Some string matching problems from Bioinformatics which still need better solutions

Gaston H. Gonnet

ETH Informatik, 8092 Zürich, Switzerland

Abstract

Bioinformatics, the discipline which studies the computational problems arising from molecular biology, poses many interesting problems to the string searching community. We will describe two problems arising from Bioinformatics, their preliminary solutions, and the more general problem that they pose. The first problem is searching for $\alpha$-helices in protein sequences. This particular instance of the search is based on matching of hydrophobicity/hydrophilicity. We find an algorithm which is linear in the sequence length for fixed helix length and is $O(n \log n)$ for any helix length. The second problem is on matching probabilistic sequences against sequences or against other probabilistic sequences. In both cases we derive efficient formulas to compute scores according to a Markovian model of evolution.

Keywords: String matching; Bioinformatics; Alpha-helix prediction; Pattern matching; Homology detection; Profile searching; Probabilistic sequence searching; Probabilistic sequence matching

1. Introduction

String searching problems in bioinformatics are normally posed in terms of returning a score as opposed to a binary answer. This is quite fundamental, since most of the time these searching problems return the likelihood of a certain event. Consequently, we will assume that the common denominator for the problems described herein is that our hits are not binary (yes or no) but rather return a score, and we will be searching for the highest scoring strings. Furthermore, we will assume that this score is not trivial to compute. Many of these problems are not solved well or efficiently.
The nature of the biological problems, and the omnipresence of random perturbations, quickly indicates that exact matches are never interesting. Furthermore, string searching problems are seldom a goal in themselves in bioinformatics, usually they are one of the many building blocks for other computations. Typically, the results of our searching goes into higher-level processes, like classification, structure prediction, multiple sequence alignments or phylogenetic trees. It should also be pointed out that these are real problems, which we encounter everyday in our work with biologists and in the development of Darwin, a system for doing bioinformatics computations [7].

Molecular biology information, for the purpose of this paper, consists of sequences of DNA/RNA (4 symbols) or sequences of amino acids (20 symbols) normally called proteins.

An example of a DNA sequence is:

\[ \text{ATGATCGTAAATAACACACGTC} \ldots \]

An example of a RNA sequence (messenger RNA, mRNA in this case) derived from the above and the protein it encodes are:

\[ \text{AUG AUC GUA AAU AAC ACA CAC GUG CUU ACC CUA CCA CUU UAU} \ldots \]

An example of a protein or amino acid sequence is:

\[ \text{MIVNNTHVLTLPLY} \ldots \]

Molecular biology information, for the purpose of this paper, consists of sequences of DNA/RNA (4 symbols) or sequences of amino acids (20 symbols) normally called proteins.

An example of a DNA sequence is:

\[ \text{ATGATCGTAAATAACACACGTC} \ldots \]

An example of a RNA sequence (messenger RNA, mRNA in this case) derived from the above and the protein it encodes are:

\[ \text{AUG AUC GUA AAU AAC ACA CAC GUG CUU ACC CUA CCA CUU UAU} \ldots \]

An example of a protein or amino acid sequence is:

\[ \text{MIVNNTHVLTLPLY} \ldots \]

An example of a protein or amino acid sequence is:

\[ \text{MIVNNTHVLTLPLY} \ldots \]

For computer scientists, the difference between DNA and RNA is that all the T’s are replaced with U’s. Three consecutive bases of RNA, properly synchronized, are called a codon. Codons code for proteins as shown in the second/third line above. We can assume that all forms of life use 20 amino acids which are the components of all proteins and are also encoded in a single-letter alphabet. The above sequence comes from the yeast Saccharomyces cerevisiae genome.

As we said we before, we are scoring all of our hits with a scoring function. Typically, a scoring function has the form

\[ S_i = F_i(s[i], s[i+1], \ldots, s[i+L-1]), \]

where \( s[i] \) is the \( i \)th symbol in the sequence and \( L \) is the length of the pattern to be scored.

E.g., for \( L = 6 \)

\[ S_1 = F_1(M, I, V, N, N, T), \]
\[ S_2 = F_2(I, V, N, N, T, H), \]
\[ S_3 = F_3(V, N, N, T, H, V), \]

etc. Our searching problems can be described as finding the \( i \) values for which \( S_i \) is largest.

