INRIA Rennes – Bretagne Atlantique

Compressing (genomic) sequences by grammar inference

Internship Report

Francisco Dorr

Director: François Coste

Rennes, France, March-August 2014

CONTENTS

1. STATE OF ART

Work in grammar-based compression has been made with different approaches. The most common way was using straight line grammars (SLG).

As stated in [5], a *SLG* is a grammar that generates exactly one sequence, so it doesn't contain loops nor ambiguities.

The general idea is to look for a grammar that represents a sequence without losing information.

The way to do that is searching for subsequences in such sequence that could be replaced by a Non Terminal symbol. By doing so, we could reduce the size of the sequence, because if we have a repeated subsequences, we can just assign a symbol to it and write the symbol instead of it.

Each time we make a replacement we start over looking for new subsequences in the new sequence to replace. The iteration process continue until we cannot compress it anymore.

For example, an SLG from the follow sequence [5]:

$$ttatatttctttctttctttttttcctcagcctcagagt$$

could be:

$$S \rightarrow N_3 ata N_3 N_2 N_2 N_2 N_3 N_3 N_3 N_1 N_1 agt$$
$$N_1 \rightarrow cctcag$$
$$N_2 \rightarrow tcN_3$$
$$N_3 \rightarrow tt$$

When using this kind of grammars, the search of constituents to be replaced by non Terminals is commonly based on different types of repeats. Those could be Repeats, Maximal Repeats, Longest Maximal Repeats, etc. Explanation about different repeats and its taxonomy can be founded here: [1, p.42-46].

In the example given above those constituents obtainded from repeats in the sequence are $cctcag$, tcN_3, tt.

But this approach has a limitation: what happend if there are words $w_1..w_n$ similar (with just some characters changed) to a repeat R_i choosen as constituent? We would ignore them but they could be useful. If we only use repeats, we could skip constituents of interest.

For example, if we have the next sequence S:

$S \rightarrow R_i \ --- R_i \ --- \ w_1 \ w_2 \ --- w_3 \ --- \ ... \ --- \ R_i \ --- \ w_n$

Using the classical approach we would replace only the R_i:

$S \rightarrow N \ --- \ N \ --- \ w_1 \ w_2 \ --- \ w_3 \ --- \ ... \ --- \ N \ --- \ w_n$

But if we add some new rule N with ambiguity we could also replace the similar words $w_1, .., w_n$:

$S \rightarrow N \ --- \ N \ --- \ N \ N \ --- \ N \ --- \ ... \ --- \ N \ --- N$

Based on this question, we started our work. In the next section we propose a way to be more flexible and to find interesting structural information.

2. NEW APPROACH PROPOSED

Adding rules with a certain type of ambiguity allows us to take more constituents to be replaced, because we aren't limitated only by the repeats.

That's why we are going to introduce a kind of CFL (Context-Free languages) that can be used for that purpose: the *k,l substitutable*.

2.1 Local-context K,L Substitutable Language

We will denote by **yield** of N ($yields(N)$) the strings that the non terminal N yields.
$\boldsymbol{u}$ and $\boldsymbol{v}$ are, respectively, a prefix and a suffix of a yield and the will be called the **contexts** of the yield.

In an informal way, we can define the local-context k,l substitutable languages, as a subset of the CFL, where we have yields formed by contexts u and v of size k and l, sourrounding different insides y_i that can be interchangeable within them.

A language L is, k,l-local context substitutable if for any $x_1, y_1, z_1, x_2, y_2, z_2, x_3, z_3 \in \sum^*$ and $u, v \in \sum^{k,l}$, such that
$uy_1v, uy_2v \neq \lambda$,
$x_1uy_1vz_1 \in L \wedge x_3uy_2vz_3 \in L \Rightarrow (x_2uy_1vz_2 \in L \Leftrightarrow x_2uy_2vz_2 \in L)$.
Definition could be find in [2, p. 3-4].

