CSE 397-497: Computational Issues in Molecular Biology

Lecture 5

Spring 2004
Title: “Speech Processing for Multimedia Communications”
Speaker: Professor Joseph Olive (University of Nebraska)
Date: Thursday, February 5
Time: 4:15 pm
Place: PL 466

(Speech can be viewed as sequence data, too!)
Student lectures: important points to remember

Remember – check schedule for your lecture on Blackboard. Roughly one third of your grade in course is determined by this!

<table>
<thead>
<tr>
<th>Date</th>
<th>Lecture Topics</th>
<th>Meetings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tu 2/3</td>
<td>sequence comparison &amp; alignment – Dan Lopresti</td>
<td>AL1</td>
</tr>
<tr>
<td>Th 2/5</td>
<td>sequence comparison &amp; alignment – Dan Lopresti</td>
<td>JW1</td>
</tr>
<tr>
<td>Tu 2/10</td>
<td>TBD – Dan Lopresti</td>
<td>AL2, LN1</td>
</tr>
<tr>
<td>Th 2/12</td>
<td>TBD – Dan Lopresti</td>
<td>JW2, UM1</td>
</tr>
<tr>
<td>M 2/16</td>
<td></td>
<td>AL3</td>
</tr>
<tr>
<td>Tu 2/17</td>
<td>sequence comparison &amp; alignment – Arthur Loder</td>
<td>LN2, ST1</td>
</tr>
<tr>
<td>W 2/18</td>
<td></td>
<td>JW3</td>
</tr>
<tr>
<td>Th 2/19</td>
<td>sequence comparison &amp; alignment – Jesse Wolfgang</td>
<td>UM2, SC1</td>
</tr>
<tr>
<td>M 2/23</td>
<td></td>
<td>LN3</td>
</tr>
</tbody>
</table>

Those taking CSE 497 must send me a ranked list of top 3 student lectures you wish to scribe for by 5:00 pm on Friday.
A cartoon you'll learn to appreciate all too soon ...

The Night Before the Big Meeting Frank Receives a Visit from the PowerPoint Fairy.
Recall: basic approaches to sequence comparison

Algorithm 1 (best global)

Algorithm 2 (best prefixes)

Algorithm 3 (best local)

Algorithm 4 (best suffixes)

Algorithm 5 (semiglobal a: ignore one prefix)

Algorithm 6 (semiglobal b: ignore one suffix)

Algorithm 7 (semiglobal c: ignore one prefix & one suffix)

All are variations on same dynamic programming algorithm requiring time and space $O(mn)$. 
Nomenclature

So far, the sequence comparison measures we've been discussing can go by any of a variety of names:

- edit or evolutionary distance,
- Levenshtein distance,
- Needleman-Wunsch (biologists),
- Wagner-Fischer (computer scientists),
- Viterbi (EE's),
- Smith-Waterman (in the case of local alignments),
- word-spotting (in the case of semi-global alignments).

There is another special case worth discussing because you'll see the terminology: *longest common subsequence (LCS)*.
**Longest Common Subsequence (LCS)**

A sequence $u$ is a **common subsequence** of $s$ and $t$ if it is a subsequence of both.

- TACG is a subsequence of TATCTG
- ACG is a common subsequence of TATCTG and ACTGA

A sequence $u$ is a **longest common subsequence (LCS)** of $s$ and $t$ if it is at least as long as any subsequence of $s$ and $t$.

- ACTG is an LCS of TATCTG and ACTGA
Longest Common Subsequence (LCS)

How does longest common subsequence relate to the notion of an alignment and the optimal dynamic programming algorithm?

... ignore non-match pairings ...

Then same recurrence computes LCS:

\[
a[i, j] = \max\left\{\begin{array}{l}
a[i-1, j] \\
a[i, j-1] \\
a[i-1, j-1] + c_{sub}(s[i], t[j])
\end{array}\right. \quad 1 \leq i \leq m, 1 \leq j \leq n
\]

We only care about total number of matches.
So set \( c_{sub} = 1 \) when symbols match, and set all other costs to 0.
“The String-to-String Correction Problem with Block Moves,”
W. F. Tichy, ACM Transactions on Computer Systems,

Title is bit of misnomer. Really, Tichy is concerned with a simplified editing model that involves a single operation.

