A Toolbox for Context-Sensitive Grammar Induction by Genetic Search

Wojciech Wieczorek WWIECZOREK@ATH.BIELSKO.PL
University of Bielsko-Biala, Willowa 2, 43-309 Bielsko-Biala, Poland

Łukasz Strąk LUKASZ.STRAK@US.EDU.PL

Arkadiusz Nowakowski ARKADIUSZ.NOWAKOWSKI@US.EDU.PL
University of Silesia in Katowice, Bankowa 14, 40-007 Katowice, Poland

Olgierd Unold OLGIERD.UNOLD@PWR.EDU.PL
Wroclaw University of Science and Technology, Wyb. Wyspianskiego 27, 50-370 Wroclaw, Poland

Editors: Jane Chandlee, Rémi Eyraud, Jeffrey Heinz, Adam Jardine, and Menno van Zaanen

Abstract

This paper introduces a new tool for context-sensitive grammar inference. The source code and library are publicly available via GitHub and NuGet repositories. The article describes the implemented algorithm, input parameters, and the produced output grammar. In addition, the paper contains several use-cases. The described library is written in F#, hence it can be used in any .NET Framework language (F#, C#, C++/CLI, Visual Basic, and J#) and run under the control of varied operating systems.

Keywords: Grammatical inference, context-sensitive grammar, Kuroda normal form, steady-state genetic algorithm

1. Introduction

In this paper, we focus on a task for the grammatical inference (GI)—learning context-sensitive from an informant, which means that the induction algorithm learns from positive and negative examples. The inference leverages evolutionary computation, namely steady-state genetic algorithm (ssGA). We have implemented this algorithm and made it available to the community through public repositories[1]. The implementation has the form of .NET Framework library and can be freely utilized in more advanced systems. It is worth noticing that the library can be integrated with any language compatible with Microsoft .NET Framework. Our library consists of two modules: 'GA' for running genetic search and 'Parsing' for handling type-1 grammars.

As a result of the genetic search, the user gets an induced grammar along with its classification score. The balanced accuracy—known from statistical measures of binary classification—was selected as our main classification evaluation because we allow the numbers of examples and counter-examples to be substantially different. Please note that even an imperfect grammar (for instance, with 55% accuracy) could be valuable and useful when we have more such inaccurate classifiers. This approach is called ensemble methods (Hearty, 2016; Rokach, 2010), which is based on the observation that multiple classifiers can improve

1. https://www.nuget.org/packages/ContextSensitiveGrammarInduction.GeneticSearch
https://github.com/w-wieczorek/ConsoleCSI

the overall performance over a single classifier if they are mutually independent. A classifier that will give us the correct result 70% of the time is only mediocre, but if we have got 100 classifiers that are correct 70% of the time, the probability that the majority of them are right jumps to 99.99%. The probability of the consensus being correct can be computed using the binomial distribution: $\sum_{k=51}^{100} \binom{100}{k} \cdot (0.7)^k \cdot (0.3)^{100-k} \approx 0.9999$ (Lam and Suen, 1997).

The paper's content is organized into six sections. In Section 2, we present necessary definitions and facts originating from formal languages. Section 3 gives a brief overview of context-sensitive related works. In Section 4, we propose a new evolutionary-based inference algorithm. Section 5 shows the examples of using our software. Concluding comments and future work are contained in Section 6.

2. Preliminaries

We assume the reader to be familiar with basic formal language theory and automata theory, e.g., from the textbook by Hopcroft et al. (2001), so that we introduce only selected notations used later in the article. The basic knowledge of an elementary genetic algorithm is supposed to be known as well, see for instance (Salhi, 2017, Chapter 4).

