Fast parser for biological sequences and a new algorithm for the inference of substitutable languages

Mikaïl Demirdelen

Master research Internship

Internship report

Fast parser for biological sequences and a new algorithm for the inference of substitutable languages

Domain: Machine Learning - Formal Languages and Automaton Theory

Author:
Mikaïl Demirdelen

Supervisor:
François Coste
Jacques Nicolas
Dyliss team

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Abstract: Grammatical inference, or grammar induction, studies how to learn automatically implicit rules behind some sequential data. This domain has a real scientific purpose and can be useful in numerous domains like natural language processing or bioinformatics as they often manipulate sequences.
The tool we use to describe these data is the formal grammar. There exists some categories of grammars that are more expressive than others and therefore, more complicated to learn. In order to infer these expressive grammars some options have been developped. One of them is to make substitutable languages assumptions. The goal of my internship is to search for methods to improve the results of expressive grammar inference using these subsitutable languages. These improvements will be especially made for practical applications, and more particularly for biological sequences.
In this report, I will first describe the state-of-art algorithm that can learn an expressive class of substitutable language and how it is currently implemented. Then, I will develop how I improved the current parser to make it useful for real cases. Finally, I will talk about my contributions to improve the learning capability of the state-of-the-art algorihtm adapting it to a new class of subsitutable languages.

Contents

Acknowledgements

I would like to deeply thank François COSTE, my tutor, for his patience and his pedagogy throughout my internship. He let me a freedom of work that I truly appreciated. I learned a lot thanks to him. He teached me how to be more rigorous in my work, to manage my time more efficiently and to trust more my instincts.

I am also very grateful to Jacques NICOLAS who really helped me a lot when I had a hard time on Prolog. His expertise prevented me to lose a lot of time to debug my program and he also teached me numerous useful logic programmation techniques.

I would also like to thank Marie LE ROÏC and the whole Symbiose teams for their warm welcome in this work environment and their advices throughout my time here.

I am also very thankful to have been able to work with all my colleagues in the intern room. There was a great ambiance throughout my time here, our conversations were very interesting and we had a lot of fun. They also helped me when I had questions or problems and I am also truly grateful for that.

More particularly, I would like to thank Lucas BOURNEUF and Sébastien FRANÇOIS that gave me a great help when I needed the most. They gave me ideas when I had none, and they gave me support when my motivation was in the lowest. Thank you, guys.

Finally, I would like to thank Maëliss MARCHAND for all her help for the re-reading of my reports, her assistance to prepare my defense and all her advice.

Introduction

Grammatical inference [8] is a field of machine learning that aims at learning the implicit rules modeling a given set of sequences. We can represent these rules with formal grammars and languages. A first application is natural language processing but grammatical inference has also been applied for data compression, or music recognition.

Another domain of grammatical induction is bioinformatics. That is the research field of the team Dyliss. They had significant results on automata learning with Protomata Learner. They also had promising preliminary results on context-free grammar learning with the algorithm ReGLiS. My internship consists in improving ReGLiS for practical cases.

I will first introduce ReGLiS and how it is used in practical cases in section 1. Then, in section 2, I will develop a new parsing algorithm I created to deal with biological sequences. I will show that we can have a huge gain on time with it. Finally, in section 3, I will show the limitations of ReGLiS for the k, l-local contextually substitutable languages. I will propose a new algorithm that will be able to solve this problem. I will prove this algorithm to solve all possible conflicts and to create the more general paths possible.

1 Learning substitutable languages

1.1 Languages and Grammars

I will start by introducing the formal grammars definitions and annotations. Let Σ be the alphabet containing a non-empty finite set of atomic symbols, and let Σ^* be the set of all possible strings composed of the concatenation of the symbols in Σ. A formal language L is a subset of Σ^*. We note λ the empty string. We note $|x|$ the length of the string x and Σ^k the set of strings of length k.

A formal grammar is defined by a set of four attributes: $G = \langle \Sigma, N, S, P \rangle$. In this set, Σ is an alphabet of terminal symbols, N is an alphabet of non-terminals, S, included in N, is the non-terminal used as a start symbol and P is a finite set of production rules. In this context, a terminal symbol is a symbol which composes the strings of the language. Non-terminal symbols, also simply called non-terminals, are the symbols used in the different rules but which do not appear in the language. The production rules are rewriting rules, composed by a left-hand side and a right-hand side, from which the different strings of the language can be derived. We note the derivation operator $\Rightarrow$. We also note its reflexive transitive closure $\Rightarrow^*$. We say that a non-terminal symbol is lexical if it only produces only terminal symbols. Two grammars are said equivalent if they produce the same language.

The formal grammars can be classified in four categories, as described by Noam Chomsky in 1959 [1]:

- Unrestricted Grammars (type 0): These grammars are all the possible grammars, without any restrictions on their forms. They can produce languages that can be represented by a Turing machine.
- Context-Sensitive Grammars (type 1): These grammars are the ones whose production rules

have the form of $\alpha A\delta \rightarrow \alpha\beta\delta$, where A is a non-terminal symbol and α, β and δ are strings of terminal or non-terminal symbols. This kind of grammar can produce languages that can be represented by a linear bounded automaton.

- Context-Free Grammars (type 2): These grammars are the one whose production rules have the form of $A \rightarrow \alpha$, where A is also a non-terminal symbol and α is also a string of terminal or non-terminal symbols. A context-free grammar can produce languages that can be represented by a pushdown automaton.

- Regular Grammars (type 3): These grammars are the one with exactly one non-terminal symbol in the left-hand side of the rules and one terminal symbol in the right-hand side with occasionally a nonterminal symbol with it. These grammars can produce languages that can be represented by a finite state automaton and produce the same strings as regular expressions.

1.2 Grammatical Inference and Evaluation

Grammatical inference is a field of machine learning, that studies the inference of grammars, from sequences. In order to do this inference, we will need to use an algorithm to concretely build a grammar. Let suppose a set of strings, S, defined over the alphabet Σ. We suppose that this set of strings is a subset of the target language L_{target}. Giving this set of examples to the learning algorithm, we want that the language, $L(G_{output})$, recognized by the output grammar G_{output}, is equal to L_{target}. The success of these algorithms can be asserted either in theory or in practice.

The principal method of asserting theoretically an algorithm is called the identification in the limit, proposed by Gold [11]. This approach makes the assumption of infinite time and data. The algorithm will take an example and make a guess. It will iterate this process until it finds a correct and stable answer. The answer is considered stable when the algorithm does not change his guess after the consideration of an additional example. The problem of this approach is we make an infinite number of assumptions. Indeed, although this learning paradigm works in theory, it is unusable in practice, as we do not have infinite time and data.

To have a theoretical framework closer to reality, a learnability criterion of polynomial time and data identification in the limit was proposed by Colin de la Higuera [7]. An algorithm respects this criterion iff it can give a result in a polynomial time and it has a characteristic set for each target, noted CS, whose size is polynomial with respect to of the size of the target. The characteristic set is a special set with the following property: if the CS is included in the learning set, the learning algorithm will always give the correct result. The polynomial time and data criteria of identification are more relevant for practical cases so I will mainly focus on this paradigm.

In practice, we evaluate our algorithms with real data sets called test sets. We need to have two test sets : one for positive examples, one for negative examples. Indeed, it is important that a grammar is both able to maximize its acceptation of positive examples and minimize its acceptation of negative examples. To put a number behind these notions we will use two statistical measures : the recall and the precision.

1.3 Learning Substitutable Languages

The main problem of regular languages is their lack of expressivity. Indeed, they are only capable to model all the strings that can be produced by a regular expression. The class of context-free grammar, the next one in the Chomsky hierarchy, is more expressive. Indeed, their construction allows more complex dependencies between within the strings of the corresponding language. For instance, the Dyck language, the language of well-formed parenthesis, is a context-free language. However, context-free grammars are proven not to be identifiable in the limit in polynomial time and data in the general case. One solution is to define another class of languages that has good properties of identification and is still expressive. The main property of distributional languages was enunciated by Harris in [12] this way: words that are used and occur in the same contexts tend to have similar meanings. Inspired by this property, Clark introduced the class of substitutable languages [2].

Before entering the properties of this class of languages, I need to define some properties. Let us define L as a language: then, its set of substring is $Sub(L) = \{y \in \Sigma^* : x, z \in \Sigma^*, xyx \in L\}$ and its set of context is $Con(L) = \{\langle x, z\rangle \in \Sigma^* \times \Sigma^* : y \in \Sigma^*, xyx \in L\}$. We note the distribution $D_L(y)$ of a string $y \in \Sigma^*$ as the set of contexts it has within the language L: $D_L(y) = \{\langle x, z\rangle \in \Sigma^* \times \Sigma^* : xyz \in L\}$. Two strings y_1 and y_2 in Σ^* are syntactically congruent for a language L, noted $y_1 \equiv_L y_2$, iff $D_L(y_1) = D_L(y_2)$. If $y_1 \equiv_L y_2$, then we have, naturally, $\forall x, z \in \Sigma^*, xy_1z \equiv_L xy_2z$. We note the congruence class of y: $[y]_L$. We have $[y]_L = \{y' \in \Sigma^* : y \equiv_L y'\}$. We note the set of syntactic congruence classes, or simply congruence classes, for a language $\mathcal{L}$ as $\mathcal{C}_\mathcal{L}$.

Weak substitutability in L is defined for two non-empty strings y_1 and y_2 this way: $\exists x, z \in \Sigma^* : xy_1z \in L \wedge xy_2z \in L$. It is noted $y_1 \doteq_L y_2$,. Substitutable languages are the class of languages in which weak substitutability implies syntactic congruence, or strong substitutability: $y_1 \doteq_L y_2$ implies $y_1 \equiv_L y_2$. In other words, it means that if y_1 and y_2 have at least one context in common then they are part of the same congruence class. A substitutable language is formally defined in definition 1, as depicted in [6].

