Structured Motifs Recognition in DNA sequences

Yuridia P. Mejía[1], Ivan Olmos[1], Jesus A. Gonzalez[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 is presented a methodology for structured motifs recognition (SMR) in DNA sequences. The SMR problem consists of finding all instances of a triple-pattern $P_L - P_C - P_R$ in a DNA sequence, where P_L, P_C and P_R are based on the IUPAC alphabet, and P_L and P_R are both separated from P_C by a distance no greater than "n" characters, which is provided as input. In this problem an inexact association between P_L, P_C, P_R and the DNA sequence is allowed, which is limited by an error based on the number of insertions, deletions, and substitutions operations. In this paper we propose a methodology for finding SMR patterns based in two stages: first, an automaton is used to search all P_C instances (where only substitutions are only allowed); second, a dynamic programming technique is proposed to find the P_L and P_R patterns (where substitutions, insertions and deletions, are allowed) based on the Levenshtein algorithm. This methodology is useful to biologists in real DNA patterns recognition tasks, where it is necessary to find DNA regions with a biological meaning.

1 Introduction

The DNA motif search problem consists of finding a pattern P (the motif to search for) from a text T (the DNA database created from the nucleotides alphabet: "A" for adenine, "C" for cytosine, "G" for guanine, and "T" for thymine) where we output the positions in T where P appears, the instances of P in T. In this problem, P is formed from an extended alphabet defined under the IUPAC (International Union of Pure and Applied Chemistry) rule where a character in this alphabet may represent more than one nucleotide, this means that P may represent a set of patterns to search for in T.

In Biology, the motif search problem is very important, because it allows us to search or discover biological segments from an organism that is being analyzed. In general, such motifs to search for are restricted to single motifs. However, there exist biological problems where it is necessary to find motifs that are formed by two or more single motifs (structured motifs, or SM for short), which are separated by a distance defined by the biologist. In this work, we consider that a SM is formed by two or three single motifs, represented by $P_L - P_C$ or $P_L - P_C - P_R$ respectively. In the biological

field, P_C is called a "central pattern or motif", and P_L, and P_R are called satellites, mini-satellites or micro-satellites, depending of their length. In both cases, and based on biological restrictions, the SM recognition problem always start search of the P_C motif, which has an important meaning for the organism that is being analyzed. Because of its importance, at the moment of searching P_C in a DNA database, it is only possible to allow minimal inexact associations, limited by a threshold in the number of substitutions of characters considering the IUPAC alphabet. The number of such substitutions (the threshold) is defined by the biologist. On the other hand, at the moment of searching a satellite, more flexibility is allowed. This means that a satellite could be found after a number of insertions, substitutions, and deletions in the DNA database has been applied (considering a larger threshold).

The problem addressed in this paper consists of finding motifs with the structure $P_L - P_C - P_R$, where P_L is located at the left of P_C and P_R at the right of P_C, and both of them are located at a distance (#bases) no longer than d_1 and d_2 respectively, which are provided as input parameters. As a solution to this problem, we propose a two phases methodology: at the beginning we build an automaton that searches all instances of P_C in a DNA database, reporting their positions; after that, we propose a dynamic programming strategy based on the Levenshtein algorithm, which search possible P_L and P_R satellites in the DNA database restricted to a distance d_1 and d_2 respectively.

The paper has the following structure: in section 2 we introduce important notation used in this work. In section 3 we present the problem to be addressed. Our proposal for finding the structured motifs is described in section 4. Finally, conclusions and future work is presented in section 5.

2 Definitions and notation

In this section, we introduce the notation on strings, sequences, and automata, that we use along this work.

An ***alphabet***, denoted by Σ is a finite nonempty set of letters. A ***string*** on alphabet Σ is a finite subset of elements on Σ, one letter after another. As an example, if $\Sigma = \{A, C, G, T\}$ then, examples of strings on this alphabet are *AAC*,*ACC*, and *AAG*. We denote by ε the zero letter sequence, called the ***empty string***. The ***length*** of a string x is defined as the number of letters on x, denoted by $|x|$. With $x[i]$, $i = 1, \ldots, |x|$, we denote the letter at position i on x. A ***substring*** γ of a string α is a string where: $|\gamma| \leq |\alpha|$, and $\gamma[i] = \alpha[i + k]$, $i = 1, \ldots, |\gamma|$ and $0 \leq k \leq |\alpha| - |\gamma|$.

