Structured Motifs Identification in DNA Sequences

Yuridia P. Mejía [1], **Ivan Olmos** [1], **Jesús A. González** [2]
[1] Facultad de Ciencias de la Computación,
Benemérita Universidad Autónoma de Puebla,
14 sur y Av. San Claudio, Ciudad Universitaria,
Puebla, México
{yuripmt,ivanoprkl}@gmail.com
[2] Instituto Nacional de Astrofísica, Óptica y Electrónica,
Luis Enrique Erro No. 1, Sta. María Tonantzintla, Puebla, México
jagonzalez@inaoep.mx

Abstract

In this paper, we present an algorithm that finds structured motifs in a DNA sequence. A structured motif consists of a central motif and one or two satellite motifs, which may be located to the left and / or right of the central motif. The search of the motifs is performed in two stages: first, the central motifs are located through an exact set matching process, which is implemented by a deterministic finite automaton; in the second stage, the satellite motifs are located from the position of the central motifs at a distance defined as input. This last phase requires two steps: first, a matrix is calculated through a dynamic programming technique using the Levenshtein algorithm. After this, we identify the satellite motifs using the matrix. Based on our results, our method is fast at the moment to search for central patterns (in linear time), and the second phase is most expensive because it is necesary to identify all the possible alignments and after that, perform the alignment with their respective satellite.

Introduction

Searching for interesting regions (patterns) in DNA databases is a frequent problem in biology, allowing to discover biological functions of organisms (Liang 2003). The DNA motif search problem consists of finding all instances of a string (pattern) in a text (a DNA database). The DNA database consists of the nucleotide alphabet: A for adenine, C for cytosine, G for guanine, and T for thymine. The output is a list of positions where the pattern appears. In general, a pattern is a string that is not only defined by the letters A, C, G, and T. This happens because in the biological area it is necessary to represent ambiguities associated with certain changes in the organism that is being studied (mutations, hereditary processes, and so on) (Schmollinger 2004). Because of this, a pattern is generally defined under the IUPAC nomenclature (*International Union of Pure and Applied Chemistry*), where each character represents two or more letters of the nucleotide alphabet.

Sometimes the patterns to search for are simple (strings), but there are cases where we need to search for patterns with a complex structure (structured motifs), i.e. looking for patterns composed of a main motif (central pattern, or P_C for short) and one or two motifs located at some distance to the left or right of the central pattern (P_L and P_R respectively). In the biological area, the patterns associated with a central pattern are called satellites (Crochemore and Sagot 2000).

There is an important difference in the way that we search for a P_C and how we search for a P_L or a P_R. The P_C is defined according to the IUPAC alphabet, P_C can be transformed into a set of strings to search for in a DNA database. This is known as the exact set matching problem (Gusfield 1994). One of the most representative approaches to solve the exact set matching problem is the Aho-Corasick algorithm (Gusfield 1994), but new approaches with performance improvements, such as the MFA algorithm, (Pérez et al. 2009) have been developed. On the other hand, P_L and P_R are also based on the IUPAC alphabet (it is possible to derive a set of strings from P_L or P_R). However, at the moment to search for these satellites, it is possible to apply operators such as insertions, deletions, and substitutions. This is done with the aim to find an alignment of the satellite with a segment of the DNA database (based on our domain experts, Candelario Vazquez and Patricia Sanchez, these operations are allowed because satellites may have a higher level of ambiguity) but limited by a threshold.

In this paper we propose an algorithm that finds all structured motifs of the form P_C - P_R in a DNA database, which are separated at most by a distance of d_1. Our approach is divided in two main phases: the first one searches for the central patterns using the MFA algorithm, and the second one uses a dynamic programming technique with the Levenshtein function, to find such structured motifs.

Notation

With the aim to define the concepts of the structured motifs problem, we introduce the following notation.

An alphabet, denoted by Σ, is a non empty finite set of letters. A string of the alphabet Σ is a finite subset of elements of Σ, where a letter is placed one after the other. The empty string, denoted by ϵ, is the sequence of zero letters. The length of a string X, is defined as the number of letters in X, denoted as $|X|$. With $X[i]$, we denote the i^{th} letter in X, where $i = 1, 2, ..., |X|$. The concatenation of two strings X and Y is the string composed by the letters of X followed by the letters of Y, denoted as XY.