Let us clarify with an example:

With contexts we mean u and v subsequences of our grammar. For example if we had:

$$S \rightarrow agtggtgaaccaagatggaccacacggttggaccg$$

Two possible contexts could be, for example, $u = ag$ and $v = gt$.

$$S \rightarrow \mathbf{u}tg\mathbf{v}gaacca\mathbf{u}atggaccacacg\mathbf{v}tggaccg$$

and by pairing them in an ordered way we can obtain different inside subsequences surrounded by those given contexts. In this case, those insides are:

$$y_1 \rightarrow tg$$
$$y_2 \rightarrow atggaccacacg$$

The insides will give us the ambiguity:

$$S \rightarrow \mathbf{N}gaacca\mathbf{N}tggaccg$$
$$N \rightarrow uIv$$
$$I \rightarrow y_1|y_2$$

The problem here, is that if we want to decode the grammar and go back to our first sequence, there is no way to know which Inside to use in each replacement, as we did with the SLG approach.

In following sections we will see how to resolve it.

2.2 Choosing yields to replace

We could try all possible repeats pairs configurations of a given sequence S, and compare all of them to see which is the best one.

To do so, we should find the set R which contains all the repeats and, after that, we calculate the cartesian product RxR obtaining all possible contexts combinations, with a complexity of $\emptyset(n^2)$ where $|R| = n$.

For each pair in RxR we should apply some heuristic (see section *Heuristics*) and select the best one.

But checking all possible contexts pair is time and resources demanding as it depends on the size of the maximal repeat set. This is a naive approach because we could obtain extra information about each repeat to avoid crossing them all to improve the performance.

As stated in [1], the best way to manage the repeats is with a Suffix Array data structure. Information about it could be founded in: [1, p. 40].

To choose which contexts pair to replace we have many possibilites that we will study in the next chapter, but first we need to pair the contexts without overlaps.

2.3 Greedy Pairing

Suppose we take u and v contexts. We start analysing the sequence from left to right. If an u appears we store its position and continue our way. If another u appears we overwrite the last one. We continue this way until a v appears. At that moment, we mate the last u viewed with the first v, and that yield will be replacerd by a non-terminal. We repeat until the end.

This take $\Theta(n)$, where n is the size of the sequence.

It can be done directly in the suffix tree using the already known contexts positions ($pos_s(u)$ is the positions in the sequence of context u).

For example:

- $pos_s(u) \rightarrow |0|6|22|30|49|62|$
- $pos_s(v) \rightarrow |12|17|26|45|$

we start looking what's the first v position:

- $pos_s(u) \rightarrow |0|6|22|30|49|62|$
- $pos_s(v) \rightarrow |\mathbf{12}|17|26|45|$

and then we look for $\max_i(pos_s(u)[i]) < pos_s(v)[j]$, where j is the current index in $pos_s(v)$. In our first step, this would be $\mathbf{6-12}$. This way we can obaint all possible yields:

- $y_1 \rightarrow \mathbf{6-12}$
- $y_2 \rightarrow \mathbf{22-26}$
- $y_3 \rightarrow \mathbf{30-45}$

To avoid overlapping we can just ask if the start of a new pair doesn't step over the last context pair added. For example $6-17$ is avoided this way.

Once done, we use an heuristic to select the adequate yields to be replaced.

2.3.1 Heuristics

Until now we only focused in a greedy heuristic, that assigns a score in each iteration to each possible yield, but we have more possibilities.

Some are enumerated here:

1. Greedy Score Function
   - Assign to each contexts pair a score based on its occurrences, insides, contexts, length, etc.
   - The one with the best score is chosen to be replaced in each iteration.
2. Coloring a graph:
   - Each node represents a u-y_i-v yield founded
   - Each edge means that the two nodes it's joining are overlapped.
     If we color that graph and then choose the set of nodes with maximal coloring number we could obtain a good set of u-y-v to replace.
     As there are lot of u-y-v substring in a very large sequence S, we could split S in subsequences Ss of the same size and for each Ss build a graph and search for the best nodes set.
3. Max-Clique:
   - Each node represents an u-y-v substring
   - Each edge means that the two nodes it's joining are NOT overlapped.
     After building the graph we can search for the Max-Clique and that would be our set of nodes.
     If there're a lot of overlapped substrings perhaps is better to use Max-Clique, because graphs would be smaller.
     Using the G^t from Max-Clique approach, we could search for the maximum independent set.