The above is too generic, quite often we have scores which are of the form:

\[ S_i = R(F_i(s[i]), F_{i+1}(s[i+1]), \ldots, F_{i+L-1}(s[i+L-1])) \]
and $R$ is a relatively simple reduction function (we use the terminology from relational databases, where a reduction function is a function that maps a set of elements into a single one, for example $\max$, etc., e.g.).

$$S_1 = \frac{\max(F_1(M), F_2(I), F_3(V), F_4(N), F_5(N), F_6(T))}{1/6(F_1(M) + F_2(I) + F_3(V) + F_4(N) + F_5(N) + F_6(T))}.$$ 

An important class of scoring functions are derived from probabilities (sometimes from likelihoods). If $\Pr\{\text{event}_i\}$ denotes the probability of a certain event, for example the probability that the sequence is homologous to some pattern, then

$$S_i = - \log \Pr\{\text{event}_i\}$$

is a very practical and reasonable scoring function. There are many good reasons to use the above, among others:

(a) it provides a sound foundation to the score,
(b) scoring of independent events results in the sum of the scores. This is a very desirable property for building other algorithms on top of scores, most notably dynamic programming algorithms;
(c) it allows to relate/compare scores obtained from different sources.

In this context it makes perfect sense to search for the highest scoring sequences, and not just a single one or a single target score.

2. Identifying $\alpha$-helices

The first problem is related to the identification of $\alpha$-helices in protein sequences. A similar, but simpler, problem arises with the identification of $\beta$-strands. We will concentrate on the first problem.

An $\alpha$-helix is a unit of secondary structure of proteins, found in many proteins. About 3/4 of the proteins contain $\alpha$-helices [12]. Recognition of $\alpha$-helices from the primary structure (sequence of amino acids) is usually called “secondary structure prediction”, and is a very important and active area of research. An $\alpha$-helix is a helical structure which rotates $100^\circ$ with every amino acid. Its existence and stability is normally due to its hydrogen (weak) bondings and its surroundings: a helix normally does not stand by itself. In particular, many helices find themselves in the interface between the inner and outer part of the protein. This information is very useful to search (or confirm) helices, as one side of the helix is hydrophobic (the inner side) and the other is hydrophilic (the outer side, typically exposed to water). The inner part of a protein is almost always hydrophobic, while the protein normally is water soluble, hence the outer part should be hydrophilic.

Hydrophobicity is the property of “disliking” water, like oily substances. Each amino acid has an index of hydrophobicity. Hydrophobicity indices range from $-1$ to $1$ or some other similar range which can be mapped to $-1 \ldots 1$. An index of $1$ means that the amino acid is highly hydrophobic, and index of $-1$ means it is highly hydrophilic (likes water). Hence a natural score for a sequence of $L$ amino acids starting at position $j$ is the internal
product:

\[ R = \sum_{i=j}^{j+L-1} H_i \sin \left( \frac{2\pi i \times 100}{360} + \theta \right) \]

where \( \theta \) is the phase angle of the helix with respect to the inner/outer limit and \( H_i \) is the hydrophobicity index of amino acid \( i \) [9]. The score for each \( j \) and \( L \) depends on \( \theta \), so we first have to find the best \( \theta \) for each position \( j \).

Expanding and collecting we obtain:

\[ R = S_j \cos \theta + C_j \sin \theta \]

with

\[ S_j = \sum_{i=j}^{j+L-1} H_i \sin \frac{5\pi i}{9}, \quad C_j = \sum_{i=j}^{j+L-1} H_i \cos \frac{5\pi i}{9}. \]

The phase angle \( \theta \) which gives the maximum score for each \( j \) is easily obtained by solving the derivative of \( R \) equated to 0. The phase angle \( \theta \) giving the maximum (\( \theta = \tan^{-1}(C_j/S_j) \)) is not very interesting in itself, the maximum score given by this \( \theta \) is the interesting value:

\[ R = \sqrt{C_j^2 + S_j^2}. \]

A linear time searching algorithm is easy to derive (for a fixed \( L \)). The values of \( C_j \) and \( S_j \) can be updated in constant time by adding the \((j + L)\)th term and subtracting the \( j \)th term.

It is clear that such an algorithm will work for any score which is a function of sums (or other reduction functions with an inverse) over individual values.

Although this algorithm is linear, it is still not good enough, as the parameter \( L \) depends on nature, and it has values from 5 to 30. This imposes a new constraint on the problem, we would like to solve this searching problem for a score which is

\[ R_j = \max_{L=5\ldots30} \sqrt{C_j^2 + S_j^2} = \sqrt{\max_{L=5\ldots30} C_j^2 + S_j^2}. \]

2.1. Geometrical interpretation

This optimal score has a surprisingly simple geometrical interpretation. If every \( H_i \) is viewed as a vector from \((0, 0)\) to \((H_i \cos \frac{5\pi i}{9}, H_i \sin \frac{5\pi i}{9})\), then we are looking for the longest path of \( L \) consecutive vectors. So what we are doing is searching for long relatively straight stretches in a consecutive graph of the \( H_i \)s (Fig. 1).

