A memetic grammar inference algorithm for language learning

Dejan Hrnčiča,∗, Marjan Mernika,†, Barrett R. Bryantb, Faizanc

a University of Maribor, Faculty of Electrical Engineering and Computer Science, Maribor, Slovenia
b The University of Alabama at Birmingham, Department of Computer and Information Sciences, Birmingham, AL, USA
c Intermark Group, Interactive Division, Birmingham, AL, USA

A R T I C L E   I N F O

Article history:
Received 4 March 2010
Received in revised form 15 June 2011
Accepted 27 November 2011
Available online 11 December 2011

Keywords:
Rule learning
Incremental learning
Grammatical inference
Memetic algorithm
Evolutionary computations

A B S T R A C T

An unsupervised incremental algorithm for grammar inference and its application to domain-specific language development are described. Grammatical inference is the process of learning a grammar from the set of positive and optionally negative sentences. Learning general context-free grammars is still considered a hard problem in machine learning and is not completely solved yet. The main contribution of the paper is a newly developed memetic algorithm, which is a population-based evolutionary algorithm enhanced with local search and a generalization process. The learning process is incremental since a new grammar is obtained from the current grammar and false negative samples, which are not parsed by the current grammar. Despite being incremental, the learning process is not sensitive to the order of samples. All important parts of this algorithm are explained and discussed. Finally, a case study of a domain-specific language for rendering graphical objects is used to show the applicability of this approach.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

Under the broad umbrella of Evolutionary Algorithms (EAs) [1] are algorithms that mimic models and mechanisms from biological evolution (e.g., genetic algorithms, evolution strategies, evolutionary programming, genetic programming). Such nature inspired EAs simulate evolution and its mechanisms such as selection, crossover, and mutation. They have been used successfully for planning, design, simulation and identification, controlling, classification, and for solving many other hard optimization problems [1]. However, many studies show that evolutionary algorithms produce better results if they are hybridized with techniques such as local search and domain-specific knowledge [2].

The term memetic, i.e., was for the first time used in [3] to represent EA which was blended with local search. We successfully applied a memetic algorithm in grammatical inference [4], which is the process of learning the grammar from positive and optionally negative sentences. Grammatical inference is a subfield of Machine Learning (ML), where the grammar needs to be learned from artifacts represented as sentences/programs written in some unknown language. The solution to this problem can be successfully applied to various problems in pattern recognition, computational biology, computational linguistics, speech recognition and natural language acquisition. In this paper we show how grammatical inference can be applied to the process of Domain Specific Language (DSL) development [5]. Such a scenario would be feasible when domain experts can provide complete DSL programs or excerpts of such programs. The results of grammatical inference, namely the inferred grammar, can be directly used to generate the DSL parser or be further examined by a software language engineer with the aim to further enhance the design of the language.

Grammatical inference also poses many theoretical problems. As an implication of Gold's theorem [6] learning algorithms have been developed that exploit knowledge of negative samples, structural information, or are applied to restrictive subclasses of grammars [7]. Yet on the other hand, exhaustive search is not feasible due to the immense search space in grammatical inference. Hence, a need for a different and more efficient approach to explore the search space arose. All this culminated in a memetic algorithm for grammatical inference, which is presented in this paper with some applications to DSL development.

The structure of this paper is as follows. In Section 2, the background and motivation of this work are discussed. Section 3, the heart of the paper, presents MAGIC – the memetic grammatical inference algorithm. In Section 4 we provide experimental results of grammatical inference on tiny languages, as well as on a realistic example from the graphics domain. In Section 5 related work is discussed. We conclude in Section 6.

2. Background and motivation

Any language, whether natural or artificial, has some well defined syntactic structure, which can be succinctly described with a context-free grammar (CFG). Formally, a CFG G is a quadruple

...
\[ G = (N, T, P, S). \] 
\[ N \text{ is a finite set of nonterminal symbols, } T \text{ is a finite set of terminal symbols such that } N \cap T = \emptyset. \]
P represents a finite set of production rules of the form \( A ::= a \) for \( A \in N \) and \( a \in V^* (V = N \cup T) \) representing a string (sequence) of nonterminal and terminal symbols.

Note, that this sequence can be empty (\( A ::= \epsilon \)). \( S \) represents a special nonterminal called the start symbol. Informally, \( T \) is a set of symbols that we actually use when writing in the language, and 
\[ N \] is a set of symbols representing a particular class of phrases (e.g., expressions, commands, declarations). Nonterminal symbols are usually written using capital letters (or at least first letter is a capital letter), while terminal symbols are usually written in non capital letters.

Production rules \( P \) specify how phrases are composed from terminal and nonterminal symbols, while the starting nonterminal symbol \( S \) represents a root class of all phrases.

If a string \( a \in (N \cup T)^* \) can be derived from nonterminal \( A \), we write \( A \Rightarrow^* a \). In a case \( A = S \) and \( a \in T^* \) a string \( a \) is called a sentence of our language. Formally, a language generated by a CFG \( G \) is represented with \( L(G) \), where \( L(G) = \{ w | S \Rightarrow^* w \text{, } w \in T^* \} \).

Informally, a language \( L(G) \) is a set, finite or infinite, of all sentences which can be derived from starting nonterminal \( S \).

A parse tree of CFG is an ordered labeled tree having the following properties:

\begin{itemize}
  \item a root node is labeled by \( S \) – starting nonterminal,
  \item production \( p \in P \) of the form \( A ::= V_1, V_2, \ldots, V_n \) represents a subtree with root node \( A \) and subtrees labeled \( V_1, V_2, \ldots, V_n \),
  \item terminal symbol \( t \in T \) represents a leaf or terminal node labeled by \( t \).
\end{itemize}

Recognition of a string \( a \) in a grammar \( G \) is deciding whether or not a string \( a \) is a sentence of \( L(G) \). Parsing of a string \( a \) in a grammar \( G \) is recognition and construction of a parse tree. Parsing a string \( a \) which is not a sentence will yield no parse tree. A grammar is ambiguous if a string \( a \) can be parsed in multiple ways, leading to different parse trees.

From now on, a language DESK [8] will be used as an example to describe the aforementioned definitions and how our algorithm for grammatical inference works. The DESK grammar productions \( P \) are presented in Fig. 1.

From Fig. 1 we can notice that:

\begin{itemize}
  \item \( N = \{ \text{DESK, E, F, C, Ds, D} \} \),
  \item \( T = \{ \text{num, id, print, where, =, +, ;} \} \),
  \item \( S = \text{DESK} \).
\end{itemize}

From set \( T \) it can be noticed that some terminal symbols also must have some simple structure (e.g., terminal symbol \( \text{num} \) represents a sequence of digits). For describing the structure of terminal symbols, in other words the language syntax, regular expressions [9] are used which are less powerful than CFGs and only suitable for describing simple structures. Regular expressions for the DESK language are:

\begin{verbatim}
Regular expression            Terminal symbol
[0-9]num                      | id
[a-z]+print                  | print
\end{verbatim}

Hence, before parsing a sentence, a stream of characters needs to be transformed into a stream of terminal symbols (also called tokens). This is done in the phase of lexical analysis, which is the first phase of a compiler [9]. A lexical analyzer reads the stream of characters given as input and groups the characters into sequences of terminal symbols. For example, \text{print} a where \( a = 10 \) is transformed after lexical analysis into \text{print id where id = num}.

Some other sentences which can also be derived from starting nonterminal DESK are:

\begin{verbatim}
print 8
print abc +25 where abc = 1
print a +b where a = 5; b = 6
\end{verbatim}

The sentence \text{print a where a = 10} belongs to language DESK since it can be derived from starting nonterminal DESK:

\begin{verbatim}
DESK  \Rightarrow print E C
      \Rightarrow print F C
      \Rightarrow print id C
      \Rightarrow print id where Ds
      \Rightarrow print id where id = num
or DESK \Rightarrow print id where id = num
\end{verbatim}

The parse tree for the sentence \text{print a where a = 10} is presented in Fig. 2.