Definition 1. *A language L is substitutable iff:*
$\forall x_1, y_1, z_1, x_2, y_2, z_2 \in \Sigma^*, y_1, y_2 \neq \lambda$:

$$x_1y_1z_1 \in L \wedge x_1y_2z_1 \in L \Rightarrow (x_2y_1z_2 \in L \Leftrightarrow x_2y_2z_2 \in L).$$

To give a meaningful example, let suppose that we know that these three sentences are part of a language L: "The cat drinks.", "The dog drinks" and "The cat is hungry". With the two first sentences, we see that the words "cat" and "dog" are part of a congruence class, in the context $\langle$the, drinks$\rangle$. As the third sentence is also part of the language L, and according to the definition 1, we can deduce that the sentence "The dog is hungry" is also part of this language. Roughly, that means that the substitutability criterion can be similar to the notion of synonyms in natural languages, on a very sharp scale.

Yoshinaka defined another type of substitutable language, introducing size in the context. This class, the k,l-substitutable languages are formally established in definition 2.

Definition 2. *A language L is k,l-substitutable iff:*
$\forall x_1, y_1, z_1, x_2, y_2, z_2 \in \Sigma^*, u \in \Sigma^k, v \in \Sigma^l, uy_1v, uy_2v \neq \lambda$:

$$x_1uy_1vz_1 \in L \wedge x_1uy_2vz_1 \in L \Rightarrow (x_2uy_1vz_2 \in L \Leftrightarrow x_2uy_2vz_2 \in L).$$

For application to bioinformatics, where protein sequences can need a more local context substitutability criterion, Coste defined, in [4], a new class of substitutable languages. It is formally defined in definition 3.

Definition 3. *A language L is k,l-local substitutable iff*
$\forall x_1, y_1, z_1, x_2, y_2, z_2, x_3, z_3 \in \Sigma^*, u \in \Sigma^k, v \in \Sigma^l, uy_1v, uy_2v \neq \lambda$:
$$x_1uy_1vz_1 \in L \wedge x_3uy_2vz_3 \in L \Rightarrow (x_2y_1z_2 \in L \Leftrightarrow x_2y_2z_2 \in L).$$

Finally, by crossing the last two type of languages we have another class of languages, also defined in [4] and formally written in definition 4.

Definition 4. *A language L is k,l-local contextually substitutable iff*
$\forall x_1, y_1, z_1, x_2, y_2, z_2, x_3, z_3 \in \Sigma^*, u \in \Sigma^k, v \in \Sigma^l, uy_1v, uy_2v \neq \lambda$:
$$x_1uy_1vz_1 \in L \wedge x_3uy_2vz_3 \in L \Rightarrow (x_2uy_1uz_2 \in L \Leftrightarrow x_2uy_2uz_2 \in L).$$

From the first substitutable language (1), there have been mainly two big adding to the definition. The first is the locality in the substitutability definition. The second one is the strong contextualization of the substitutability criterion. Indeed, we can see that the class of languages in definitions 2 and 4 conserve the contexts in the use of the substitutability. If we have the strings "I need to pack", "I need to eat" and "The pack of wolves is terrifying", we do not want to substitute pack to eat in the third sequence. These languages have the specificities that the contexts should be conserved to keep a certain meaning before the substitutability. For this example, we only want to be able to produce a verb after the word "to". Therefore, only the words substitutable with the verb "pack" (and not the noun) can be replaced in this context. It gives, for this example, a notion of part-of-speech additionnaly to the one of synonyms but it also makes sense for the general cases.

In spite of these differences, the definitions of these classes are still very similar. According to the data available in the domain in which in algorithm is used, one definition can be more relevant to use than another. We can also note that the difference between those languages is the definition of weak and strong substitutability as I defined earlier. For instance, for the k,l-substitutable languages, the weak substitutability is defined this way: $uy_1v \doteq_L uy_2v$ implies $uy_1v \equiv_L uy_2v$, with $u = \Sigma^k$ and $v = \Sigma^l$. This works in analog ways for the languages defined in 3.

In the rest of this review, I will refer to the substitutable languages as defined in definition 1 by a zero-substitutable language and I will keep the term "substitutable languages" to talk about all types of the substitutable languages. Indeed, we can see that the substitutable languages defined in 1 are equivalent to a 0,0-substitutable language as defined in 2.

In the rest of this section, I will review different algorithms that learn substitutable languages represented by context-free grammars. As a reminder from the section 1.1, a context-free grammar is a formal grammar whose production rules have the form $A \to \alpha$, where $A \in N$ and $\alpha \in (N \cup \Sigma^+)$. A context-free grammar is in Chomsky Normal Form, or CNF, if all of its production rules are in the form $A \to BC$, $A \to a$ or $S \to \lambda$, where A, B and C are non-terminal symbols, S is the start symbol, a is a terminal symbol and λ the empty string.

Despite substitutable languages can be modeled by context-free grammars, their learning is feasible and have properties of identification in the limit with polynomial time and data with positive

example only. It can be very useful in practical tasks, as bioinformatics, because we may need to learn expressive grammars with only biological sequences as training set. In the section 1.4, I will first detail a basic algorithm, SGL, that learn context-free substitutable grammars. Then, in section 1.6, I will detail the state-of-the-art algorithm, ReGLiS, and explain how it improves the SGL algorithm.

1.4 SGL algorithm

A basic algorithm to do this task is with the Substitution Graph Learner algorithm, or SGL, defined in algorithm 5 as written in [6]. This algorithm was first enunciated by Clark in [2]. This algorithm first creates a graph where each connected component is a congruence class and then builds the associated context-free grammar in Chomsky Normal Form. In this grammar, for each of the substitutability class, there is a corresponding non-terminal symbol. The branching rules, defined in line 14, allow the generalization over the different substitutability classes. Finally, the terminal rules, defined in line 18, give the terminal symbols.

To be noted that the algorithm shown in 5 is defined for k,l-substitutable languages. To make it learn other classes of substitutable languages, it is only necessary to change the second line, where the substitutability graph is created. Also to be noted, in this line, we use the symbols $\Sigma^{\triangleleft k}$ and $\Sigma^{l\triangleright}$. They were created to avoid the edge effect of the context, with the beginning and the end of the words, respectively $\triangleleft$ and $\triangleright$. They are formally defined this way $\Sigma^{\triangleleft k} = (\Sigma \cup \triangleleft)^k \cap (\triangleleft^*.\Sigma^*)$ and $\Sigma^{l\triangleright} = (\Sigma \cup \triangleright)^l \cap (\Sigma^*.\triangleright^*)$

In spite of the fact that context-free grammar cannot be learnt with polynomial time and data, generally speaking, it has been proved that this algorithm can learn a target substitutable language in polynomial time. It just needs to have a characteristic set of polynomial cardinality with respect to the size of the grammar[14]. Unfortunately, there are no properties of size of the characteristic set.

However, this set can be bounded polynomially in size with respect to the number of rules of the grammar and an additional parameter, the grammar's thickness. This parameter is the maximum length of the smallest word generated from each rule. It allows taking into account not only the size of the grammar but also the actual size of generated words and can counteract some compression effect that might happen. Additionally, the SGL algorithm has another drawback: the grammar it generates tends to have too many ambiguities and redundancy. With real data experiments, one of the advantages of generating a grammar is its readability, and there, the result interpretation can be very fastidious.

Input: Set of strings K on alphabet Σ, int k, int l
Output: Grammar $G = \langle \Sigma, N, S, P \rangle$ in CNF

```
/* Build substitutability graph on substrings of K */
```
$V \leftarrow \{y \in \Sigma^+ : y \in \text{Sub}(K)\}$
$E \leftarrow \{\{y_1, y_2\} \in V \times V : uy_1v \in \text{Sub}(K), uy_2v \in \text{Sub}(K), y_1 \neq y_2, u \in \Sigma^{\lhd k}, v \in \Sigma^{l \rhd}\}$
```
/* Get congruence classes */
```
$\mathcal{C}_K \leftarrow Connected_components\langle V, E \rangle$
$\forall y \in \text{Sub}(K), C(y) \leftarrow C \in \mathcal{C}_K : y \in C$
```
/* Build associated grammar */
```
$N \leftarrow \emptyset, P \leftarrow \emptyset$
for $C \in \mathcal{C}_K$ **do**
  ```
  /* Creation of a non-terminal for each congruence class */
  ```
  $N \leftarrow N \cup \{[\![C]\!]\}$
  ```
  /* Start symbol */
  ```
  if $C \cap K \neq \emptyset$ **then**
    $S \leftarrow [\![C]\!]$
  ```
  /* Creation of production rules for each substring in the class */
  ```
  for $y \in C$ **do**
    if $|y| > 1$ **then**
      ```
      /* Creation of a 'CNF' rule for each split of the substring */
      ```
      for $y_1 \in \Sigma^+, y_2 \in \Sigma^+ : y_1y_2 = y$ **do**
        $P \leftarrow P \cup \{[\![C]\!] \rightarrow [\![C(y_1)]\!][\![C(y_2)]\!]\}$
    else
      ```
      /* Terminal rule */
      ```
      $P \leftarrow P \cup \{[\![C]\!] \rightarrow y\}$
return $\langle \Sigma, N, S, P \rangle$

Algorithm 1: *SGL* (Substitution Graph Learner for k, l-local substitutable languages)

1.5 Grammar reduction

One solution that has been proposed by Coste in [5] is to directly learn reduced canonical grammar. A reduced canonical grammar is a grammar in which the number of non-terminal is minimized and where the right-hand sides of the rules are reduced to avoid unnecessary derivations. In the following parts, I will call reduced grammars the ones in the reduced canonical forms.

The first step, the non-terminal minimization, works in multiple steps. The first one is the remove different non-terminal symbols that produce exactly the same thing. The second step is to remove the symbols that have only one derivation. In other words, if there exists a unique rule $A \rightarrow \alpha$, with $A \in N$ and $\alpha \in (N \cup \Sigma^+)$, and a more general rule that says $G \rightarrow \delta A \gamma$, where δ and γ can be any kind of sequences, we must replace the general rule by $G \rightarrow \delta a \gamma$. Indeed, in this example, the non-terminal symbol A is useless.

