CSE 397-497: *Computational Issues in Molecular Biology*

Lecture 5

Spring 2004

ECE seminar this afternoon

Title: “Speech Processing for Multimedia Communications”

Speaker: Professor Joseph Olive (University of Nebraska)

Date: Thursday, February 5

Time: 4:15 pm

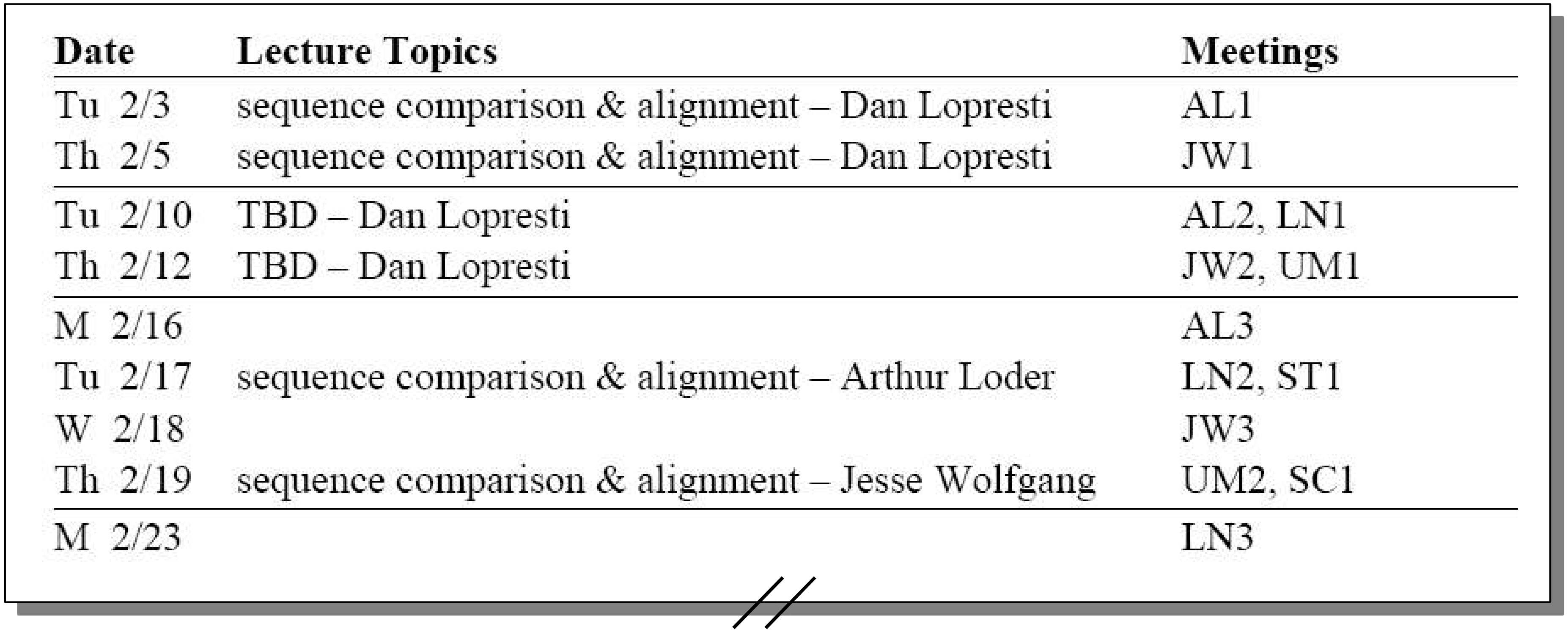
Place: PL 466

(Speech can be viewed as sequence data, too!)

Student lectures: important points to remember

Remember – check schedule for your lecture on Blackboard.