2.3.2 Score function

We will call $|G| = \sum_{N \to \alpha \in P}(|\alpha| + 1)$.

Based on MDL principle [1, p. 8] we want to find an hypothesis H that minimize our sequence S:

$$|S| > min_H(|H| + |S/H|)$$

Where $|S|$ and $|H|$ are, respectively, the sizes in characters.
In our case, that would be a grammar:

$$|G| + |S/G|$$

Naming G_0 the original Sequence:

$$G_0 : S \to -uy_1v...uy_nv-$$

We want to find G_1 so: $|G_1| + |G_0/G_1| < |G_0|$

Or in general we want: $|G_n| + |G_{n-1}/G_n| < |G_{n-1}|$, to reduce the size in each iteration.

The score function is used to chose the yields to be replaced. It says us what is the the gain in our grammar size if we use a given contexts pair u-v and what's the penalty.

It's based on a greedy approach. It means, in each iteration we are trying to get the best possible reduction in the grammar size.

We use to different scores, one for the single words constituents (single repeats) and one for the contexts yields (paired repeats with insides). Then we compare them.

Single Repeat Score

Let w be a substring of the a sequence S, $|w|$ its lenght and $\emptyset(w)$ its occurrences in S. We want to replace each occurrence of w with a non terminal N ($\emptyset(w) * |w| - \emptyset(w)$), and add the new rule N to the grammar productions ($|w| - 1$). So we add this changes to the score function.

Then, we have:

$$\text{score} = \emptyset(w) * |w| - \emptyset(w) - |w| - 1$$

Contexts Yield Score

Beeing u and v prefix and suffix, respectively, of a chosen yield of the sequence S:

- $k = |u|$
- $l = |v|$

- p: number of non overlapped u-v contexts pairs in the sequence.
  For example, in the next sequence, $p = 3$

$$uy_1v...uy_2v...uy_3v$$

- q: number of different yields (uy_iv) (or number of different insides y).
  If all insides are distinguibles, then $q = p$, but in general, $q \leq p$.
- $\emptyset(y_i)$: occurrences of inside y_i.
- $pipes = (q - 1)$. Those are the pipes that separate the options in the I rules.

Finally, we got the score function:

$$score = k(p-1) + l(p-1) + \sum_{y \in Y} |y|(\emptyset(y) - 1) - p - 3 - pipes$$

That means:

- $k(p-1) + l(p-1)$: remove of contexts u-v occurrences from grammar, and the add of one of each to the new N rule.
- $\sum_{y \in Y} |y|(\emptyset(y) - 1)$: remove of insides within contexts, and the add of one of each to the new I rule.
- $-p$: added p N symbols when replacing the yields.
- -3: - N - I production symbol - I symbol inside N rule.
- $-(q-1)$: pipes in I production

If we obtain a $score > 0$ we are gaining in the next grammar size, it means it's going to be reduced.

So we compare the best yield score with the best single word score.

3. ALGORITHMS

3.1 Grammar with disambiguation list for Disambiguation: GELD

The problem with the new kind of grammar, compared to the old SLG is that we have ambiguity in it. For example:

$$S \rightarrow \mathbf{N}gaaccа\mathbf{N}tggaccg$$
$$N \rightarrow uIv$$
$$I \rightarrow y_1|y_2$$

When decoding, each time an N appears we don't know if we have to replace the inside I with and y_1 or a y_2

Hence, we have to add some information to avoid this issue. That could be and disambiguation list telling us what inside should be used. But this introduce a penalty in the grammar size in each iteration k:

$$|G_k| + |Encoding_k|$$

So, in this approach we should add to the score function the size of the disambiguation list in each iteration.