For two strings $s[1]s[2]...s[m]$ and $t[1]t[2]...t[n]$, a block move is a triple $(p,q,l)$ such that $s[p]...s[p+l-1] = t[q]...t[q+l-1]$.

In other words, a substring of length $l$ starting at index $p$ in $s$ that corresponds exactly to a substring starting at index $q$ in $t$.

For example, $(0,2,2)$ is a block move from TACG to CCTATC.
For two strings $s$ and $t$, a *covering set* (of $t$ with respect to $s$) is a set of block moves such that every symbol $t[i]$ that also appears in $s$ is the target of exactly one block move.

A trivial covering set consists of block moves of length 1, one for each symbol in $t$ that appears in $s$.

The goal is to find a *minimal covering set*, i.e., a covering set that is no larger than any covering set for the two strings.

Note how this differs significantly from the notion of an alignment.

**covering set** = \{(1,0,1), (3,1,3), (1,4,1)\}

Covering sets
Covering sets vs. LCS

Tichy notes that traditional alignments (and LCS) don't allow:

1. Crossings:
   - LCS: $S = a b c d e$
   - Covering set: $S = a b c$

2. Re-use of substrings:
   - LCS: $S = a b c$
   - Covering set: $S = a b c$

Repeated application of LCS doesn't help, either:

- Repeated LCS: $S = a b c d e a$, $T = c d a b$
- Covering set: $S = a b c d e a$, $T = c d a b$
Algorithm for computing minimal covering set

Tichy's algorithm for computing a minimal covering set:

1. Set $j = 0$.

2. Find longest substring beginning at $t[j]$ that appears in $s$.

3. If no such substring exists (i.e., if $t[j]$ is not in $s$), then $j++$.

4. If such a substring of length $l$ does exist, record it as a block move, set $j = j + 1$, and return to step 2.

5. Repeat until all of $t$ is consumed.

Note: step 2 is only non-trivial step. Going beyond naïve implementation, there are several smart ways to do this.
Computing minimal covering sets

Example:

Step 1:

\[ S = v \ w \ v \ w \ x \ y \]
\[ T = |z \ v \ w \ x \ w \]  
longest block move starting with \( T[0] \): none

Step 2:

\[ S = v \ w \ v \ w \ x \ y \]
\[ T = z \ |v \ w \ x \ w \]  
longest block move starting with \( T[1] \): (2, 1, 3)

Step 3:

\[ S = v \ w \ v \ w \ x \ y \]
\[ T = z \ v \ w \ x \ |w \]  
longest block move starting with \( T[4] \): (1, 4, 1)

Time complexity is \( O(mn) \), space complexity is \( O(m + n) \).

Time complexity can be reduced to linear using suffix trees (Arthur will cover suffix trees as part of his lecture on Feb. 17).
How do we know this is yields optimal solution?

Informal proof of correctness (for real proof, see paper):

Look at resulting string $t$:

\[ \ldots X(\ldots)X(\ldots)(\ldots)X(\ldots)(\ldots)(\ldots)X\ldots \]

- $X$ = unmatched symbols
- $(\ldots)$ = symbols matched in single block move

The only way this can fail to be a minimal covering set is if there's a way to coalesce some region of $k > 1$ blocks into $(k - 1)$ or fewer blocks.
**How do we know this is yields optimal solution?**

What properties do we know about blocks in such a region?

1. Each block consists of consecutive symbols,
2. Each block is maximal,
3. Blocks themselves are consecutive.

How could three blocks become two blocks, for example?

<table>
<thead>
<tr>
<th>Case 1:</th>
<th>a block in new matching subsumes prefix of next block in old matching.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( . . . . ) ( . . . ) ( . . ) ( . . . . . . . ) ( . . . )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case 2:</th>
<th>a block in new matching subsumes suffix of previous block in old matching.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( . . . ) ( . . . ) ( . . ) ( . . ) ( . . . . . . . . . . )</td>
</tr>
</tbody>
</table>

Both of these would violate condition (2). QED.
What are the advantages of Tichy's model?

- Allows for a different kind of operation that seems like it might be biologically relevant.
- An efficient algorithm exists.

What are the limitations of Tichy's model?