The last step is linked to the fact that we learn substitutable languages. In SGL, when we build the substitution graph, we take all the possible classes and generate our rules with them. The matter is that some congruence classes are a composition of other ones. For instance, in the example I gave in section 1.3, I said that there was a congruence class [cat] = {cat, dog} but there is also the singleton [the] = {the} (as each word is, at least, its own congruence class) and the class [the cat] = {the cat, the dog} (with the same contexts $\langle \lambda$, drinks$\rangle$). In this example, we can clearly see that

the third class can be decomposed this way: [the cat]=[the][cat]. Also, we can notice that both of the two other classes are atomic classes. We called a decomposable class a composite congruence class and we define them this way, as depicted in [6], in definition 5.

Definition 5 (Composite congruence class)**.** *Let define a language L whose set of non-zero and non-unit congruence classes is* C^+*. A class* $[y] \in C^+$ *is* composite *for L iff:*

$$\exists [x_1], \dots [x_m] \in C^+, m \geq 2, [y] = [x_1] \dots [x_m]$$

A congruence class which is not a composite class is called a *prime* congruence class and is formally defined, as depicted in [6], in definition 6.

Definition 6. *Let define a language L whose set of non-zero and non-unit congruence classes is* C^+*. A class* $[y]$ *in* C^+ *is* prime *for L iff* $\forall y_1, y_2 \in \Sigma^+ : y_1 y_2 \in [y], [y] \not\subset [y_1][y_2]$.

In these definitions, we call the unit congruence class the one of the empty string λ: $[\lambda]$. We call a zero congruence class, the one defined this way $\{y : D_L(y) = \emptyset\} = \Sigma^* \backslash \text{Sub}(L)$. A congruence class is not empty if it is a subset of Sub(L) [5]. We say that a grammar is *composite-free* if there is no rule producing a composite congruence class, with the exception of the start symbol [5]. To get back to the last step of the non-terminal symbol reduction, a canonical grammar must be composite free.

However, we must limit ourselves to grammars that have a finite number of prime classes. We note this subset of substitutable languages $\mathcal{L}_{Sc}$. It has been proven that this subset is included in the context-free languages [3].

To have a canonical grammar, we still need one more step: rule reduction. This step consists in replacing any sequences of any type, named α, that can be derived from a non-terminal symbol named A by this non-terminal. This allows reducing the length of the rules and a generalization of the language produced by the right-hand side of the changed rules. However, reducing the rules this way can produce different overlapping reductions. Indeed, if there are the following rules: $S \to \alpha_1\alpha_2\alpha_3$, $N_1 \to \alpha_1\alpha_2$ and $N_2 \to \alpha_2\alpha_3$, there are two reduction possible of S: $S \to \alpha_1 N_2$ or $S \to N_1\alpha_3$.

1.6 ReGLiS algorithm

In order to solve these problems, Coste *et al.* created a new algorithm named ReGLiS, for Reduced Grammar inference by Local Substitutability, which learns directly a reduced grammar during the inference. This algorithm has been defined in [5] and is detailed in algorithm 6. This algorithm works in two steps. In the first step, from line 1 to 19, ReGLiS builds the initial grammar. It is a grammar in which there is the start symbol and a non-terminal symbol for each congruent class. However, in this grammar, no generalization is done. Indeed, the language recognized by the grammar is only the language containing only the examples of the training set.

The next step is to generalize this initial grammar. The generalization tool used by ReGLiS is the rule reduction that I talked about in the last section. This is done by the Reduced_rhs function written in line 24 and developed in algorithm 2. To avoid the overlapping possible reductions, they

Input: String α, Set of rewriting rules P
Output: Set R of all non-redundant right-hand sides generating α

`/* Build parsing graph */`
$V \leftarrow \{i \in [1, |\alpha| + 1]\}$ `/* vertices */`
$E \leftarrow \emptyset$ `/* labeled directed edges */`
foreach (i, j): $\alpha_{i,j}$ proper substring of α **do**
  if $\exists([\![C]\!] \rightarrow \alpha_{i,j}) \in P$ **then**
    $E \leftarrow E \cup (i, j, [\![C]\!])$
  else if $|\alpha_{i,j}| = 1$ **then**
    $E \leftarrow E \cup (i, j, \alpha_{i,j})$
`/* Search for irreducible paths */`
$Ipaths[1] \leftarrow \{\{1\}\}$
for $j \leftarrow 2$ **to** $|V|$ **do**
  `/* Memorize irreducible paths arriving in` j `*/`
  $P \leftarrow \bigcup_{(i,j,l) \in E}(Ipaths[i].j)$
  $Ipaths[j] \leftarrow \{x \in P : \nexists y \in P, y \prec_r x\}$
$IP \leftarrow Ipaths[n]$
`/* Return corresponding rhs */`
$R \leftarrow \emptyset$
for $path \in IP$ **do**
  $rhs \leftarrow \epsilon$
  for $i \leftarrow 1$ **to** $|path| - 1$ **do**
    $rhs \leftarrow rhs.\beta_i$ with $\beta_i : (path[i], path[i+1], \beta_i) \in E$
  $R \leftarrow R \cup rhs$
return R

Algorithm 2: `Reduced_rhs`

decide to start by reducing the shortest right-hand sides first, in an optimal way, using a minimal grammar parsing approach. They create a parsing graph from an example and the production rules of the initial grammar. This parsing graph allows them to build all the possible paths for this example with the different production rules. From this graph, with a dynamic programming method, they optimize the reductions minimizing the right-hand side of each rules.

There exist multiple versions of ReGLiS. The one presented in algorithm 3 is defined for k,l-local substitutable languages. For the inference of zero-substitutable languages, contextually substitutable languages or local contextually substitutable languages, there exist algorithms named ReGis, ReGCiS and ReGLCiS, respectively. The intuition is that it is sufficient to change only the criterion of weak substitutability for the creation of the class in order to be able to learn it.

The grammar produced by ReGLiS is much more understandable than the ones produced by SGL. Moreover, they are built in much faster time. To give some numbers, with zero-substitutable languages, with 130 strings of length 20, SGL takes around 300 seconds to give a result where ReGLiS only takes 50 seconds. With the same string characteristics but with a 3,3-local substitutability language, SGL takes around 100 seconds whereas ReGLiS takes less than 50 seconds. It has also been proved that, contrary to SGL, ReGLiS is identifiable in the limit from polynomial time and data[5].

Input: Set of strings K on alphabet Σ, int k, int l
Output: Grammar $G = \langle \Sigma, N, S, P \rangle$

`/* Partition` $\mathrm{Sub}(K)$ `in substitutability classes */`

`/* Initial partition */`

$\mathcal{C} \leftarrow \{\{y\} : y \in \mathrm{Sub}(K) \setminus \{\lambda\}\}$

`/* Merge classes with respect to weak local susbtitutability evidence */`

foreach $uy_1v \in \mathrm{Sub}(K), uy_2v \in \mathrm{Sub}(K)$ with $u \in \Sigma^{\lhd k}, v \in \Sigma^{l\rhd}, y_1 \in \Sigma^+, y_2 \in \Sigma^+, y_1 \neq y_2$ **do**
  $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{C(y_1), C(y_2)\}) \cup \{C(y_1) \cup C(y_2)\}$

`/* Initial grammar */`

$N \leftarrow \emptyset, P \leftarrow \emptyset$

for $C \in \mathcal{C}_K$ **do**
  `/* Start symbol */`
  if $C \cap K \neq \emptyset$ **then**
    $N \leftarrow N \cup \{[\![C]\!]\}$; $S \leftarrow [\![C]\!]$
    for $y \in C$ **do**
      $P \leftarrow P \cup \{[\![C]\!] \to y\}$
  `/*` K`-primes */`
  else if $(\forall C' \in C_K : C \subseteq\, !C'\Sigma^+$ **and** $C \subseteq\, !\Sigma^+C')$ **then**
    $N \leftarrow N \cup \{[\![C]\!]\}$
    for $y \in C$ **do**
      $P \leftarrow P \cup \{[\![C]\!] \to y\}$

`/* Generalization */`

repeat
  $P' \leftarrow P$; $P_s \leftarrow ((\![C]\!] \to \alpha) \in P$, sorted wrt $\alpha)$
  $P \leftarrow \emptyset$
  `/* Reduce rules */`
  for $([\![C]\!] \to \alpha) \in P_s$ **do**
    for $\beta \in$*Reduced_rhs*$(\alpha, P_s \cup P)$ **do**
      if $\exists([\![C']\!] \to \beta) \in P, [\![C']\!] \neq [\![C]\!]$ **then**
        `/* Unification by substituting` $[\![C]\!]$ `for` $[\![C']\!]$ `*/`
        $P \leftarrow P_{\{[\![C']\!] \mapsto [\![C]\!]\}}$; $S \leftarrow S_{\{[\![C']\!] \mapsto [\![C]\!]\}}$; $N \leftarrow N \setminus [\![C']\!]$
        $P_s \leftarrow P_{s\{[\![C']\!] \mapsto [\![C]\!]\}}$
      else
        $P \leftarrow P \cup \{[\![C]\!] \to \beta\}$
  `/* Remove newly detected composite classes */`
  while $\exists [\![C]\!] \neq S : \exists! \beta\, ([\![C]\!] \to \beta) \in P$ **do**
    $P \leftarrow P_{\{[\![C]\!] \mapsto \beta\}}$
    $N \leftarrow N \setminus \{[\![C]\!]\}$
  end
until $P = P'$
return $\langle \Sigma, N, S, P \rangle$

Algorithm 3: ReGLiS (**Re**duced **G**rammar inference by **L**ocal **S**ubstitutability)