The list can be represented in many different ways. For example,

- for each non terminal we could have a queue with the elections of the insides of the non terminal. When we decode, each time we see this non terminal appear we pop the next element from the queue and see which option should be used:

  $S \rightarrow \mathbf{N}gaaccа\mathbf{N}tggaccg$
  $N \rightarrow uIv$
  $I \rightarrow y_1|y_2$ encoding queue: [1, 2]

  In this approach we are already compressing the disambiguation list, because we don't need to add an encoding element for each election.
  For example, if we have the next rule:

  $S \rightarrow \boldsymbol{N_2}gaaccа\mathbf{N}tggaccg\mathbf{NN}aggtg\mathbf{N}$
  $N \rightarrow \boldsymbol{N_2}agt$
  $N_2 \rightarrow uIv$
  $I \rightarrow y_1|y_2$ encoding queue: [1, 2]

  doesn't matter how many times we have the N non terminal, because we always know that the N_2 inside it has always the encoding 2.

- A sequence ordered in preorder, postorder, inorder, etc. based on the parse tree generated by the grammar.

  By ordering the sequence, we could also get intereseting structural motifs. But we need an encoding element for each time a non terminal with elections appears in the parse tree.

  reusing our last example:

  $S \rightarrow \boldsymbol{N_2}gaacca\mathbf{N}tggaccg\mathbf{NN}aggtg\mathbf{N}$
  $N \rightarrow \boldsymbol{N_2}agt$
  $N_2 \rightarrow uIv$
  $I \rightarrow y_1|y_2$ encoding queue: [1, 2, 2, 2, 2]

3.1.1 Pseudocode

Algorithm 1 GELD

Require: Sequence S

```
1: while Grammar size can improve do
2:     bestYield = getBestYieldBasedOnScore(allPossibleYields(S))
3:     replaceYield(S, bestYield)
4:     addProductionToSequence(N, bestYield)
5:     N++
6: end while
```

Algorithm 2 replaceYield

Require: Sequence S, yield

```
1: yieldPos = getYieldPositions(S, yield)
2: for pos in yieldPos do
3:     i = currentInside(pos)
4:     replace yield by N
5:     encodingList.insert(i)
6: end for
```

3.1.2 Compressing disambiguation list

If we manage to construct the disambiguation list in a way that is structurally interesting we could compress it effectively.

Efforts have been made to do it in a Depth-first search (DFS) approach. *DFS is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.* (`http://en.wikipedia.org/wiki/Depth-first_search`)

Complications were found during the implementation and DFS couldn't be done directly from the desambiguation list, but from a parse tree. Building Parse Trees is big time and resource consuming. In future work this will be a priority.

There are some ways not to add this type of penalties as a list outside the grammar, but inside it as new symbols. This approach will be explained in the next section.

3.2 Straight Line Inplace Disambiguation (SLID)

Instead of adding the same non terminal N to replace each yield in each iteration, and an disambiguation list to remember the inside selection, we can use a new symbol (N_{+i}) to refer to the inside Y_i inside the N rule. By doing so, we don't need an encoding scheme outside the grammar, because the information remains implicit in the new symbol. The counterpart is that we are adding new symbols and this affect as the iterations increase.

Our grammar would be like this:

$$S \rightarrow \boldsymbol{N_{+1}}gaacca\boldsymbol{N_{+2}}tggaccg$$
$$N \rightarrow uIv$$
$$I \rightarrow Y_1|Y_2$$

The decode is straightfoward rightnow, because we construct the grammar and do the replacements in such a way that there's no possible ambiguity. Now we have a symbol for each inside used. So, if we have for example N_{+1} we know we are refering to inside Y_1.

The SLID approach is extremelly useful in small sequences that doesn't need so many iterations. When the sequence get bigger, we reduce in each iteration the possibility of compressing by adding new symbols.