2.1. Formal Grammars and Languages

Definition 1 *A **string rewriting system** on an alphabet A is a relation defined on A^*, i.e., a subset of $A^* \times A^*$.*

Remark 2 *Let R be a string rewriting system. Usually, $(\alpha, \beta) \in R$ is written in the form $\alpha \to \beta$ and it means that α can be replaced by β in some string.*

Definition 3 *A **grammar of type-0** (or unrestricted grammar) is written as (N, T, P, S), where*

*(i) N, T are disjoint finite sets so-called **non-terminal** and **terminal** symbols respectively;*

*(ii) $S \in N$ is the **start symbol**;*

*(iii) P is a finite string rewriting system on $N \cup T$, so-called **production rules**.*

Definition 4 *Let G be a grammar of type-0. For $x, y \in A^*$ we write $x \Rightarrow y$ iff $x = x_1 \alpha x_2$, $y = x_1 \beta x_2$, for some $x_1, x_2 \in A^*$ and $\alpha \to \beta \in P$. The reflexive and transitive closure of the relation $\Rightarrow$ is denoted by $\Rightarrow^*$.*

Definition 5 *The language $L(G)$ generated by a grammar G is defined by:*

$$L(G) = \{w \in T^* \colon S \Rightarrow^* w\}. \tag{1}$$

Definition 6 (Chomsky hierarchy) *Beside type-0 grammar, Noam Chomsky (Chomsky, 1956) defined the subsequent three types of grammars. A grammar is of:*

Type-1 if it is of type-0 and if all productions have the form $\alpha \to \beta$ with $1 \leq |\alpha| \leq |\beta|$.

*Type-2 (or **context-free grammar**) if it is of type-1 and if all productions have the form* $A \to \alpha$, *where* $A \in N$ *and* $\alpha \in (N \cup T)^+$.

*Type-3 (or **regular grammar**) if it is of type-2 and if all productions have the form* $A \to aB$ *or* $A \to a$, *where* $A, B \in N$ *and* $a \in T$.

Remark 7 *It can be shown (Chiswell, 2009) that if* G *is type-1, then* $L(G) = L(G')$ *for some **context-sensitive** grammar, that is a grammar in which all productions have the form* $\alpha A \beta \to \alpha \gamma \beta$, *where* $A \in N$, $\alpha, \beta, \gamma \in (N \cup T)^*$ *and* $\gamma \neq \varepsilon$ *(naturally* ε *stands for the empty string).*

Definition 8 *A language* L *is of type-n if* $L = L(G)$ *for some grammar of type-n* $(0 \leq n \leq 3)$.

Definition 9 *A grammar* $G = (N, T, P, S)$ *is in **Kuroda normal form** (Kuroda, 1964) if each rule from* P *take one of the form*

$$\begin{aligned} AB &\to CD \\ A &\to BC \\ A &\to B \\ A &\to a \end{aligned}$$

where $A, B, C, D \in N$ *and* $a \in T$.

Remark 10 *Every grammar in Kuroda normal form generates a context-sensitive language. Conversely, every context-sensitive language can be generated by some grammar in Kuroda normal form (Rozenberg and Salomaa, 1997).*

2.2. Steady-State Genetic Algorithm

The basic idea of the steady-state genetic algorithm is to change the generational approach in classic genetic algorithms. At each iteration, the population is replaced and reproduces new genotypes on an individual basis. Only one or a small part of the population is replaced in each iteration, which causes individuals to receive faster feedback relative to the total number of the solution space evaluation compared to genetic algorithms, where a large part of the population is replaced (Whitley, 1989; Agapie and Wright, 2014). This model is used particularly often, wherever the fitness function requires a high computational complexity (Agapie and Wright, 2014). The description below points out the most important ssGA characteristics comparing to the classical GA algorithm.

Selection policy During the selection, a tournament is applied, in which the best two individuals from a randomly selected subset of the population become parents.

Deletion policy Parents and offspring can co-exist together in the population. After creating a new individual, it replaces another individual in the population.