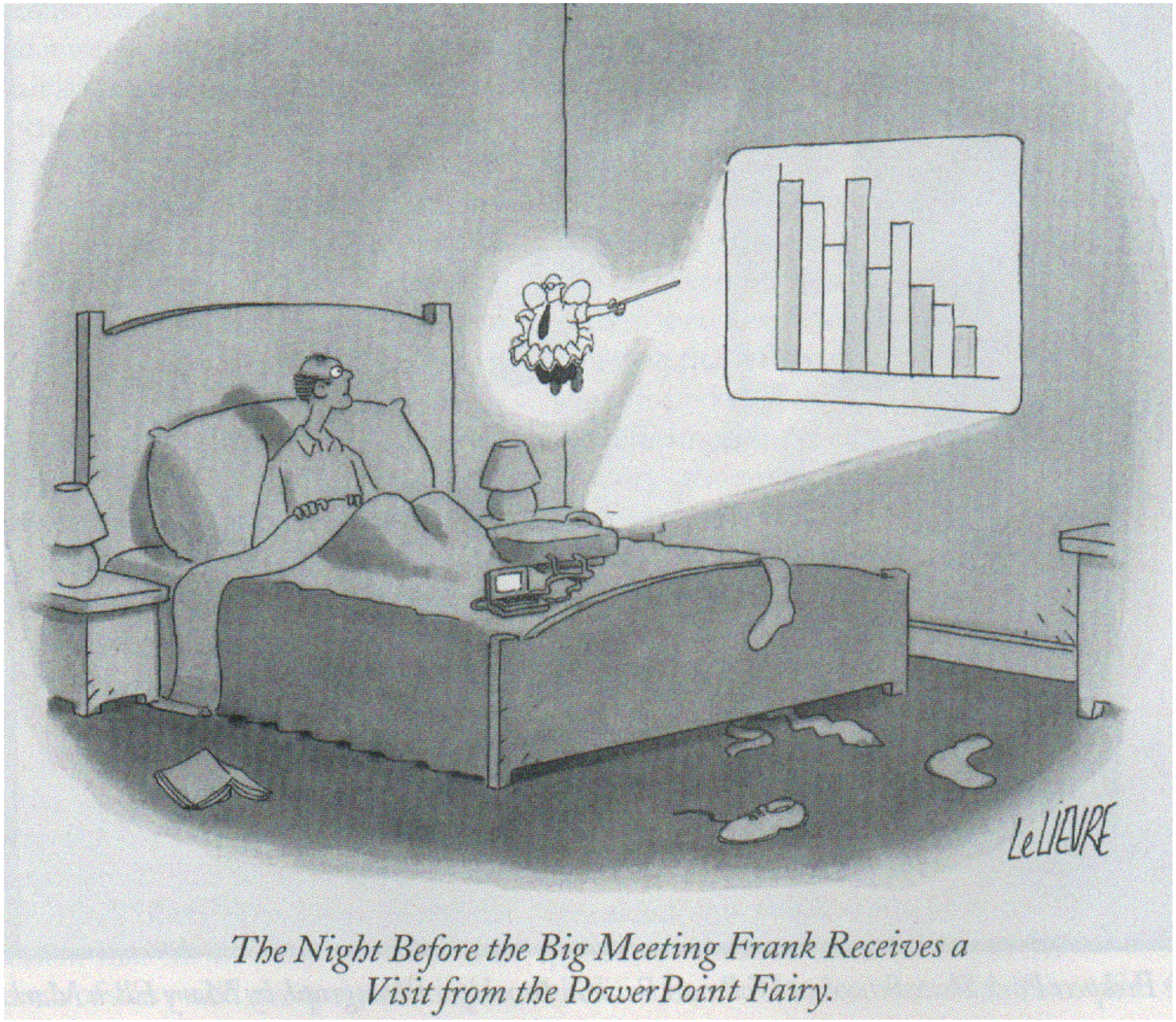
Roughly one third of your grade in course is determined by this!



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

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.

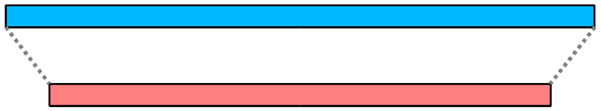
A cartoon you'll learn to appreciate all too soon ...



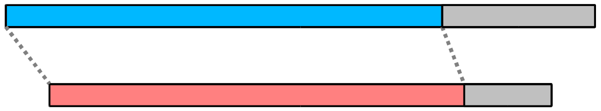
The Night Before the Big Meeting Frank Receives a Visit from the PowerPoint Fairy.