3.2.1 Pseudocode

Algorithm 3 SLID

Require: Sequence S

```
while Grammar size can improve do
    bestYield = getBestYieldBasedOnScore(allPossibleYields(S))
    replaceYield(S, bestYield)
    addProductionToSequence(N)
    N += numberOfDifferentInsides(N)
end while
```

Algorithm 4 replaceYield

Require: Sequence S, yield

```
yieldPos = getYieldPositions(S, yield)
for pos in yieldPos do
    i = currentInside(pos)
    replace yield by N_{+i}
end for
```

3.3 Parse Tree + SLID (PT+SLID)

To avoid adding new symbols in each iteration, we said one thing we can do is to have the disambiguation list outside the grammar. In each iteration we replace by just an N as in $GELD$ instead of N_{+1}, N_{+2} as in $SLID$.

At the end, based on the encoding, we can parse the grammar and traverse the tree generated.

By traversing it in a preorder manner, we add new Non Terminals to differenciate the N that we didn't differenciate before. It means, that if we have to subtrees with the same root but different nodes inside it, we separate it in as many rules as differences we have.

A preordering is a list of the vertices in the order that they were first visited by the depth-first search algorithm. This is a compact and natural way of describing the progress of the search (`http://en.wikipedia.org/wiki/Depth-first_search`).

Basically, we follow $GELD$ and when we reach the final grammar we convert it to a $SLID$ grammar.

In the next section this concept will be clarified.

3.3.1 Algorithm

Starting from the result of GELD, a grammar and and its disambiguation list, we want to obtain a grammar with the encoding implicit in it, as in the SLID approach.

We must use a different heuristic score for each iteration, because we are not having an disambiguation list at the end. Until now we are using the same score but without the encoding penalty.

We can think the algorithm as traversing a Parse Tree generated by the grammar and changing/adding some new non terminals on the fly.

We start in the root of Parse Tree and traverse the tree in a preorder way.

For each Non Terminal N node we reach, we add it with his children as a production to a productions set P.

If this non Terminal was an Inside, we assign a j value based on the option we are adding.

j: is refered to the inside election of the current non terminal. For example:
M_{+j} means that we are in the j option from the inside I of M in our parse tree traversing.

Now, each non terminal N, must be separated in N_1, ..., N_n, based on the M_{+j} inside them.

For example, watching the parse tree, we got that N could be:

$N \rightarrow M_{+\mathbf{1}}agtI$

But in another subtree it could be:

$N \rightarrow M_{+\mathbf{2}}agtI$

so we separate these two cases in:

$N^{\mathbf{1}} \rightarrow M_{+\mathbf{1}}agtI$

$N^{\mathbf{2}} \rightarrow M_{+\mathbf{2}}agtI$

So we need to add the index i to make a distinction within them: N^i

N^i_{+j} means that, we choose the N non terminal with index i and with option j in its *Inside*.

Example of the resulting grammar:

Original $S = gtggaccgaccacagagacctgaccgttggaacgaccacagaagaacacca$

$S \rightarrow g - 10^1 - 8_{+1} - t - 8_{+1} - g - t - 10^2_{+1} - a - 8_{+2} - a - c - c - a$

$7 \rightarrow c|a$

$8 \rightarrow g - a - 7 - c$

$9^1 \rightarrow 8_{+1} - a - c$

$10^1 \rightarrow t - g - 8_{+1} - 9^1 - a - g - a$

$10^2 \rightarrow t - g - 8_{+2} - 9^1 - a - g - a$

With all these new symbols added, there's no ambiguity, so we can just follow the non terminals and unfold them.

The resulting grammar will be different to the original SLID version, because in each iteration we weren't adding new symbols that decrease drastically our compression power.