The above graph produced by a protein shows typical behaviour. There are some portions which are randomly twirled, and then we have a relatively straight stretch going up starting on the 13th amino acid (possibly broken in the middle). The rest of the protein shows less significant straight stretches.

We can solve this problem for all possible values of \( L \) by finding the convex hull of this set of points and then finding its diameter. This would require \( O(n \log n) \) time. It is clear
from the graph, that the diameter is not very interesting, what is really interesting are the straight, relatively short, stretches.

This leaves us with two interesting open problems:

**Open problem 1.** Is there a more efficient algorithm to search an amino acid sequence for a subset of values of $L$, e.g., $L = 5\ldots 30$? (Any improvement is interesting, even if it is not asymptotic, but just a constant factor.)
Other scoring functions for α-helices may also be interesting, for example

\[ R_j = \max_{L=5...30} \sqrt{\frac{C_j^2 + S_j^2}{\sum_{i=j}^{j+L-1} H_i^2}} \]

**Open problem 2.** Are there any mathematical properties of the score \( R \) which makes this search generally more efficient?

### 2.2. Reduction functions in \( O(1) \) time

We call reduction functions any function which maps a set of values into a single value, e.g., sum, max, ave, etc. Earlier we stated that a reduction function which accepts an inverse is easy to compute. For example for sum, we can compute:

```plaintext
run_sum := sum( F(s[i]), i=1..L );
for i from 1 to n do
  .... evaluate scoring function ....
  run_sum := run_sum - F(s[i]) + F(s[i+L]);
od;
```

Is it possible to do this efficiently for functions which are associative but do not necessarily have an inverse? (such an object is called a monoid). For example, \( \text{max}() \)? Alternatively we may wish to use this property when the inverse is extremely expensive to compute, like matrix multiplication.

A little known fact is that monoid operations over a running sequence can also be updated in constant time using in total \( O(L) \) space. It is worth reviewing this idea which goes as follows:

Let \( k \) be the largest integer such that \( 2^k \leq L/2 \), \( k = \lfloor \log_2 L \rfloor - 1 \).

We compute \( k \) vectors \( iF_j \) which contain \( 2^i \) summarized results each. More precisely

\[ i+1 F_j = \max( i F_j, i+2 F_j ) \]

\[ 0 F_j = F_j[s[j]] \]

The only \( i F_j \) which are kept are the ones for which \( j \) is a multiple \( 2^i \).

Computing these values requires an average of \( \frac{2^{2i}-1}{2^{i+1}} \) operations per position. For example:
<table>
<thead>
<tr>
<th>(j)</th>
<th>(0F_j)</th>
<th>(1F_j)</th>
<th>(2F_j)</th>
<th>(3F_j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8.1467</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6.1208</td>
<td>7.2938</td>
<td>7.7943</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7.2938</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2.4366</td>
<td>7.7943</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7.7943</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.1776</td>
<td>6.5518</td>
<td>9.9351</td>
<td>9.9351</td>
</tr>
<tr>
<td>9</td>
<td>6.5518</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>9.9351</td>
<td>9.9351</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4.4696</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>6.8383</td>
<td>6.8383</td>
<td>6.8383</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1.9662</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>5.1864</td>
<td>6.3316</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>6.3316</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The values of \(iF_j\) can be kept in circular arrays of length \(\lceil L/2^i \rceil\). No data structure other than arrays is needed.

From the above, it is now trivial to see that we could compute a running max of \(L\) elements in \(O(\log L)\) steps per position (which is not a bad practical variant). However, if we want to save the \(\log L\) factor or the monoid operation is very expensive, then we can use the following algorithm.

We will compute the values of the maximum for increasing values of \(j\) in \(O(1)\) operations by grouping the “head” operations and the “tail” operations. This grouping is best illustrated by an example. Suppose that \(L = 20\) hence \(k = 3\), and \(j = 77\) then

\[
\max(F_{77} \ldots F_{96}) = \max\left(\max(F_{77}, \max(1F_{78}, 3F_{80})), \max(3F_{88}, F_{96})\right).
\]

Notice the grouping, which identifies the \(2^k\)-size blocks in the middle and groups on the right for the tail part and on the left for the head part.