The ***concatenation*** of two strings x and y is the string composed of the letters of x followed by the letters of y, denoted by xy. A string α is a ***prefix*** of the string x, denoted by $\alpha \sqsubset x$, if $x = \alpha y$. On the other hand, β is a ***suffix*** of x, denoted by $\beta \sqsupset x$, if $x = y\beta$. The empty string ε is both a prefix and a suffix of every string (not empty).

In this work we denote a ***finite automaton*** M as a 5-tuple $(Q, q_0, A, \Sigma, \delta)$ where: Q is a finite set of ***states***, $q_0 \in Q$ is the ***start state***, $A \subseteq Q$ is a distinguished set of ***accepting states***, Σ is a finite ***input alphabet***, and $\delta : Q \; x \; \Sigma \rightarrow Q$ is the ***transition function*** of M.

Let P be a pattern (string) to search for. In biochemistry, P is a string called a *motif*, wherein all of its letters are defined by the union of two alphabets: a main alphabet $\Sigma^B = \{A, C, G, T\}$ (every letter represents a nuceotide as described before), and the

IUPAC alphabet, (denoted as the extended alphabet in this work) used to represent ambiguities in the pattern. The IUPAC established an ambiguity alphabet, represented by $\Sigma^E = \{R, Y, K, M, S, W, B, D, H, V, N\}$, where $R = \{G, A\}$, $Y = \{T, C\}$, $K = \{G, T\}$, $M = \{A, C\}$, $S = \{G, C\}$, $W = \{A, T\}$, $B = \{G, T, C\}$, $D = \{G, A, T\}$, $H = \{A, C, T\}$, $V = \{G, C, A\}$, and $N = \{A, G, C, T\}$.

Let X be a string, where $X = AWB$, A and B are substring of X separated by the string W. The ***distance*** between two strings A and B is denoted by $d_{(A,B)}$, where $d_{(A,B)} = |W|$.

A ***match*** between two strings A and B is a process where each character of A is associated with a character of B. In this paper, we define three different associations (matchings) between two strings: exact matching, exact set matching, and inexact matching (for the sake of simplicity, consider the strings X, Y, and Z, where X and Y are defined in Σ^B, and Z is defined in $\Sigma^B \bigcup \Sigma^E$):

1. *Exact matching between X and Y*: Process where each character of X is equal to its corresponding character on Y. This match is also known as "equality between strings", denoted by $X = Y$. Formally, $X = Y$ if for each $i = 1, 2, ..., m$, $X[i] = Y[i]$ and $|X| = |Y| = m$. For example, if $X = ACG$ and $Y = ACG$, then $X = Y$.
2. *Exact set matching between X and Z*: A exact set matching between X and Z is established if it is possible to derive a string S from Z, where $X = S$. The string S is derived through substitutions of characters of Σ^E in Z to characters in Σ^B. Formally, an exact set matching between X and Z is established, denoted by $X \sim Z$, if for each $i = 1, 2...m : X[i] = Z[i]$ if $Z[i] \in \Sigma^B$ or $X[i] \in Z[i]$ if $Z[i] \in \Sigma^E$. For example, if $X = ACG$ and $Z = AMG$, where $M = \{A, C\}$, then by replacing characters Σ^E in Z we generate two strings AAG and ACG. Then, it is possible to generate a string $S = ACG$, where $X = S$, then $X \sim Z$.
3. *Inexact matching between X and Y*: An inexact matching between X and Y (denoted by $X \approx Y$) exists if it is possible to generate a string X' from X through substitutions, insertions, deletions (or combinations of these operations), such that $X' = Y$. These operations are explained as follows:
   - A *substitution* in $X = \alpha\gamma\beta$ is the operation that exchanges a character γ by another character γ' in Σ^B, deriving a new string $X' = \alpha\gamma'\beta$. This operation is used if we need to associate two strings X and Y in a exact match, where $|X| = |Y|$, but some characters are different. If $\gamma \in \Sigma^B$, then γ is replaced by a $\gamma' \in \Sigma^B$ such that $\gamma \neq \gamma'$. On the other hand, if $\gamma \in \Sigma^E$, then γ is replaced by a $\gamma' \in \Sigma^B$ - γ. For example, if $X = TGTCA$, such that $\alpha = TG$, $\gamma = T$ and $\beta = CA$, and $Y = TGGCA$, then we need to generate a string $X' = TGGCA$, where $\alpha = TG$, $\gamma' = G$ and $\beta = CA$, such that $X' = Y$.
   - An *insertion* in $X = \alpha\beta$ is the operation where we add a character γ to X, resulting in a string $X' = \alpha\gamma\beta$. This operation is used if we need an exact matching between X and Y, where $|X| < |Y|$. For example, if $X = TAG$, $Y = TGAG$ ($|X| < |Y|$), then we apply an insertion in X. If $X = \alpha\beta$, where $\alpha = T$ and $\beta = AG$, then we add $\gamma = G$, generating $X' = TGAG$ such that $X' = Y$.
   - A *deletion* in $X = \alpha\gamma\beta$, is the operation that removes a character γ from X, such that we generate a new string $X' = \alpha\beta$. This operation is used if we want an exact matching between X and Y such that $|X| > |Y|$. For example, if $X =$