3.3.2 Pseudocode

Algorithm 5 PT+SLID

Require: Grammar G

```
1: Run GELD algorithm with modify score
2: Build Parse Tree PT from G based on disambiguation list
3: prods = map{int, list} {the list will save the different right sizes (rhs) of N}
4: preNodeList = getNodesInPreorder(PT)
5: for N in preNodeList do
6:    rhs = N.rhs()
7:    if N not in prods then
8:       prods.insert(N, rhs)
9:    end if
10:   if N in prods but with a different rhs then
11:      prods(N).append(rhs)
12:      i = rhsPosInProds(N)
13:   end if
14:   if lastInside is N's inside then
15:      Modify N adding j in parse tree: N_{+j}
16:      update preNodeList
17:   end if
18:   if isInside(N) then
19:      lastInside = N
20:      j = i
21:   end if
22: end for
```

3.3.3 Possible Problems with PT+SLID approach

- The problem here is that we have to generate a parse tree, or an implicit parse tree, carrying implementation difficulties and performance problems (memory consumption with large corpus).
- The problem is that the final grammar has many similar rules, so we should find a way to factorize the rules to avoid this problem.

  For example:

  $10^1 \rightarrow t - g - 8_{+1} - 9^1 - a - g - a$

  $10^2 \rightarrow t - g - 8_{+2} - 9^1 - a - g - a$

  could be written as:

  $10^1 \rightarrow t - g - 8_{+1}, 8_{+2} - 9^1 - a - g - a$

  or something similar.

This carries again implementation issues.

4. RESTRICTING SET OF CONTEXT CANDIDATES

There are many ways to choose different contexts in our sequence. To test all of them in each iteration is kind of expensive. Because of that, we should find an heuristic, that gives us good results by searching in a smaller subset of all possible contexts.

4.1 Crossing Maximal Repeats

When replacing simple exact words, a good way is choosing them within the sequence Maximal Repeats (MR) as stated in [1], because it is fast using a suffix tree data structure.

We start thinking in a similar approach, by crossing all MR and obtaining contexts u-v from them ($u = MR_i$, $v = MR_j$).

This approach could only be used in small datasets, because it's highly demandant in the resources it uses, because we should cross them all, with a complexity of $\emptyset(|MR|^2)$ that gets easily really big.

4.2 Contexts as subsequence of MR

The idea is to obtain the contexts u-v from the very same maximal repeat. As replacing words from MR gave great results in [1], if we get the u-v from them the intution says that we should get similar results. The computation time and resources used improve, because we are just crossing prefix and suffix of each MR and this is less costly than crossing all MR. The counterpart is that we are cutting some possibilities.

4.3 Contexts from best MR

Here we first select the best MR as the SLG problem in [1] and we obtain from it the possible contexts. It's a really fast approach because we are using only one MR, but we don't know if the best MR word will guide us to best contexts replacements.

4.4 Limitations on U, V sizes

If we are using Maximal Repeats, it is usually common in the experimental results, that the sizes of u and v based on them don't go over a certain small limit s, so we can fix the search to maximal repeats of size s increasing the performance and obtaining similar results.

We think, this happens because the possibility of pairing two small contexts is greater than pairing big contexts.

5. RESULTS

We run three experiments:

1. Fragment DNA Corpus: a fragment from the canterbury Ecoli corpus
2. XML: a simple xml file
3. Small Ecoli: a tiny subsequence from the Ecoli corpus.

Bigger experiments were unaffordable, because of the resources consumption. With better heuristics this could be done.

For comparition purpose, we have here the results obtained with the same dataset with the classical SLG approach from [1].

Fragment DNA Corpus (Initial Size: 4757)
Final Size: **1977**

XML (Initial Size: 3886)
Final Size: **1082**

Small Ecoli Sequence (Initial Size: 386)
Final Size: 268

5.1 Results 1

- Contexts obtained from Crossing all MR
- k and l < 3
- score with encoding penalty 0

5.1.1 Fragment DNA Corpus (Initial Size: 4757)

GELD:
Encoding Size: 2517
Final Size: 1670
Grammar + Encoding: 4187
Words Chosen: 41(60.2941 per cent)
Contexts Chosen: 27(39.7059 per cent)
Size compared to SLG: +111%

