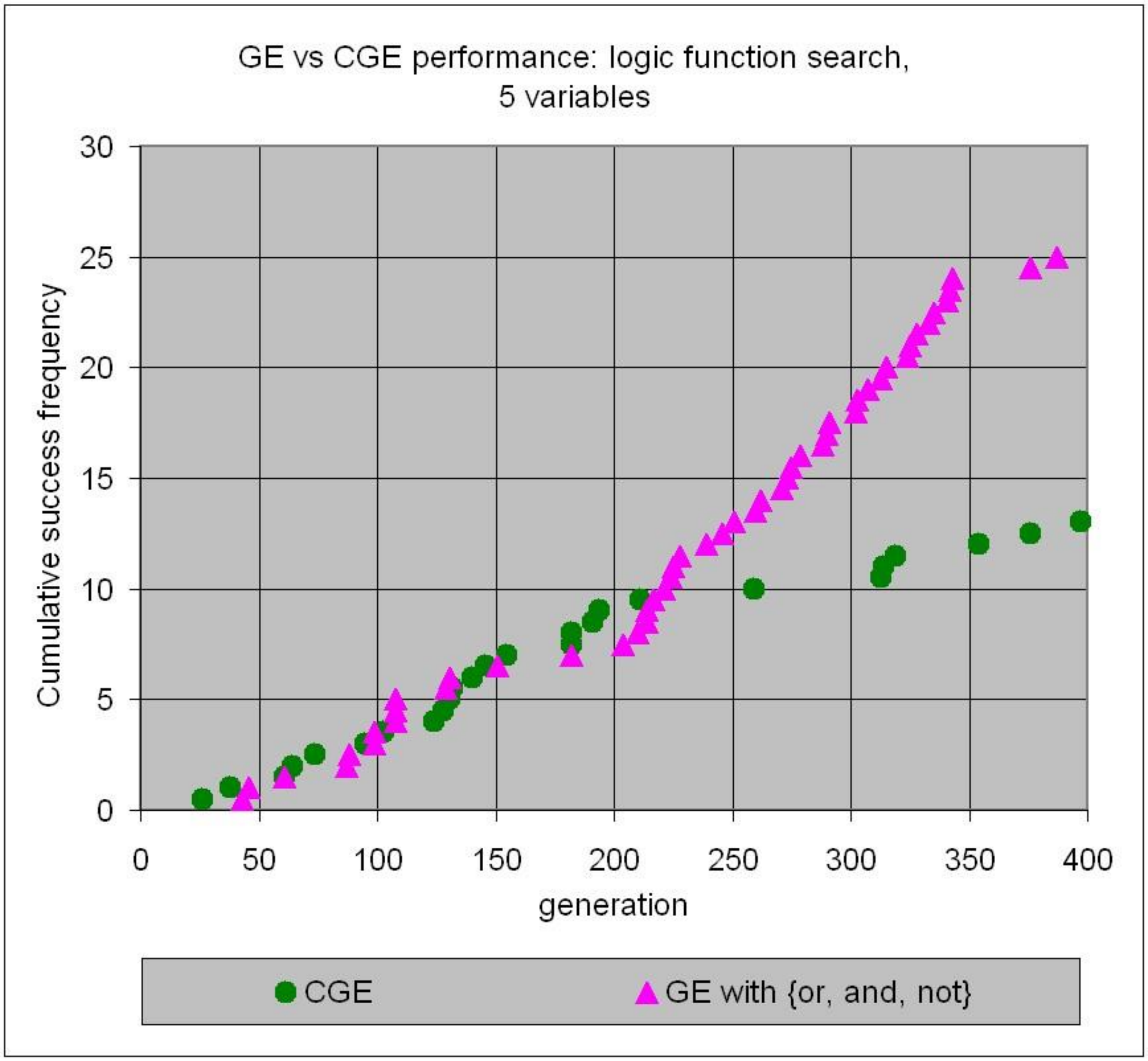
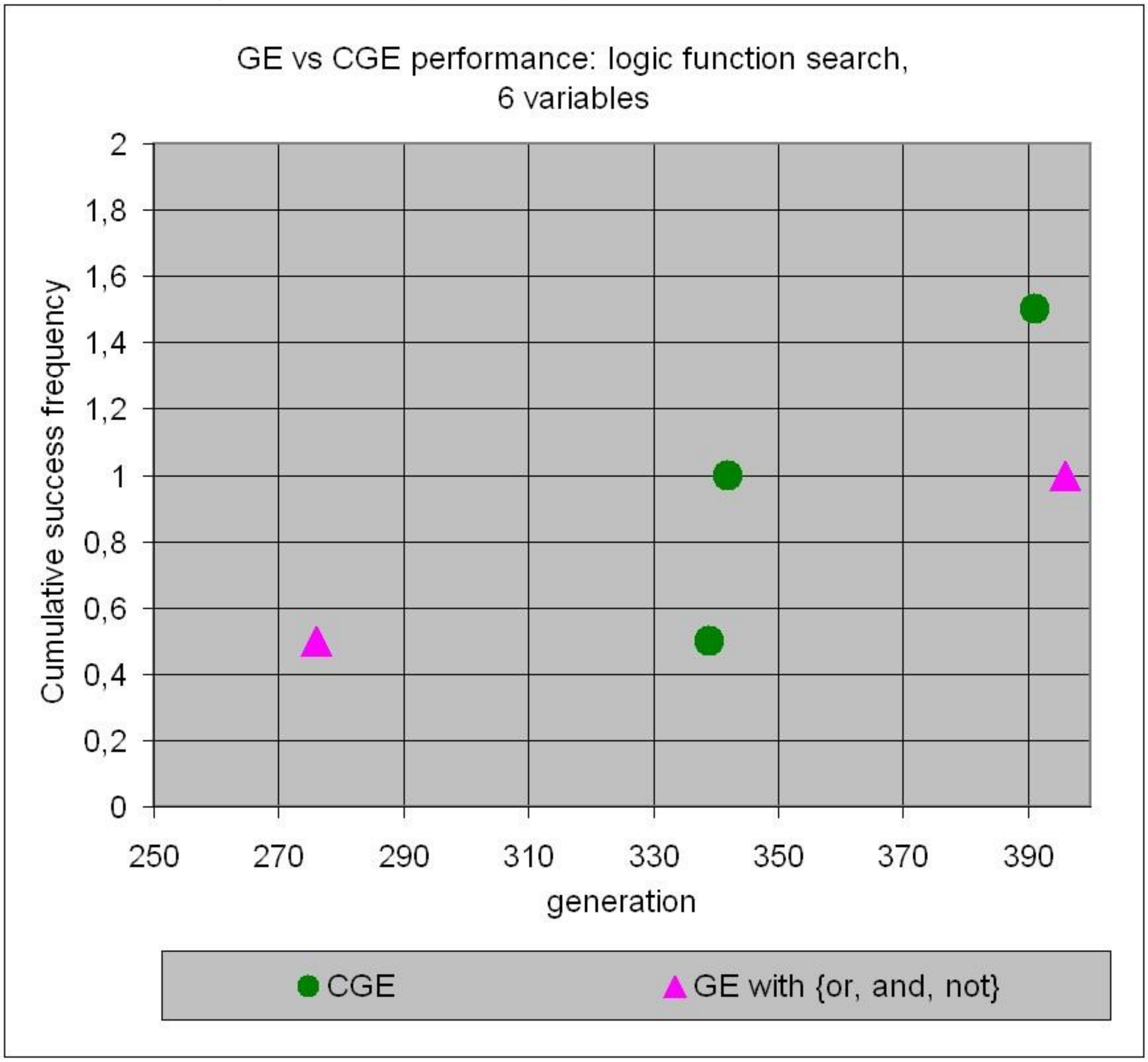


Figure 11: CGE vs. GE performance comparison for the third experiment: 6 variables


GE vs CGE performance: logic function search, 6 variables
Cumulative success frequency
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
250
270
290
310
330
350
370
390
generation
● CGE
▲ GE with {or, and, not}