We define Σ as the union of two sets: Σ^B and Σ^E. The base alphabet, denoted as Σ^B, is defined as an alphabet Σ^B

= {A, C, G, T}. The extended alphabet, denoted as Σ^E, is defined as an alphabet Σ^E = {R, Y, K, M, S, W, B, D, M, V, N}, where R = {G, A}, Y = {T, C}, K = {G, T}, M = {A, C}, S = {G, C}, W = {A, T}, B = {G, T, C}, D = {G, A, T}, H = {A, C, T}, V = {G, C, A}, N = {A, G, C, T}. The extended alphabet is guided by the IUPAC nomenclature and will be used to represent ambiguities in patterns.

A pattern P of size m, is defined as a string $P = P[1]\ P[2]\ ...\ P[m]$, where for every $P[i] \in P$: $P[i] \in \Sigma$. A DNA database is defined as a string $\zeta = \zeta[1]\ \zeta[2]\ ...\ \zeta[n]$, such that for every $\zeta[i] \in \zeta$: $\zeta[i] \in \Sigma^B$.

Let X be a string of the form $X = AWB$, such that A and B are two strings separated by a string W, the distance between A and B is defined as $d_{(A,B)}$, where $d_{(A,B)} = |W|$.

An association (*matching*) between two strings A and B, is the process where each character of A is associated with a character of B. In this paper, we distinguish three types of string matches: exact matching, exact set matching, and inexact matching. For simplicity, with the aim to define these concepts, we will use the strings X, Y, and Z, where X and Z are defined in Σ^B and Y is defined in $\Sigma^B \cup \Sigma^E$.

1. *Exact Matching between X and Z*: It is the process where each of the characters of X is equal to its corresponding character on Z, this matching is also known as *equality between strings* and is denoted by $X = Z$. Formally, $X = Z$ if for each $i = 1, 2, ..., m$, $X[i] = Z[i]$. For example, if $X = ACG$ and $Z = ACG$, then $X = Z$.
2. *Exact Set Matching between X and Y*: This matching is established between X and Y if there is a string S generated from Y, where $S = X$. The string S is derived by replacing the Σ^E characters in Y by characters in Σ^B. Formally, an exact set matching, denoted by $X \sim Y$, is present if for each $i = 1, 2...m$: $X[i] = Y[i]$ if $Y[i] \in \Sigma^B$ or $X[i] \in Y[i]$ if $Y[i] \in \Sigma^E$. For example, if $X = ACG$ and $Y = AMG$, where $M = \{A, C\}$, by replacing the characters in Σ^E of Y by characters of Σ^B we generate two strings AAG and ACG. Since we can generate a string $S = ACG$, where $X = S$, then $X \sim Y$.
3. *Inexact Matching between Z and X*: An inexact matching between Z and X, denoted by $Z \approx X$, is present if it is possible to generate a string Z' from Z through a series of substitutions, insertions and / or deletions, so that $Z' = X$ (Navarro 2001).
   - A *substitution* on $Z = \alpha\gamma\beta$, is the operation where a character γ is exchanged by another character γ', generating $Z' = \alpha\gamma'\beta$. We apply a substitution when we want to match two strings X and Z, where $|X| = |Z|$ but they differ in one or more characters. If $\gamma \in \Sigma^B$, then γ is replaced by $\gamma' \in \Sigma^B$ such that $\gamma \neq \gamma'$. On the other hand, if $\gamma \in \Sigma^E$, then γ is replaced by $\gamma' \in \Sigma^B$ - γ. For example, let the string $Z = TGTCA$, such that $\alpha = TG$, $\gamma = T$ and $\beta = CA$, we want to make an exact matching between Z and X, where $X = TGGCA$, then we apply the substitution operation γ for γ' to generate the string $Z' = TGGCA$, where $\alpha = TG$, $\gamma' = G$ and $\beta = CA$, so that $Z' = X$.
   - An *insertion* on $Z = \alpha\beta$, is the operation where we add a character γ to Z, generating $Z' = \alpha\gamma\beta$. We apply this operation when we want to match two strings Z and X, where $|Z| < |X|$. For example, if $Z = TAG$ and $X = TGAG$, where $Z = \alpha\beta$, then $\alpha = T$ and $\beta = AG$. If we apply an insertion in Z, we insert γ in Z such that $\gamma = G$, we generate $Z' = TGAG$ such that $Z' = X$.
   - A *deletion* on $Z = \alpha\gamma\beta$, is the operation where we remove a character γ to the Z string, generating $Z' = \alpha\beta$. We apply a deletion operation when we want to match two strings Z and X such that $|Z| > |X|$. For example, if $X = TCAG$ and $Z = TCAAG$, where $\alpha = TC$, $\gamma = A$, and $\beta = AG$. When we apply a deletion in Z, we eliminate γ from Z and generate $Z' = TCAG$, such that $Z' = X$.