Recall: basic approaches to sequence comparison

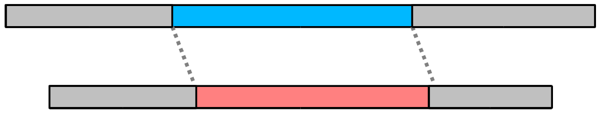
Algorithm 1 (best global)



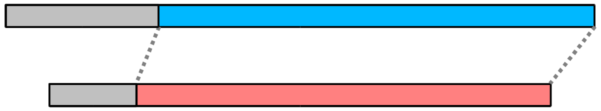
Algorithm 2 (best prefixes)



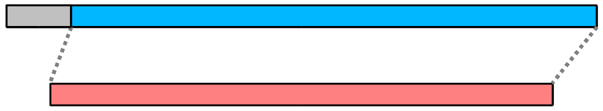
Algorithm 3 (best local)



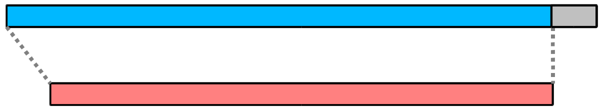
Algorithm 4 (best suffixes)



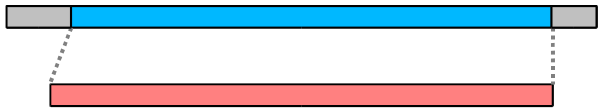
Algorithm 5 (semiglobal a: ignore one prefix)



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 $O(mn)$.

LEHIGH UNIVERSITY

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)

subsequence	sequence *u* that can be obtained from *s* by removing some symbols.

`TACG` is a subsequence of `TATCTG`

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

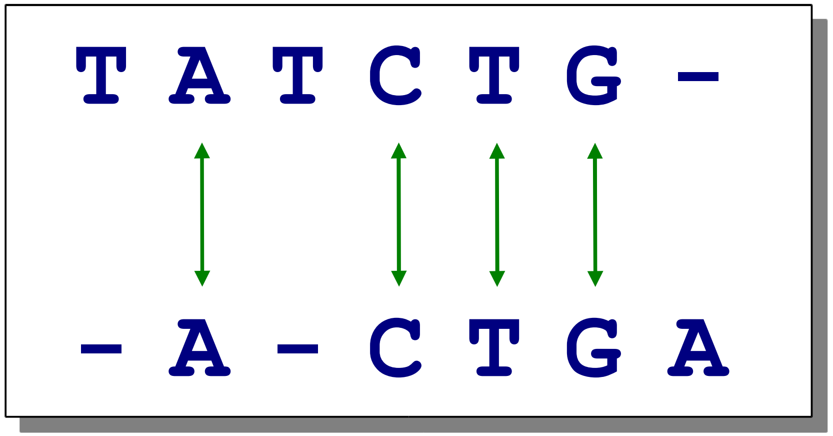
`ACG` is a common subsequence of `TATCTG` and `ACTGA`

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?



We only care about total number of matches.

So set c_{sub} = 1 when symbols match, and set all other costs to 0.

... ignore non-match pairings ...

Then same recurrence computes LCS:

$$a[i,j] = \max \begin{cases} a[i-1,j] \\ a[i,j-1] \\ a[i-1,j-1]+c_{\text{sub}}(s[i],t[j]) \end{cases} \qquad 1 \le i \le m,\ 1 \le j \le n$$

Paper: block moves and covering sets (Tichy '84)

"The String-to-String Correction Problem with Block Moves," W. F. Tichy, *ACM Transactions on Computer Systems*, 2(4):309-321, November 1984.

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

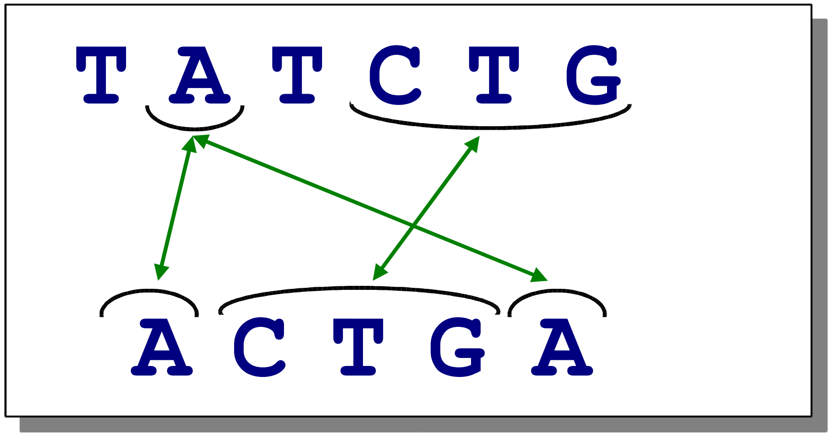
For two strings $s[1]s[2]...s[m]$ and $t[1]t[2]...t[n]$, a *block move* is a triple (p,q,l) such that $s[p]...s[p+l-1] = t[q]...t[q+l-1]$.