$TCAAG$ and $Y = TCAG$ ($|X| > |Y|$), it is necessary to remove a character from X. If $X = \alpha\gamma\beta$, $\alpha = TC$, $\gamma = A$ and $\beta = AG$, then it is possible to remove γ from X, generating a new string X', such that $X' = Y$.

It is usual that the total number of insertions / deletions / substitutions in the inexact matching process is limited to a percentage (permissible error) defined by the user. In this work, we define the error of an inexact matching between two strings X and Y, denoted by $\sigma_{(X,Y)}$, as follows: $\sigma_{(X,Y)} = (\#ins_Y + \#sub_Y + \#del_Y)/|X|$, where Y is a string derived from X through insertions ($\#ins_Y$), substitutions ($\#sub_Y$) and deletions ($\#del_Y$).

Finally, we introduce the concept of a structured motif. This term is introduced in this work because it is based on the composition of simple motifs. Formally, a ***structured motif*** is a string $X = P_L W P_C Z P_R$, where $|W| = d_1$, $|Z| = d_2$ (this means that P_L and P_R are located at a distance from P_C no longer than d_1 and d_2 respectively), P_L is the left satellite, P_C is the central pattern, P_R is the right satellite, and P_L, P_C and, P_R are defined in $\Sigma^B \bigcup \Sigma^E$.

As an example, consider the string $X = A\ G\ T\ G\ A\ C\ G\ A\ C\ T\ C\ A$, where $P_C = ACG$, $P_L = TG$, $P_R = TC$ and $d_1 = 3$ and $d_2 = 4$. Then, in X we find a P_C in position 5 - 7, a P_L at position 3 (with $d_1 = 0$) and P_R at position 10 (with $d_2 = 2$).

In Section 3, we introduce a detailed description of our problem.

3 Problem Description

In this section we describe in detail the input and output of the SMR problem that is considered in this work.

First, we describe the input of the problem. Let T be a DNA database, P_L, P_C, and P_R three motifs to search for, where P_C is a motif called "central pattern", P_L and P_R called satellites (left and right respectively), two distances d_1 (the distance between P_L and P_C) and d_2 (the distance between P_C and P_R), and a percentage error σ (#mismatches allow in a matching process between a substring of T and a satellite). Based on these input parameters, the output of the SMR problem consists of finding all substrings S of T, such that:

1. There exists a substring S of T such that $S \sim P_C$, and
2. There exists a left substring S' of T at a distance d_1 of S such that $S' \approx P_I$, and
3. There exists a right substring S'' of T at a distance d_2 of S such that $S'' \approx P_D$, and
4. S' and S'' must both be generated within an error σ, ie, $\sigma_{(P_L,S')} \leq \sigma$ and $\sigma_{(P_R,S'')} \leq \sigma$.