- Not really full editing – all of $t$ may not be accounted for.
- Blocks themselves must match exactly – no differences allowed between corresponding blocks.
“Block Edit Models for Approximate String Matching,”

Introduces concept of *block edit distance*:

two strings are compared by optimally extracting sets of substrings and placing them into correspondence.

E.g., these could represent biological “motifs” shared between two genetic sequences but appearing in an unknown order.
What are we looking for? An optimal way to solve this problem:

The quick brown fox jumps over the lazy dog.

String A

Jump over the brown fox, lazy dog. Quick!

String B

• Don't know block boundaries in advance (to be determined as part of optimization).
• Blocks don't have to be matched in sequential order.
• Allow lower-level editing between blocks (don't depend on finding exact matches).
File comparison (copy-and-paste with editing)

This is some text ...

And yet more text ...

Still more text ...

This is some text ...

Still more text ...

And yet more text ...

Still more text ...
**Hand-drawn sketch matching**

**Sketch Level** = block editing

```
<table>
<thead>
<tr>
<th>house</th>
<th>tree</th>
<th>car</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>car'</th>
<th>tree'</th>
<th>house'</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Motif Level** = approximate string matching (time warping)

```
<table>
<thead>
<tr>
<th>frame</th>
<th>roof</th>
<th>door</th>
<th>window</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>frame'</th>
<th>roof'</th>
<th>door'</th>
<th>window'</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Stroke Level** = VQ or elastic distance

```
<stroke type>
(x_1, y_1) (x_2, y_2) ... (x_n, y_n)
```

```
<stroke type>'
(x_1, y_1)' (x_2, y_2)' ... (x_n, y_n)'
```
“Global dynamic programming alignments of such rearranged sequences yield unpredictable, evolutionarily confusing results ...

Global alignment methods are generally incapable of dealing with intrasequence rearrangements, yet this phenomenon is quite common ...

Dot matrix analysis is the only currently available tool that deals sensibly with this phenomenon.”

Related work

Tichy '84

Extends LCS to determine minimal covering set for one of the strings using block moves (i.e., requires exact matches between blocks).


Waterman & Eggert '87

Locates best local alignment, then iterates process to determine next-best alignment that doesn’t share any pairings with previous one (“greedy” algorithm).

Block edit distance: preliminaries

Let $A = a_1a_2...a_m$ be a string over a finite alphabet.

$A \mid_t \equiv t$-block substring family of $A$ (multiset of $t$ substrings).

String A: The quick brown fox jumps over the lazy dog.

Some substring families:

- quick  brown fox  jumps over the  lazy dog.
- The quick  quick brown fox  jumps over  the lazy dog.
- The quick brown  fox jumps over  the lazy dog.
Block edit distance: preliminaries

Cover ≡ each symbol appears in some substring

Disjoint ≡ substrings in family do not overlap

Variations:

- **C**: substring family must be a cover (more restrictive)
- **C'**: substring family need not be a cover (less restrictive)
- **D**: substring family must be disjoint (more restrictive)
- **D'**: substring family need not be disjoint (less restrictive)

A block edit problem is specified by a tuple:

\[ \{C,C'\} \times \{D,D'\} - \{C,C'\} \times \{D,D'\} \]
So, for example,

String A
The quick brown fox jumps over the lazy dog.

String B
Jump over the brown fox, lazy dog. Quick!

represents a specific ____________ matching.
Block edit distance: definition

Let \( \text{dist} \) be an underlying distance function between two substrings \( A_{i,j} \) and \( B_{k,l} \):

\[
\text{dist}: \{i, j \mid 1 \leq i \leq j \leq m\} \times \{k, l \mid 1 \leq k \leq l \leq n\} \rightarrow \mathbb{R}
\]

\( \text{dist} \) could be standard string edit distance, for example.

Then block edit distance between \( A \) and \( B \) is defined as:

\[
\text{bdist}(A, B) \equiv \min_t \min_{|A|, |B|} \min_{\sigma \in S(t)} \left\{ t \cdot c_{\text{block}} + \sum_{i=1}^{t} \text{dist}(A^{(i)}, B^{\sigma(i)}) \right\}
\]

where \( S(t) \) is the permutation group on \( t \) elements.
How hard is it to compute block edit distance?