The aim of parsing is to build a parse tree, explicitly showing a syntactical structure of a sentence. This can be done in a top-down manner, building a parse tree from the starting nonterminal downwards to the leaves, or in a bottom up manner, building a parse tree from the leaves upwards to the starting nonterminal. Hence many parsing algorithms for CFGs exist. The LR(k) parsers, where \( k \) stands for a number of lookahead terminal symbols that are used in making parsing decisions, are the most prevalent type of bottom-up parsers today [9]. The LR(1) parser is used also in our approach.

Using the CFG formalism we are now able to solve the membership problem. Given a sentence \( ps \) and CFG \( G \) we can tell whether \( ps \) belongs to \( L(G) \) (\( ps \in L(G) \)). Such sentence is also called positive sample. The set of positive samples is denoted by \( S^+ \). Conversely, a negative sample \( ns \) belongs to \( L(G)^c \), where \( L(G)^c = \{ w | w \notin L(G) \} \). The set of negative samples is denoted by \( S^- \).

Some negative samples for the DESK language are:

\begin{verbatim}
print where a = 1
print a where 10 = 11
print b; c = 3
\end{verbatim}
A set of positive samples \( S^+ \) of a language \( L(G) \) is structurally complete if each grammar production is used in the generation of at least one sentence in \( S^+ \).

The grammar inference process (grammar induction or grammar learning) can then be formulated as follows. Given a set of positive samples \( S^+ \) and set of negative samples \( S^- \), which might be also empty, find at least one grammar \( G \) such that \( S^+ \subseteq L(G) \) and \( S^- \subset L(G) \). However, there is no unique relationship between a language and a grammar since many different grammars may generate the same language. Two grammars are equivalent, \( G_1 = G_2 \), if and only if \( L(G_1) = L(G_2) \). Hence, there is no unique solution for the grammar inference problem unless we pose additional constraints upon a grammar being inferred (e.g., minimum complexity).

Grammar inference has been investigated now for more than 40 years, and has found applications in several research domains such as language acquisition, pattern recognition, computational biology, and software engineering. In language acquisition a child, being exposed only to positive samples, is able to discover the syntactic representation of the language \( G \). The aim of research on grammar inference is to provide different models how language acquisition takes place [10]. Grammars have been also used as an efficient representation of artifacts that are inherently structural and/or recursive (e.g., neural networks, structured data and patterns) [11]. In pattern recognition, pattern grammars are used for pattern description and recognition [12]. Such a pattern grammar consists of primitives (e.g., circle, square, line), a set of predicates that describe the structural relationships among defined primitives (e.g., left, above, inside), and a set of productions which describe composition of predicates and primitives. Given the set of patterns the problem is to infer a pattern grammar that fits the given set of patterns. In computational biology grammar inference has been used for analysis of DNA, RNA, or protein sequences. For example, grammar inference has been successfully applied to predict secondary structures and functions of the biological molecules [13]. An early application of grammar inference in software engineering was programming language design [14], where an inference algorithm for a very restricted grammar, operator precedence grammar, has been proposed.

So far, grammar inference has been mainly successful in inferring regular languages. Researchers have developed various algorithms which can learn regular languages from positive and negative samples. A number of algorithms (e.g., RPNI [15]) first construct the automaton from positive samples, and generalize the automaton by using a state merging process. By merging states, an automaton is obtained that accepts a bigger set of strings and generalizes according to the increasing number of positive samples presented. CFG inference is more difficult than regular grammar inference. Using structurally complete positive samples along with negative samples did not result in the same level of success as with regular grammar inference. Hence, some researchers resorted to using additional knowledge to assist in the inference process. Sakakibara [16] used a set of skeleton parse trees (unlabeled parse trees), where the input to the inferring process is sentences with parentheses inserted to indicate the shape of the parse tree. An enhancement to this algorithm was proposed in [17], where CFG inference was possible from partially structured sentences. However, in many application domains it is impractical to assume that completely or partially structured samples exist. Despite the fact that many researchers have looked into the problem of CFG inference [18–21] there has been no one convincing solution to the problem yet.

In this work we have been interested in one of the earliest applications of grammar inference, namely programming language design. Our research is oriented towards various aspects (e.g., domain analysis, design, implementation) of domain-specific languages (DSLs) [5]. In contrast with general-purpose languages (GPLs), where one can address large classes of problems (e.g., scientific computing, business processing, symbolic processing, etc.), a DSL facilitates the solution of problems in a particular domain (e.g., aerospace, automotive, graphics, etc.). One of the open problems in DSL research stated in the survey paper on DSLs [5] is: “How can DSL design and implementation be made easier for domain experts not versed in GPL development?” Namely, domain experts are not versed in compilers and designing languages, but know how to express problems and their solutions in their domain of expertise. In other words, they know domain notations and abstractions and can provide DSL programs. Here, grammar inference can find the underlying structure of the provided DSL programs. Hence, a DSL grammar can be constructed and a DSL parser generated. On the other hand, the inferred grammar can be further examined by a software language engineer with the aim to further enhance the design of the language. DSLs, also called little languages, are usually small and declarative. Hence, it is more likely that the grammar inference process would be successful. Note, that the syntax structure of GPLs is too complex for current grammar inference algorithms to be successful. Overall, our main motivation is to, at least partially, solve one of the open problems in DSL research.

3. MAGIc

MAGIc is a Memetic Algorithm for Grammatical Inference (Fig. 3). A memetic algorithm [3] is a population-based evolutionary algorithm enhanced with local search. Using only positive samples, MAGIc can infer CFGs, which are non-ambiguous and of type LR(1). MAGIc is composed of the following steps, which are graphically shown also in Fig. 3:

- initialization: in this step the initial population is constructed;
- improve and evaluate: the local search operator;
- mutation: grammars are exposed to random changes;
- generalization: grammars are generalized and simplified;
- selection: the best grammars are selected for the next generation.

The algorithm makes the first initialization step only once, at the beginning of the algorithm run. The next four steps represent the evolutionary cycle and are repeated until the exit condition, the maximum number of generations, is met. In the evolutionary cycle the population individuals are changed through the local search, mutation and generalization step. Finally the best individuals are selected and compose the next generation. In this section all algorithm steps are discussed in detail.

3.1. Initialization

For the initial population, grammars are generated using the Sequitur algorithm [22], that detects repetition in a sample and factors it out by forming grammar rules. Note that the generated grammar by Sequitur only parses a sample from which the grammar was generated. The size of the initial population is therefore equal to the number of input samples.

For example, Sequitur produces for the sample print id where id = num; id = num the following grammar rules:

\[
N1:=\ \text{print id where } N2; N2
\]

N2 := id = num

Note, that above grammar does not parse the aforementioned statements:

\[
\text{print } 8
\]

\[
\text{print abc +25 where abc = 1}
\]

\[
\text{print a +b where } a = 5; b = 6
\]
3.2. Local search

After the initial population is built, each generated grammar parses one positive sample. In the inference process each (partial) grammar divides the positive samples into two groups. The successfully parsed positive samples are named true positive samples and unsuccessfully parsed positive samples are named false negative samples. MAGiC tries to improve partial grammars to be able to parse more positive samples, by adding new or extending existing productions. To achieve this task, differences identified among samples are used. From the differences, new productions are made and inserted at proper positions into the existing grammars. For comparing samples we used the Linux diff command \cite{difftool} that returns the difference between two files, which can be ADD, REPLACE, or DELETE. The diff command compares files line by line. The algorithm is based on solving the longest common subsequence problem (LCS) \cite{longest_common_subsequence}. Since comparison has to be done at the token level, not at the character level, we inserted the lexical analysis phase \cite{lexical_analysis} before the diff command.

After the differences among samples are identified the objective is to change the grammar in such a manner that both samples are parsable. Here MAGiC exploits the parsing history (an LR(1) parser is used). When the parser fails to parse the sample, information about the parser stack and the LR(1) item set \cite{lexical_analysis} are returned. Because the parser puts the processed states on the stack, it is also possible to examine all previous states and therefore also the parsing history.