It is easy to see, that as the computation proceeds, with this grouping, \(O(1)\) computation is needed at each step.

\[
\max(F_{78} \ldots F_{97}) = \max\left(\max(1F_{78}, 3F_{80}), \max(3F_{88}, 1F_{96})\right)
\]

the tail requires no computation, it reuses the previous partial result, the head requires one new max to be computed and finally the outermost max (which is always required).

\[
\max(F_{79} \ldots F_{98}) = \max\left(\max(F_{79}, 3F_{80}), \max(\max(3F_{88}, 1F_{96}), F_{98})\right)
\]

the tail requires one max computation, the head also one (reuses the inner one) and the outer max.

\[
\max(F_{80} \ldots F_{99}) = \max(3F_{80}, \max(3F_{88}, 2F_{96}))
\]

again the tail requires no computation, the head one.
We are not going to give a formal description of the algorithm or its proof, which are straightforward once that we observe that:

(a) the tail groups $F$ values in increasing left-superscript (the head in decreasing left-superscript);
(b) a head operation $\max(\ldots, F_j)$ (or a tail operation $\max(F_j, \ldots)$) will be reused $2^i$ times;
(c) every time that a $kF_j$ disappears from the tail, we move the next one from the head to the tail.

For each step we have an average of four monoid operations, one to maintain the $F$ values, one for the tail, one for the head and one for the final result.

3. Matching profiles or probabilistic sequences

A probabilistic sequence (also called profile) is a sequence which is defined by probability vectors for each position. It is interpreted as each position having several choices (all the characters which have a corresponding non-zero probability). This is a natural generalization of matching certain regular expressions, where we additionally want to find the most likely match (match with highest likelihood) of all the possible ones.

The following is an example of a profile or probabilistic sequence of length 4 for DNA:

<table>
<thead>
<tr>
<th>Pos</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0.5</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Notice that the second and fourth positions are equivalent to a deterministic match and that the third position is equivalent to a “don’t care” match.

There are four main uses of probabilistic sequences:

- Probabilistic ancestral sequences. This can be used to build the root (or other internal nodes) of phylogenetic trees. It is the best method to obtain information about the ancestral sequence [8,11].
- Multiple sequence alignments. Multiple sequence alignments can be constructed by aligning groups of sequences against groups of sequences using dynamic programming over the probabilistic sequences representing each group [7,14].
- Patterns/Motifs. An excellent example of patterns and motifs is the PROSITE [2,5,13] database. PROSITE is a database of protein families and domains. It consists of biologically significant sites, patterns and profiles that help to reliably identify protein families and functional sites.
- Profiles. It is a common practice to summarize the common parts of a set of related sequences by a profile (probabilistic sequence) which is then used to search an
tire database. This method detects more distant homologies than by searching with sequences alone [3].

The scores that we compute in this section can be used for direct string matching or can be used in the dynamic programming alignment of sequences with indels. In both cases we need to compute the score of matching one probabilistic position against another position (given or probabilistic).

3.1. Matching a given sequence against a probabilistic sequence

Scoring an amino acid against a probabilistic sequence is not completely trivial if it is done according to the standard Markovian model of evolution. The first approach of taking the expected value of the score for each entry is not correct. A much better score is given by the logarithm of the quotient between the probability of being homologous, i.e., coming from a common ancestor divided the probability of being matched by chance (null hypothesis). This quotient is very relevant, as it allows us to compare the two events which are the main question in sequence analysis: are two given sequences descendants of the same ancestor or not? We follow the theory developed by Dayhoff et al. [6]

\[
S(v, A) = 10 \log_{10} \frac{\Pr\{\text{common ancestor}\}}{\Pr\{\text{random occurrence}\}}.
\]

Let \( v \) be the probability vector to be scored against amino acid \( A \) and \( d_1 \) the distance between \( x \) and \( v \) and \( d_2 \) the distance between \( x \) and \( A \) as shown in Fig. 2. The quotient of probabilities is

\[
\frac{\Pr\{\text{common ancestor}\}}{\Pr\{\text{random occurrence}\}} = \sum x f_x \left( \sum_i \Pr\{x \rightarrow i\} v_i \right) \Pr\{x \rightarrow A\} \left( \sum_i v_i f_i \right) f_A.
\]

![Fig. 2. Matching a probabilistic position against an amino acid.](image-url)
The probability is summed over all possible values of the ancestor \((x)\) and all possible values of the probabilistic position \((i)\). We will develop the simplification of this formula in detail, as there are several important steps in its computation.

Notice that the event of having a common ancestor is the product of three probabilities which are well visualized in Fig. 2. First we have the natural frequency of the root, \(f_x\), then the probability that \(x\) mutates to one of the \(v\)’s and finally that \(x\), independently of the previous, mutates to an \(A\).