<table>
<thead>
<tr>
<th>String A</th>
<th>C'D'</th>
<th>C'D</th>
<th>CD'</th>
<th>CD</th>
</tr>
</thead>
<tbody>
<tr>
<td>C'D'</td>
<td>O(m^2n^2)</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>C'D</td>
<td>O(m^2n)</td>
<td>NP-complete</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>CD'</td>
<td>O(m^2n^2)</td>
<td>NP-complete</td>
<td>NP-complete</td>
<td>*</td>
</tr>
<tr>
<td>CD</td>
<td>O(m^2n)</td>
<td>NP-complete</td>
<td>NP-complete</td>
<td>NP-complete</td>
</tr>
</tbody>
</table>

* By symmetry, C'D'-CD is the same problem as CD-C'D', etc.
CD-CD block edit distance is most restrictive form of problem.

Sketch of proof (for full proof, see paper): reduction is from uniprocessor scheduling [GJ79].

Instance: Set $T$ of jobs and, for each Job $j \in T$,

- a length $l(j) \in \mathbb{Z}^+$
- a release time $r(j) \in \mathbb{Z}_0^+$
- a deadline $d(j) \in \mathbb{Z}^+$

Question: Is there a uniprocessor schedule for $T$ that satisfies release time constraints and meets all deadlines?

CD-CD block edit distance is NP-complete

Example:

<table>
<thead>
<tr>
<th>Job</th>
<th>Length</th>
<th>Release</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Details are intricate because you must construct job substrings so that they can only be matched at appropriate time slots.

With this construction, solution to CD-CD block edit distance problem would imply solution to uniprocessor scheduling. QED.
Poly-time algorithm for CD-C'D' block edit distance

If at least one of the substring families is unconstrained, we can solve the problem in polynomial time.

We define matrix $W^1$ for $1 \leq i \leq j \leq m$ as:

$$W^1(i, j) = \min_{k \leq l} \{ \text{dist}(a_i \ldots a_j, b_k \ldots b_l) \}$$

That is, $W^1$ records the cost of the best possible match between $a_i \ldots a_j$ and any substring of $B$. 
Poly-time algorithm for CD-C'D' block edit distance

\[ W^1(3,8) = \text{cost of best match between } a_3...a_8 \text{ and any substring of } B \]

String A
Poly-time algorithm for CD-C'D' block edit distance

\[ M(i) = \text{cost of best block match between } a_1...a_i \text{ and all of } B. \]

Once we have computed \( M(i) \) for \( i = 1, 2, ..., m \), our final answer is \( bdist(A, B) = M(m) \).

As in past, consider index \( i \):

\[
\begin{array}{cccccccc}
\text{String A} & \text{W1}^{j+1,i} \\
\hline
a_1 & a_2 & \cdots & a_j & a_{j+1} & \cdots & a_i & \cdots & a_{m-1} & a_m \\
\end{array}
\]

Final block ends there and starts at some index \( j \).

Can be solved using dynamic programming:

\[
M(i) = \min_{j<i} \left\{ M(j) + W^1(j+1,i) \right\}
\]

... and takes time \( O(m^2) + T(W^1) \).
Recall that
\[ W^1(i, j) \equiv \min_{k \leq l} \{ \text{dist}(a_i \ldots a_j, b_k \ldots b_l) \} \]

Naively, we must ...

- ... fill in \( O(m^2) \) entries by ...
- ... comparing \( O(n^2) \) values ...
- ... each of which takes time \( O(mn) \) to compute when \( \text{dist} \) is standard edit distance.

Which yields an \( O(m^3n^3) \) algorithm (ugh).
Fortunately, $W$ can be computed much more efficiently:

1. As we know, local comparison modification of the dynamic programming algorithm allows best match in $B$ for a fixed substring in $A$ to be found in time $O(mn)$. This saves $O(n^2)$.

2. Also, table generated for matching $a_i...a_n$ to $B$ contains information about best substring matches for all prefixes $a_i...a_k$ as well, for $i \leq k \leq n$. Hence, only $O(m)$ such tables need be built, saving another $O(m)$.

Thus, $T(W) = O(m^2n)$, which is also determines overall time.
Wrap-up

Readings for next time:

• Section 3.5 in your textbook.


Remember:

• Come to class prepared to discuss what you have read.

• Check Blackboard regularly for updates.