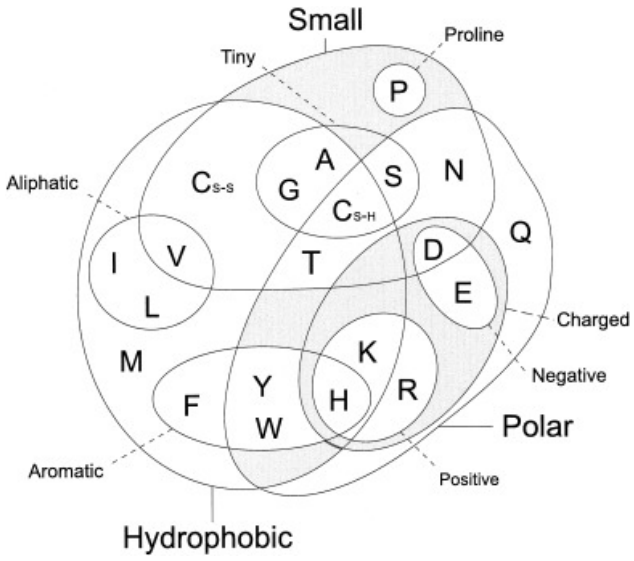

Figure 1: All the amino acids grouped by their biological characteristics

1.7 Pre and post-processing for biological data

ReGLiS has no particular conditions on the data it needs to learn, except it needs to be sequential. Therefore, it is possible to learn protein families. To give a brief biological background, a protein is a sequence composed of amino acids, molecules used as the bricks of proteins. It can only be composed by one of the different amino acids. There exist 20 amino acids so we can consider that the language of the proteins is composed of an alphabet of 20 characters. We represent an amino acid by a letter and a protein by the concatenation of the different amino acid's letter at each position. For instance, a protein can be represented this way: *MASSNLLTLALFLVLLTHANSSNDA*.

As for the nucleotides in the DNA, the amino acids can mutate over the years and, therefore, evolve. There exists, then, proteins that descends of other proteins, proteins that have the same evolutionary parents, etc. A protein family is a group of proteins that are evolutionary related. It means that they tend to have similar structures, similar sequences, and similar biological functions. As the domain of my team is bioinformatics, we want to focus on how to learn these protein families.

Before giving the protein sequences to the learning algorithm, we decided to process them, because of the natural mutation of amino acids. Sometimes, two protein sequences are very close together and there is only a difference of a few amino acids between them. Moreover, two amino acids can be very close, biologically speaking. They can have the same chemical properties (electrically charged, hydrophobic, etc) and behave virtually the same way. The classification of the different amino acid, according to their chemical behavior is shown in Figure 1. In the case of a relatively meaningless mutation, we do not want our algorithm to use irrelevant information.

To avoid these problems we transform our amino acid sequences into PLMA block sequences. This stands for Partial Local Multiple Alignment. This is a process applied to our proteins that groups amino acids that repeat similarly throughout the sequences together. They are called PLMA

blocks and they are sequentially numbered. An example of this step is shown in Figure 2,(a) and (b.1). With this transformation, there is no repetition of vocabulary within one sequence and the alphabet size is increased.

Doing this preprocessing also prevents, as a side effect, the formation of hubs. A hub is a substring we can find on the learning set given to ReGLiS that is repeated a large number of times, in different contexts. Therefore, it groups together a lot of different substrings in the same congruence class. If there is another hub within the other substrings, more classes are then merged. This effect is particularly true when the sequences are long, the alphabet is small and the k and l parameters are low. This tends to create a few big classes and a lot of information can be lost.

With this preprocessing step, additionally to plma blocks, gaps can be created. A gap a substring of the protein that cannot be matched with any plma blocks. Gaps can have various sizes and are not numbered. The example of Figure 2 (point (b.1)) contains a gap, noted, naturally, Gap.

Once we have determined where are the different plma blocks and gaps in the different sequences we index and remove the gaps from the sequences. The gap-removed sequences are given as an input to the learning algorithm (ReGLiS). Gaps are reintroduced, afterwards, in the output grammar. We do this because we consider that the gaps do not carry relevant information for the learning process. However, they still exist so they must appear in the final grammar.

In parallel, we keep a grammatical description of the composition of the plma blocks and the gaps. A gap is simply a non-determined number of amino acid. It can be translated in two production rules: $Gap \to \Sigma\ Gap$ and $Gap \to \lambda$, where Σ is an amino acid. For the plma blocks, we have a non-terminal for each column of the blocks that can produce all the amino acids we can find in this column. This step is shown in (b.2) of Figure 2.

Moreover, we also code the amino acids lexically. It means that each time we want to be able to produce it, we call a auxiliary non-terminal instead. To give an example, let us take a plma block, called $plma1$, of one column, that can only produce a "V" amino acid. Then, we would have: $plma1 \to N_V$ and $N_V \to "V"$ instead of $plma1 \to "V"$. This choice can be very convenient because N_V represent not only the amino acid "V" but mainly whatever representation we want of the amino acid "V". For instance, to generalize more this workflow, whenever we can produce an amino acid, we want to also be able to produce all amino acid close to it. This way, we do not have simply $N_V \to V$ but $N_V \to V|I|L$ without changing the rest of the grammar.

If a certain lexical non-terminal produces only the terminal symbol it is linked with (*e.g.* V for the non-terminal N_V), we say the grammar is coded with lexicalized rules. If this non-terminal produces also all amino acids closely related to the one it's linked with (*e.g.* V, L and I for the non-terminal N_V), we say it is coded with generalized lexical rules. The default behavior of the program codes the grammar with generalized lexical rules.

Finally, to form the final grammar, it is only needed to concatenate the grammar of plma blocks produced by ReGLiS, including gaps, with the grammar coding of these blocks. An illustration representing the complete current workflow is presented in Figure 2.

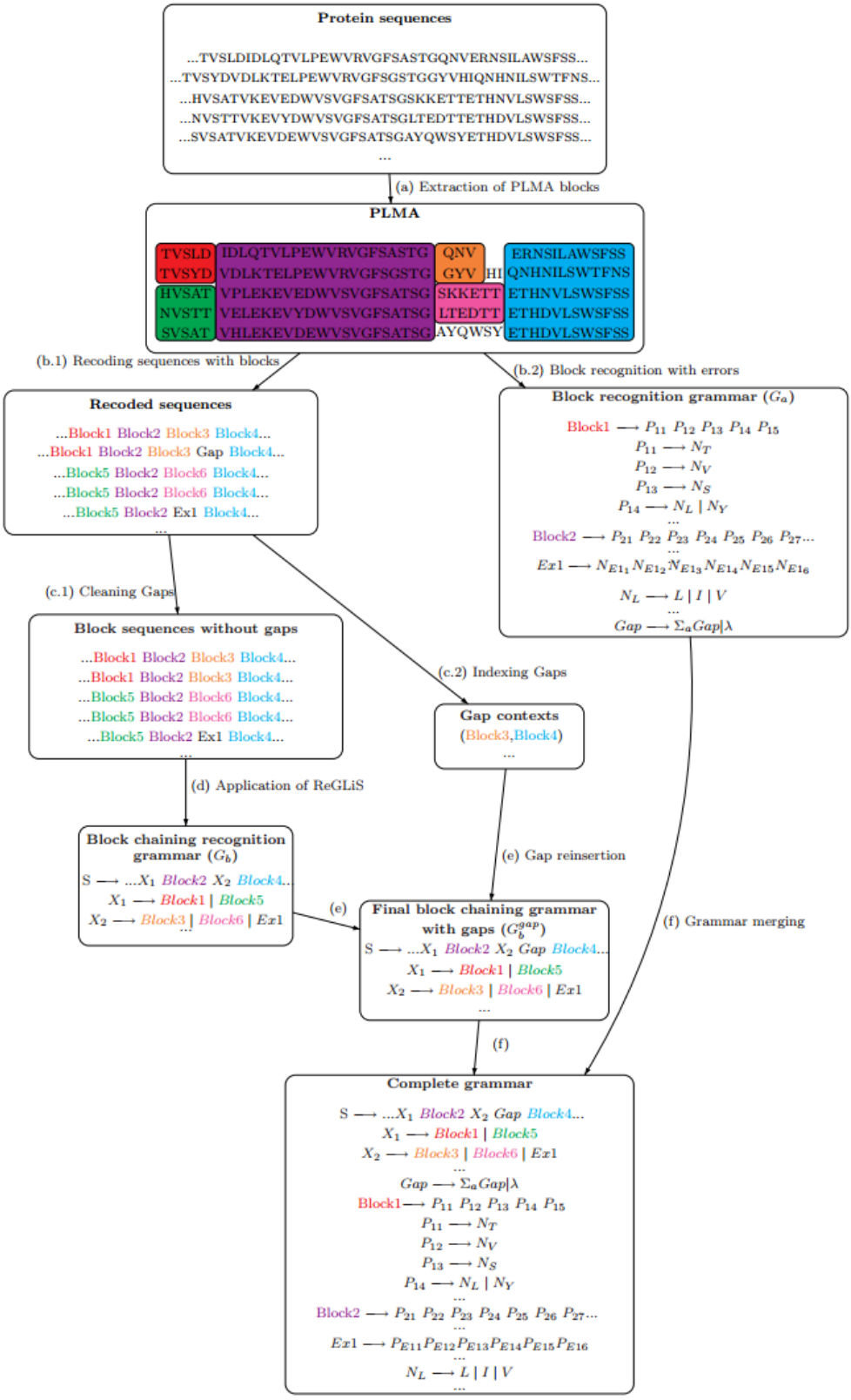

Figure 2: The workflow of ReGLiS adapted for proteins

2 Improving the parsing algorithm

2.1 Previous implementation of parser

The goal of my internship is to improve the ReGLiS algorithm for practical cases. Working on this project, I identified multiple problems. The first one was that the parser used to test the output grammar of ReGLiS was taking too much time for large scale experiments.

A parser is a program that will take, as input, a grammar and a sequence and will decide whether the sequence can be recognized by the grammar or not. If it is the case, we say that the sequence can be parsed. A parser usually offers, when the sequence can be parsed, a data structure representing its hierarchical structure with respect to the grammar. This structure is usually a tree, called a parsing tree, as it is a well-known structure particularly suited to emphasize hierarchies. In our case, the parser will be the tool we will need to calculate a prediction score for ReGLiS, our learning algorithm. Thus, having a quick parser is necessary for practical applications.