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`**.

Covering sets

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.



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

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

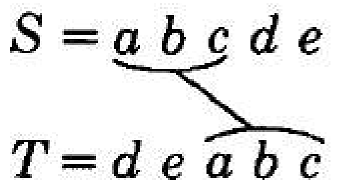
A trivial covering set consists of block moves of length 1, one for each symbol in *t* that appears in *s*.

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.

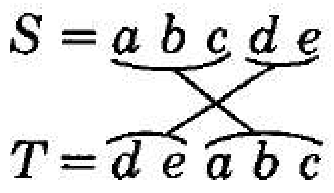
Covering sets vs. LCS

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

(1) crossings:

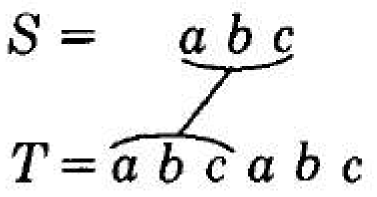


LCS

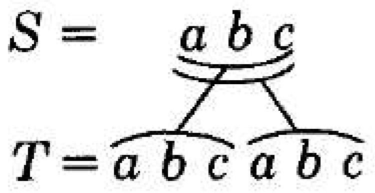


covering set

(2) re-use of substrings:

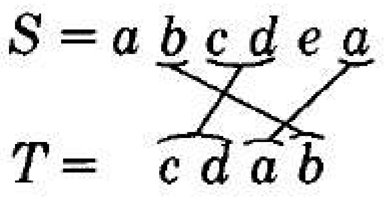


LCS

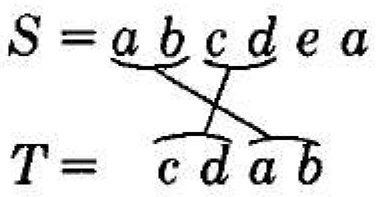


covering set

Repeated application of LCS doesn't help, either:



repeated LCS



covering set

Algorithm for computing minimal covering set

Tichy's algorithm for computing a minmal 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*:

```
...X(...)X(...)(...)X(...)(...)(...)X...
```

`X`	= unmatched symbols
`(...)`	= 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?

Case 1:

a block in new matching subsumes prefix of next block in old matching.

```
( . . . . ) ( . . . ) ( . . )
( . . . . . . . ) ( . . . . )
```

Case 2:

a block in new matching subsumes suffix of previous block in old matching.

```
( . . . . ) ( . . . ) ( . . )
( . . ) ( . . . . . . . . . )
```

Both of these would violate condition (2). QED.

Discussion: block moves and covering sets (Tichy '84)

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.

Paper: block editing (Lopresti & Tomkins '97)

"Block Edit Models for Approximate String Matching,"
D. Lopresti and A. Tomkins, *Theoretical Computer Science*, vol. 181, no. 1, 1997, pp. 159-179.

