

Motif Discovery fixing mismatch positions

Marco Zantoni, Alberto Policriti
Università degli Studi di Udine
Via delle Scienze, 206 - Udine
{zantoni,policrit}@dimi.uniud.it

Emiliano Dalla, Claudio Schneider
Laboratorio Nazionale CIB
Area Scienze Park, Padriciano - Trieste
dalla@area.trieste.it
schneider@lncib.it

1 Introduction

The discovery/identification of short strings occurring approximately in a set of longer strings/sequences is one of the major tasks in today's computational biology. In our particular case we refer to these short strings as *motifs*. In our initial setting the notion of "approximate occurrence" means that motifs must match a segment of (each) sequence with at most some specified number of mismatches. Motif discovery abstracts many problems encountered during the analysis of biological sequence data, where the sequences can be nucleotide or protein molecules and motifs represent short functionally important patterns. In this work we have in mind a new computational approach to the problem of looking for *Transcription Factor Binding Sites* (TFBS): the search for genomic motifs responsible for the binding of transcription factors to Promoters and other regulative elements, the major event underlying gene expression control. More specifically, in this paper we focus our attention on the problem of, given a set of strings, finding a substring common to (a significant portion of) the strings in the input set, allowing a *fixed layout* for mismatches in our output. Our general strategy used to tackle the problem is based on the introduction of a data structure encoding 'a la Karp-Rabin substrings of the strings in the input set. This idea, together with the fixed layout error assumption and the observation that layout mismatches can be ordered, allows a quick and efficient motif indexing and storing based on an incremental construction highly reducing the overall number of basic operations.

2 Our Proposal

Let $\mathcal{F} = \{s_1, \dots, s_m\}$, d the number of errors allowed in comparisons, and q a parameter denoting the minimum size of the set of \mathcal{F} -elements containing a closest substring of length ℓ with at most d errors. Without loss of generality, we consider input strings of the same length n .

When using a solution to the above problem in a biological framework like the identification of known and putative TFBS, we take advantage of a useful assumption: given a motif (TFBS), only some of its characters (nucleotides) are important for the binding of the transcription factor. In fact, transcription factors interact together chemically and physically then it is important their position respect to the DNA helix, and therefore

respect to nucleotides to which they can bind. This feature simplifies and reduces the number of consensus TFBS generated and analyzed by the algorithm.

Thus, we propose an algorithm based on the concept of localized nucleotide mutation: a protein can “tolerate” one or more mutations in a binding site, but always in the same positions (fixed layout error).

Definition: A solution for the *fixed-layout* (ℓ, d, q) -consensus problem over \mathcal{F} is S_{fl} if and only if

- $S_{fl} \subseteq \Sigma^\ell$
- for all $s' \in S_{fl}$, there exists $s_i \in \mathcal{F}$ such that $s' \trianglelefteq s_i$
- $|\{i \mid \exists s' \in S_{fl}, s' \trianglelefteq s_i\}| \geq q$
- there exists a set of indexes $fl = \{i_1, \dots, i_d\}$, $1 \leq i_1 < \dots < i_d \leq \ell$, such that for all s'_i and s'_j in S_{fl} , $s'_i[k] \neq s'_j[k] \Rightarrow k \in fl$.

We assume to produce in output all the positions of the common subsequences of length ℓ with d errors discovered in at least q sequences, with the addition of the constraint that the errors, if occur, are in the same position (called layout).

2.1 ScanPro

In the above considerations we can specify our approach based on an adaption of the Karp-Rabin technique to the solution of the *fixed-layout* (ℓ, d, q) -consensus problem.

2.1.1 The algorithm.

We classify error layouts into two classes: *basic* and *shifted*. The algorithm exploits the relation between these classes to perform a cost-effective encoding of substrings according to all possible layouts. A generic error layout $fl = \{i_1, \dots, i_d\}$, such that $1 \leq i_1 < \dots < i_d \leq \ell$, is the set of the d positions where an error may occur during a comparison between strings. Without loss of generality, we assume that i_1 is strictly greater than 1, that is no error can be in the first position; in this way there are $tfl = \binom{\ell-1}{d}$ error layouts called $FL = \{fl_1, \dots, fl_{tfl}\}$. *Basic layouts*, bfl , are characterized by having $i_d = \ell$ and are $\binom{\ell-2}{d-1}$ overall. *Shifted layouts* relative to a given basic layout bfl_x are denoted by $sfl_{x,j}$ and look like $\{i_1 - j, \dots, i_d - j\}$, with $i_1 - j \geq 2$.

For a given error layout, the function $fl_code : (\Sigma^\ell \times FL \rightarrow \mathbb{N})$ gives the encoding of a string of ℓ characters. For each string $s \in \mathcal{F}$, with $|s| = n$, and each basic layout $bfl_x = \{i_1, \dots, i_d\}$, we have to encode all possible $n - \ell + 1$ substrings of length ℓ in s , thus perform $\mathcal{O}(n \cdot \ell)$ operations. Considering a shifted layout $sfl_{x,j} = \{i_1 - j, \dots, i_d - j\}$, with $j = 1, \dots, i_1 - 2$, we obtain the encoding of the substring $s[i, \dots, i + \ell - 1]$ for the given layout by taking the encoding of $s[i - 1, \dots, i + \ell - 2]$ for the error layout $sfl_{x,j-1}$, performing a left shift and adding the value relative to $s[i + \ell - 1]$, making $\mathcal{O}(n)$ operations only. Since $|\Sigma| = 4$, we map the alphabet on \mathbb{Z}_4 using only 2 bits for each character. Hence, an ℓ -characters-long string is mapped into a 2ℓ bits integer number, so any shift involves only 2 bits. The implementation (in language C, available upon request) of the algorithm exploits this compact binary representation to improve performances. This idea was suggested by the *Shift-Or* algorithm.

