数理モデル化と問題解決 22－6
（1998. 11. 26）

New Algorithms for Multiple Sequence Alignment

PAUL HORTON
REAL WORLD COMPUTING PARTNERSHIP
TSUKUBA MITSUI BLDG.
TSUKUBA, IBARAKI 305-0032, JAPAN
HORTON@RWCP.OR.JP

We have developed two new exact algorithms for gapless multiple alignment of DNA sequences. Unlike previous exact algorithms, for sufficiently short motifs, the complexity of our algorithms do not grow exponentially in the number of sequences. One algorithm, which assumes a class of scoring functions which includes the information theoretic entry scoring function, is polynomial in the number of sequences although it is exponential in the width of the motifs to be aligned. The other algorithm is super exponential in the width of the motif but linear in the number of sequences for a very general class of scoring functions.

1. Introduction

The function of DNA and protein sequences can often be characterized by the presence of important substrings or "motifs". This observation has led to the field of local multiple sequence alignment which attempts to find meaningful substrings from a collection of sequences which have a common biological function. For most problem formulations there have been no known algorithms which do not depend exponentially on the number of sequences n. Thus most work in this field has concentrated on finding good heuristic algorithms such as beam search[1], expectation maximization[2], and Gibbs Sampling[3]. A branch and bound algorithm has been developed which can solve some problem instances with more that 10 DNA sequences[4]. However, the worst case running time of the algorithm is still exponential in n and indeed the running time of the algorithm becomes unacceptable when the input sequences contain only very weak patterns.

In this paper we describe two algorithms which do not depend exponentially on n, although they depend strongly on the size of the alphabet and the width of the motif. This paper is divided into the following sections:

- Definition of Gapless Local Multiple Sequence Alignment
- Definition and Significance of Consistent Alignments
- A Specific Algorithm for Enumerating Consistent Alignments
- Description of a Separate $O(n^{w(\sigma-1)}nl)$ Algorithm
- Discussion and Conclusion

2. Definition of Gapless Local Multiple Sequence Alignment

Gapless local multiple sequence alignment, (for convenience, hereafter we will simply write local multiple alignment or sometimes just alignment), is the problem of choosing one substring from each sequence in a set of sequences such that the *score* of the chosen substrings is maximal. Throughout this paper we assume that the scoring function can be viewed as the sum of some function applied to each substring in the alignment, this is true for the likelihood ratios of sequence generating Markov Models and the information theoretic entropy scoring function. Furthermore we assume that the substrings are of fixed length w. Even given this restriction lo-

$$P = \begin{bmatrix} \text{(row a)} & 2 & 0 & 0 \\ \text{(row c)} & 0 & 0 & 1 \\ \text{(row g)} & 0 & 0 & 1 \\ \text{(row t)} & 0 & 2 & 2 \end{bmatrix}$$

```
-g ATG c
tc ATC -          score = 10
```

Fig.1 The application of a scoring matrix to an alignment of the sequences "gatgc", and "tcatc" is shown.

cal multiple alignment is difficult because the scoring function generally depends on the choice of substrings. For both algorithmic and statistical reasons most scoring functions are column independent, that is the score of an alignment is simply the sum of the scores of its columns. In general a column independent scoring function can be represented as a $\sigma \times w$ matrix P, where w is the width of the motif and σ is the size of the alphabet. The entry $P(c, i)$ gives the score of the cth character at position i. Figure 1 illustrates this definition.

3. Definition and Significance of Consistent Alignments

In this section we introduce the concept of a consistent alignment. This idea views choosing an alignment as being analogous to a tournament; just as each game in a tournament tells us something about the relative strengths of the players, each choice of a substring in an alignment tells us something about which substrings will score well.

In general the scoring function of an alignment depends on the alignment itself, however we observe that all scoring functions should be consistent in the order in which they score substrings. For example, if a scoring function chooses "at" over "ac" once, then we can conclude that it will always choose "at" over "ac". We denote this relationship as "at" *dominates* "ac". We observe that in an optimal alignment the dominance relationships implied by the choice of substrings in each sequence should be consistent. An example of an alignment which violates this observation is shown in figure 2. We refer to that kind of inconsistency as a *simple inconsistency* and note that it applies to any scoring function. For column independent scoring functions we can make a stronger observation If, as in the first sequence in figure 2, the scoring function chooses "at" over "ac" then we can conclude that the scoring function should always choose "ct" over

```
AT gac
AC atg
```

Fig.2 An example of an alignment with a simple inconsistency. "at" is favored in the first sequence over "ac" but the opposite is true for the second sequence.