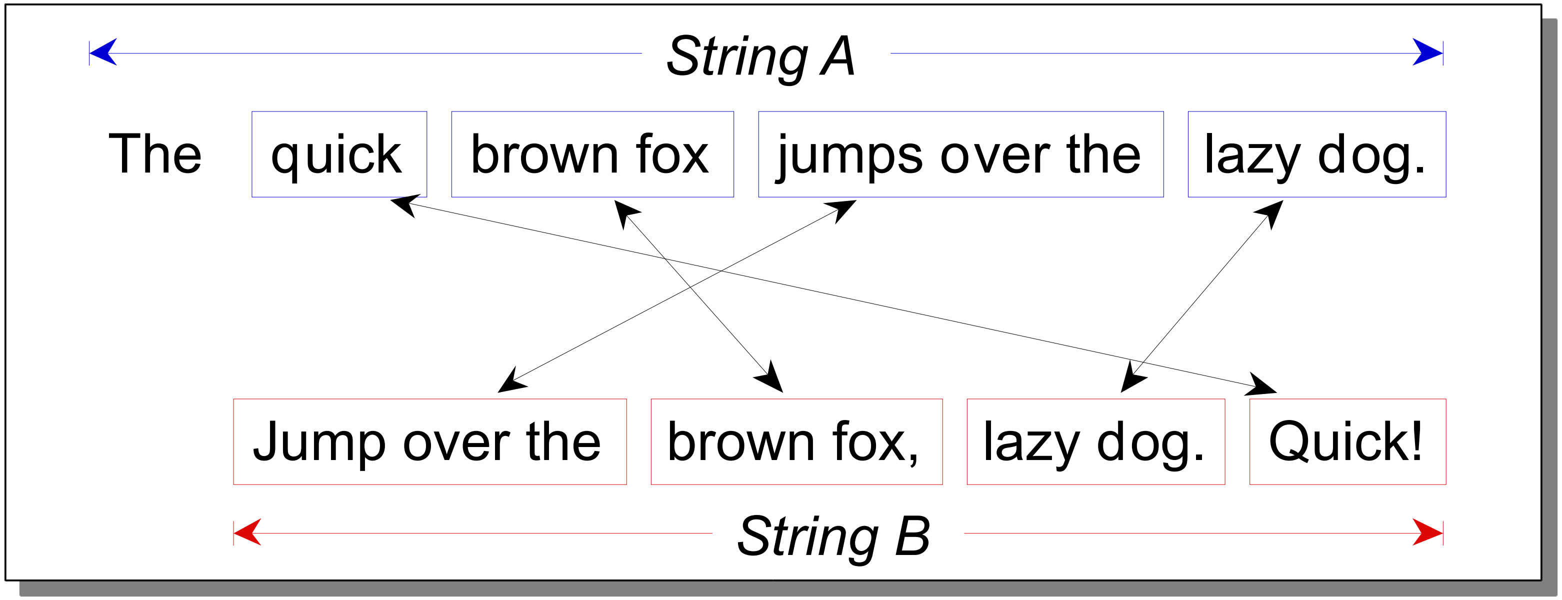
Introduces concept of *block edit distance*:

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.