In the current workflow, the parser that is used is the one from the NLTK (Natural Language Tool Kit) library. It is a python library that is mainly used for natural language processing applications, including formal grammar manipulations. In this library, there exist multiple types of parsers. The implemented one was called a BottomUpChartParser. It is a parser particularly made for parsing large grammars.

Quickly, after a few experiments, I realized that this parser was really slow, even for small grammars that only have around a few hundred rules. So when, in practice, grammars can have more than 2000 rules, it seemed necessary to develop another solution to parse our sequences.

2.2 Earley's parser

There exist multiple very used parsing algorithms. One of the most wildly used is called the Earley's parser, first defined in [9]. It is a powerful parser that uses a dynamic programming approach. It can be used to parse context-free grammars that can be ambiguous or even non-deterministic. For these advantages, it is very useful, particularly in natural language processing applications. It is well adapted for our case, as our grammars can be both ambiguous and non-deterministic, and do not need any special format, like Chomsky normal form, for instance.

To briefly explain its working of process, beginning at the start production rule of the grammar, it moves forward a '$\bullet$' representing the progress of the parsing. If there exists a rule $X \to \alpha\beta$, having $X \to \alpha \bullet \beta$ means that we have already parsed α and we have still to parse β. Word after word, on the tested sequence, the Earley's parser changes of states. A state is a set of tuples noted $S(k) = \{(X \to \alpha \bullet \beta, i), ...\}$, where k is the position of the current word and i the position in which the rule was created. The aim of the algorithm is to move the parameter k from the 0^{th} index to the n^{th}, where n is the length of the tested sequence.

In order to do so, the parser can do three operations. The first one is called prediction. When a rule of the form $(X \to \alpha \bullet Y\beta, i)$ is in a state $S(k)$ and there exists a rule $Y \to \gamma$ in the grammar, the parser adds the rule $(Y \to \bullet\gamma, k)$ to the state $S(k)$. The second operation called scanning states

that if a is the next symbol to be parsed, then for each rule in $S(k)$ of the form $(X \to \alpha \bullet a\beta, k)$, the parser adds the rule $(X \to \alpha a \bullet \beta, k+1)$ to $S(k+1)$. The last rule, name completion, states that for every rule of the form $(Y \to \gamma\bullet, j)$ in $S(k)$, and $(X \to \alpha \bullet Y\beta, i)$ in the state $S(j)$, the parser adds the rule $(X \to \alpha Y \bullet \beta, i)$ to the state $S(k)$.

Inspired by [13], a convincing approach to implement this algorithm is called deductive parsing. It is an approach particularly suited for logic programming, a paradigm that we were comfortable with. It is also well suited for high level model changes, as heuristics or adaptation to biological sequences.

2.3 Deductive parsing

To describe the approach used in [13], deductive parsing sees the parsing as a deductive process and builds the algorithm accordingly. Deductive parser uses many types of logical structures. The first one is called an item. An item states a truth on the derivative possibilities of the grammar. The start item is called an axiom, from which the parsing process can begin, and the end item, from which it can finish, is called the goal. A deductive parsing system can have multiple axioms and goals. In order to go from the axiom(s) to the goal(s), we use inference rules. The general form of an inference rule is:

$$\frac{A_1 \qquad \dots \qquad A_k \qquad \langle \text{ opt. cond. on } A_i \text{ and } B \rangle}{B}$$

This means that from the statements A_i, if the side conditions are respected, we can deduce the B statement. The A_i statements are called antecedents and B is called the consequence.

The deductive system of Earley's algorithm is composed by the same four types of logical structures. The items, representing the content of the states of the algorithm, have this form: $[i, A \to \alpha \bullet \beta, j]$. The j index represents in which states the item is, *i.e.* $S(j)$, and the i index represents the origin number of the rule. Another way of seeing this, is to say that in the tested sequence, this rule parses the i^{th} to the j^{th} symbols. Formally, this item means that for a tested sequence $\omega = \omega_1 \dots \omega_n$, we have $S \Rightarrow^* \omega_1 \dots \omega_i A\gamma$ and $\alpha \Rightarrow^* \omega_{i+1} \dots \omega_j$.

The only axiom defined is $[0, Start \to \bullet S, 0]$. In other words, in the beginning of the parsing, we have parsed nothing, from index 0 to 0. The goal item is $[0, Start \to S\bullet, n]$. It states that we have parsed our sequence with the production of the start symbol of the grammar, from the index 0 to n. Then, the three inference rules, scanning, prediction and completion, have these forms, respectively:

$$\frac{[i, A \to \alpha \bullet \omega_{j+1}\beta, j]}{[i, A \to \alpha\omega_{j+1} \bullet \beta, j+1]}$$

$$\frac{[i, A \to \alpha \bullet B\beta, j] \qquad B \to \gamma}{[j, B \to \bullet\gamma, j]}$$

$$\frac{[i, A \to \alpha \bullet B\beta, j] \qquad [j, B \to \gamma\bullet, k]}{[i, A \to \alpha B \bullet \beta, k]}$$

It has been proven that the deductive parsing system of the Earley's parser is sound and complete. The authors of the papers also gave an implementation of this system, in Prolog.

2.4 Improvements of the Earley's deductive parser

Prolog is one of the principal languages of logic programmation. It is a language very used in artificial intelligence and natural language processing, amongst other domains. It is very efficient as it provides a very high level description of the system, particularly adapted in these domains. A big interest of this high level description is that it is particularly suited to model a problem and adapt it to our needs. As we will need to do some improvements on the parser, this option seemed the best.

With the collaboration of Jacques Nicolas, I took the implementation that was done by Schieber *et al.* and we adapted it for our workflow. Therefore, we made some improvement to the original parser. What took a lot a time, with the NLTK parser, was that the gaps were handled naively. Indeed, as state in section 1.7, a gap could produce both another gap and an amino acid, or nothing at all. Parsing a biological sequence without a technology made for this has been proven to be very time consuming [10]. Indeed, in this paper, Fredouille *et al.* states that an optimized parsing algorithm can have an important speed-up, which can be very interesting for our problem.

Therefore, we improved the logic system behind our deductive parser adding these inference rules dedicated to handling the gaps. We note $gap(N)$ a gap of size N, with $N \in \mathbb{N}$. If there exists a gap(N) in the sequence $\omega = \omega_1 \omega_n$, that means that for an index i, we have $\omega_1 \omega_i gap(N) \omega_{i+N+1} \omega_n \Rightarrow^* \omega_1 \omega_n$.

To take the gaps into consideration we have to take two cases into consideration, the gaps of size 0 and the gaps of size greater than that. To do so, we added two simple rules to handle them, named respectively *empty gap scanning* and *gap scanning*:

$$\frac{[i, A \to \alpha \bullet gap(0)\, \beta, j] \qquad gap(0) \Rightarrow \lambda}{[i, A \to \alpha\, gap(0) \bullet \beta, j]}$$

$$\frac{[i, A \to \alpha \bullet gap(G)\, \beta, j] \qquad gap(G) \Rightarrow \omega_{j+1} \dots \omega_{j+G} \qquad G \neq 0}{[i, A \to \alpha\, gap(G) \bullet \beta, j + G]}$$

Proof. To prove the soundness of the *gap scanning* rule, let us take the item $[i, A \to \alpha \bullet gap(G) B \beta, j]$. It means we have $\alpha \Rightarrow \omega_i \dots \omega_j$ and, by definition, $gap(G) \Rightarrow^* \omega_{j+1} \dots \omega_{j+G}$. We have then $\alpha gap(G) \Rightarrow \omega_i \dots \omega_{j+N}$ and then $[i, A \to \alpha gap(G) \bullet B\beta, j + G]$. Thus, the *gap scanning* rule is proved sound. Analogously, the *empty gap scanning* rule is also sound. Except that we have $gap(0) \Rightarrow \lambda$ and $\alpha gap(0) \Rightarrow \omega_i \dots \omega_j$. Moreover, as the size of a gap can only be equal or greater to 0, we have handled all possible gaps. Our rules are therefore sound and complete. □

In addition to these two rules, we added another rule to discard an empty gap (*i.e.* a gap which size is 0) if we have already parsed a non-empty gap:

$$\frac{[i, A \to \alpha\, gap(G) \bullet gap(0)\, \beta, j]}{[i, A \to \alpha\, gap(G) \bullet \beta, j]}$$

Proof. As this rule only treats a particular case, it does not conflict the completeness of our deductive system but we still have to prove its soundness. In order to do so, we can see that as $gap(0)$ can derive λ and only lambda, we have $(A \to \alpha\, gap(G) \bullet gap(0)\, \beta) \Leftrightarrow A \to \alpha\, gap(G) \bullet \beta)$. This inference rule is sound and complete. □

With these new rules, we can decide to skip over some gap rules instead of developing all the gap possibilities. Additionally, this way of working can be very efficient to analyze the parsing trees of the sequences. As the gaps are displayed with only their length, it gives more visibility to the trees. Indeed, sometimes, gaps can be longer than 200 amino acids.

Finally, the last improvement we have made is to delete the scanning inference rule, from the original Earley's parser. We have replaced them by more axioms that create the corresponding rules. Our complete deductive system is the one depicted in Figure 3.

Additionally to the improvements on the deductive system, we added two new predicates: *purescan* and *ahead*. To define them formally we have $purescan(X, a)$ iff $(\exists a \in \Sigma, X \Rightarrow a) \vee (\forall D, X \Rightarrow D, \exists nt \in (\Sigma \setminus a), purescan(D, nt) \wedge purescan(D, a)$. In other terms, a non-terminal is a purescan if it is only scanning the grammar through a certain degree, in this case, 2. We have $ahead(X, a)$ iff $X \Rightarrow B, purescan(B, a)$ and $\forall D, X \Rightarrow D, \exists nt \in \Sigma, purescan(D, nt)$. It gives the information about which characters are to be find after a non-terminal, if this one only derives purescans.