Generally, the number of operations that can be applied to a string is limited by a threshold defined by the user. In this paper, we denote by $\sigma_{(X,Y)}$ the maximum percentage of operations that can be applied to generate Y from X.

Finally, we introduce the concept of structured motif. This term is used to refer to a motif formed by three motifs separated by a distance. Formally, a structured motif is a string X, where $X = P_L\ W\ P_C\ Z\ P_R$, $|W| = d_1$ and $|Z| = d_2$. Starting from the P_C (central pattern), we have a P_R (right satellite) located to the right of P_C, where $d_{(P_C,P_R)} \leq d_2$; Likewise, there is a P_L (left satellite), which is located to the left of P_C, such that $d_{(P_C,P_L)} \leq d_1$.

For example: Let $X = AGTGACGACTCA$, where we need to search for the structured motif $P_C = ACG$, $P_L = TG$, $P_R = TC$, where $d_1 = 3$ and $d_2 = 4$. First, we find that P_C is located at position 5 in X. From this position we search for the left and right satellites. In this example, we found a satellite P_L at position 3 with $d_{(P_L,P_C)} = 0$ and a satellite P_R at position 10 where $d_{(P_C,P_R)} = 2$.

It is important to mention that we also consider as structured motifs those composed by the central pattern and a right satellite (P_C - P_R), the central pattern and a left satellite (P_L - P_C) or those containing both satellites (P_L - P_C - P_R). For simplicity, this paper describes the solution to locate structured motifs of the form P_L - P_C, where $d_{(P_L,P_C)} \leq d_1$ and a threshold σ.

Proposal

The methodology proposed for searching structured motifs of the form P_L - P_C in a DNA sequence ζ, where $d_{(P_C,P_L)} \leq d_1$, and a threshold σ, consists of two phases:

- We first implemented a candidates generation phase, where all substrings S of a DNA sequence ζ are located, such that $S \sim P_C$. As the output of this phase, we get a set $C = \{S : S$ is a substring of ζ and $S \sim P_C\}$. At this stage, we proposed an automaton that searches for all instances of P_C in ζ.
- Second, we implemented a candidates evaluation phase, where each string S from C is procesed with the aim to search for the substrings S' of ζ, where $S' \approx P_L$ and $d_{(P_L,S')} \leq d_1$. As the output of this phase, we obtain a list of positions of central patterns where it is possible to generate a left satellite after we apply a finite sequence of insertions, deletions, and sustitutions operations (restricted by σ). Finally, due to the fact that the last phase does

not report the explicit satellites, we use a dynamic programming technique with the aim to identify the actual satellites found in ζ.

Candidates Generation Phase

As we mentioned before, this phase is implemented through an automaton called MFA (Pérez et al. 2009), which finds the longest suffix of a pattern that is associated with the prefix of previously recognized patterns. With the aim to explain the MFA algorithm in this section, we assume that our input parameters are: $P_C = AMS$, $P_L = GK$, $d_1 = 4$ and $\zeta = ATGGACAACC$ (a fragment of a DNA sequence).

The MFA algorithm consists of two phases: a preprocessing and a searching phase. In turn, the preprocessing consists of three phases: the expansion of P_C, the creation of the states matrix *matQ*, and the construction of the transitions matrix of the automaton δ.