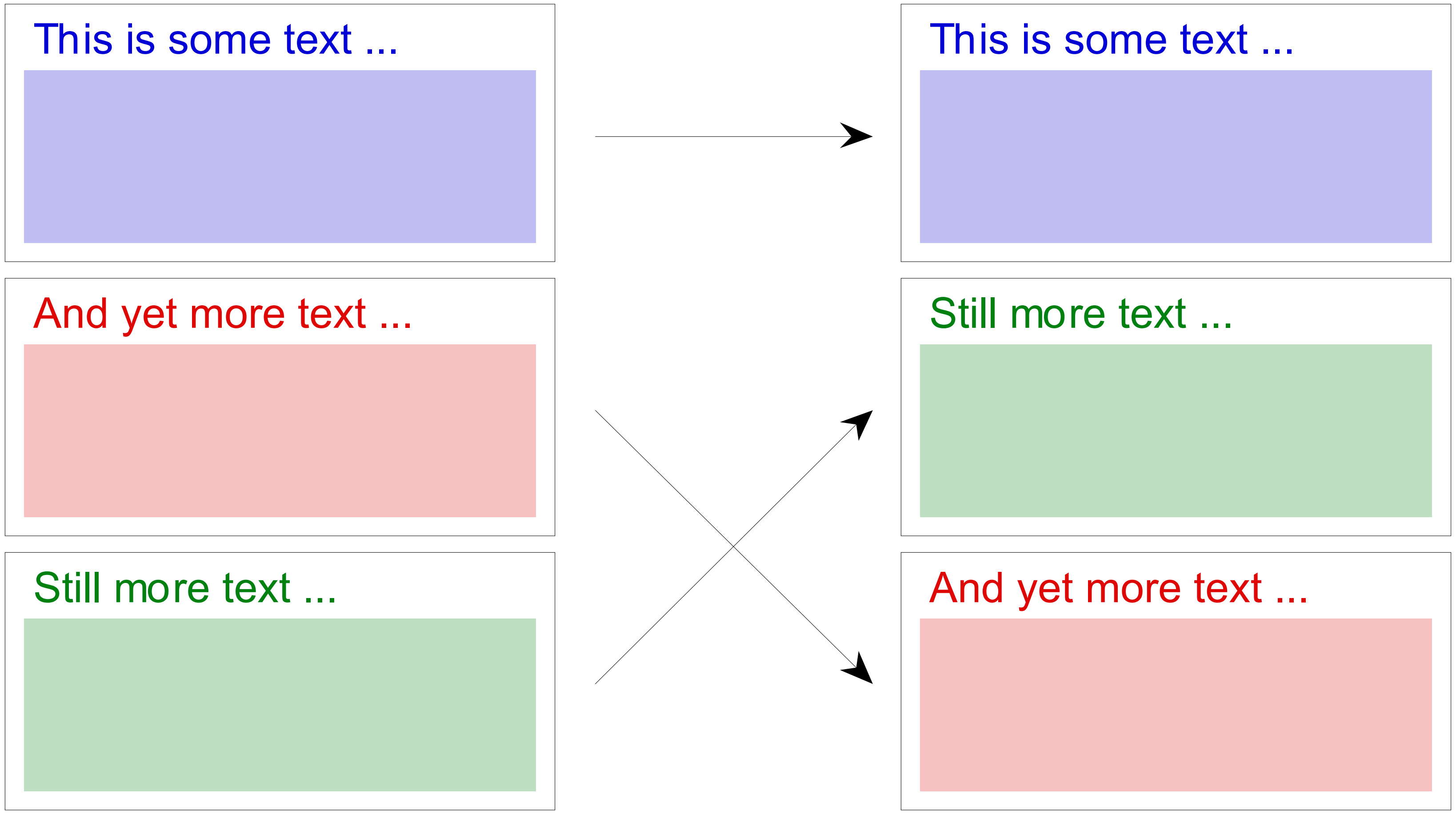
Block edit distance

What are we looking for? An optimal way to solve this problem:



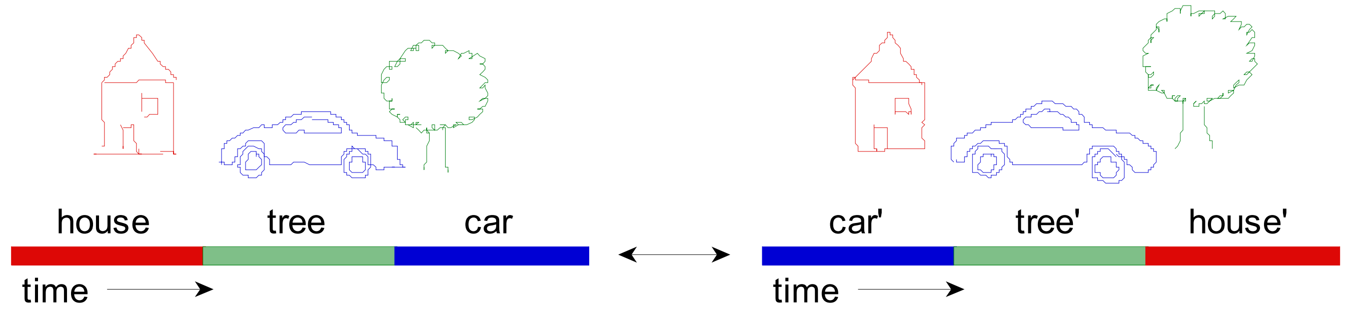
- Don't know block boundaries in advance (to be determined as part of optimization).
- Blocks don't have to be matched in sequential order.
- Allow lower-level editing between blocks (don't depend on finding extact matches).

File comparison (copy-and-paste with editing)

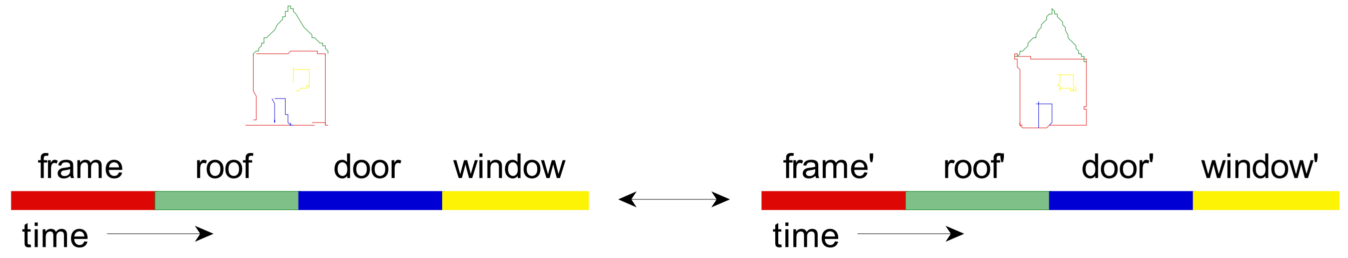


Hand-drawn sketch matching

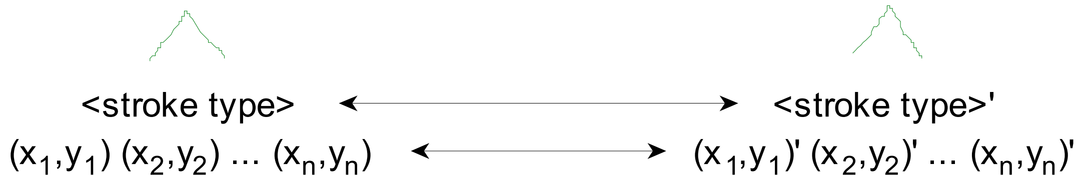
Sketch Level = block editing



Motif Level = approximate string matching (time warping)