Fig. 3. The memetic algorithm for grammatical inference.
The form of an LR(1) item is $[A \rightarrow \alpha \cdot \beta]$, where $A::=\alpha\beta$ is a production. An LR(1) item indicates how much of a production has been recognized at a given point in the parsing process. The previous example indicates that we have just recognized a string derivable from $\alpha$ and that we expect to recognize a string derivable from $\beta$. Fig. 4 shows a goto graph containing LR(1) items for the simple grammar obtained with Sequitur algorithm.

**Example.** Let us suppose we have a grammar $G_1$:

\[
N1::= \text{print id where id = num} \\
N2::= \text{id}
\]

Since the statement doesn’t belong to $L(G_1)$, the LR(1) parser cannot parse this sample. When the parse error occurs, the item set returned is:

\[
N1 \rightarrow \text{print id where id = num} \\
N1 \rightarrow \text{id}
\]

which means that tokens in positions $k+1$ to $k+n$ ($a_k$) from false negative sample have to be added into the true positive sample after the token in position $k$.

When the difference between samples is identified, the position in the grammar where to add this difference needs to be found. The state’s configurations on the LR(1) parser stack where an error occurs can be of the following forms:

- $Nx \rightarrow \alpha_1 \cdot \alpha_2$
- $Ny \rightarrow \beta$
- $Nz \rightarrow \gamma$

If configuration $Nx \rightarrow \alpha_1 \cdot \alpha_2$ is returned, then the first check is if $s_{k+1} \in \text{FIRST}(\alpha_2)$, where set $\text{FIRST}(\alpha_1)$ includes all terminal symbols that appear first in $\alpha$ [9]. If true then the following productions are inserted into the grammar (note that the parser might recognize $i$ tokens from the difference):

\[
N_x::= \alpha_1 N_2 \alpha_2 \\
N_1::= \alpha_1 \ldots \alpha_n \\
N::= \cdot
\]

**Example.** Let us suppose we have a grammar $G_2$:

\[
N1::= \text{print id where id = num} \\
N2::= \text{id}
\]

The LR(1) parser for grammar $G_2$ successfully parses sample 1, but fails to parse sample 2. The next step is to identify the difference between both samples. The $\text{diff}$ command returns:

\[
2 \rightarrow 3 \rightarrow 4 \\
> + \\
> \cdot \text{id}
\]

which means: add the third and fourth token from sample 2 after the second token of sample 1. To determine the position where to insert the difference inside the grammar, the state of the parser is used at the time the parse error occurs by parsing the false negative sample. The item set on top of the parser stack is:

\[
N1 \rightarrow \text{print id where id = num}
\]

The token following the difference ($s_{k+1}$) is where it is in $\text{FIRST}(\text{id = num})$, therefore the new nonterminal $N_2$ can be inserted at the position of the dot and the extended grammar is:

\[
N1::= \text{print id where id = num} \\
N2::= \text{id} \\
N2::= \cdot
\]

If $s_{k+1} \notin \text{FIRST}(\text{id})$, then it is checked if $s_{k+1} \in \text{FOLLOW}(Nw)$, where set $\text{FOLLOW}(N)$ includes all terminal symbols that first appear after nonterminal $N$ [9]. If true then the following productions are inserted into the grammar:

\[
N_x::= \alpha_1 N_2 \\
N_1::= \alpha_2 \\
N::= \cdot
\]

**Example.** Let us suppose we have a grammar $G_3$:

\[
N1::= \text{print id where id = num} \\
N2::= \text{id} \\
N2::= \cdot
\]

and two samples of the DESK language (after lexical analysis):

1. $\text{print id where id = num}$ (true positive)
2. $\text{print id = id where id = num}$ (false negative)

The difference returned by the $\text{diff}$ command is:

\[
2 \rightarrow 3 \rightarrow 4 \\
> + \\
> \cdot \text{id}
\]

which means: add the third and fourth tokens from sample 2 after the second token of sample 1. The item set on top of the parser stack by parsing the false negative sample is:

\[
N2 \rightarrow \cdot \text{id}
\]
The token following the difference (\(s_{k+1}\)) is where and it is not in \(\text{FIRST}(\text{id})\), but it is in \(\text{FOLLOW}(\text{Nz})\). Therefore new nonterminal \(\text{N}3\) can replace the part from the dot position until the end of the production and the extended grammar is:

\[
\begin{align*}
\text{N}1::= & \text{print id N2 where id = num} \\
\text{N}2::= & \text{+ N3} \\
\text{N}3::= & \text{e} \\
\text{N}3::= & \text{id} \\
\text{N}3::= & \text{num} \\
\end{align*}
\]

The grammar cannot be changed at this configuration if \(s_{k+1} \notin \text{FIRST}(\gamma) \land s_{k+1} \notin \text{FOLLOW}(\text{Nz})\) and the algorithm goes to the next configuration returned from the parser.

The next form of parser configuration, that can also be returned, is \(\beta \rightarrow \gamma\). In this case we check if \(s_{k+1} \in \text{FOLLOW}(\text{Ny})\). If this is true, then the following productions are inserted into the grammar:

\[
\begin{align*}
\text{Ny}::= & \beta \text{N}1 \\
\text{N}1::= & \text{a}_{1}...\text{a}_{4} \\
\end{align*}
\]

**Example.** Let us suppose we have a grammar \(G_4\):

\[
\begin{align*}
\text{N}1::= & \text{print id N2 where id = num} \\
\text{N}2::= & \text{id} \\
\text{N}2::= & \text{e} \\
\text{N}3::= & \text{+ num} \\
\end{align*}
\]

and two samples of the DESK language (after lexical analysis):

1. \(\text{print id +id where id = num}\) (true positive)  
2. \(\text{print id +id where id = num}\) (false negative)

The difference returned by the \textit{diff} command is:

\[
4a5.6 \\
> + \\
>num
\]

which means: add the fifth and sixth tokens from sample 2 after the fourth token of sample 1. The item set on top of the parser stack by parsing the false negative sample is:

\[
\text{N}2 \rightarrow +id \]

The token following the difference (\(s_{k+1}\)) is where and it is in \(\text{FOLLOW}(\text{N}2)\). Therefore new nonterminal \(\text{N}3\) can be inserted at the end of production and the extended grammar is:

\[
\begin{align*}
\text{N}1::= & \text{print id N2 where id = num} \\
\text{N}2::= & \text{id N3} \\
\text{N}2::= & \text{e} \\
\text{N}3::= & \text{+num} \\
\end{align*}
\]

If \(s_{k+1} \notin \text{FOLLOW}(\text{Ny})\) then change the grammar at this configuration cannot be made and the next one is processed.

In case of configuration \(\text{N}2 \rightarrow \gamma\) we first check if \(s_{k+1} \in \text{FIRST}(\gamma)\). If this check is true then the following productions are inserted into the grammar:

\[
\begin{align*}
\text{N}1::= & \text{N1 \gamma} \\
\text{N}1::= & \text{a}_{1}...\text{a}_{4} \\
\text{N}1::= & \text{e} \\
\end{align*}
\]

**Example.** Let us suppose we have a grammar \(G_5\):

\[
\begin{align*}
\text{N}1::= & \text{print N2 id where id = num} \\
\text{N}2::= & \text{id} \\
\text{N}2::= & \text{+} \\
\text{N}3::= & \text{num} \\
\end{align*}
\]

and two samples of the DESK language (after lexical analysis):

1. \(\text{print id +id where id = num}\) (true positive)  
2. \(\text{print num +id where id = num}\) (false negative)

The difference returned by command \textit{diff} is:

\[
1a2.3 \\
>num \\
>+
\]

which means: add the second and third tokens from sample 2 after the first token of sample 1. The item set on top of the parser stack from parsing the false negative sample is:

\[
\text{N}2 \rightarrow +id \\
\]

The token following the difference (\(s_{k+1}\)) is id and it is in \(\text{FIRST}(\text{id}+)\). Therefore new nonterminal \(\text{N}3\) can be inserted at the position of the dot in the production and the extended grammar is:

\[
\begin{align*}
\text{N}1::= & \text{print N2 id where id = num} \\
\text{N}2::= & \text{N3 id} \\
\text{N}2::= & \text{e} \\
\text{N}3::= & \text{num} \\
\text{N}3::= & \text{e} \\
\end{align*}
\]

If \(s_{k+1} \notin \text{FIRST}(\gamma) \land s_{k+1} \notin \text{FOLLOW}(\text{N}z)\) is true, then the following productions are inserted into the grammar:

\[
\begin{align*}
\text{N}2::= & \gamma \\
\text{N}2::= & \text{a}_{1}...\text{a}_{4} \\
\end{align*}
\]

Note, that this case could not happen in the DESK example.