Algorithm 1: General steady-state genetic algorithm

Input: n_i – the number of iterations
n_p – the size of the population
p_m – probability of mutation
Output: highest fitness value individual in population P

1 generate random initial population P of size n_p
2 **for** $i \leftarrow 1$ **to** n_i **do**
3 select $T \subseteq P$ individuals to the tournament
4 select $p_1, p_2 \in T$ with highest fitness
5 create offspring p_n based on p_1, p_2 crossover
6 execute mutation function on p_n with probability p_m
7 calculate fitness for p_n
8 replace lowest fitness $p_r \in T$ with p_n in P
9 **end**
10 **return** an individual from P with highest fitness

In the nomenclature of evolutionary strategies, this model is denoted by $(\mu + 1)$ GA (Agapie and Wright, 2014). Algorithm 1 represents the generic ssGA algorithm.

In line 1, the population is initialized randomly. The procedure details are strongly related to the problem and its solution space. The same applies to crossover and mutate operations, as well as a fitness function. On that level, those functions are very general, just like in the classic genetic algorithms. Some implementation of a crossover function creates two offspring, but it does not affect the presented logical flow of the Algorithm 1. In line 8, an individual with a lowest fitness value is chosen from the group of individuals forming tournament and replaced by a newly created individual in the population.

3. The Context-Sensitive Realms

Most operations with the use of context-sensitive grammars have exponential complexity (Grune and Jacobs, 2008), which is why it is hard to find any grammar inference algorithm for finding a contextual grammar based on examples and counterexamples. It is also difficult to find scientific packages related to this subject in literature. In this section, we have collected the most related algorithms and packages.

3.1. VEGGIE

Visual Environment for Graph Grammar Induction and Engineering (VEGGIE) is a tool that consists of three modules: SGG, SubdueGL, and GraphML (Ates and Zhang, 2007). The first one is responsible for graph formalism and parsing. The second provides a context-sensitive graph grammar induction system. The last one provides a commonly used data file format. The aim of creating the system is to producing complex graph grammar for visual languages easily and for graph databases exposing hidden structures. The system was used in practical applications, for example, to program behavior verification (Zhao and

Zhang, 2008; Zhao et al., 2010) and formal approach to discovering the semantic structure underlying a Web Page interface (Kong et al., 2011).

3.2. Compression and Pattern Matching

Maruyama et al. (2010) proposed Knuth-Morris-Pratt (KMP)-type compressed pattern matching algorithm (CPM). A framework uses Σ-sensitive grammar (subclass of context-sensitive grammar) to enhance compression and pattern matching to speed up searching text in compressed data. The authors show solution robustness compared to other compression algorithms.

3.3. Vehicle Movement Description

The cameras in autonomic cars sampling many images that the objects detection system needs to process. Based on that data, a decision has to be made. Piecha and Staniek (2010) used movement descriptors languages and predefined context-sensitive grammar that simplify making complex vehicles maneuver. The set of terminal contains: `w`–driving ahead, `l`–turning left, `p`-turning right, `c`–reverse. Grammar definition contains rules to begin a sequence of moves.

3.4. English Text Analysis

Context-sensitive grammars were successfully applied for parsing news articles (Simmons and Yu, 1991). Additionally, the solution includes the acquisition of a context-sensitive grammar from examples. In the algorithm, words are coded from the text to parts of speech as alphabet symbols and limiting the size of the created sample.

4. Genetic Search for Context-Sensitive Grammar Induction

Let us assume that we are given a sample $S = S_+ \cup S_-$ $(S_+, S_- \neq \emptyset,\ S_+ \cap S_- = \emptyset,\ \varepsilon \notin S)$ over an alphabet T. The goal is to induce a grammar $G = (N, T, P, S)$ in Kuroda normal form of an expected size $|P| = k$ that accepts as many examples S_+ as possible and accepts as few counter-examples S_- as possible.

We follow the before-given procedure of steady-state genetic algorithm. In order to put Algorithm 1 to work, we have to define the following elements and routines of GA: the structure of an individual, the initialization, genetic operators, and the fitness function.

An individual is an array composed of non-negative integers, and it represents a grammar in Kuroda normal form. Each integer in an array is less then a constant value (determined by $n = |N|$ and $t = |T|$) and represents one production rule. For example, if we have two non-terminal (S, A) and two terminal (a, b) symbols, then each possible production is associated with a number: $S \to a$ gets 0, $S \to b$ gets 1, $A \to a$ gets 2, $A \to b$ gets 3, $S \to S$ gets 4, and so on up to 31, which goes to the rule $AA \to AA$.