To explain it more informally, the *purescan* and *ahead* predicates describes non-terminal that can only be derived, according to a certain maximal depth, into a terminal symbol. This depth is, here, described as being three (two with the *purescan* and one for the *ahead*) as it is the empirical optimal size for the grammars constructed by ReGLiS for biological sequences. This is a high value and, therefore, using these predicates can potentially gain us some parsing time.

The predicate *purescan* is useful as it allows us, in addition to computing the different predicates *ahead*, to define the axioms (except $[0, Start \rightarrow \bullet S, 0]$) without searching again which non-terminal have the correct form. This can be interesting when we have a numerous sequences to parse in a row.

The predicate *ahead* is used to efficiently calculate the length G of a gap in the *gap scanning* rule. Indeed, to calculate the length of a gap, we need to take all the tokenized words that happen after the start of the gap (in the rules, this is j). This is a particularly expensive operation when the gaps are in the beginning of the tested sequence and all of the possibilities have to be kept for the exploration of the rules. If we have, in the *gap scanning* rule, $\beta \Rightarrow \delta\gamma$, we also have:$\exists nt \in \Sigma, \omega_{j+G+1} = nt \wedge ahead(\delta, nt)$. Therefore, having the *ahead* predicate, we can decrease the search time, and, in consequence, increase the parsing time.

We made two versions of the parser: one for the SICSTUS implementation of Prolog and one for SWI-Prolog. The first one is the state-of-the-art implementation but is, however, a paying version. SWI-Prolog is a wildly used open source implementation of Prolog that has also good performances. This implementation is found, by default, on Linux but is also available for Windows. Having both these implementations allows the user to choose between accessibility and performance.

Items:

$$[i, A \to \alpha \bullet \beta, j]$$

Axioms:

$$[0, Start \to \bullet S, 0]$$
$$[j-1, A \to \omega_j \bullet, j] \text{ if } \textit{purescan(A)} = \omega_j$$

Goal:

$$[0, Start \to \bullet S, 0]$$

Inference rules:
Prediction

$$\frac{[i, A \to \alpha \bullet B\beta, j] \qquad B \to \gamma}{[j, B \to \bullet\gamma, j]}$$

Completion

$$\frac{[i, A \to \alpha \bullet B\beta, j] \qquad [j, B \to \gamma\bullet, k]}{[i, A \to \alpha B \bullet \beta, k]}$$

Empty gap scanning

$$\frac{[i, A \to \alpha \bullet gap(0)\,\beta, j] \qquad gap(0) \Rightarrow \lambda}{[i, A \to \alpha gap(0) \bullet \beta, j]}$$

Gap scanning

$$\frac{[i, A \to \alpha \bullet gap(G)\,\beta, j] \qquad gap(G) \Rightarrow \omega_{j+1} \ldots \omega_{j+G} \qquad G \neq 0}{[i, A \to \alpha\, gap(G) \bullet \beta, j+G]}$$

Gap discarding

$$\frac{[i, A \to \alpha\, gap(G) \bullet gap(0)\,\beta, j]}{[i, A \to \alpha\, gap(G) \bullet \beta, j]}$$

Figure 3: The complete deductive parsing system

2.5 Results

In order to test the relevance of these improvements, we have made some tests on different grammars. The first one is a voluntarily simple grammar, named BmOr. It contains only 1 plma block and no gap. It has been coded with generalized lexical rules and has 142 rules. We test our results with this grammar to show how the presence or absence of gaps in a grammar influences the parsing time.

The two following grammars are made with real examples belonging to the family protein PS00308, described on the website PROSITE. It is a protein database registering protein families. In order to create the grammar, we give to ReGLiS a training set of the first 15 proteins of this family (recognized as true positives). We set the parameters k and l to 1. The final grammars both have 19 plma groups and multiple gaps. The first of them, called PS00308simple is coded with

	BmOr	PS00308simple	PS00308group
Original Earley's	0.50	639.93	> 1000
Earley's + *purescan*/*ahead*	0.58	439.65	> 1000
Earley's + gap handling	0.50	4.89	6.47
Complete adapted parser	0.56	4.94	6.08

Figure 4: The table of results comparing the different improvements made on the parser

lexicalized rules and has 98 rules. The other one, named PS00308group, is coded with generalized lexical rules and has 198 rules. Having those two grammars allows us to see the influence of these two types of coding with the different parser improvements. These grammars are grammars that can be parsed in practical case and yet are enough small to have reasonable parsing times.

These grammars have been tested with 4 different parsers. The original parser from [13], one with only gap handling rules in the inference rules, one with only the *purescan* and *ahead* predicates and finally, our final parser. For each of these tests, we launch the parsing process 5 times. We then take the average of the parsing times in order to minimize the fluctuations of the operating system. We put an arbitrary time limit at 1000 seconds for one parsing procedure as it would not be particularly relevant to go further.

For both of the PS00308 grammars, the set of tested sequences is composed of the training set used for ReGLiS. As the prediction result of ReGLiS is not significant for this experiment, taking the learning set allows us to see if all sequences are recognized (*i.e.* the parser works) or not. The set of tested sequences for BmOr is also the set used for its creation. It is composed of 11 sequences. Also to be noted that all of these tests are done with the SWI-Prolog implementation of the parser. The results are shown in Figure 4.

We can note that for the tests, the size of the different test sets were not identical. However, this is not a problem to analyze the results. Indeed, what is relevant is the difference of parsing times between two different parsers, and much less between grammars.

In this sense, we can see that the adding of the *purescan* and *ahead* predicates increases slightly the parsing speed on the complicated grammar but this is not very significant. On the simpler grammars, or when the gap are not handled, adding the *purescan*/*ahead* predicates increases slightly the parsing time but, in the worst case, to the tenth of an already very quick parsing. Therefore, we can conclude that this lost, for very simple grammar, is not significant and does not impact the parser on real, more complicated cases.

However, we have an enormous time difference between the parsers that handle gaps and the ones that do not with an order of magnitude of 100 with simple grammars. This seems to be even greater when the grammar becomes more complicated.

Then, we have to compare our final parser with the NLTK BottomUpChartParser that was previously implemented. We also did the test 5 times but, this time, with two test sets: one with positive and one with negative examples. The set of positive examples are composed the same way

	BmOr	PS00308simple	PS00308group
Complete prolog parser	1.66	9.46	15.27
NLTK BottomUpChartParser	1.78	163.58	340.52

Figure 5: The table of results comparing the final prolog parser and the NLTK parser

than for the previous experiment. The set of negative examples for both of the PS00308 grammar is built with the 10 first false positive sequences on the PS00308 PROSITE web page. The set of negative test for the BmOr sequence is composed of the 7 first sequences of the PS00889 protein family, on PROSITE. This way, we have results closest to parsing times in real cases. The results are shown in Figure 5.

In this sense, we can see that there is almost no difference between the two parsers with the BmOr grammar that means when no gaps are involved. However, when gaps are involved we have a parsing time that is around 20 times inferior with the new parser. Let us also take into consideration that the grammars that were tested were very simple. Indeed, the more complex was PS00308group with only 192 rules. In real cases we can have grammar with over 2000 rules. The gain becomes, therefore, necessary for practical uses.

Finally, we can notice that the score we have with the NLTK parser is much lower than the original deductive Earley's parser. This can be easily explained by multiple facts. First, the Earley's parser was not created to deal, in any sort, of a combinatorial created by the gap rules. Indeed, the Earley's parser is a top-down algorithm that means that it starts with the start production rule of the grammar and derives it until it finds the characters in the tested sequence. The NLTK parser is a bottom-up algorithm that means it does the opposite. It starts with the characters in the tested sequences and goes back to the start rule. This inherent difference between both algorithms may explain the different of the parsing times.

3 Generalization of ReGLiS

3.1 Limitations of ReGLCiS

Another issue I found in ReGLiS lies within its version for k, l-local contextually substitutable languages, ReGLCiS. It is said in [6] that it is only needed to change the definition of the substitutability when the substitutability graph is created (line 2-4 of algorithm 3) to adapt it to the corresponding class of language. However, it is possible to find an example with which the generalization with the parsing graph cannot be done.

First, let us define some terms. The first notion we have to explicit is the difference between the class of congruence and a class of substitutability. We already defined the congruence classes in Section 1.3. We note the set of substitutability classes, for a certain set $\mathcal{C}_\mathcal{L}$, as $\mathcal{N}_{\mathcal{C}_\mathcal{L}}$. A substitutabilty class is defined as followed: ${N^{\langle x,z\rangle}}_y \in \mathcal{N}_{\mathcal{C}_\mathcal{L}}, \exists x, z\Sigma^*, \exists [y] \in \mathcal{C}_\mathcal{L}, [y] = [x.{N^{\langle x,z\rangle}}_y.z]$. We can note that $\forall y, y' \in N_y^{\langle x,z\rangle}, N_y^{\langle x,z\rangle} = N_{y'}^{\langle x,z\rangle}$. Therefore, we note $N^{\langle x,z\rangle}$ all the strings that can be substituted in the context $\langle x, z\rangle$

To explain it more informally, a congruence class is composed of strings that have exactly the same syntactic meaning. You can substitute one complete string to another one from the same class in all situations. A substitutability class represents what we want to truly substitute inside a syntactic congruence class.

As a short-cut, We note LCS languages for local contextually substitutable languages defined in section 1.3. Similarly, we say LS languages for local substitutable languages. We can note that if L is a LCS language, and $N_y^{\langle x,z\rangle} \in \mathcal{N}_{mathcalC_L}, |x| = k$ and $|z| = l$. Moreover, if L is a LS language and $N_y^{\langle x,z\rangle} \in \mathcal{N}_{\mathcal{C}_L}, |x| = 0$ and $|z| = 0$.