In case that \(s_{k+1} \notin \text{FIRST}(\gamma) \land s_{k+1} \notin \text{FOLLOW}(\text{N}z)\) then the grammar at this configuration cannot be changed and the next configuration returned from the stack is processed.

3.4. REPLACE difference

If the true positive sample is \(s_{1}^{1.2}...s_{k}^{k+1}...s_{m}^{m+1}...s_{k+1}^{k+1}\) and the false negative is \(s_{1}^{1.2}...s_{k}^{k+1}...s_{m}^{m+1}...s_{k+1}^{k+1}\) the output of the \textit{diff} command is:

\[
k + 1, k + n \n c k + 1, k + m
\]

which means, that the tokens on positions \(k + 1\) to \(k + n + 1\) from the true positive sample have to be replaced with tokens on positions \(k + 1\) to \(k + n + m\) from the false negative sample.

In order to successfully parse the false negative sample, a production with the \(b_{1}...b_{m}\) part of the false negative has to be added into the grammar. This new production represents an alternative for part of the grammar, where the first symbol is \(a_{1}\) and the last symbol is \(a_{n}\). The problem here is to find the beginning and the end where to insert the alternative production. To solve this problem, we modified the LR(1) parser. When the true positive sample is parsed, we memorize for each token the dots in the configuration items before and after this token. This way we can get all the configuration items where the dots are before \(a_{1}\) and all configuration items where the dots are after \(a_{n}\). The new grammar can be made when the dot before \(a_{1}\) and the dot after \(a_{n}\) are in the same production. The latter can be seen as a limitation of our approach and will be removed in the future. Hence, symbols \(a_{1}\) to \(a_{n}\) can be spread through many productions.

A case where the grammar can be changed is shown in production \(\text{N}x::= \text{a}_{1} < \beta > \text{a}_{2}\), where \(\beta\) represents part of grammar containing tokens \(a_{3}...a_{n}\). Note that \(a_{1}\) and \(a_{2}\) can also be \(e\). For easier understanding we introduce a symbol < that represents the dot before \(a_{1}\) and symbol > that represents the dot after \(a_{2}\). Both dots are in the same production, therefore \(\beta\) can be replaced with a new nonterminal and the following change to the grammar is made:

\[
\begin{align*}
\text{N}x::= & \text{a}_{1} \text{ N1} \text{ a}_{2} \\
\text{N}1::= & \beta \\
\text{N}1::= & \text{b}_{1}...\text{b}_{m} \\
\end{align*}
\]

**Example.** Let us suppose we have a grammar \(G_6\):

\[
\begin{align*}
\text{N}1::= & \text{print id where id = num} \text{ and two samples of the DESK language (after lexical analysis):} \\
1. & \text{print id where id = num} \text{ (true positive)} \\
2. & \text{print num where id = num} \text{ (false negative)} \\
\end{align*}
\]

The difference returned by command \textit{diff} is:

\[
2>2 \\
\text{id} \\
>num
\]

\[
\text{\textit{diff} command is:}
\]
which means: replace the second token of sample 1 with the second token of sample 2. Combining the configuration points from item sets obtained by parsing the true positive sample after first token (beginning of difference) and after the second token (end of difference) returns:

\[ N1 \rightarrow \text{print } < \text{id} > \text{where id} = \text{num} \text{where the < sign represents the configuration point after parsing the first token of the true positive sample and the > sign represents the configuration point after parsing the second token of the true positive sample. Since both configuration points are in the same production, the difference from true positive sample can be replaced with the difference from the false negative sample. The extended grammar is:} \]

\[
N1::= \text{print } N2 \text{ where id} = \text{num} \\
N2::= \text{id} \\
N2::= \text{num} \\
\]

**3.5. DELETE difference**

If the true positive sample is \(s_1^1 s_2^1 \ldots s_{k+n}^1\) and the false negative is \(s_1^2 s_2^2 \ldots s_{k+n}^2\), the output of the `diff` command is:

\[
k + 1, k + n + k \left< a_1 < \ldots < a_n = \right.
\]

which means, that the tokens in positions \(k + 1\) to \(k + n\) from the true positive sample have to be removed.

In this case part of the true positive sample \(a_1 \ldots a_n\) has to be found in the grammar and made optional. To find the part \(a_1 \ldots a_n\) the configuration dots before \(a_1\) and after \(a_n\) are used. This information is obtained from the output of parsing the true positive sample. The grammar can be changed if both configuration dots are in the same production. Again, this limitation will be removed in the future.

In the case of production \(N_{x::=} = \alpha_1 < \beta > \alpha_2\) the following change to the grammar is made:

\[
N_{x::=} = \alpha_1 N_{z::=} \alpha_2 \\
N_{z::=} = \beta \\
N_{z::=} = \varepsilon
\]

**Example.** Let us suppose we have a grammar \(G_f:\)

\[ N1::= \text{print } \text{id} + \text{id} \text{ where id} = \text{num} \text{ and two samples of the DESK language (after lexical analysis):} \]

1. \(\text{print } \text{id} + \text{id} \text{ where id} = \text{num} \) (true positive)
2. \(\text{print } \text{id} \text{ where id} = \text{num} \) (false negative)

The difference returned by command `diff` is:

\[
3. \text{4} \text{ id 2} \\
< = < \\
< \text{id}
\]

which means delete the third and fourth tokens from sample 1. Combining the configuration points from item sets obtained by parsing the true positive sample after the first token (beginning of difference) and after the second token (end of difference) returns:

\[ N1 \rightarrow \text{print } < \text{id} > \text{where id} = \text{num} \text{where the < sign represents the configuration point after parsing the second token of the true positive sample and the > sign represents the configuration point after parsing the fourth token of the true positive sample. Since both configuration points are in the same production, the difference from true positive sample can be made optional. The extended grammar is:} \]

\[
N1::= \text{print } N2 \text{ where id} = \text{num} \\
N2::= \text{id} \\
N2::= \varepsilon
\]

**3.6. Mutation**

After local search grammars undergo transformation through mutation. The mutation exploits the domain-specific knowledge of grammars, namely extended Backus Naur Form (EBNF), where grammar symbols often appear optionally or iteratively (option operator, iteration* operator, and iteration’ operator). For each grammar symbol that appears on the right-hand side of a production a random value is generated. If this value is smaller than the probability of mutation \(p_m\) then the grammar symbol is mutated. Option, iteration*, or iteration’ operators are chosen randomly. The results of these operators on the production \(N_{x::=} = \alpha_1 \text{ny} \alpha_2\), where grammar symbol `ny` (nonterminal or terminal) is selected for mutation, are:

**Option operator:**

\[
N_{x::=} = \alpha_1 \text{ny} \alpha_2 \\
N_{y::=} = \text{ny} \\
N_{z::=} = \text{ny}
\]

**Iteration* operator:**

\[
N_{x::=} = \alpha_1 \text{ny} \alpha_2 \\
N_{y::=} = \text{ny} \text{ny} \\
N_{z::=} = \text{ny}
\]

**Iteration’ operator:**

\[
N_{x::=} = \alpha_1 \text{ny} \alpha_2 \\
N_{y::=} = \text{ny} \text{ny} \\
N_{z::=} = \text{ny}
\]

**Example.** Let us suppose we have a grammar \(G_f:\)

\[ N1::= \text{print } \text{id} N2 \text{ where id} = \text{num} \\
N2::= + \text{id} \\
N2::= \varepsilon
\]

and that the mutation operator selects the third element from the first production \(N_{z::=} \) for mutation. After the grammar element is selected, the type of mutation is randomly chosen. Let us suppose that the iteration’ operator is chosen. In this case the grammar is changed to:

\[ N1::= \text{print } \text{id} N3 \text{ where id} = \text{num} \\
N2::= + \text{id} \\
N2::= \varepsilon \\
N3::= N2 N3 \\
N3::= \varepsilon
\]

**3.7. Generalization**

After mutation an important step of generalization is performed. Since our goal is not to over-generalize grammars only simple rules are applied. Note also that we keep the original grammars in the intermediate population. New grammars which are found by a generalization process do not replace them; instead the new grammars are added to the current population.