An initial population is built completely randomly: each of its element is an k-length array and each array entry is a random integer number taken uniformly from the range $[0,\ n(t + n + n^2) + n^4]$. The expected size of a grammar, the terminal symbols, and the number of non-terminal symbols are given as the parameters of the algorithm. The start symbol is determined by the actual values stored in an array. Every non-terminal has its

own index starting from 0. An array (an individual) is decoded to a grammar, and then the smallest symbol from the left hand side of all rules of the form $A \to \ldots$ becomes the start symbol ($A \in N$).

The Crossover has been designed in the following way. Let $a = [a_1, a_2, \ldots, a_j]$ be an array ($j \geq 1$). By head(a) we will denote a_1, and by tail(A) the remain part, i.e., $[a_2, \ldots, a_j]$. Now, assume we have two arrays: x and y. Their offspring should inherit some values of x and some of y. To that end, let us consider four cases in the below-given recursive procedure.

1. Both arrays, x and y, are non-empty. Then compare head(x) with head(y). If they are equal, append head(x) at the beginning of the result of the Crossover operation performed on tail(x) and tail(y). If head(x) $<$ head(y), then with 50% change append head(x) at the beginning of the result of the Crossover operation performed on tail(x) and y. If head(x) $>$ head(y), then with 50% chance append head(y) at the beginning of the result of the Crossover operation performed on x and tail(y).

2. Array x is non-empty and y is empty. Then every element of x has 50% chance to be copied.

3. Array x is empty and y is non-empty. Then every element of y has 50% chance to be copied.

4. Both arrays are empty. Then return the empty array.

During the mutation a randomly taken entry is changed to a new randomly generated value that is not already present in the array. Finally, the fitness function measures a grammars's balanced accuracy based on a sample S with Equation (2):

$$f(G) = \frac{1}{2} \cdot \left(\frac{|\{w \in S_+ : w \in L(G)\}|}{|S_+|} + \frac{|\{w \in S_- : w \notin L(G)\}|}{|S_-|} \right). \tag{2}$$

5. Exemplary Use-Cases

In this section, we will show how to use the published library. We will give three examples: the first in C++/CLI, the second in F#, and the last one in the C# language.

In below-given listing we can see how to manage grammar inference from the sets of examples and counter-examples.

```
#include "pch.h"

using namespace System;
using namespace System::Collections::Generic;

int main(array<System::String ^> ^args) {
    GA::GAParams^ parameters = gcnew GA::GAParams(
        500,            // population size
        4,              // tournament size
        0.1,            // mutation probability
        4000,           // iterations
        400,            // verbose=how often show intermediate results
```

```
        gcnew Random(), // random numbers generator
        6,              // initial number of rules
        3,              // number of non-terminals
        gcnew array<Char>(2),       // alphabet
        gcnew SortedSet<String^>(), // examples
        gcnew SortedSet<String^>()  // counter-examples
    );
    parameters->alphabet[0] = L'a';
    parameters->alphabet[1] = L'b';
    parameters->examples->Add(L"a");
    parameters->examples->Add(L"ab");
    parameters->examples->Add(L"aab");
    parameters->examples->Add(L"aaab");
    parameters->counterexamples->Add(L"b");
    parameters->counterexamples->Add(L"abb");
    parameters->counterexamples->Add(L"aba");
    parameters->counterexamples->Add(L"aabb");
    Tuple<Parsing::Type1Grammar^, double>^ result = GA::runGA(parameters);
    Console::WriteLine(L"grammar:\n" + result->Item1);
    Console::WriteLine(L"with balanced accuracy = " + result->Item2);
    int i = 0;
    for each (Parsing::Rule^ prod in result->Item1->Rules) {
        Console::WriteLine(++i + String::Concat(": ", prod));
    }
    Console::WriteLine(L"The start symbol: " + result->Item1->Start->name);
    return 0;
}
```