"cc", "gt" over "gc", and "tt" over "tc". i.e.

$\forall x \in \{a, c, g, t\}$ xt dominates xc

Why? We know that the score in the second column for "c" must be greater than the score for "t" since score("at") is greater than score("ac") and for each of the pairs of substrings shown the character in the first column is identical, thus the order of the scores within each pair is determined by the score of the second column. This observation can be stated as a general rule. Let $s1$ be any substring chosen in an alignment and $s2$ be another substring from the same sequence, i.e. one of the ones which was not chosen; Likewise let $s3$ be any substring chosen in the alignment and $s4$ be another substring from the same sequence as $s3$. Define $I(s1, s2)$ as the set of identical positions between $s1$ and $s2$, for example I("agtc", "actc") would be {1, 4}. Likewise let $D(s1, s2)$ be the set of differing positions between $s1$ and $s2$. Now we can state our rule. If for any $s1$, $s2$, $s3$, $s4$ defined as above, $I(s1, s2) = I(s3, s4)$ and for each position in $D(s1, s2)$ $s4$ matches $s1$ while $s3$ matches $s2$ then the alignment is *inconsistent*. The significance of this observation is that regardless of the particular scoring function, we need only consider consistent alignments when searching for an optimal alignment. We must admit the possibility of different substrings having the same score, thus the strongest statement we can make is that there exists at least one optimal solution which is consistent for any reasonable scoring function.

4. Specific Algorithm for Enumerating Consistent Alignments

In this section we describe an algorithm for enumerating the consistent alignments of a set of sequences. The algorithm utilizes the fact that any consistent alignment of n sequences must be an extension of a consistent alignment of the first $n-1$ sequences. The algorithm conducts a depth first search (with pruning) on the search tree of local multiple alignment shown in figure 3. The algorithm uses a binary matrix **Dominance_table** of dimensions $\sigma^w \times \sigma^w$. As the algorithm builds up a *partial alignment* by descending the search

tree it stores information about which words dominate other words in **Dominance_table**. Pseudocode for enumerating simply consistent alignments is shown in table 1. Note that the algorithm exploits the transitivity of the dominance relationship to assert extra dominance relationships. For simplicity the pseudocode makes many copies of **Dominance_table**, however in practice it is only necessary to keep one global copy of the **Dominance_table**. Each invocation of Consistent_aux remembers the ones it added to **Dominance_table** before it makes a recursive call and resets them to zero when the call returns.

The algorithm for enumerating column independent consistent alignments is similar but whenever it asserts that a substring **s1** dominates another substring **s2** it uses a function Generate_implied_substring_pairs to generate all dominance relationships that follow directly from **s1** dominates **s2** for column independent scoring functions. Pseudo-code for the functions which need to be added to the simply consistent case to enumerate column independent consistent alignments are shown in table 2.

4.1 Efficiency of the Algorithm

For a fixed w, in the limit as the number of sequences n grows, the number of consistent alignments approaches a constant. This can be seen by considering the function Consistent_aux in table 1. Assume that the algorithm has followed a path down the search tree of alignment and is now executing Consistent_aux at some level. Let N be the set of substrings that reach the label "ADD SUBSTRING **s1** TO NEXT LEVEL". If N has only one element then there is no branching and the number of alignments which follow from that path do not increase. If $|N| > 1$ note that when reaching the label with a particular substring $s \in N$,

$\forall_{s' \neq s} \in N$, $Dominance\_table[s][s'] = 0$

This follows from the definition of N. In the loop directly after the label the relationship