It is important to mention that this problem has some variations based on the input parameters. These variants are described bellow:

- If the input of the SMR problem is: T, P_L, P_C, d_1 and σ, then we need to find all instances S and S' from T such that $S \sim P_C$, $S' \approx P_I$, $d(S, S') \leq d_1$ and $\sigma_{(P_L,S')} \leq \sigma$. This problem is called $P_L - P_C$.
- If the input of the SMR problem is: T, P_C, P_R, d_2 and σ, then we need to find all instances S and S' from T such that $S \sim P_C$, $S' \approx P_R$, $d(S, S') \leq d_1$ and $\sigma_{(P_R,S')} \leq \sigma$. This problem is called $P_C - P_R$.

– If the input of the SMR problem is: T, P_L, P_C, P_R, d_1, d_2 and σ, then we need to find all instances S, S' and S'' from T such that $S \sim P_C$, $S' \approx P_L$, $S'' \approx P_R$, $d(S,S') \leq d_1$, $d(S,S'') \leq d_2$, $\sigma_{(P_L,S')} \leq \sigma$, and $\sigma_{(P_R,S'')} \leq \sigma$. This problem is called $P_L - P_C - P_R$

For the sake of simplicity, in this paper we describe the solution of the P_L - P_C problem. Note that in order tho solve the other two versions of the problem, we only need to change the orientation of the distances, or combine the results of P_L - P_C and P_C - P_R. In the next section we introduce a proposal to solve the P_L - P_C problem.

4 Proposal

With the aim to solve the P_L - P_C problem, we propose a method divided in two phases:

– We first implement a searching phase, where all substrings S of T, $S \sim P_C$ are located. As result of this phase, we obtain a set $C = \{S : S$ is a substring of T, and $S \sim P_C\}$. To do this, we propose an automaton that searches for all instances of P_C in T. At the end of this phase, all results are stored in a matrix, including the end position of each pattern.
– In a second phase, all $S \in C$ are processed, searching if there exists a substring S' of T (satellite), such that $S' \approx P_L$ and $d(S',S) \leq d_1$. Since insertions, deletions, and substitution operations are needed in this phase, we propose a technique based on a dynamic programming strategy. As output, this phase reports all central patterns with their left-satellites.

We proceed to explain in detail the way in which each phase is implemented.

4.1 Searching the Central Patterns

As we mentioned above, we implement an automaton to search for all the instances of P_C in T. This automaton, called MFA [5] is based on the idea of storing in a temporal memory each pattern that has previously been recognized during the searching phase [2]. In other words, this automaton implements a strategy that stores knowledge about how the pattern matches overlap with itself, avoiding the computation of the prefix that matches with the last largest suffix. As result, in the searching phase, each character in T is examined exactly once (linear time with respect to $|T|$). In our explanation and for the sake of simplicity, consider that $P_C = AMS$, $P_L = GK$ and a distance $d_1 = 4$.

The MFA automaton must be constructed from the pattern P_C in a preprocessing step before it can be used to search P_C in T. This preprocessing phase is divided in three stages: first, we perform a phase called *expansion of* P_C, where all characters from P_C in Σ^E are substituted by characters in Σ^B (based on the IUPAC nomenclature); then, in a second step we build a matrix that stores the states of the MFA automaton; finally, in a third phase we generate the transition matrix of the MFA automaton.

The expansion of P_C is performed by a substitution process, where each character of P_C in Σ^E is "expanded" with each valid character in Σ^B, restricted to the combinations derived from the IUPAC specification. As result, we generate a set called *setP*, where

we store each combination of the pattern created with the substitution of characters. As an example, if $P_C = AMS$, where $M = \{A, C\}$, $S = \{C, G\}$, then M and S are replaced by characters in Σ^B, from right to left. As final result, $seqP = \{AAC, ACC, AAG, ACG\}$.

Based on *seqP*, the second phase consists on building a matrix called *matQ*, which stores all states of the MFA automaton. With the aim to identify with precision all the final states, this matrix is filled sequentially, from top to bottom (by rows), and from left to right (by columns), where the number of columns is equal to $|P_C|$ (each column corresponds to a character). The values assigned to the matrix start from 1, and with unit increments. Each column is assigned a number of values that is equal to the product of each character cardinality associated to the previous columns, including the current column. The last column of this matrix stores all the final automaton states. Resuming to our example, *matQ* for $P_C = AMS$ is filled as shown in Fig. 1.