Please notice that the function that launches genetic search is `runGA` and requires the set of GA parameters along with the input data. As a result, the user obtains a type-1 grammar and a floating-point number denoting its balanced accuracy. In the last few lines in the listing we show how to manage the resultant data structure. The following is an exemplary console output.

```
400: best bar = 3/4
800: best bar = 7/8
1200: best bar = 7/8
1600: best bar = 7/8
2000: best bar = 7/8
2400: best bar = 7/8
2419: best bar = 1
grammar:
V0 -> a
V0 -> V2 V0
V0 -> V2 V1
V0 V1 -> V2 V2
V1 -> b
V2 -> a

with balanced accuracy = 1
```

```
1: V0 -> a
2: V0 -> V2 V0
3: V0 -> V2 V1
4: V0 V1 -> V2 V2
5: V1 -> b
6: V2 -> a
The start symbol: V0
```

The second example is written in F# and similar to the previous example, the program induces a grammar from a sample. Please note that the number and order of the parameters should be the same as in the first case. Moreover, this code also utilizes the same library as previously because we load the same NuGet package regardless of the implementation language.

```
open System
open GA
open System.Collections.Generic

let parameters = {
  pop_size = 200;
  tournament_size = 4;
  p_mutation = 0.01;
  iterations = 4000;
  verbose = 1000;
  rnd = Random();
  grammar_size = 8;
  variables = 5;
  alphabet = [|'a'; 'b'; 'c'|];
  examples = SortedSet ["ab"; "bc"; "abbc"; "aabb"; "bbcc"; "aaabbb";
      "bbbccc"; "abbbcc"; "aabbbc"]
  counterexamples = SortedSet ["a"; "b"; "c"; "aab"; "abb"; "bcc"; "bbc";
      "aabbc"; "abbcc"; "abbbc"]
}

let grammar, bar = runGA parameters
printfn $"{grammar}"
printfn $"with bar = {bar}"
```

The following is an exemplary console output.

```
1000: best bar = 2/3
2000: best bar = 17/18
3000: best bar = 17/18
4000: best bar = 17/18
4000: best bar = 17/18
V0 -> V2 V1
V1 -> c
V1 -> V2 V2
V1 V3 -> V1 V1
V2 -> b
```

```
V2 -> V1 V4
V2 -> V4 V4
V3 -> V2 V2
V3 V3 -> V2 V2
V3 V3 -> V2 V3
V4 -> a
V4 -> V1

with bar = 0,9444444444444444
```

The third example, written in C#, illustrates how to define a grammar and perform parsing.

```
using System;
using System.Collections.Generic;
using Rule = Parsing.Rule;
using Symbol = Parsing.Symbol;
using Terminal = Parsing.Terminal;
using Nonterminal = Parsing.Nonterminal;
using Microsoft.FSharp.Collections;

namespace ConsoleApp {

    class Program {
        static void Main(string[] args) {
            var S = new Nonterminal(0);
            var A = new Nonterminal(1);

            var a = new Terminal('a');
            var b = new Terminal('b');

            var rule1 = new Rule(
                    ListModule.OfSeq(new Symbol[] { S }),
                    ListModule.OfSeq(new Symbol[] { a, b })
                );

            var rule2 = new Rule(
                    ListModule.OfSeq(new Symbol[] { S }),
                    ListModule.OfSeq(new Symbol[] { a, A })
                );

            var rule3 = new Rule(
                    ListModule.OfSeq(new Symbol[] { a, A }),
                    ListModule.OfSeq(new Symbol[] { a, S, b })
                );

            var grammar = new Parsing.Type1Grammar(
                    new SortedSet<Rule>() { rule1, rule2, rule3 }, S
                );
            Console.WriteLine($"accept aaabbb: {grammar.accepts("aaabbb")}");
            Console.WriteLine($"accept aabbb: {grammar.accepts("aabbb")}");
```

```
        }
    }
}
```

More examples of using particular modules that are parts of the package can be found in unit tests included in source code (GitHub repository).