To give an example, let us take the set of strings defined in example 1. If we consider the $\mathcal{L}(T) \in LS$, we would have the creation of at least the following congruence classes:$[a]$ and $[c]$. The associated parsing graph is shown in Figure 6. We can see that we can generate a path included both congruence classes.

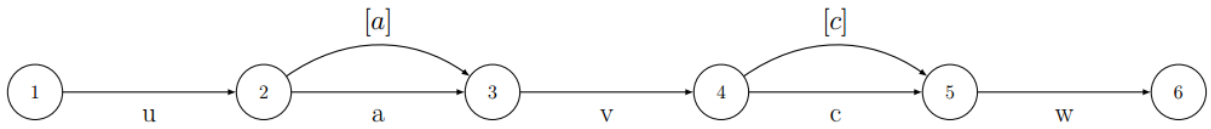

Figure 6: Previous parsing graph with the input 'uavcw' and the congruence classes $[a]$ and $[c]$

However, if we consider $\mathcal{L}(T) \in LCS$, we have creation of, at least, the following congruence classes : $[uav] = \{uav, ubv\}$ and $[vcw] = \{vcw, vdw\}$. The substitutability class, however, would ony be: $N_a^{\langle u,v\rangle} = \{a, b\}$ and $N_c^{\langle v,w\rangle} = \{c, d\}$.

Example 1. Let us define T :='uav', 'ubv', 'vcw', 'vdw', 'uavcw'

The issue with LCS is that there exists a difference between a congruence class and its associated substitutability class. For the k, l-local substitutable languages, we had, for prime congruence classes, $[y] = N_y^{\langle\lambda,\lambda\rangle} = N_y$ so there was no problem when we used the *Reduced_rhs* (algorithm 2). In the case of LCS languages, there is no direct equivalence between a substitutability class and its associated congruence class because of the possibly non-empty context.

Therefore, with the precedent example, we want to be able to produce, thanks to the example 'uavcw', the strings 'ubvcw', 'uavdw' and 'ubvdw'. However, when we would want to generalize 'uavcw' we would have an overlap of the two substitutability classes as shown in Figure 7. It is impossible to generate the example 'ubvdw' without a bigger characteristic set that would contain all the possibilities of the different congruence class used. Despite this overlap, we want to be able to generalize it as we have the evidence that the right context of the first class is strictly the same as the left context of the second class. But we cannot simply take the substitutability classes instead of the congruence ones: if there exists a class $N_v^{\langle a,c\rangle}$, it would be able to change the need contexts of the other classes.

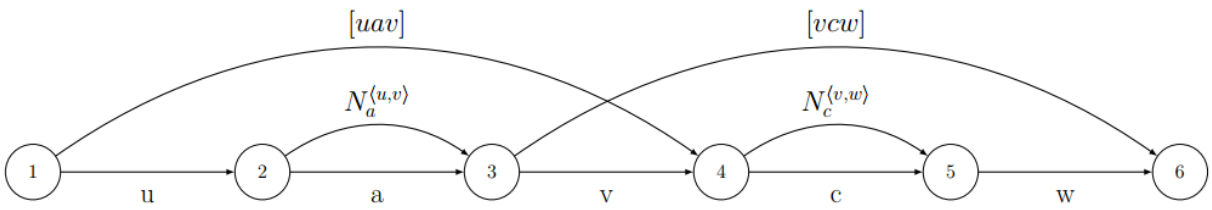

Figure 7: Previous parsing graph with the input 'uavcw', the congruence classes $[uav]$ and $[vcw]$ and the substitutability classes $N_a^{\langle u,v\rangle}$ and $N_c^{\langle v,w\rangle}$.

3.2 A new algorithm for k,l-local contextually substitutable languages

The difficult task, trying to adapt ReGLiS to LCS languages, is to build a correct parsing graph. Indeed, when the substitutability classes have contexts, we have to deal with the possibility of overlap between these contexts while being sure they cannot be modified. A complete algorithm, adapted for LCS languages, is described in algorithm 4, in the end of this section.

In order to solve this problem, we define a value of minimal inter substitutability distance: $d = Max(k,l)$. Adding this distance in the edges creation of the parsing graph solves the issue of overlapping contexts. Indeed, if we take only the substitutability classes in the graph (instead of the congruence classes), this distance is sufficient to force each substitutability class to have both a correct left and right context. This will allow more generalization as there will be no more concurrent contexts.

In order to manipulate correctly the terms of the input sequence of the algorithm, let us define some more terms. Let us note $\alpha = \alpha_1 \dots \alpha_n$ as the sequence that is parsed. We note $\alpha[i]$ the string α_i, $\alpha[i,j]$ the string $\alpha_i \dots \alpha_j$ and, similarly, $\alpha[i,j[$ the string $\alpha_i \dots \alpha_{j-1}$, for $j,i \in [1,n]$ with $j > i$. Finally, we note that $\alpha[i,i[= \lambda$.

There are multiple differences between this algorithm and the original *Reduced_rhs* (algorithm 2). The first one is how the parsing graph is built. In this new algorithm, we consider each vertex as a parsing position in α. In other words, when the parsing is in the i^{th} position of the parsing graph, it means that the next character to be parsed is a non-terminal that contains α_i. On this graph, an edge (i,j,l) represents how to parse the sequence from the i^{th} index to the $(j-1)^{th}$ with a label l. In other words, the edge (i,j,l) means that we are able to parse in α, $\alpha[i,j[$. Indeed, as the edge goes to the j^{th} vertex, it means the next character we will have to parse will be j. The label of an edge is the sequence of the more general right-hand sides between the index i and $j-1$.

The vertices are built in the beginning, as we know the size of α and, therefore, all the possible parsing positions. Then, we build the edges corresponding to the parsing of $alpha[i,i+1[$, or, in other words, from the letter i to the next one. We note $N_{\alpha[i,i+1[}$ the non-terminal that can rewrite the letter α_i because we have $N_{\alpha[i,i+1[} = N_{\alpha[i]} = \{\alpha[i]\} = \{\alpha_i\}$. We also define $N_{\alpha[i,j[} = N_{\alpha[i,i+1[} \dots N_{\alpha[j-1,j[} = \alpha[i,j[$, for $j \leq i+2$. We can do this because, as we work only with α, we can consider the contexts of these classes implicit.

Then, we build the edges for substitutability classes. We do it the same way but we make sure that there exists a distance minimum d between two classes. This is done by adding d to the original arriving index of the edge. We label this edge with the non-terminal of the substitutability class concatenated with the non-terminal of the additional distance like this:$(i+k,(j-l)+d+1,N^{u,v}.N_{\alpha[j-l,(j-l)+d+1[})$ where we have $i,j \in V, u.N^{u,v}.v = \alpha[i,j]$.We use this step to make sure that a rewriting rule that is able to rewrite a part of α has its right context.

Then, similarly to the previous parsing graph algorithm, we use a dynamic programming approach to find the more general rewriting paths. For each vertex, we associate a set of irreducible paths. This set is built as all the irreducible paths of its predecessors in the graph for which, within this set, there is no other path that is a subsequence of it. Then, we can transform the paths into the sequence of final right-hand sides. We do this computation for all vertices.

Finally, the last step consists in rebuilding all the right-hand sides that can generate α given the vertices indexes on all the irreducible paths. This is done by simply concatenating the labels of the parsed edges.

We say the parsing graph generated by algorithm 4 is *contextually non-conflicting* if we have:

Definition 7. $\forall r \in R, N_y^{\langle x,z\rangle} \in r \Rightarrow N_{\alpha[i-k,i[}.N_y^{\langle x,z\rangle}.N_{\alpha[j+1,j+l+1[} \in r$

Proof. Let us note $y = \alpha[i,j]$. Then we would have $N_{\alpha[i,j]}^{\langle x,z\rangle}$. In order to complete this proof, we have to consider two cases. First, let us take the case where $k \leq l$. Then, the edge representing the rule would be $(S,E,L) = (i, j+l+1, N_y^{\langle x,z\rangle}.N_{\alpha[j+1,j+l+1[})$. This works for all sizes of $N_y^{\langle x,z\rangle}$ (in the worst case $i = j$) and for all k and l (in the worst case $l = k = 0$).

We know, for sure, that we will recode $N_{\alpha[j+1,j+l+1[}$ as it is included in the label of (S,E,L). However, for $N_{\alpha[i-k,i[}$, we have to look by which vertices the paths goes to be sure the left context of $N_y^{\langle x,z\rangle}$ will be conserved. There can be a conflict only on the vertices indexed from $i-k$ to i, as it is the needed context of $N_y^{\langle x,z\rangle}$. Therefore let represent the conflictual edge closest to (S,E,L) this way: $(S',E',L') = (i', j'+l+1, N_{y'}^{\langle x',z'\rangle}.N_{\alpha[j'+1,j'+l+1[})$. As this edge is a conflictual one, we can rewrite $(j'+l+1)$, the vertex that poses a conflict, as $(i-k+k')$, we have: $(S',E',L') = (i', i-k+k', N^{\langle x',z'\rangle}y'.N_{\alpha[i-k+k'-l,i-k+k'[})$, with $k' \in [0,k]$. By construction, we also have: $\nexists k'' > k', (S'', i-k+k'', L'') \in E$. Therefore, a substring of one of the final paths we will have will be:$(i').(i-k+k')\ldots(i)$.

This path would be recoded as $(N^{\langle x',z'\rangle}y').(N_{\alpha[i-k+k'-l,i'-k+k'[}).(N_{\alpha[i-k+k',i[}).(N_y^{\langle x,z\rangle}))$. And as this sequence of right-hand sides is equivalent to $(N^{\langle x',z'\rangle}y').(N_{\alpha[i-k+k'-l,i-k[}).(N_{\alpha[i-k,i[}).(N^{\langle x,z\rangle}y))$. This rewriting is correct as we have $i-k+k'-l \leq i-k$ because of the conditions $k \leq l$ and $k' \leq k$. Thus, we can see that we have conserved both the left and the right context of the non-terminal $N_y^{\langle x,z\rangle}$.