The expansion of P_C is performed by substituting from right to left, each of the characters in Σ^E for characters in Σ^B (with respect to the IUPAC nomenclature). The output of this stage is a set of strings *seqP*. For example, if $P_C = AMS$, where $M = \{A, C\}$ and $S = \{C, G\}$, replacing M and S by their characters in Σ^B we obtain $seqP = \{AAC, ACC, AAG, ACG\}$.

In the second step we create the array of states *matQ*, which is sequentially filled from top to bottom (rows) and from left to right (columns), so that the final states are stored in the last column. The number of rows is controlled by the product of the cardinalities of each element of P_C, and $|P_C|$ is the number of columns. In Fig. 1 we show how to fill *matQ*.

matQ:

	A	M	S
1	1	2	4
2	0	3	5
3	0	0	6
4	0	0	7

For column A:
|A| = 1, a row will be filled sequentially
For column M:
|M| = 2, be filled to the line |A| * |M| = 1 * 2 = 2
For column S:
|S| = 2, be filled to the line |A| * |M| * |S|= 1 * 2 * 2 = 4

Figure 1: *Filling of matQ*

In the last stage, the transition matrix δ is created and filled row by row, and their values depend on the longest suffix of a string which is a prefix of an element of *seqP*. As an example, consider vertex 2 with label "AA", which is obtained through the concatenation of the letters from the root to vertex 2. The transitions from this vertex are created by concatenating each character of Σ^B to the vertex label, generating the strings *AAA*, *AAC*, *AAG*, and *AAT*. After that, we search for the longest suffix of each string that is a prefix of an element in *seqP*. This operation is performed with a transition function proposed for the MFA algorithm (Pérez et al. 2009). For this example, the strings that match with this condition are *AAC* and *AAG*, generating two transitions (one with *C* and a second with *G*).

Each of these transitions defines if the cell $\delta[i, j]$ needs to be updated, where i represents the vertex that is being analyzed and j is the j^{th} element of Σ^B that is used in a transition. In our example, $\delta[2, 2]$ and $\delta[2, 3]$ are updated (transtions that are not being used are filled with 0). The

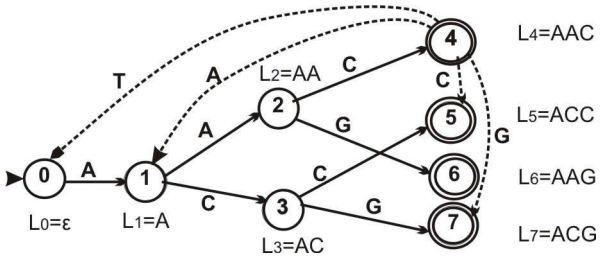

Figure 2: *Generation of prefixes to fill δ*

value assigned to $\delta[i, j]$ comes from $matQ[k, l]$, where k is the k^{th} element of *seqP* where we locate the prefix of vertex i, and l is the cardinality of this prefix. According with our example, $\delta[2, 2] = matQ[1, 3]$, because $k = 1$ (the first element of *seqP* has the prefix *AAC*) and $l = 3$ ($|AAC| = 3$). In this example, $\delta[2, 2] = 4$, generating a transition from vertex 2 to vertex 4 with the letter *C* (in the MFA algorithm, *matQ* is indexed starting in 1, because these indices are associated to the i^{st} element of *seqP*, starting from 1; unlike rows and columns in δ, where for simplicity the rows start from 0 and the columns start from 1). Resuming to our example, matrix δ of Fig. 3 is generated after we perform the preprocesing phase.

δ:

	1	2	3	4
	A	C	G	T
0	1	0	0	0
1	2	3	0	0
2	2	4	6	0
3	1	5	7	0
4	1	5	7	0
5	1	0	0	0
6	1	0	0	0
7	1	0	0	0

Figure 3: *Transition matrix of the automaton*

Once we built the automaton with the MFA algorithm, the pattern search process in a DNA database ζ is performed in the same way as we do with a traditional automaton, recording in a matrix *PCS* the found patterns (first column), and its position in ζ. For example, if $\zeta = ATGGACAACC$ and δ is the matrix of Fig. 3, we found the patterns *AAC* and *ACC* at positions 9 and 10 respectively (see Fig. 4).