$\forall_{s' \neq s} \in N$, $Dominance\_table[s][s'] \leftarrow 1$

is asserted. Thus during any invocation of Consistent_aux in which branching occurs the number of ones in **Dominance_table** increases. Since total number of ones in **Dominance_table** is bounded by $\binom{\sigma^w}{2}$, any path in the alignment search tree has a limited number of branch points even when n approaches infinity. The running time of the algorithm for each consistent alignment is linear in n and thus for a fixed w the running time is linear in n. The number of possible consistent alignments is bounded by the number of orderings of the possible substrings which is $(\sigma^w)!$. For the simply consistent case this bound seems to be reasonably tight. If we ignore the fact that overlapping substrings are not independent of each other then we can construct examples which have $(\sigma^w)!$ simply consistent alignments. In fact when $w = 1$ the example shown in figure 4 does indeed have 4! consistent alignments. For larger values of w the bound $(\sigma^w)!$ is probably quite loose for the number of column independent consistent alignments

```
aa
ac
ag
at
cc
cg
ct
gg
gt
tt
```

Fig.4 This problem instance has 4! consistent alignments for $w = 1$.

5. A Separate Algorithm Which is Polynomial in n

In this section we describe a $n^{w(\sigma-1)}nl$ time algorithm for gapless local multiple sequence alignment, where σ is the size of the alphabet, w is the motif width, n is the number of length l sequences. This running time assumes that the scoring function is column independent, that is, the score of an alignment is the sum of the scores of its columns. Moreover we assume that the score of a column depends only on the distribution of characters found in the column, for example the information theoretic entropy scoring function sums the entropy of the distribution of characters found in each column. The algorithm is based on the following observation:

Theorem1 Given the distribution of characters in the columns of an optimal alignment, that optimal alignment can be reconstructed in $O(nl)$ time.

Proof: By our assumptions stated above, the scoring function is determined by the distribution of characters in the columns of the optimal alignment. With the scoring function determined, it is easy to choose the highest scoring substring in each sequences in $O(ln)$ time.

Thus it is sufficient try all the possible distribu-

```
# All function calls are call by value.
# Consistent() prints all consistent alignments. This function is shared
# by the simple and column independent consistent case.
Consistent()
    Sequence[0...total number of input sequences-1] ← input sequences
    Dominance_table ← σ^ω × σ^ω boolean array with all entries initialized to false
    Alignment ← Empty stack
    Consistent_aux( 0, Alignment, Dominance_table )


# Consistent_aux( , , ) builds consistent alignments by depth-first search
# and then prints them. Shared by both the simple and column independent cases.
Consistent_aux( level, Alignment, Dominance_table )
    If( level = total number of sequences )
        print Alignment
        Return

    Foreach substring s1 in Sequence[ level ]
        Foreach substring s2 ≠ s1 in Sequence[ level ]
            If Dominance_table[ s2, s1 ] = true      # if s2 dominates s1
            Next s1

        # TEXT_LABEL: ADD SUBSTRING s1 TO NEXT LEVEL
        # s1 not dominated by any other substring in the current sequence,
        # so add any dominances that choosing s1 implies.
        New_dominance_table ← Dominance_table      # Create a copy.
        Foreach substring s2 ≠ s1 in Sequence[ level ]
            New_dominance_table ← Add_dominance_and_implied_dominances(
                                                New_Dominance_table, s1, s2 )

        # Now add s1 to the alignment and descend the tree.
        Push( Alignment, s1 )
        Consistent_aux( level + 1, Alignment, New_dominance_table )


# Add_dominances_and_implied_dominances doesn't do anything in the simply consistent case.
Add_dominances_and_implied_dominances( Dominance_table, good_s, bad_s )
    Return Add_dominance_and_close( Dominance_table, good_s, bad_s )


# Add good_s > bad_s dominance and any other dominances implied by transitivity.
# This function is shared by both the simple and column independent case.
Add_dominance_and_close( Dominance_table, good_s, bad_s )
    Dominance_table[ good_s ][ bad_s ] ← true
    Foreach substring better_s such that Dominance_table[ better_s, good_s ] = true
        If Dominance_table[ better_s ][ bad_s ] = false
            Dominance_table ← Add_dominance_and_implied_dominances(
                                        Dominance_table , better_s, bad_s )

    Foreach substring worse_s such that Dominance_table[ bad_s, worse_s ] = true
        If Dominance_table[ good_s ][ worse_s ] = false
            Dominance_table ← Add_dominance_and_implied_dominances(
                                        Dominance_table, good_s, worse_s )

    Return Dominance_table
```

Table 1. Pseudocode for an algorithm which prints all of the simply consistent alignments of a set of sequences.