The other case, when $k > l$, is very similar to the precedent one. In this case, the edge representing $N_y^{\langle x,z\rangle}$ is written this way: $(S,E,L) = (i, j+k+1, N_y^{\langle x,z\rangle}.N_{\alpha[j+1,j+k+1[})$. Its closest

conflictual edge is: $(S', E', L') = (i', j' + k + 1, N_{y'}^{\langle x', z' \rangle}.N_{\alpha[j'+1,j'+k+1[})$. We note $(j' + k + 1)$ as $(i - k + k')$ and similarly, we can rewrite (S', E', L') as $(i', i - k + k'1, N_{y'}^{\langle x', z' \rangle}.N_{\alpha[i-k+k'-k,i-k+k'[})$. The substring of one of the final paths would then be:$(i').(i - k + k') \dots (i)$ and it can rewrite this way $(N^{\langle x', z' \rangle}y').(N_{\alpha[i-k+k'-k,i-k[}).(N_{\alpha[i-k,i[}).(N^{\langle x,z \rangle}y))$. In the same way, this is a valid rewriting as $i - k + k' - k \leq i - k$ because of the condition $k' \leq k$. Thus, we have proved that algorithm 4 produces parsing graph that are *contextually non-conflicting*. □

The other advantage of this algorithm, additionally to this property, is that it gives the more general paths possible.

Proof. The only edges that can be incompatible are the ones with overlap within the distance d, after a certain substitutability class. Therefore, if $k > l$, it would be in conflict with the left context of the next substitutability class. If $k \leq l$, it would be in conflict with the right context of the first substitutability class. Therefore, there cannot be a conflict that could be avoided, preventing some generalization.

Moreover, we use the more restrictive generalizing classes, the substitutability ones. Finally, we can note that, by construction, we have: $\forall N_y^{\langle x,z \rangle} \in \mathcal{N}_\mathcal{C}, \exists r \in R, substring(N_y^{\langle x,z \rangle}, r)$.

In conclusion, as we have the more generalizing classes, all classes are included and that no conflict can under generalize the results, we have the more general paths possible. □

With these two advantages, the result of this algorithm, for the example 1, is the new parsing graph, for $k = l = 1$, depicted in Figure 8. We can see that there is not overlap anymore and that the more general path contains both the two substitutability classes and their contexts. With this new parsing graph, we are able to generalize α and, thus, produce 'ubvdw'.

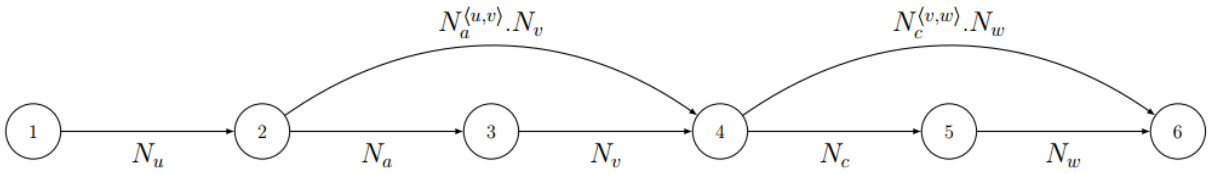

Figure 8: New parsing graph with the input 'uavcw' and the substitutability classes $N_a^{\langle u,v \rangle}$ and $N_c^{\langle v,w \rangle}$.

Input: Parsed string α, language parameters k and l, set of substitutability classes $\mathcal{N}_\mathcal{C}$
Output: Set R of all non-redundant right-hand sides generating α

`/* Building parsing graph */`
$V \leftarrow \{i : i \in [1, |\alpha| + 1]\}$`/* vertex for each parsing position */`
$E \leftarrow \{(i, i+1, N_{\alpha[i,i+1[}) : i \in [1, |\alpha|]\}$`/* symbol parsing edges */`
$d \leftarrow \max(k, l)$ `/* inter substitutability minimal distance */`
$E \leftarrow E \cup \{(i+k, (j-l)+d+1, N^{u,v}.N_{\alpha[j-l,(j-l)+d+1[}) : i, j \in V, u.N^{u,v}.v = \alpha[i,j]\}$`/* parsing edges for each non-terminal and their minimal substitutability distance */`
`/* Searching for irreducible paths */`
$IP[1] \leftarrow \{'1'\}$ `/* Starting irreducible path */`
for $j \leftarrow 2$ ***to*** $|\alpha| + 1$ **do**
  $P \leftarrow \bigcup_{(i,j,l)\in E}\{p.l : p \in IP[i]\}$
  `/* Irreducible paths are the ones that have no other paths as subsequence */`
  $IP[j] \leftarrow \{p \in P : \nexists p' \in P, is\_subsequence(p', p)\}$
`/* Building right-hand sides from indices */`
$R \leftarrow \emptyset$
foreach $p \in IP[|\alpha| + 1]$ **do**
  $rhs \leftarrow \lambda$
  for $i \leftarrow 1$ ***to*** $|p| - 1$ **do**
    $rhs \leftarrow (rhs.l : (i, i+1, l) \in E)$
  $R \leftarrow R \cup rhs$
return R

Algorithm 4: Reduced_rhs adapted for k, l-local contextually substitutability languages

4 Conclusion

In this internship, I worked on two very different parts of the learning process of ReGLiS: the parser and the learner. For the parser, my contribution is an adaptation of the deductive system of the Earley's parser for biological sequences that allows realistic parsing time for practical cases. Indeed, with this new parser we have a parsing time around 20 times faster than previously.

However, the Earley's parser is a top-down algorithm and it may not be the most optimized type of parser to search through gaps. We have been able to see it comparing the original Earley's algorithm with the NLTK BottomUpChart parser. Both parsers do not handle gaps, but the NLTK one was much faster. One improvement one can made, in the future, is to adapt a bottom up parser for handling gaps. In consequence, the fact we use Prolog proves to be very useful as it would be much easier to do such a task : we only had to change the different items, axioms, goals and inference rule to completely transform the parser into another one. In other words, we will almost only need to change the model and the parser will work.

Currently, the main purpose of the research behind the improvement of the parser was to accelerate it. One other lead that might be interesting would be to use the parser to give a score to the positively parsed sequences. Indeed, when a grammar is coded with generalized lexical non-terminals, one amino acid can be closer to the main one than another. Moreover, inside a plma group, sometimes, on a certain column, an amino acid is found very frequently compared to another. These informations can be transformed as probabilities or score and the structure of our parser can be updated to compute it while parsing. We just need to be careful about the complexity of such an algorithm.

The second part on which I worked was the learning algorithm. Seeing the limitation of ReGLiS for LCS languages, I proposed a new algorithm that can successfully learn this class of languages. This algorithm works by creating a parsing graph using only substitutability classes and a minimal inter substitutability distance. I also proved that this algorithm can create contextually nonconflicting parsing graphs and the more general paths possible.

The next step would be to integrate this new algorithm into a whole new learning process, adapted from ReGLiS. More research will probably necessary to find an optimized form of reduced output grammar. Similarly, we can also wonder if the definition of composite and prime classes will be identical to the one used in ReGLiS. At term, it would be interesting to compare k, l-local substitutable languages and k, l-local contextually substitutable languages in practical cases. Thus, we will be able to gauge more precisely the concrete interest of this new class of language.

The class of language that is the closest to k, l-local substitutable languages is the k, l substitutable languages. Ryo defined an algorithm, in [14], very similar to SGL that is able to learn this last type of language. In light of the limitations of ReGLCiS, we would like to better understand the validity scope of $SGL_{k,l}$. Further investigation is needed to better understand the limits of this algorithm and in which way our new algorithm would improve that.

Always in the aim of improving ReGLiS, we could also use other approaches as learning heuristics. For instance, we could use negative examples in the learning process to prevent overgeneralization. Moreover, another possibility could be to define a minimum support of evidence to create a congruence class.

References

[1] Noam Chomsky. On certain formal properties of grammars. *Information and control*, 2(2):137–167, 1959.

[2] Alexander Clark. Learning deterministic context free grammars: The omphalos competition. *Machine Learning*, 66(1):93–110, 2007.

[3] Alexander Clark. Learning trees from strings: A strong learning algorithm for some context-free grammars. *The Journal of Machine Learning Research*, 14(1):3537–3559, 2013.

[4] François Coste, Gaëlle Garet, and Jacques Nicolas. Local substitutability for sequence generalization. In *Proceedings of the Eleventh International Conference on Grammatical Inference, ICGI 2012, University of Maryland, College Park, USA, September 5-8, 2012*, pages 97–111, 2012.

[5] François Coste, Gaëlle Garet, and Jacques Nicolas. A bottom-up efficient algorithm learning substitutable languages from positive examples. In *Proceedings of the 12th International Conference on Grammatical Inference, ICGI 2014, Kyoto, Japan, September 17-19, 2014.*, pages 49–63, 2014.

[6] François Coste, Franco Luque, Gaelle Garet, and Jacques Nicolas. Learning efficiently substitutable context-free languages from positive examples by reduction. (6958):1001–1028, 2016.

[7] Colin de la Higuera. Characteristic sets for polynomial grammatical inference. *Machine Learning*, 27(2):125–138, 1997.

[8] Colin De La Higuera. A bibliographical study of grammatical inference. *Pattern recognition*, 38(9):1332–1348, 2005.

[9] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[10] Daniel Fredouille and Christopher H Bryant. Speeding up parsing of biological context-free grammars. In *Combinatorial Pattern Matching*, pages 241–256. Springer, 2005.

[11] E Mark Gold. Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

[12] Z. S. Harris. Distributional Structure. *Word*, (23):146–162, 1954.

[13] Stuart M Shieber, Yves Schabes, and Fernando CN Pereira. Principles and implementation of deductive parsing. *The Journal of logic programming*, 24(1):3–36, 1995.

[14] R. Yoshinaka. Identification in the limit of (k,l)-substitutable context-free languages. In *Proceedings of the 9th international colloquium conference on Grammatical inference: theoretical results and applications*, ICGI'08, pages 266–279, 2008.