PCS:

AAC	9
ACC	10

	A	T	G	G	A	C	A	A	C	C
0	1	0	0	0	1	3	1	2	(4)	(5)

Figure 4: *Searching Phase*

Candidates Evaluation Phase

This phase consists of evaluating each element S of PCS and locating substrings S' in ζ where $S' \approx P_L$ and $d_{(S',S)} \leq d_1$. This phase is divided in two stages: we first create a matrix D based on the Levenshtein algorithm (Bofivoj 1995) and after that, we verify D with the aim to locate satellites with an exact match or, the existence of an inexact match with P_L based on a finite set of insertions, deletions, and sustitutions, without exceeding a threshold σ.

Levenshtein Algorithm Before we build the dynamic tables derived from the Levenshtein algorithm, as first step we have to replace all the extended characters of P_L to base characters, just as the expansion process of the candidates generation phase. Let $\mathbb{S}$ be the output set of this expansion. For example, if $P_L = GK$ and $K = \{G, T\}$, then $\mathbb{S} = \{GG, GT\}$.

Once we create $\mathbb{S}$, we need to locate all the instances of each element of $\mathbb{S}$ in ζ, based on the initial position of P_C and the distance value d_1. The segment of ζ where we search for the satellites is defined as a search window. First, the limits of the search window are computed as follows: $W_{start} = Pos_{start} - d_1 - |P_L|$ and $W_{end} = Pos_{start} - 1$, where $Pos_{start} = Pos_{end}$ - $|P_C|$ + 1 and Pos_{end} is the final position stored in PCS. Resuming to our example, where $\zeta = ATGGACAACC$, $\mathbb{S} = \{GG, GT\}$, and $d_1 = 4$, we show how to compute W_{start} and W_{end} for each element in $\mathbb{S}$, and in Fig. 5 we show the search window for $P_C = AAC$:

- AAC: $W_{start} = Pos_{start} - d_1 - |GK| = 7 - 4 - 2 = 1$, $W_{end} = Pos_{start} - 1 = 7 - 1 = 6$
- ACC: $W_{start} = Pos_{start} - d_1 - |GK| = 8 - 4 - 2 = 2$, $W_{end} = Pos_{start} - 1 = 8 - 1 = 7$

1	2	3	4	5	6	7	8	9	10
Wstart					Wend	Posstart		Posend	
A	T	G	G	A	C	A	A	C	C
G	G								

Figure 5: *Initial and Final Positions of the Search Window*

After we identify the search window W, we generate a matrix $D_{(m+1)(n+1)}$ (where $m = |S'|$, such that $S' \in \mathbb{S}$ and $n = |W|$). This matrix stores the minimum number of insertions, deletions, and substitutions operations that we need to apply to a substring of the current window W such that it matches with S' in an inexact way. This matrix is filled using the Levenshtein algorithm (Bofivoj 1995), which is guided by the equations shown in Fig. 6.

$$D[0,j] = 0, \text{ where } 0 \leq j \leq n$$
$$D[i, 0] = i, \text{ where } 0 \leq i \leq m$$
$$D[i,j] = \text{Min}(D[i-1,j]+1, D[i,j-1]+1, D[i-1,j-1]+cost)$$

Figure 6: *Levenshtein Algorithm*

In this expresion, $cost = 0$ if $\mathbb{S}_{k,i} = W[j]$, otherwise $cost = 1$, where $\mathbb{S}_{k,i}$ is the i^{th} character of the k^{th} element in $\mathbb{S}$, and $W[j]$ is the j^{th} character of the search window that is being analyzed.

It is important to mention that we need to fill a matrix D for each element in $\mathbb{S}$ associated to each element in PCS.

For example, in Fig. 7 we show the matrices D associated to the satellite GK, where the central patterns are AAC and ACC.