The function fl_code returns a value used as index of an array: in each position c of this array there is a pointer to a matrix M_c of dimension $tfl \times (m + 1)$, with $m = |\mathcal{F}|$. $M_k(i, j) \neq NULL$ if in s_j there exists a substring w such that $|w| = \ell$ and $fl_code(w, fl_i) = k$. In column $m + 1$ there is the “rank”, that is the number of different strings $s \in \mathcal{F}$ in which we can find that specific motif. A rank is incremented when an element is put in a empty position in the relative row. If this last value is greater than the quorum, the motif is a solution for the *fixed-layout problem* and we retrieve all positions of the occurrences using a modified generalized suffix tree able to manage don’t care positions.

Using these occurrences, we are able to generate a consensus c_k , according to a consensus string model (i.e. the majority string). To obtain a most biologically informative consensus, during its creation we use an extended alphabet $\Sigma' = \{A, C, G, T, R, Y, N\}$, where $R = A$ or G (purine), $Y = C$ or T (pyrimidine), and N is any character, in according to the IUPAC alphabet.

We can observe that basic layouts of dimension d are easily obtainable from shifted layouts of dimension $d - 1$ and, similarly, values of corresponding encodings. Iterating this argument, it is not difficult to show that we can reach the previous results in time $\mathcal{O}(mdl^{d-1}n)$. Asymptotically such performance matches the one of the described algorithm, but additional information is gathered in the process.

The space needed ($\mathcal{O}(|\Sigma|^{l-d}l^{d-1}m)$) depends on the number of strings in input but not on their length. It is possible to reduce the space needed at least of a factor m worsening time complexity. Furthermore, with our implementation, this algorithm is completely on-line.

2.1.2 On the road.

The table below shows the results of a test performed on a 2.0GHz AMD processor with 2GB ram on a dataset of 90 sequences 1000 nucleotides long with the quorum at 30% (low filtering). T_{comp} denotes the encoding and data structure building time, while T_{out} the output file generation time.

| | | | | | | | |
|------------|--------|--------|--------|--------|------|------|-------|
| ℓ | 6 | 8 | 10 | 10 | 12 | 13 | 16 |
| d | 1 | 2 | 3 | 2 | 3 | 3 | 4 |
| $\ell - d$ | 5 | 6 | 7 | 8 | 9 | 10 | 12 |
| T_{comp} | 0.2sec | 0.4sec | 1sec | 0.8sec | 3sec | 7sec | 11sec |
| T_{out} | 1.2sec | 0.8sec | 1.8sec | 1sec | 5sec | 4sec | 4sec |

Another test were performed on a dataset of 30 sequences 50k nucleotides long with the quorum at 50%. The hardest combination of parameters has required less than 3 minutes.

REFERENCES

1. T. Akutsu, H. Arimura, and S. Shimozono. On approximation algorithms for local multiple alignment. *RECOMB*, 2000.
2. B. Brejova, C. Di Marco, T. Vinar, S. Hidalgo, G. Holguin, and C. Patten. Finding patterns in biological sequences, 2000.

3. M. Frances and A. Litman. On covering problems of codes. *Theor. Comput. Syst.*, 30:113–119, 1997.
4. Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
5. R. M. Karp and M. O. Rabin. Efficient randomized patten-matching algorithm. *IBM J. Res. Dev.*, 31(2):249–260, 1987.
6. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. *Information and Computation*, 185(1):41–55, 2003.
7. M. Li, B. Ma, and L. Wang. Finding similar regions in many strings. *Proc. 30th ACM Symp. Theory of Computing (STOC'99)*, pages 473–482, 1999.

Algorithm 2.1 *Encoding(Seq, LEvect)*

```

1: Seq: one of the  $m$  sequences in input;
2: LEvector: vector containing error layouts;
3: codeVect: temporary vector containing subsequences encodings;
4: cArray: array of matrices;
5: for ( $l=1$  to  $tf$ ) do
6:   if (LEvect[ $l$ ] is basic) then
7:     for ( $i=1$  to  $length(Seq) - l + 1$ ) do
8:        $codeVect[i] = fl\_code(Seq[i \dots i + l - 1], LEvect[l]);$ 
9:     end for
10:  else
11:     $codeVect[1] = fl\_code(Seq[1 \dots l], LEvect[l]);$ 
12:    for ( $i=2$  to  $length(Seq) - l + 1$ ) do
13:       $codeVect[i] = 4 * (codeVect[i - 1]/4^l) + Seq[i + l - 1];$ 
14:    end for
15:  end if
16: end for
17: for ( $i=1$  to  $length(Seq) - l + 1$ ) do
18:    $Matr = cArray[codeVect[i]];$ 
19:   if ( $Matr[l, n\_seq] == 0$ ) then
20:      $Matr[l, n\_seq] = 1;$ 
21:      $Matr[l, m + 1] = Matr[l, m + 1] + 1;$ 
22:   end if
23: end for

```