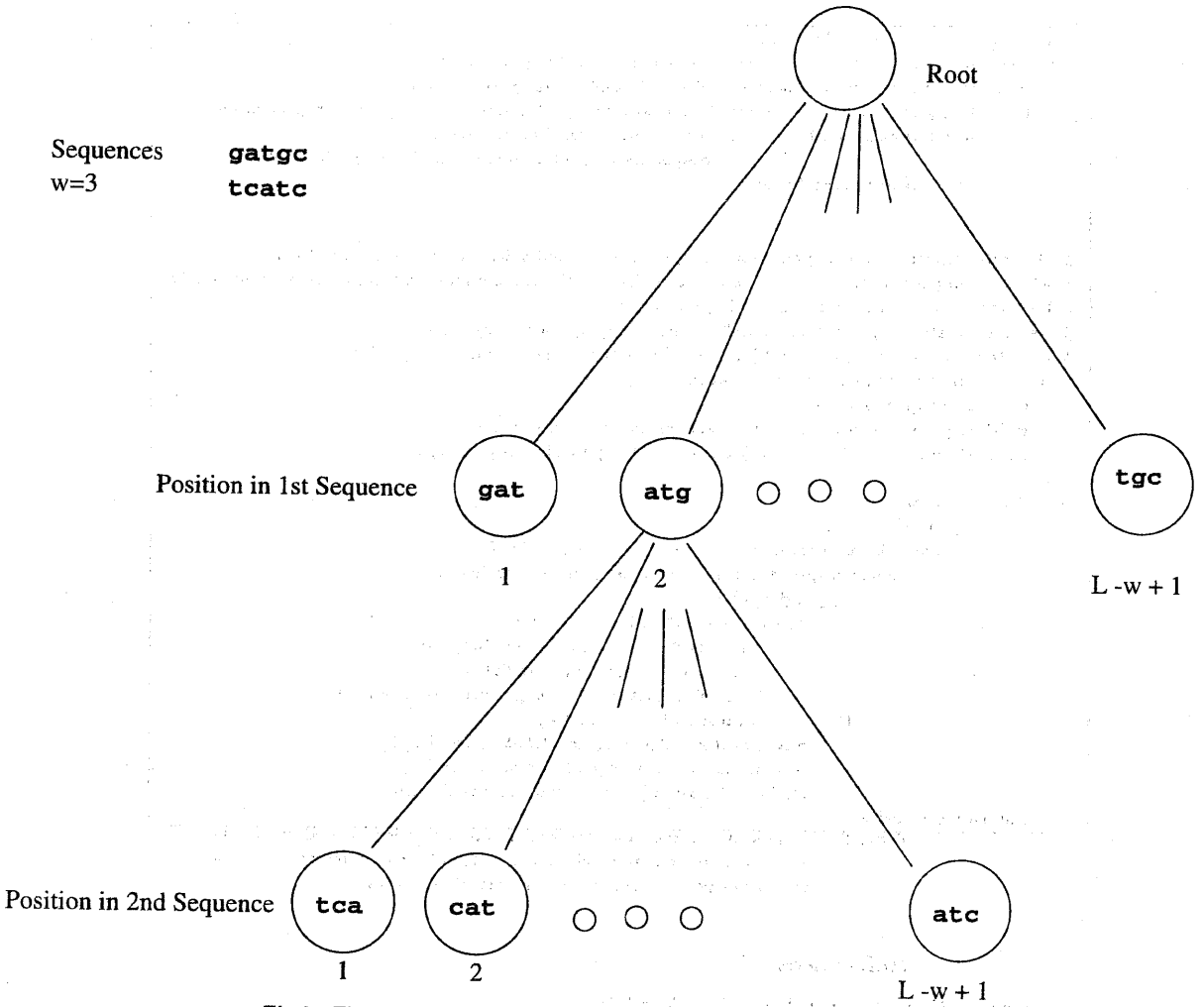

Fig.3 The search tree for local multiple alignment is demonstrated with the sequences "gatgc"and "tcatc".

tions of characters in each of the w columns. A column in a local multiple alignment is a collection of n characters from a set of size σ. Thus the number of possible distributions of characters in a single column is $\binom{n+\sigma-1}{\sigma-1}$. The number of possible combinations for w columns is $\binom{n+\sigma-1}{\sigma-1}^w$ which is $O(n^{w(\sigma-1)})$. For each of the combinations we can find the highest scoring alignment in $O(ln)$ time so the overall running time is $O(n^{w(\sigma-1)}nl)$ time.