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 sequentially filled
For column M:
|M|=2, two rows will be filled up to line |A|*|M|=2
For column S:
|S|=2, four rows will be filled up to line |A|*|M|*|S|=4

Fig. 1. *Filling matQ*

As final step of the preprocessing phase, we generate the transition matrix δ of the MFA automaton. This matrix is filled row by row, its values depend of the largest suffix that is prefix of some element of *seqP* (for further reference, see [5]).

For example, consider vertex 2 of Fig. 2 with label "AA", which is obtained by concatenating the letters in the path from the root to this vertex. From AA, we derive a new expansion by adding at the end each character in Σ^B, obtaining the set of strings AAA, AAC, AAT, and AAG. We then test these strings, searching their largest suffix that is a prefix for some element in *seqP*. In this example, AAC and AAG satisfy this condition, since they are suffixes of the first and third element of *seqP* respectively.

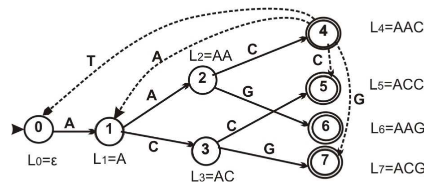

Fig. 2. *Generation of prefixes to fill* δ

As next step, we compute states that are reached from vertex 2 through transitions with C and G. Each result is stored in $\delta[i, j]$, where i is the ith element of *seqP* associated

to the prefix, and j is the jth element of Σ^B (used in the respective transition). Resuming to our example, from vertex 2 with transition throughout "C" (third element of Σ^B), we obtain the string AAC, which is suffix of the first element of *seqP*. Therefore, $\delta[2,2] = matQ[k,l]$, where k is the kth element of *seqP* associated to the prefix, and l is the cardinality of this prefix. In this example, $k = 3$ (AAC is a sufix of the first element of *seqP*), and $l = 3$ (|AAC| = 3). As result of this, from vertex 2 with label AA, there exists a transition with C towards vertex 4. This process is used to compute each transition for each state of the automaton.

Before we continue and for the sake of simplicity in our explanation, we enumerate in our example the rows of *matQ* starting from 1, and the rows of δ starting from 0.

At the end of the preprocessing phase, the MFA automaton obtained from our example is shown in Fig. 3.

$$Q = \{0, 1, \ldots, n \mid n = \sum_{i=1}^{|P|} \prod_{j=1}^{i} |P_j|\}$$
$$q_0 = 0$$
$$A = \{i \mid i = matQ[j][|P|], j = 1, 2, \ldots, |seq\mathbb{P}|\}$$
$$\Sigma = \{A, C, G, T\}$$

δ:

	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

Fig. 3. *Automaton created by the algorithm*

After that we build δ, the next step consists of search of each pattern from *setP* in T. However, this phase is simple because δ is computed in linear time T, character by character. Consider that our results are stored in a matrix called $PCS_{npc,2}$, where npc is the total number of patterns located in T. For simplicity, consider that the first column of PCS stores each pattern, and the second column the position of the last character of the pattern in T.

As an example, if $T = ATGGACAACC$ and δ is the automaton shown in Fig. 3, we obtain the output illustrated in Fig 4. In this example, the circles are used to mark the final states, which represent patterns found in the searching phase. In this example, the found patterns are AAC and ACC. These results are stored in the PCS matrix illustrated in Fig. 5.

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

Fig. 4. *Results of the localization of central pattern*

PCS:

AAC	9
ACC	10

Fig. 5. *PCS Matrix*

4.2 Searching the P_L pattern

The next step in our methodology consists of a process where based on a string S in PCS, we search for substrings S' of T where $S' \approx P_L$, and $d(S', S) \leq d_1$.

Since P_L, as the central patterns, may contain characters in Σ^E, we first replace these characters by characters in Σ^B (in this phase we could use the same substitution process explained before). Let $\mathbb{S}$ be a set that stores each pattern generated from S.