**Step 1.** In the first step the algorithm is searching for nested structures. In particular, all productions of the form

\[
\text{Nxx::= } \text{ny} \text{ ny} \text{ nx} \text{ nx} \text{ are generalized into the form} \\
\text{Nxx::= } \text{ny} \text{ ny} \text{ nx} \text{ nx}
\]

where \(\text{ny}\) and \(\text{nx}\) are grammar symbols (nonterminals or terminals).

Since the DESK language contains no recursive structures, this step cannot be shown on the DESK example.

**Step 2.** In the second step, productions where the right-hand side (RHS) of another production appears in that production are replaced with the left-hand side (LHS) symbol. For example, in the grammar

\[
\text{Nx::= } \text{ny} \\
\text{Ny::= } \text{ny} \\
\text{Ny::= } \text{nx}
\]

the symbol \(\text{nx}\) represents the RHS of the second production. This RHS can be also found in first production before nonterminal \(\text{ny}\). Therefore it is replaced with the nonterminal from the LHS of the second production and the grammar is generalized to

\[
\text{Nx::= } \text{ny} \text{ ny} \\
\text{Ny::= } \text{nx}
\]
Example. Let us suppose we have a grammar $G_0$:

\[
\begin{align*}
N_1 &::= \text{print num } N_2 \text{ where id } = \text{num} \\
N_2 &::= + N_3 \\
N_2 &::= e \\
N_3 &::= \text{num } N_2 \\
N_3 &::= \text{id } N_2
\end{align*}
\]

Using the second generalization step, the RHS of the fourth production ($\text{num } N_2$) is found and replaced in the first production:

\[
\begin{align*}
N_1 &::= \text{print } N_3 \text{ where id } = \text{num} \\
N_2 &::= + N_3 \\
N_2 &::= e \\
N_3 &::= \text{num } N_2 \\
N_3 &::= \text{id } N_2
\end{align*}
\]

**Step 3.** In the next step a repeated sequence of symbols is searched for. If such a sequence is found there is a possibility for iteration of symbols. For example, the result of the second step contains two repeating nonterminals $N_2$ $N_2$. Here are two possibilities, the iteration and iteration” can be made. After the third step the grammar is generalized to:

\[
\begin{align*}
N_2 &::= N_2 N_2 \\
N_2 &::= \alpha N_2 \\
N_2 &::= \beta N_2 \\
N_2 &::= e
\end{align*}
\]

**Step 4.** Lastly, the simplification of grammars is performed and unit productions [9], those with only one non-terminal on the right-hand side, are removed. If the previous grammars are simplified, they become:

\[
\begin{align*}
N_2 &::= \alpha N_2 \\
N_2 &::= \beta N_2 \\
N_2 &::= e
\end{align*}
\]

and

\[
\begin{align*}
N_2 &::= \alpha N_2 \\
N_2 &::= \beta N_2 \\
N_2 &::= e
\end{align*}
\]

3.8. Selection

The last step of the evolutionary cycle (see Fig. 3) is selection. We are using deterministic selection where all grammars are ranked based on their fitness and only the best $pop \_size$ grammars are selected into the next population. Note that during the evolutionary cycle, the population of grammars is not fixed and can grow, but at the end of the evolutionary cycle only $pop \_size$ grammars survive.

Note also that currently the neutral solutions, which have the same fitness value, are not differentiated between each other and are selected for the next population randomly. The first $pop \_size$ grammars survive, which means if there are more grammars with equal fitness value, some of them are lost. The extensions, which could handle such cases, are planned for future work.

For the fitness function we experimented with several options. The simple measure (i.e., the number of positive samples that are parsed) performed well. Since MAGIc learns from positive samples only, we will investigate the use of the MDL principle [25] to prefer smaller grammars and use of negative samples to avoid overgeneralization. The algorithm runs for $\text{num} \_\text{gen}$ generations, where $\text{num} \_\text{gen}$ is an input parameter of the algorithm. In Section 4 we provide an extensive test on control parameters, while in the future we might apply a more sophisticated approach using adaptive or self-adaptive parameter control [26].

4. Experimental results

In this section first the results using MAGIc on the DESK language are presented. Then the influence of control parameters ($\rho_m$, $pop \_\text{size}$, $\text{num} \_\text{gen}$) on successfullness of inferred grammars with respect to the success rate (SR) and to the average number of evaluations to solution (AES) [27] is discussed. Next, the comparison with the approach that uses a tabular representation for CFG learning [28] is presented. Finally, a realistic application of language learning from the graphics domain is shown.

4.1. Application of MAGIc to domain-specific language DESK

MAGIc was tested on the DSL DESK [8]. Testing was done with 12 positive samples, which exercise all possible combinations of the original grammar productions. Positive samples used are:

1. print id
2. print num
3. print id + num
4. print id + id + id
5. print id where id = num
6. print num where id = num
7. print num + id where id = num
8. print num + num where id = num
9. print id where id = num; id = num
10. print num num where id = num; id = num
11. print num + num where id = num; id = num
12. print id + id + id where id = num; id = num; id = num

An example of an inferred grammar, constructed from samples 6, 7, 12, 5, 10 and 2 is shown in Fig. 5. The samples are ordered in the same way as they were used by MAGIc to infer this grammar.

Comparing the inferred and original grammar, it can be seen that the number of productions as well as the size of the RHS are similar. Nonterminal $N_2$ in the inferred grammar can be correlated with nonterminal $e$ in the original grammar, because both of them infer the expression part of the language. Furthermore the nonterminal $N_5$ from the inferred grammar can also be correlated with nonterminal $c$ in the original grammar. Both infer the definitions part of the DESK language.

4.2. Control parameter tuning

To be successful, any search algorithm needs to find a good balance between exploration and exploitation, where exploration is a process of visiting entirely new regions of a search space, while exploitation is a process of visiting neighborhood regions. In evolutionary algorithms exploration and exploitation are achieved by selection and recombination (mutation and/or crossover) processes [29]. In contrast, the memetic algorithm additionally emphasizes exploitation by performing local search [3]. However, the line between exploration and exploitation in evolutionary algorithms is blurred since it is difficult to delimit exploration from

\[
\begin{align*}
1. \ N_1 &::= \text{print } N_3 N_5 \\
2. \ N_2 &::= + N_3 \\
3. \ N_2 &::= e \\
4. \ N_3 &::= \text{num } N_2 \\
5. \ N_3 &::= \text{id } N_2 \\
6. \ N_4 &::= ; \text{id } = \text{num } N_4 \\
7. \ N_4 &::= e \\
8. \ N_5 &::= \text{where id } = \text{num } N_4 \\
9. \ N_5 &::= e
\end{align*}
\]

Fig. 5. Inferred DESK grammar with MAGIc.
exploitation in selection and recombination processes [30]. Furthermore, population size and the representation of individuals have an important impact on exploration and exploitation, too. Typically, the balance between exploration and exploitation in evolutionary algorithms is managed by proper control parameters (e.g., \(p_m\), pop_size) settings. In [31] an excellent overview of this problem has been given, where the authors distinguish between parameter tuning and parameter control. What control parameter settings are most likely to produce the best results is the question that every evolutionary algorithm developer/user has to face. Currently, we are only using parameter tuning, where control parameters are defined before running and do not change during the evolutionary process. Several experiments have been performed on the DESK language, where we studied the influence of control parameters on MAGiC’s success (SR = success rate, AES = average number of evaluations to solution), on the number of necessary samples to infer the complete grammar (MNS = minimal number of samples), ANS = average number of samples needed to infer grammar), and on the structure of inferred grammars (ANN = average number of nonterminals, ANP = average number of productions, ARHS = average size of production right hand size, ANG = average number of grammars found). The following control parameters have been taken into account: probability of mutation (\(p_m\)), population size (pop_size), and number of generations (num_gen).