SLID:
Encoding Size: 0
Final Size: 2383
Grammar + Encoding: **2383**
Words Chosen: 64(86.4865 per cent)
Contexts Chosen: 10(13.5135 per cent)
Size compared to SLG: +20,5%

5.1.2 XML (Initial Size: 3886)

GELD:
Encoding Size: 655
Final Size: 1025
Grammar + Encoding: 1680
Words Chosen: 31(62 per cent)
Contexts Chosen: 19(38 per cent)
Size compared to SLG: +55%

SLID:
Encoding Size: 0
Final Size: 1084
Grammar + Encoding: **1084**
Words Chosen: 31(60.7843 per cent)
Contexts Chosen: 20(39.2157 per cent)
Size compared to SLG: +0,1%

5.1.3 Small Ecoli Sequence (Initial Size: 386)

GELD
Encoding Size: 122

Final Size: 231
Grammar + Encoding: 353
Words Chosen: 12(75 per cent)
Contexts Chosen: 4(25 per cent)
Size compared to SLG: +31,7%

SLID
Encoding Size: 0
Final Size: 249
Grammar + Encoding: **249**
Words Chosen: 13(81.25 per cent)
Contexts Chosen: 3(18.75 per cent)
Size compared to SLG: -7,1%

5.2 Results 2

- Contexts obtained from Crossing all MR.
- k and l < 3
- score with encoding penalty $log(|disambiguationlist|)$

5.2.1 Fragment DNA Corpus (Initial Size: 4757)

GELD:
Encoding Size: 2354
Final Size: 1634
Grammar + Encoding: 3988
Words Chosen: 56(77.7778 per cent)
Contexts Chosen: 16(22.2222 per cent)
Size compared to SLG: +101%

SLID:
Encoding Size: 0
Final Size: 2358
Grammar + Encoding: **2358**
Words Chosen: 77(91.6667 per cent)
Contexts Chosen: 7(8.33333 per cent)
Size compared to SLG: +19%

5.2.2 XML (Initial Size: 3886)

GELD
Encoding Size: 594
Final Size: 1009
Grammar + Encoding: 1603
Words Chosen: 42(73.6842 per cent)
Contexts Chosen: 15(26.3158 per cent)
Size compared to SLG: +48%

SLID
Encoding Size: 0
Final Size: 1064
Grammar + Encoding: **1064**
Words Chosen: 38(69.0909 per cent)
Contexts Chosen: 17(30.9091 per cent)
Size compared to SLG: -1,7%

5.2.3 Small Ecoli Sequence (Initial Size: 386)

GELD
Encoding Size: 98

Final Size: 225
Grammar + Encoding: 323
Words Chosen: 16(88.8889 per cent)
Contexts Chosen: 2(11.1111 per cent)
Size compared to SLG: +20%

SLID
Encoding Size: 0
Final Size: 247
Grammar + Encoding: **247**
Words Chosen: 15(88.2353 per cent)
Contexts Chosen: 2(11.7647 per cent)
Size compared to SLG: -7,9%

5.3 Results 3

- Contexts obtained from Crossing all MR.
- No restrictions in k,l sizes.
- score with encoding penalty 0

5.3.1 Fragment DNA Corpus (Initial Size: 4757)

GELD Encoding Size: 2504
Final Size: 1664
Grammar + Encoding: 4168
Words Chosen: 52(71.2329 per cent)
Contexts Chosen: 21(28.7671 per cent)
Size compared to SLG: +110%

SLID: Encoding Size: 0
Final Size: 2388
Grammar + Encoding: **2388**
Words Chosen: 71(87.6543 per cent)
Contexts Chosen: 10(12.3457 per cent)
Size compared to SLG: +20%

5.3.2 XML (Initial Size: 3886)

GELD:
Encoding Size: 581
Final Size: 987
Grammar + Encoding: 1568
Words Chosen: 59(92.1875 per cent)
Contexts Chosen: 5(7.8125 per cent)
Size compared to SLG: +44,9%