Let $P_L = GK$ be the pattern to search for. After we replace each extended character from P_L, our output is $\mathbb{S} = \{GG, GT\}$. We then search for each element of this set in a segment of T, limited by the position of the central pattern and a left distance no larger than d_1. Resuming to our example where $T = ATGGACAACC$, $P_C = AMS$ and $d_1 = 4$, we have two possible central patterns, AAC and ACC. If we first process AAC as P_C and GG as P_L, then we need to find all possible alignments as we show in Fig. 6.

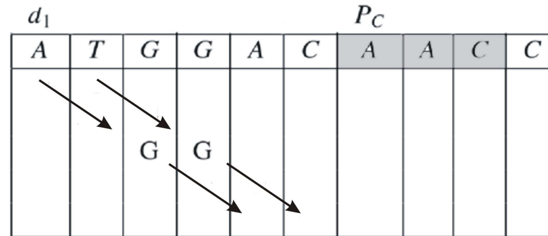

Fig. 6. *Searching for satellites*

As we mentioned in section 4, it is possible to use insertions, deletions, and substitutions in the searching phase of P_L. The total number of these operations is limited by σ, which is the error allowed by the user (this value is defined as a percentage with respect to the dimension of P_L).

Since this problem is related to the string alignment problem [2], we propose a solution based on a dynamic programming strategy with the Levenshtein distance [1], which determines the total number of insertions, deletions and substitutions that we need to transform a string (original pattern) into other string (target pattern).

As we see in Fig. 6, the first step consists of defining a start and end position where we will search for P_L in T. This gap is called a *search window*, and there is a window search per each pattern in PCS. The limits of a search window are computed from the initial position of each P_C in PCS, where: $Pos_{start} = Pos_{end} - |P_C| + 1$ (Pos_{end} value is

retrieved from PCS). Based on our example, where PCS includes the central patterns AAC and ACC, we obtain the following initial positions:

- AAC: $Pos_{start} = Pos_{end} - |AAC| + 1 = 9 - 3 + 1 = 7$
- ACC: $Pos_{start} = Pos_{end} - |AAC| + 1 = 10 - 3 + 1 = 8$

Based on Pos_{start}, we compute the initial and final positions of each search window of each P_L in $\mathbb{S}$, where: $W_{ini} = Pos_{start} - d_1 - |P_L|$, and $W_{end} = Pos_{start} - 1$. In our example, where $P_L = GK$, $\mathbb{S} = \{GG, GT\}$ and $d_1 = 4$ we compute these positions per each P_C:

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

As an example, Fig. 7 shows the central pattern $P_C = AAC$ and its corresponding window search, where we need to find if there exists an alignment with each pattern in $\mathbb{S}$.

Wstart					Wend	Posstart		Posend	
A	T	G	G	A	C	A	A	C	C
G	G								

Fig. 7. *Initial and final positions of the window*

Note that if we do not implement an intelligent strategy to search for possible alignments, we need to test a character more than once if there exist intersections between two or more search windows. In order to avoid this computation, we compute the intersection between two search windows. This value is stored in a variable called Mem, where

$$Mem = \begin{cases} W_{end}^{previous} - W_{start}^{current} & \text{if } W_{end}^{previous} > W_{start}^{current} \\ 0 & \text{otherwise} \end{cases}$$

$W_{end}^{previous}$ is the end position of the previous search window and $W_{start}^{current}$ is the start position of the current search window. For our example of Fig. 7, we compute Mem per each central pattern as follows:

- AAC: $W_{end}^{previous} - W_{start}^{current} = 0 - 1$, then $Mem = 1$
- ACC: $W_{end}^{previous} - W_{start}^{current} = 6 - 2$, then $Mem = 4$

The next step consists of building a matrix that stores the total number of insertions, deletions, and substitutions which are needed to align P_L with a segment of characters into the current search window. This matrix, denoted by $D_{(m+1)\times(n+1)}$, has $m + 1$ rows, where $m = |S'|$ (S' is an element of $\mathbb{S}$), and $n + 1$ is the total number of columns (where n is the cardinality of the search window associated to the current central pattern). This matrix is filled with the Levenshtein algorithm [1]:

- $D[0, j] = 0$, where $0 \leq j \leq n$
- $D[i, 0] = i$, where $0 \leq i \leq m$
-

$$D[i, j] = \begin{cases} D[i-1, j-1] & \text{if } \mathbb{S}_{k,i} = T[j] \\ D[i-1, j-1] + 1 & \text{otherwise} \end{cases}$$

where $\mathbb{S}_{k,i}$ is the ith character of the kth element of $\mathbb{S}$, and $T[j]$ is the jth character of the search window that is currently being processed. These rules are used to fill each D for each element in $\mathbb{S}$. When we changed of search window, we compute Mem to know if there exist columns in D that are a copy of the matrix of the previous search window (these columns are a copy starting from the second column of the current matrix D).

Resuming to our example, consider the case of ACC with GG as P_L with $Mem = 4$. Therefore, the current matrix D is filled as follows: the first column is computed based on the Levenshtein algorithm; the next four columns are copied from the matrix of AAC with GG as P_L; the last column is also filled based on the Levenshtein algorithm. This example is shown in Fig. 8.

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

Fig. 8. *Example where matrix D is generated for $P_C = AMS$ and $P_L = GK$*

Finally, these matrices are used to find possible alignments based on the error σ. The procedure is simple: we select all $D[|P_L|, j]$, where $D[|P_L|, j] \leq \sigma$, and j is the final position of a valid left pattern of the corresponding search window. As an example, if we consider that $\sigma = 50\%$, then we need to find all positions where $D[|P_L|, j] \leq 1$, because $|P_L| = 2$. In our example of Fig. 8, there exist three valid results for AAC and GG as left pattern: $D_1[2, 2] = 1$, $D_1[2, 3] = 0$, $D_1[2, 4] = 1$; and three valid results for ACC and GG as left pattern: $D_4[2, 1] = 1$, $D_4[2, 2] = 0$, and $D_4[2, 3] = 1$. With this information, we retrieve the left patterns based on $T = ATGGACAACC$:

- Since the search window for $P_C = AAC$ and GG as left pattern is $ATGGAC$, then our left patterns are located at the end positions: 2 (TG is transformed in to GG with one substitution), 3 (corresponding for the pattern GG), 4 (GA, is transformed to GG with one substitution)
- Since the search window for $P_C = ACC$ and GG as left pattern is $TGGACA$, then all patterns for this search window are the same as the previous window (they are not computed again).

As we can see in this example, our proposal has the following advantages: we can find the central patterns very fast because we avoid computing any character in T twice. With a dynamic programming strategy based on the Levenshtein algorithm, it is possible to improve the runtime taken to find the left patterns, we avoid computing common

regions twice. However, it is important to see that this approach needs to generate a matrix per each possible left pattern.

5 Conclusions and Future Work

This paper presents a proposal for solving the structure motive recognition in a DNA sequence. Our approach is divided in two phases: first, we locate all central patterns with an automation, that performs this task very fast; second, we implement a dynamic programming strategy and the Levenshtein algorithm with the aim to find satellites that are located at a distance no larger than d. Actually, we implemented and tested the automaton and found that its performance is very fast, allowing us to work with large DNA databases.

We are currently implementing the dynamic programming strategy with the Levenshtein algorithm proposed in this work. After testing this phase, we will develop a tool that can be used by biologists that need to find structured patterns.

References

1. Bofivoj, Melichar. *Approximate string matching by finite automata*. In Conf. on Analysis of Images and Patterns, number 970 in LNCS. Pages 342–349. 1995.
2. Cormen, Thomas H. *Introduction to Algorithms*. 2nd Edition. The MIT Press and McGraw-Hill. Pages 805-811, 2001.
3. Maxime Crochemore and Marie-France Sagot, *Motifs in Sequences: Localization and Extraction*. Pages 26-31, 2000
4. Navarro, Gonzalo. *A Guide Tour to Approximate String Matching*. ACM Computing Surveys. vol. 33, Issue 1. Pages. 31-88, 2001.
5. Perez, Gerardo et al. *An Automaton for Motifs Recognition in DNA Sequences*. To appear in MICAI 2009, Springer - Verlag.