Tables 1 and 2 show the results after 30 runs on the DESK grammar inference using the previously presented 12 positive samples. Note that the inferred grammars didn’t successfully parse all positive samples in some runs. Although this happened only in 1 run out of 30 runs achieving SR = 0.97. On the other hand, small population size (pop_size = 20) and pop_size = 30) did not lead to successful results, while bigger population size (pop_size = 50) required many more evaluations. We did not notice a big influence of \(p_m\) on the obtained results and we chose \(p_m=0.01\) for DESK grammar learning. The influence of pop_size and num_gen on the number of evaluations is shown in column AES in Table 1, where a bigger population and more generations require many more evaluations. Looking in Table 1 at the minimal and average number of samples (MNS and ANS) used when a successful grammar was inferred, we can see that in some cases the correct grammar was inferred from only positive samples (e.g., from samples 5, 10, 8), while the average number across all 30 runs was 7.4 samples. Also, for example, that from sample 12, which contains 18 terminal symbols, we can construct 3.25786278994 \(\times 10^{40}\) possible binary parse trees [32]. For the sample in Fig. 10 with 175 terminal symbols, the number of possible binary parse trees is 2.76847263596 \(\times 10^{135}\). These are just two examples of how big the search space can be even for small sentences.

On the other hand, the structure of inferred grammars (Table 2) revealed that the obtained grammars have similar structure. After 30 runs the average values on grammar structure metrics are: ANN = 7.42, ANP = 14.19, and ARHS = 3.35, while average
ANG = 21.13. When calculating the number of grammars found (ANG) we eliminated exact copies of grammars, but didn’t perform more sophisticated analysis where an exact copy can be obtained and eliminated by renaming some nonterminals. Hence, the actual different grammars inferred are much smaller than ANG and the importance of this measure is smaller.

With control parameter values \( p_m = 0.01 \), \( \text{pop} \_\text{size} = 40 \) and \( \text{num} \_\text{gen} = 30 \), that achieved best results, the convergence of the algorithm to the solution was studied. Fig. 6 shows the convergence of fitness values of individuals through 30 generations. The results are averaged for 30 runs. The curve named best represents the fitness of the best individual in the generation, while worst represents the worst individual fitness in generation. The curve named average is the average fitness value of the whole population in each generation.

From the curve named best in Fig. 6 it can be seen, that the complete grammar, parsing all positive samples, is found in approximately 10 generations, which is less than the number of input samples. Through the 20th generation all the individuals in the population parsed all positive samples, which can be seen from the average and worst curves in Fig. 6.

A simple test was also made with different selection methods. MAGiC’s deterministic selection was replaced with tournament selection, where the size of the tournament was 2. Tables 3 and 4 show averaged results after 30 runs for the two best parameter sets chosen. Based on the results obtained from the tournament selection and the results from the deterministic selection, we can conclude that the change in the selection process did not affect results and efficiency of the algorithm. Further tests with more specialized selection procedures are planned for the future.

Finally, we test how robust MAGiC is for a different order of samples (Table 5). Five different orders of the aforementioned true positive samples were constructed (original, reverse and 3 random orders) and MAGiC was run 30 times on the following control parameters: \( p_m = 0.01 \), \( \text{pop} \_\text{size} = 40 \), \( \text{num} \_\text{gen} = 30 \) and \( p_m = 0.01, \text{pop} \_\text{size} = 40, \text{num} \_\text{gen} = 50 \). These control parameters, as discussed previously, achieved best results. We can see from Tables 1, 2 and 5 that MAGiC is fairly independent of the order of samples.

### 4.3. Comparison of MAGiC to improved version of TBL algorithm

Our approach was compared with the recently improved TBL algorithm described in [28]. The original TBL algorithm [33] is a CFG inference algorithm that stores an exponential number of grammatical structures in a tabular representation, which is similar to the parse table of the Cocke–Younger–Kasami (CYK) parsing algorithm [34]. The problem of inferring the CFG is reduced to the problem of partitioning the nonterminals. For merging distinct nonterminals into one nonterminal algorithm is used. The TBL approach uses both positive and negative samples and infers grammars in Chomsky Normal Form [34].

In the improved TBL algorithm [28] a modified fitness function and a new delete specialized operator were applied. The new delete specialized operator deletes all nonterminals that are placed in some randomly selected block. The modified fitness function does not punish the inferred grammar for generating even one negative sentence as the original fitness function, where the grammar that generates at least one negative sample, has fitness 0. The improved fitness function counts the sum of the normalized number of generated negative examples and the complement of the normalized number of non-accepted positive examples. The aim of the new fitness function is to find a CFG consistent with a whole learning set, i.e., both positive and negative examples.

Compared to the results presented in [28], the improved TBL algorithm outperforms the original in number of successful runs as well as in average number of generations needed to reach 100% fitness. The improved TBL algorithm is also not so much dependent on the block and population size.

The experiments were performed on the same languages used in [33] and [28]. The first is \( L_1 = \{ a^m b^n c^m | m \geq 1 \} \) and the second is \( L_2 = \{ a^m c^n | m \geq 1 \} \). Both languages consist of all typical

---

**Table 3**

<table>
<thead>
<tr>
<th>( p_m )</th>
<th>( \text{pop} _\text{size} )</th>
<th>( \text{num} _\text{gen} )</th>
<th>( T(\text{s}) )</th>
<th>SR</th>
<th>AES</th>
<th>ANP</th>
<th>ANPH</th>
<th>ANPHS</th>
<th>ANPHS</th>
<th>ANPHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40</td>
<td>30</td>
<td>9.71</td>
<td>1.00</td>
<td>7.42</td>
<td>14.67</td>
<td>3.51</td>
<td>19.33</td>
<td>19.33</td>
<td>19.33</td>
</tr>
<tr>
<td>1</td>
<td>40</td>
<td>50</td>
<td>9.53</td>
<td>1.00</td>
<td>7.47</td>
<td>14.80</td>
<td>3.56</td>
<td>21.61</td>
<td>21.61</td>
<td>21.61</td>
</tr>
</tbody>
</table>

**Table 4**

<table>
<thead>
<tr>
<th>( p_m )</th>
<th>( \text{pop} _\text{size} )</th>
<th>( \text{num} _\text{gen} )</th>
<th>( T(\text{s}) )</th>
<th>SR</th>
<th>AES</th>
<th>ANP</th>
<th>ANPH</th>
<th>ANPHS</th>
<th>ANPHS</th>
<th>ANPHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40</td>
<td>30</td>
<td>9.71</td>
<td>1.00</td>
<td>7.42</td>
<td>14.67</td>
<td>3.51</td>
<td>19.33</td>
<td>19.33</td>
<td>19.33</td>
</tr>
<tr>
<td>1</td>
<td>40</td>
<td>50</td>
<td>9.53</td>
<td>1.00</td>
<td>7.47</td>
<td>14.80</td>
<td>3.56</td>
<td>21.61</td>
<td>21.61</td>
<td>21.61</td>
</tr>
</tbody>
</table>