6. Discussion

We have presented two algorithms for gapless multiple sequence alignment which, for a fixed motif size, having running times which are not exponential in the number of sequences. This is important since the dramatic improvements in sequence technology in recent years means that in the future researchers will want to align larger and larger sets of sequences. The most suitable motif width, however, is determined by characteristics of protein or DNA structure, for example the number of residues which are included in a binding site, and therefore is not expected to increase dramatically in the future.

```
# All function calls are call by value.
Add_dominance_and_implied_dominances( Dominance_table, good_s, bad_s )
    Substring_pairs_list = Generate_implied_substring_pairs( good_s, bad_s )
    Foreach  ( good_substring_element, bad_substring_element ) pair in Substring_pairs_list
        Dominance_table ← Add_dominance_and_close( Dominance_table,
                                    good_substring_element, bad_substring_element )
        Return Dominance_table


# Generate_implied_substring_pairs returns a list of the ( good substring, bad substring ) pairs
# which are implied by the good_s dominates bad_s and the column independence of the scoring matrix.
# For example Generate_implied_substring_pairs( "ata", "cga" )
# Returns ( ("ata", "cga" ), ("atc", cgc"), ("atg", "cgg"), ("att", "cgt" ) ).
# Note that the notation good_s[ 2 ] is used to denote the 2nd character of good_s.
Generate_implied_word_pairs( good_s, bad_s )
    w ← length of substrings
    Pair_list[ 0..w ] ← array of empty lists, which will hold substring pairs.
    Push( Pair_list[ 0 ], ("", "" ) ) # one element, a pair of empty strings.

    For i = 1 to w
        If( good_s[ i ] = bad_s[ i  ] )
            Foreach character c in the alphabet # {a,c,g,t} for DNA
                Foreach (good_list_s, bad_list_s) pair in Pair_list[ i - 1 ]
                    If( good_s[ i ] = bad_s[ i ] )
                        Foreach character c in the alphabet
                            new_good_s = Append( good_list_s, c )
                            new_bad_s = Append( bad_list_s, c )
                            Push( Pair_list[ i ], ( new_good_s, new_bad_s ) )
                    Else    # good_s[ i ] ≠ bad_s[ i ]
                        new_good_s = Append( good_list_s, good_s[ i ] )
                        new_bad_s = Append( bad_list_s, bad_s[ i ] )
                        Push( Pair_list[ i ], ( new_good_s, new_bad_s ) )
    Return Pair_list[ w ]
```

Table 2 Pseudocode for the functions which must be added or modified to the case of simply consistent alignments, in order to enumerate only the column independent alignments of a set of sequences.

References

1) Stormo, G. and Hartzell, G. W.: Identifying protein-binding sites from unaligned DNA fragments, *Nucleic Acids Research*, Vol. 86, pp. 1183–1187 (1989).

2) Lawrence, C. E. and Reilly, A. A.: An Expectation Maximization (EM) Algorithm for the Identification and Characterization of Common Sites in Unaligned Biopolymer Sequences, *PROTEINS*, Vol. 7, pp. 41–51 (1990).

3) Lawrence, C.E., Altschul, S.F., Boguski, M.B., Liu, J. S., Neuwald, A. F. and Wootton, J. C.: Detecting Subtle Sequence Signals: A Gibbs Sampling Strategy for Multiple Alignment, *Science*, Vol. 262, pp. 208–214 (1993).

4) Horton, P.: A Branch and Bound Algorithm for Local Multiple Alignment, *Pacific Symposium on Biocomputing '96*, pp. 368–383 (1996).