Dynamics Tables for AAC

D_1		T	G	G	A	C
	0	0	0	0	0	0
G	1	1	0	0	1	1
G	2	2	1	0	1	2

D_3		T	G	G	A	C
	0	0	0	0	0	0
G	1	1	0	0	1	1
T	2	1	2	1	1	2

Dynamics Tables for ACC

D_2		G	G	A	C	A
	0	0	0	0	0	0
G	1	0	0	1	1	0
G	2	1	0	1	2	2

D_4		G	G	A	C	A
	0	0	0	0	0	0
G	1	0	0	1	1	1
T	2	2	1	1	2	2

Figure 7: *Matrix D generated for $P_C = AMS$ and $P_L = GK$*

Once D is generated, we store in the last row of D the minimum number of operations needed to transform a substring of the search window (subwindow) into the satellite P_L. For example, if we consider table D_1 of Fig. 7, the value of position $d_1[2,4] = 1$ means that we need a single operation to transform the search subwindow GGA into the satellite GG. In order to know which operation (insertion, deletion, or sustitution) is required to achieve this transformation, we need to trace a path from position $D[2,4]$ up to row zero, based on the operations that were applied in the subwindow. This process is explained in the following section.

Location of Satellites As mentioned in the previous section, the idea of this phase is to trace a path from a cell in the last row to a cell in the first row. Each cell visited in this path represents an operation that is applied to a character of the substring of the current search window.

The cells that are selected from the last row are those in which values do not exceed the threshold. Based on these cells, we build a path, step by step. In each step, we move between cells with three possible directions: left, top, or upper-left diagonal. The direction that we follow depends on where the minimum value between the three possible directions is located, which is computed based on the third restriction of Fig. 6. If two or more directions have the minimum value, then each of these directions generates a branch, which is expanded independently. This process starts with the last characters of the satellite to search for and the subwindow to search in. For each step, we moved from character i^{th} to the character $(i-1)^{th}$ in the satellite, but in the subwindow it depends on the operation used in the transition (insertions, deletions, and sustitutions are applied only to the subwindow). This idea is represented in Fig. 8.

For example, consider the satellite $S = ACGTAC$ (S is an element of $\mathbb{S}$) and a segment of a search window $W = ...ACTGAC$, where the threshold $\sigma = 2$. After we

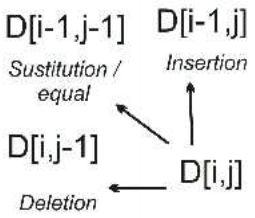

Figure 8: *Associate operations by position*

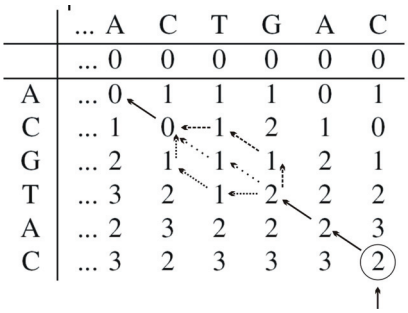

Figure 9: *Path of Operations*

build D with the Levenshtein algorithm, there are two cells in the last row where the threshold is satisfied ($D[6,2] = 2$ and $D[6,6] = 2$, as we can see in Figure 9). If we build the path from $D[6,6]$, then we first test the minimum value between $\{D[5,6]+1, D[6,5]+1, D[5,5]+0\}$ (because of $S[6] = W[6] = C$, then moving to the cell D[5,5] would not increase the cost). As result, we moved to the cell $D[5,5]$. In the same way, from $D[5,5]$ we moved to the cell $D[4,4]$ without changes in w (where w is a string that indicates the changes that a substring of the subwindow suffered to become a satellite with its respective operations) . However, from $D[4,4]$ the path is divided in two branches, because $S[4] \neq W[4]$ and all the possible directions have the same value ($\{D[3,4]+1, D[4,3]+1, D[3,3]+1\}$). As consequence of this, if we take the direction $D[4,3]$, then $w[4]$ is "deleted" in the alignment between S and W. If we select the path through $D[3,4]$, then we insert a "gap" in $w[3]$, moving the letters to the left one position, resulting in $w = AC\_TAC$ (this corresponds to an alignment with one error between S and W). Finally, if we select the path through $D[3,3]$, there is a sustitution of T by G. At the end of this process, we found three possible alignments of W with respect to S: AC_TAC, ACG_AC and $ACGTAC$.