**Table 5**

<table>
<thead>
<tr>
<th>( T(\text{s}) )</th>
<th>SR</th>
<th>AES</th>
<th>ANP</th>
<th>ANPH</th>
<th>ANPHS</th>
<th>ANPHS</th>
<th>ANPHS</th>
<th>ANPHS</th>
<th>ANPHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.2</td>
<td>3.0</td>
<td>5.7</td>
<td>5.4</td>
<td>7.1</td>
<td>6.8</td>
<td>7.5</td>
<td>7.2</td>
<td>7.1</td>
</tr>
<tr>
<td>2</td>
<td>3.3</td>
<td>3.1</td>
<td>5.8</td>
<td>5.5</td>
<td>7.2</td>
<td>6.9</td>
<td>7.6</td>
<td>7.3</td>
<td>7.2</td>
</tr>
<tr>
<td>3</td>
<td>3.4</td>
<td>3.2</td>
<td>5.9</td>
<td>5.6</td>
<td>7.3</td>
<td>7.0</td>
<td>7.7</td>
<td>7.4</td>
<td>7.3</td>
</tr>
<tr>
<td>4</td>
<td>3.5</td>
<td>3.3</td>
<td>6.0</td>
<td>5.7</td>
<td>7.4</td>
<td>7.1</td>
<td>7.8</td>
<td>7.5</td>
<td>7.4</td>
</tr>
<tr>
<td>5</td>
<td>3.6</td>
<td>3.4</td>
<td>6.1</td>
<td>5.8</td>
<td>7.5</td>
<td>7.2</td>
<td>7.9</td>
<td>7.6</td>
<td>7.5</td>
</tr>
</tbody>
</table>
grammatical structures characteristic for the CFG, but none of them are realistic and of practical importance. In [28] the algorithm needed at least 600 generations to infer \( L_1 \), while for inference of \( L_2 \) needed at least 900 generations.

Using the following positive samples for language \( L_1 \):

\[
\begin{align*}
1. & \ a \ b \ c \\
2. & \ a \ b \ c \ c \\
3. & \ a \ b \ c \ c \ c \\
4. & \ a \ a \ b \ b \ c \\
5. & \ a \ a \ b \ b \ c \ c \\
6. & \ a \ a \ a \ b \ b \ b \ c \ c \\
7. & \ a \ a \ a \ a \ b \ b \ b \ b \ c \ c \\
\end{align*}
\]

MAGIC inferred the following CFG:

\[
\begin{align*}
1. & \ N_1::= a \ N_2 \ b \ N_3 \\
2. & \ N_2::= a \ N_4 \\
3. & \ N_2::= c \\
4. & \ N_3::= c \ N_3 \\
5. & \ N_3::= c \\
6. & \ N_4::= N_2 \ b \\
\end{align*}
\]

in the 6th generation from samples 1, 6, 2, 4, 5, 7.

Using positive samples, that were also used in [28] (structured samples were eliminated), for the language \( L_2 \):

\[
\begin{align*}
1. & \ a \ c \\
2. & \ a \ c \ c \\
3. & \ a \ c \ c \ c \\
4. & \ b \ c \\
5. & \ b \ c \ c \\
6. & \ b \ c \ c \ c \\
\end{align*}
\]

MAGIC inferred the following CFG:

\[
\begin{align*}
1. & \ N_1::= N_3 \ N_2 \\
2. & \ N_2::= c \ N_2 \\
3. & \ N_2::= c \\
4. & \ N_3::= a \\
5. & \ N_3::= b \\
\end{align*}
\]

in the 5th generation from samples 3, 1, 4, 2.

Table 6 shows the average number of evaluations needed to reach 100% fitness and average performance time for the improved TBL and MAGIC algorithms. The results for MAGIC were obtained and averaged from 30 algorithm runs. The average number of evaluations for the improved TBL algorithm was calculated based on the data available in [28], where the average number of generations was multiplied by population size. The average performance time for TBL was determined from graphs presented in [28] and the smallest value was used. MAGIC inferred grammars for \( L_1 \) and \( L_2 \) in far less evaluations than the improved TBL algorithm and also in shorter time.

Figs. 7 and 8 show convergence of the fitness values in populations for language \( L_1 \) and \( L_2 \), respectively. The results are similar to convergence for the DESK language described earlier and presented in Fig. 6. In both cases less than 5 generations were needed to infer grammars that parse all positive samples (curve named best). After 15 generations all the grammars in the population parsed all positive samples (curves average and worst).

### 4.4. Application of MAGIC to graphics domain

We have used MAGIC on a real example from the computer graphics domain. In [35], Strnad and Guid developed a method for modeling trees with hypertextures, a method for describing 3D shapes and textures. The method is based on a volumetric representation of trees generated by a three dimensional variation of an Iterated Function System (IFS), which is a technique for fractal generation. Using this method a tree is a fractal object described by nonlinear and nondeterministic IFS where a combination of linear transformations (scaling, translations and rotations) and nonlinear shears are used. Scaling, translations, and rotations are used to describe fractal subparts (size, position, and orientation), while nonlinear shears are used to bend them in two coordinate directions. Nondeterminism of transformations is achieved by randomly chosen values of transformation parameters (e.g., angle of rotation). Random parameters allow the same set of transformations to produce visually different hypertrees with similar basic structure. A hypertree consists of branches which are like smaller trees. This similarity goes several levels deep and depends on the tree family. Branches need not be exact copies of the whole tree, but they may only closely resemble it. Branching structure is generated by condensation transformation. Moreover, the ideal structure of hypertrees can be distorted with noise perturbation. The whole description of a hypertree consists of resolution, number of iterations, fractal depth, number of branch levels, the description of the generator (POINTINIT or LINEINIT), the coloring scheme (DEPTH-COLOR) and transformations. In Fig. 10 an excerpt from a DSL program is given, while the generated hypertree is displayed in Fig. 11.

In [35], Strnad and Guid developed a DSL for describing trees with hypertextures. Due to lack of experience with language engineering the language was implemented from scratch. The parsing algorithm was hard coded and not based on a grammar, which was not even identified. Hence, the possibility of automatically generated parsing code was missed, while the maintenance of parsing code will be extremely difficult in the case that this language will evolve in the future. We have run the MAGIC algorithm on a sample of programs provided by Strnad and Guid [35]. Two programs from the used set of 18 samples are shown in Fig. 9. The control parameters used were: \( p_m = 1 \%, \ pop \_size = 50, \ num \_gen = 50 \). Averaged
results after 30 runs are shown in Table 7. Because of the bigger population size, more generations and especially bigger grammars the running times are higher than in DESK language sample. A lot of time is taken by the generation of the LR(1) parser. The success rate is 77% and AES is around 27K evaluations. One of the finally inferred grammars is presented in Fig. 12.

5. Related work

There have been many different and varying approaches to grammatical inference. Initial learning efforts focused on inferring regular grammars and these have resulted in successful algorithms such as L* [36], regular positive negative inference (RPNI) [15], finite automata learning from simple examples [37] and automata learning with merge constraints [38]. These algorithms use a variety of approaches ranging from membership queries and simple distributions from which samples are elicited to using both positive and negative samples and utilizing artificial intelligence approaches such as genetic algorithms. Learning CFGs on the other hand has proved to be more challenging but a multitude of potential applications in a wide variety of areas such as bioinformatics, robotics, programming languages and software engineering has recently led to concerted efforts and advancements towards learning CFGs.