According to the Markovian model of evolution, the probability of a mutation from \(a\) to \(b\) is given by
\[
\Pr(a \rightarrow b) = (M^d)_{ba},
\]
where \(M\) is a unit (1-PAM unit) mutation matrix and \(d\) is the distance (in PAM units) between \(a\) and \(b\). We place parenthesis around \(Md\) to emphasize that the powering is done before the selection of the subindices. A 1-PAM matrix means that the amount of expected mutation is 1% of the positions, or in mathematical terms:
\[
\sum_{i=1}^{20} (1 - M_{ii}) = 0.01.
\]

\(M^d\) is the mutation matrix corresponding to a \(d\)-PAM distance. The mutation matrices have two additional mathematical properties,

(1) the natural frequencies vector \(f\) is an eigenvector of \(M\) with eigenvalue 1
\[
Mf = f
\]
(in simple terms, the natural frequencies are not altered by a mutation event), and

(2) a symmetry property arising from the construction of these matrices:
\[
f_v M_{ba} = f_b M_{ab}.
\]

With this notation, the probabilities quotient becomes:
\[
\frac{\Pr[\text{common ancestor}]}{\Pr[\text{random occurrence}]} = \sum_i \sum_x f_x (M^{d_1})_{i_x} v_i (M^{d_2})_{xA}.
\]

Using the symmetry relation on the second \(M\) and inverting the summation we obtain:
\[
= \sum_i \sum_x v_i (M^{d_1 + d_2})_{iA} f_v
\]

The inner summation is a term of the matrix product:
\[
= \sum_i v_i (M^{d_1 + d_2})_{iA} f_v
\]

Which finally becomes
\[
S(v, A) = 10 \log_{10} \frac{\mu_A f_v}{f_v},
\]
where \( u = v^T M^{d_1 + d_2} \). Observe that the final score does not depend on \( d_1 \) and \( d_2 \) individually, but only on their sum. This is a well-known phenomenon, which indicates that what really matters is the distance between \( v \) and \( A \) and not where the root is placed.

\( S(v, x) \) can be precomputed for every amino acid \( x \) at the cost of a matrix-vector multiplication (computing \( u \)) and the same amount of space as \( v \) itself. With this pre-computation, scoring each position is \( O(1) \) and extremely efficient.

### 3.2. Matching two probabilistic sequences

Scoring a probabilistic sequence against another is done following similar lines. Fig. 3 describes the situation of scoring one particular position against another.

This time we have a triple sum, over the unknown ancestor and over each of the two probabilistic descendants, i.e.:

\[
\frac{\Pr[\text{common ancestor}]}{\Pr[\text{random occurrence}]} = \sum_i \sum_j f_i \Pr(x \to i) v_i \Pr(x \to j) w_j.
\]

Using the same techniques as before we obtain:

\[
S(v, w) = 10 \log_{10}(w^* v^*)
\]

where \( v^* = \frac{v}{f_v} \) can be precomputed for the first sequence, \( w^* = \frac{w^T M^{d_1 + d_2} F}{f_w} \) can be precomputed for the second sequence and \( F \) is a diagonal matrix containing the natural frequencies, i.e., \( F_{ii} = f_i \) (multiplying a vector by \( F \) can be done in linear time, so this multiplication does not affect the complexity). The precomputation takes as much space as the probabilistic sequences and a matrix-vector multiplication for each position \( w \). Clearly we choose \( w \) to be in the shortest of the two probabilistic sequences. The computation of the score, besides precomputation, requires an internal product of two vectors of size 4 or 20 (number of symbols). This is quite efficient since the precomputation is dwarfed by the matching phase.

[Fig. 3. Matching two probabilistic positions.]
Open problem 3. Is it possible to compute the scores more efficiently for this case?

Open problem 4. All these string matching algorithms assume a brute force underlying string searching strategy. Is it possible to improve the efficiency by applying the ideas of faster string searching algorithms [1,4,10] over these scores?

4. Conclusions

We have shown several string matching problems which are difficult because their matching depends on a scoring function which is non-trivial to compute. These problems occur naturally in the analysis of biological sequences.

We characterize the problems by the type of scoring function. In our first example, we find efficient (linear time) algorithms for any reduction function which forms a monoid.

Our second example is based on computing scores which are derived from likelihoods and we manage to reduce the complexity of the computation quite significantly.

Finally, four open problems are presented.

Acknowledgements

I would like to acknowledge the careful reading and good comments made by Gina Cannarozzi and Pedro Gonnet.

References