This process is shown in Algorithm 1. In line 4 of this algorithm we compute the minimum value between the possible cells where the path could be expanded from cell $D[i,j]$. In lines 5, 9, and 13 we test the values of the cells, with the aim to identify if there is more than one cell with the minimum value. If so, the corresponding paths are expanded. It is important to mention that this algorithm needs to be invoked once per each cell $D[|S|,j]$ in the last row of D that satisfies the threshold, using the call: AlignmentW(D,S,W,|S|,j).

In the following section we present a set of experiments where we use our approach.

Algorithm 1 AlignmentW

Input: D (Levenshtein matrix), S (satellite), w a string with the current alignment, i, j integers that define the expand position
Output: alignments w

1: **if** i = 0 or $D[i,j] = 0$ **then**
2: print w and return
3: **end if**
4: $Min \leftarrow \min\{ D[i-1,j]+1, D[i,j-1]+1, D[i-1,j-1]+cost \}$ is found (cost $\leftarrow$ 0 if $S[i] = w[j]$ or cost $\leftarrow$ 1 otherwise)
5: **if** $D[i-1,j] = Min$ **then**
6: $w' \leftarrow$ insert a gap at position j in w
7: AlignmentW($D, S, w', i-1, j$)
8: **end if**
9: **if** $D[i,j-1] = Min$ **then**
10: $w' \leftarrow$ delete the character at position j in w
11: AlignmentW($D, S, w', i, j-1$)
12: **end if**
13: **if** $D[i-1,j-1] = Min$ **then**
14: $w' \leftarrow$ string w where $S[i] = w[j]$
15: AlignmentW($D, S, w', i-1, j-1$)
16: **end if**

Results

In this phase we show the results of the implementation of the candidates generation phase through the MFA automata and some results about how the implementation of the Levenshtein algorithm works, but we are still working in the analysis of the results with the biological experts.

All the experiments were done in an Intel Core Duo 1.8 Ghz with 4 GB of RAM and the Windows Vista operating system. Both algorithms were implemented in C + +.

We evaluated the performance of the MFA algorithm through experiments with four real databases: Candida albicans (14,361,129 base pairs), Ustilago maydis (19,683,350 base pairs), Aspergillus nidulans (30,068,514 base pairs) and Neurospora crassa (39,225,835 base pairs). These databases correspond to the text used to search for patterns, denoted by ζ. All databases were downloaded for free from the Gene Research Center, from the fungi section (http://www.broad.mit.edu/).

In Fig. 10 we show our run-time in the preprocessing phase and the run-time in the searching phase in Fig. 11.

For simplicity and without losing generality, in our experiments we are looking for patterns of length 32, because this is the maximum length of patterns that our domain experts need to locate. In the x axis, we indicate the number of patterns generated by the combination of extended letters. The y axis indicates the dimension of each of the databases. Finally, the z axis shows the execution time in seconds.

As shown in the graphs, in both, the preprocessing phase and the searching phase, the MFA automata does not change significantly neither in terms of the dimension of the set of