6. Conclusions and Future Work

The presented open-source NuGet library for context-sensitive inference has been described. It is easy to use in all .NET Framework languages, which are gaining more and more popularity today. Although context-sensitive parsing is a computationally expensive procedure, our software can run in polynomial time as far as the lengths of input words are quite short. Otherwise, it is not possible to avoid exponential running time. The performance of the algorithm strongly relies on parameter tuning. The initial grammar size and the number of non-terminal symbols cannot be too small or too large. These numbers should be established by preliminary experiments. Furthermore, after increasing the number of iterations and the population size, the final result might change significantly. It is also worth emphasizing that the multi-start approach is perfectly suitable for our algorithm. Having some computational budget, it is often more reasonable to run a program multiple times with fewer iterations than a few times with more iterations.

The important feature of our GA routine is its non-deterministic nature, leading to the opportunity to achieve many different grammars that can be used in ensemble methods. One of the promising research directions is following a memetic algorithm, in which a local search technique to reduce the likelihood of premature convergence is used.

References

Alexandru Agapie and Alden H. Wright. Theoretical analysis of steady state genetic algorithms. *Applications of Mathematics*, 59(5):509, 2014. ISSN 0862-7940.

Keven Ates and Kang Zhang. Constructing veggie: Machine learning for context-sensitive graph grammars. In *19th IEEE International Conference on Tools with Artificial Intelligence(ICTAI 2007)*, volume 2, pages 456–463, 2007. doi: 10.1109/ICTAI.2007.59.

Ian M Chiswell. *A course in formal languages, automata and groups*. Springer Science & Business Media, 2009. ISBN 978-1-84800-939-4.

Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956. doi: 10.1109/TIT.1956.1056813.

Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques: A Practical Guide*. Springer-Verlag, 2nd edition, 2008.

John Hearty. *Advanced Machine Learning with Python*. Packt Publishing Ltd, 2016.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.

Jun Kong, Omer Barkol, Ruth Bergman, Ayelet Pnueli, Sagi Schein, Kang Zhang, and Chunying Zhao. Web interface interpretation using graph grammars. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(4):590–602, 2011.

S.-Y. Kuroda. Classes of languages and linear-bounded automata. *Information and Control*, 7(2):207–223, 1964. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(64)90120-2. URL https://www.sciencedirect.com/science/article/pii/S0019995864901202.

Louisa Lam and SY Suen. Application of majority voting to pattern recognition: an analysis of its behavior and performance. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 27(5):553–568, 1997.

Shirou Maruyama, Youhei Tanaka, Hiroshi Sakamoto, and Masayuki Takeda. Context-sensitive grammar transform: Compression and pattern matching. *IEICE transactions on information and systems*, 93(2):219–226, 2010.

Jan Piecha and Marcin Staniek. The context-sensitive grammar for vehicle movement description. In Leonard Bolc, Ryszard Tadeusiewicz, Leszek J. Chmielewski, and Konrad Wojciechowski, editors, *Computer Vision and Graphics*, pages 193–202, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15907-7.

Lior Rokach. Ensemble-based classifiers. *Artificial intelligence review*, 33(1):1–39, 2010.

Grzegorz Rozenberg and Arto Salomaa. *Handbook of Formal Languages*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1997. ISBN 3540614869.

Sad Salhi. *Heuristic Search: The Emerging Science of Problem Solving*. Palgrave Macmillan, 1st edition, 2017. ISBN 331949354X.

Robert F Simmons and Yeong-Ho Yu. The acquisition and application of context sensitive grammar for english. In *29th Annual Meeting of the Association for Computational Linguistics*, pages 122–129, 1991.

Darrell Whitley. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms*, page 116–121, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc. ISBN 1558600063.

Chunying Zhao and Kang Zhang. A grammar-based reverse engineering framework for behavior verification. In *2008 11th IEEE High Assurance Systems Engineering Symposium*, pages 449–452. IEEE, 2008.

Chunying Zhao, Jun Kong, and Kang Zhang. Program behavior discovery and verification: A graph grammar approach. *IEEE Transactions on software Engineering*, 36(3):431–448, 2010.