SLID:
Encoding Size: 0
Final Size: 1041
Grammar + Encoding: **1041**
Words Chosen: 46(79.3103 per cent)
Contexts Chosen: 12(20.6897 per cent)
Size compared to SLG: -3,8%

5.3.3 Small Ecoli Sequence (Initial Size: 386)

GELD:
Encoding Size: 122
Final Size: 231

Grammar + Encoding: 353
Words Chosen: 15(78.9474 per cent)
Contexts Chosen: 4(21.0526 per cent)
Size compared to SLG: +31,7%

SLID:
Encoding Size: 0
Final Size: 249
Grammar + Encoding: **249**
Words Chosen: 14(82.3529 per cent)
Contexts Chosen: 3(17.6471 per cent)
Size compared to SLG: -7,1%

6. CONCLUSIONS

- GELD Encoding penalty is huge, so only repeats instead are usually chosen, like in the SLG approach. The most important thing to reduce the final size is reducing the encoding penalty and taking it into account when calculating the heuristics score. Ordering the disambiguation list and applying some kind of codification like Hoffman coding could be a good way to start.

- All the results showed us that SLID is better than GELD in every corpus we test. This is because we are not compressing the disambiguation list at all.
  If we manage to get a good compression, we could achieve in some examples better results with GELD

- Results don't show great differences when choosing bounded k,l ($+-3\%$) and performance increase a lot. So this seems to be a good heuristic.

- As more contexts we choose, more complex become our grammar and the size is bigger. This made us think that for reducing the size it's better to choose just repeats instead of yields, like in SLG.
  It's difficult to find an example where the opposite states, because it's very common that we can replace U, V and the insides with words in 3 or more iterations. That's why in all experiments, but the last one with the tiny Ecoli subsequence, the classical approach gave us better results.

- PT+SLID should has great behaviour if the Parse Tree has many repeated branches, otherwise it's really bad. We could reduce the final size drastically with a good factorization in the grammar rules, because a lot of the final rules are similar (see 3.3.3). Due to implementation issues we couldn't show results with it.

- Crossing all Maximal Repeats to obtain contexts takes long time with small corpus, and its resource consuming make it impossible with big corpus. A faster and lighter heuristic should be used instead.

- SLID is great with small sequences, but when the size increase the performance decrease as we are adding new symbols in each iteration.

7. WHAT'S NEXT?

There are many roads to follow from here:

- Compression: going a step further, analysing entropy aspects, different types of encoding, etc.
- Focusing on the heuristic to choose better and faster constituents, to avoid limitations in the size of our dataset.
- More approaches besides Greedy. For example, Minimal Grammar Parsing adapted to contexts, Graphs Algorithms like the ones showed in the heuristics section, etc.
- Theorical formulas to complete the practical experiments. Why it is better to choose words? Why bounding the k,l size gave us almost the same results?
- Finding interesting patterns and structures in the final grammar representation of the sequences (for example, interesting DNA parts).
- Improve one of the approaches (for example, focusing in small sequences with SLID). This could be intereseting for specific types of dna, could be plasmids.

BIBLIOGRAPHY

[1] Matthias Gallé PhD thesis *Searching for Compact Hierarchichal Structures in DNA by means of the Smallest Grammar Problem.* Université de Rennes, Bretagne Atlantique, 2011.

[2] Coste, Garet, Nicolas *Local Substitutability for Sequence Generalization.* JMLR: Workshop and Conference Proceedings 21:97–111. INRIA, Centre Inria Rennes - Bretagne Atlantique, 2012.

[3] Brodal1, Lyngsø, Ostlin and Pedersen *Solving the String Statistics Problem in Time* $\emptyset(nlogn)$ University of Aarhus, 2002.

[4] Coste, Garet, Nicolas *Learning Context Free Grammars on Protein Sequences by Local Substitutability.* INRIA, Centre Inria Rennes - Bretagne Atlantique, 2013

[5] Benz, Kötzing *An Effective Heuristic for the Smallest Grammar Problem.* GECCO '13 P.487-494, 2013.