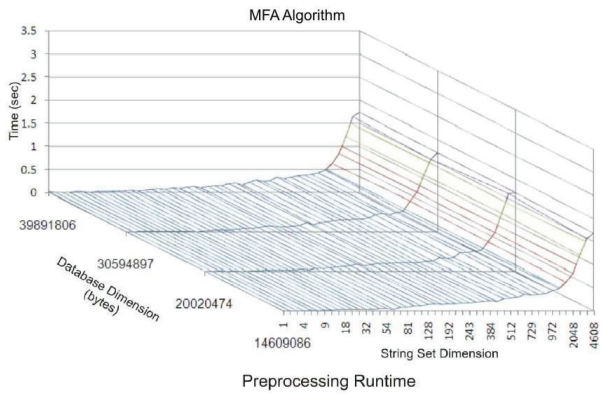

Figure 10: *Preprocessing RunTime of the MFA Algorithm*

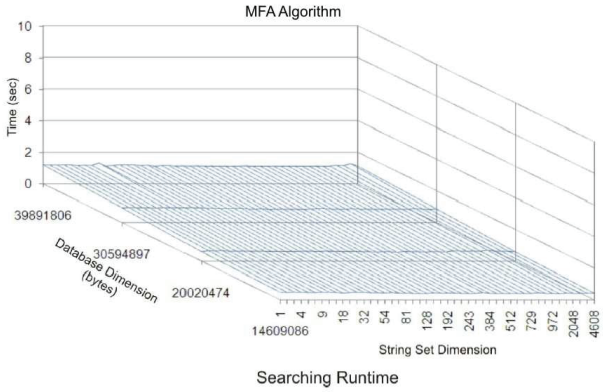

Figure 11: *Searching RunTime of the MFA Algorithm*

patterns that we locate, nor in the size of the database. This is very important because the databases can be very large.

In the phase of the Levenshtein algorithm we are still in the experimental phase, because up to now we have done just a few experiments with real data until the moment that we present this paper. However, some of the experiments that we made are with the database Candida Albicans and the patterns: $P_C = GATA$, $P_L = TGACTCA$, $P_R = TGACTCA$ with distances of 100, 200, 500 and 1000 nucleotides to the left and right of the central pattern and a threshold of 35% (which is 2 operations by satellite as maximum).

All the data for the experiments were provided by the biologists Patricia Sanchez and Candelario Vazquez, and this information has a real meaning in biology.

In figure 12 we show the results of the experiments, the columns that we present are: central pattern, satellites left and right , threshold error, distances left and right, number of satellites or alignments that we located and the total searching time in seconds.

As we can see, the number of alignments in each experiment increases significatively, this happens because calculating the Levenshtein matrix has a cuadratic cost and the alignments can be done in linear time, but both of them must be calculated every time we find a central pattern and the

Central Pattern	Left Satellite	Right Satellite	Threshold	# PCs	Left Distance	Right Distance	Alignments	Total (secs)
GATA	CTATCG	CGATATCT	35%	72232	100	100	1291128	405.322
					200	200	2568125	614.378
					500	500	6423502	1701.004
					1000	1000	12842312	4600.063

Figure 12: *Searching RunTime of the Levenshtein Algorithm*

alignments are done when we find a satellite. It is important to mention that if we increase the distance between the central pattern and the satellite we can find more alignments.

Conclusions and Future Work

The methodology presented in this work allows us to search for structured patterns in DNA sequences through a finite automata and a dynamic programming technique.

The candidate generation phase was performed based on the MFA automata, which has already been implemented. This algorithm is capable to find all the instances of the central pattern P_C in the DNA sequence, with an attractive performance.

We already finished the implementation of the candidate evaluation phase based on the Levenshtein algorithm. However, we have to make more experiments to analyze its performance, while the biologists are analyzing the results that we obtained with the aim to find a biological interpretation.

With our approach it will be possible to find structured motifs in DNA databases, providing a useful tool for biologists that need to find complex motifs.

References

Bofivoj, M. 1995. Approximate string matching by finite automata. *In Conf. on Analysis of Images and Patterns in number 970, LNCS* 342–349.

Crochemore, M., and Sagot, M. F. 2000. Motifs in sequences: Localization and extraction. 26–31.

Gusfield, D. 1994. *Algorithms on Strings, Trees, and Sequences*. Computer Science and Computational Biology. Cambridge University Press, 1st edition edition.

Liang, Mike P., B. D. L. A. R. B. 2003. Automated construction of structural motif for predictiong functional sites on protein structures. *Pacific Symposium on Biocomputing* 204 – 215.

Navarro, G. 2001. A guide tour to approximate string matching. *ACM Computing Surveys* 33(1):31–88.

Pérez, G.; Mejía, Y. P.; Olmos, I.; González, J. A.; Sánchez, P.; and Vázquez, C. 2009. An automaton for motifs recognition in dna sequences. *MICAI 2009 - LNAI 5845, Springer - Verlag* 556–565.

Schmollinger, M. e. 2004. Parseq: Searching motifs with structural and biochemical properties. *Bioinformatics* 20:1459 – 1461.