While CFGs can be identified in the limit, learning them in polynomial time is still believed to be intractable. Consequently researchers have resorted to either using a simplicity bias to converge to an appropriate grammar or exploring a vast search space using evolutionary algorithms and Bayesian methods, or a combination of the two methods. In [33], a CFG is learned from partially structured positive and negative samples. Using a variant of parse tables as in parsing algorithms such as the CYK algorithm [34], the problem is reduced to partitioning the set of nonterminals and a genetic algorithm is incorporated to search the space of possible solutions. This work was recently extended in [28] using a modified fitness function and a new delete specialized operator. The improved version finds the solutions faster than the previous version and is not as vulnerable to block and population size. By comparison, MAGIc uses only (unstructured) positive samples to infer a grammar. In Section 4 MAGIc is contrasted and compared

<table>
<thead>
<tr>
<th>Table 7</th>
<th>Results for hypertree grammar inference (T(s)) – average run time in seconds, SR – success rate, AES – average number of evaluations, ANN – average number of different nonterminals, ANP – average number of productions, ARHS – average size of right-hand side (RHS) in inferred grammars, ANG – average number of grammars found in one run.</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(s)</td>
<td>SR</td>
</tr>
<tr>
<td>2293.57</td>
<td>0.77</td>
</tr>
</tbody>
</table>

RESOLUTION 300 400 300
ITERATIONS 300000
POINTINIT 0 0 0
TREEDEPTH 5
BRANCHDEPTH 1
HYPERVOLUME 6 6 6 6 6
DEPTHCOLOR 0-1 0.3+0.0 0.3+0.0 0.2+0.0
CONDENSATION 1
CONE 1 1 1 1 CONE_Y

RESOLUTION 300 400 300
ITERATIONS 300000
POINTINIT 0 0 0
TREEDEPTH 5
BRANCHDEPTH 1
HYPERVOLUME 6 6 6 6 6
SCALE (0,0,1) (0,1,0) (1,0,0)
SCALE (0,0,1) (0,1,0) (1,0,0)
CONDENSATION 1
CONE 1 1 1 1 CONE_Y

Fig. 9. Two examples of positive samples given as input.

Fig. 10. Excerpt of domain specific program for hypertree generation.
to the algorithm presented in [28]. The comparison is made on the same languages used in [28]. The Synapse system [39] is an incremental inductive CYK system which uses both positive and negative samples, a bridging process to generate rules for incomplete parse trees, and local and global search components to infer both ambiguous and unambiguous grammars. MAGIC on the other hand is not sensitive to order effects in input samples while Synapse expects ordered presentation of samples to efficiently infer CFGs. We have run Synapse on a real example – the Hypertree DSL (Section 4.4). Synapse was not able to infer a grammar from the same positive samples as we used in MAGIC. A genetic programming-based system with local optimization in the form of a CYK table from which information is obtained to construct a new rule is discussed in [40]. Unlike MAGIC, this system uses both positive and negative samples, and is tested only on Tomita’s regular languages [41]. The Tomita languages are the set of the following seven regular languages which have been frequently used to benchmark deterministic finite automata (DFA) learning algorithms:

$$L_1 = (a^*)$$
$$L_2 = (ab)^*$$
$$L_3 = (b|aa)^*(a^*|(abb(bb|a)*))$$
$$L_4 = (a((b|bb)aa^*)^*(b|bb|a*))$$
$$L_5 = ((aa|bb)^*(a|bb)|((a(ab)^*(b|aa))|((ba))^*))$$
$$L_6 = ((b(ba)^*(a|bb))|(a(ab)^*(b|aa)))^*)$$
$$L_7 = b^*a*b*a^*$$

**Fig. 12. Inferred grammar.**

```
N1 -> resolution N2
  iterations num N3 N2
  treedepth num
  branchdepth num
  hypervolume N2 N2
  condensation num cone N2 num coney
N2 -> num num num N4
N3 -> pointinit
N3 -> lineinit num num num
N4 -> depthcolor range bpp bpp bpp N4
N4 -> \varepsilon
N4 -> name programe N4
N4 -> scale lpar num comma num comma num rpar
  lpar num comma num comma num rpar
  lpar num comma num comma num rpar N4
N4 -> rotate lpar num comma num comma num rpar
  lpar num comma num comma num rpar
  lpar num comma num comma num rpar N4
N4 -> translate lpar num comma num comma num rpar
  lpar num comma num comma num rpar
  lpar num comma num comma num rpar N4
N4 -> transform num num N4
N4 -> shear lpar num comma num comma num rpar
  lpar num comma num comma num rpar
  lpar num comma num comma num rpar shearx N4
N4 -> perturb lpar num comma num comma num comma num rpar
  lpar num comma num comma num comma num rpar
  lpar num comma num comma num comma num rpar
  lpar num comma num comma num comma num rpar
  lpar num comma num comma num comma num rpar N4
```
Our previous work, GenInc [42], is an incremental learning algorithm which infers a CFG from a set of ordered characteristic samples (i.e., they exercise every rule in a grammar) drawn from a simple distribution. Like GenInc, MAGIC is an incremental learning algorithm. However, MAGIC is more robust in that it does not rely on ordered samples; this increased inference robustness is due to a memetic programming approach which augments the main evolutionary-based technique with an incremental local search component thus foregoing the need to present samples in a particular order.

For a simplicity bias, many approaches use the concept of Minimum Description Length (MDL) [25] which states that the best hypothesis is one which best compresses the dataset. SubdueGL [43] is an iterative search method which uses MDL to induce context-free graph grammars from positive samples. The focus here is on learning databases which cannot be represented as textual grammars hence the focus on learning graph grammars. In [44] an approach is described to infer programming language dialects from an incomplete initial grammar and program samples. This approach also uses a variant of MDL to assign weights to rules; rules which are more likely to succeed are assigned a higher value and hence more likely to be chosen to complete the initial grammar. By comparison MAGIC’s focus is on inferring a complete grammar from scratch, although it does have the capability to use a partial incomplete grammar as initial input and infer the missing constructs. Kammeyer and Belew [45] were the first to apply genetic algorithms with local search optimization (the inside-outside algorithm) to infer stochastic CFGs (i.e., a CFG where each rule has an associated probability in the range [0,1]). The results reported were preliminary and on simple CFGs. A genetic algorithm with a priori distribution over the space of all possible grammars with a bias towards simpler grammars is described in [46]. This technique assumes the existence of a covering grammar which is a stochastic CFG and whose parameters are optimized based on the information contained in the samples. The main difference between these approaches and MAGIC is that they are focused on inferring stochastic CFGs. eg-GRIDS [47] is a learning system which focuses on learning natural language using a genetic algorithm as a search strategy and MDL as a heuristic to control overgeneralization. eg-GRIDS starts with an overly specific grammar and applies learning operators to converge to an appropriate grammar; hence eg-GRID assumes the existence of the entire learning corpus from the beginning while MAGIC is an incremental learning algorithm and outputs a valid hypothesis after every iteration of an input sample.

6. Conclusion

This paper has presented a memetic algorithm (MAGIC) for unsupervised learning of context-free grammars and its application to DSL development. MAGIC is an incremental population-based algorithm comprised of initialization, local search, mutation, generalization, and selection steps. The algorithm’s inner workings are explained on the DESK language, where the inferred grammar is compared to the original one. Although the inferred grammar has fewer productions, it is equivalent to the original one. On the same language the impact of different control parameters on MAGIC’s success rate is discussed. MAGIC was also proven to be independent from the order of language samples, which was the limitation of our previous work. Comparing MAGIC to the recently improved TBL algorithm showed that it can infer complete grammars for the same languages in far fewer fitness evaluations. However, languages in the TBL comparison are not realistic. To also show the practical use of MAGIC, a case study is presented, which uses the approach of a real DSL for expressing hypertextes in the computer graphics domain.

In the future we would like to extend the experimental part with more DSL examples and enhanced local search with the information from a grammar repository (collection of GPL and DSL grammars), as well as from negative samples.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 0811630. We would like to thank Damjan Strnd and Niko Guid for providing us examples for hyper-tree generation. The authors are also grateful to Olgiierd Unold, who provided samples of the L2 language used in the improved TBL algorithm, and Upendra Sapkapota who assisted with the running of the Synapse tool.

References

[38] B. Lambeau, C. Dumas, P. Dupont, State-merging DFA induction algorithms with mandatory merge constraints, in: Proc. 9th International Colloquium on Grammatical Inference (ICGI’08), 2008, pp. 139–153.