Stroke Level = VQ or elastic distance


<stroke type>
$(x_1,y_1)\ (x_2,y_2)\ \ldots\ (x_n,y_n)$
<stroke type>'
$(x_1,y_1)'\ (x_2,y_2)'\ \ldots\ (x_n,y_n)'$

Higher

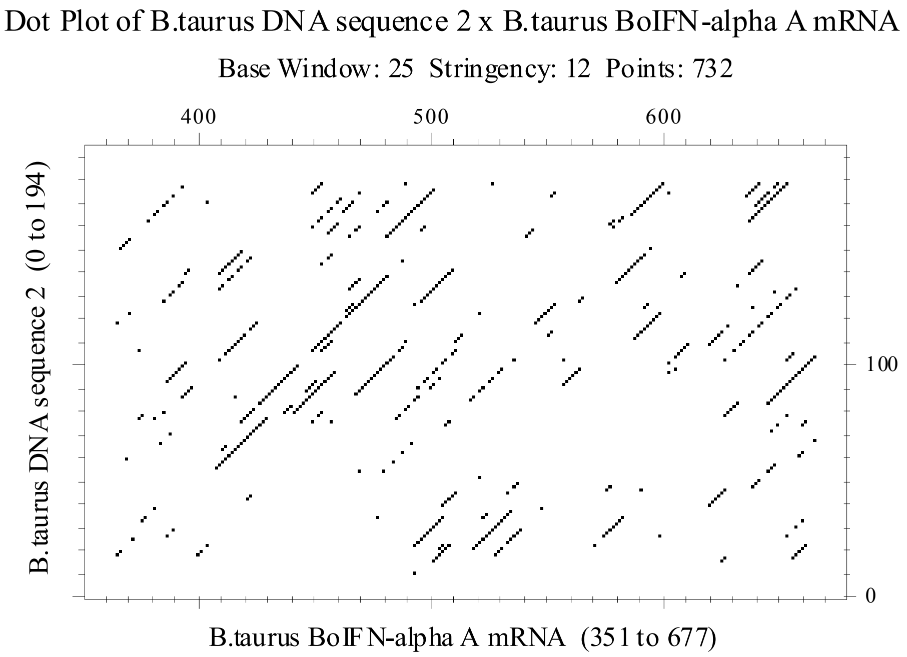
Lower

From molecular biology

"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."



M. Gribskov and J. Devereux, *Sequence Analysis Primer*, 1991, pg. 96.

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).

“The String-to-String Correction Problem with Block Moves,” W. F. Tichy, *ACM Transactions on Computer Systems*, vol. 2, 1984, pp. 309-321.

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).

“A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons,” M. S. Waterman and M. Eggert, *Journal of Molecular Biology*, vol. 197, 1987, pp. 723-728.

Block edit distance: preliminaries

Let $A = a_1 a_2 ... a_m$ be a string over a finite alphabet.

$A|_t \equiv$ *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:

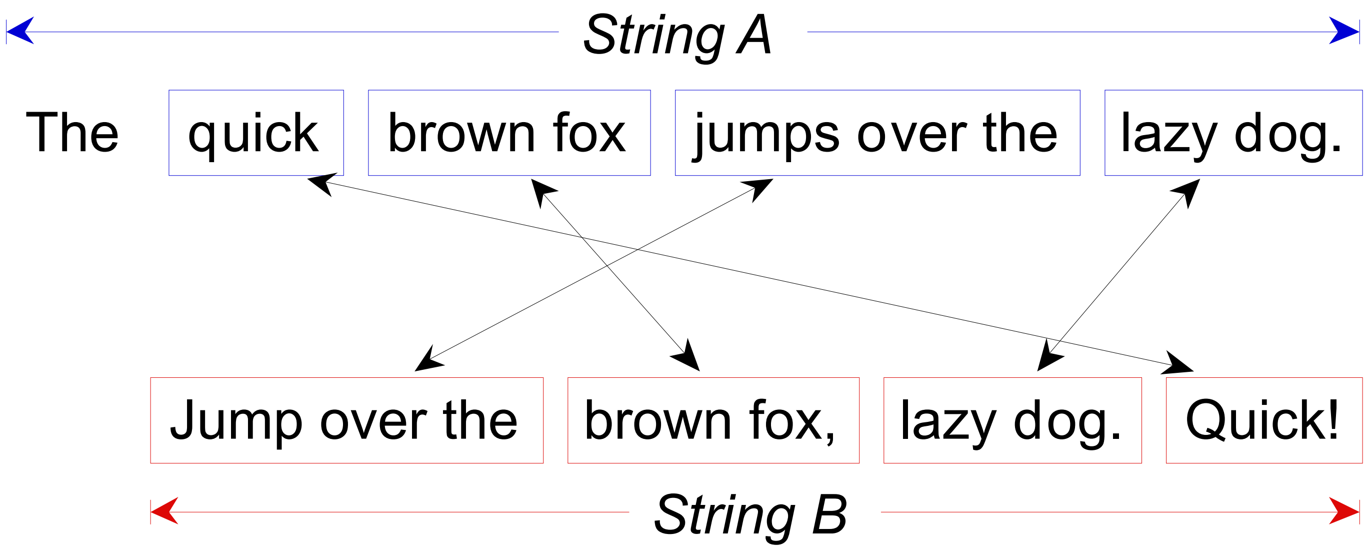
C	substring family must be a cover	← more restrictive
C'	substring family need not be a cover	← less restrictive
D	substring family must be disjoint	← more restrictive
D'	substring family need not be disjoint	← less restrictive

A block edit problem is specified by a tuple:

$$\{C,C'\} \times \{D,D'\} – \{C,C'\} \times \{D,D'\}$$

Block edit distance: preliminaries

So, for example,



represents a specific __C'D-CD__ matching.

Block edit distance: definition

Let *dist* be an underlying distance function between two substrings $A_{i,j}$ and $B_{k,l}$:

$$dist: \{i,j \mid 1 \leqslant i \leqslant j \leqslant m\} \times \{k,l \mid 1 \leqslant k \leqslant l \leqslant n\} \rightarrow \mathbb{R}$$

dist could be standard string edit distance, for example.

Then *block edit distance* between A and B is defined as:

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

where $S(t)$ is the permutation group on *t* elements.

Block edit distance: summary of results

How hard is it to compute block edit distance?

String A \ *String B*	C'D'	C'D	CD'	CD
C'D'	$O(m^2n^2)$	*	*	*
C'D	$O(m^2n)$	NP-complete	*	*
CD'	$O(m^2n^2)$	NP-complete	NP-complete	*
CD	$O(m^2n)$	NP-complete	NP-complete	NP-complete

* By symmetry, C'D'-CD is the same problem as CD-C'D', etc.

CD-CD block edit distance is NP-complete

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$_t$ $\in$ T,

- a length $l(t) \in Z+$
- a release time $r(t) \in Z_0+$
- a deadline $d(t) \in Z+$

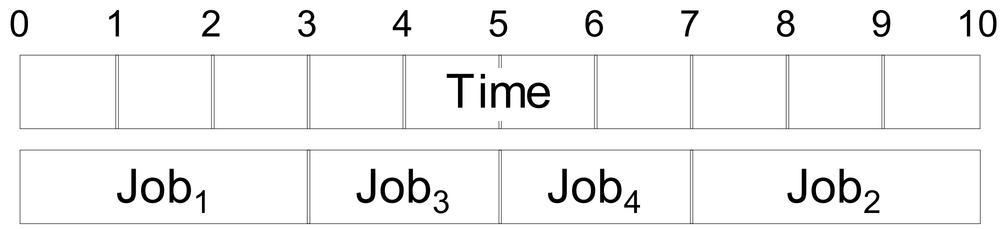
Question: Is there a uniprocessor schedule for T that satisfies release time constraints and meets all deadlines?

A Guide to the Theory of NP-Completeness, M. R. Garey and D. S. Johnson, W.H. Freeman and Company, 1979.

CD-CD block edit distance is NP-complete

Example:

Job	*Length*	*Release*	*Deadline*
1	3	0	4
2	3	1	10
3	2	2	8
4	2	5	8



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.

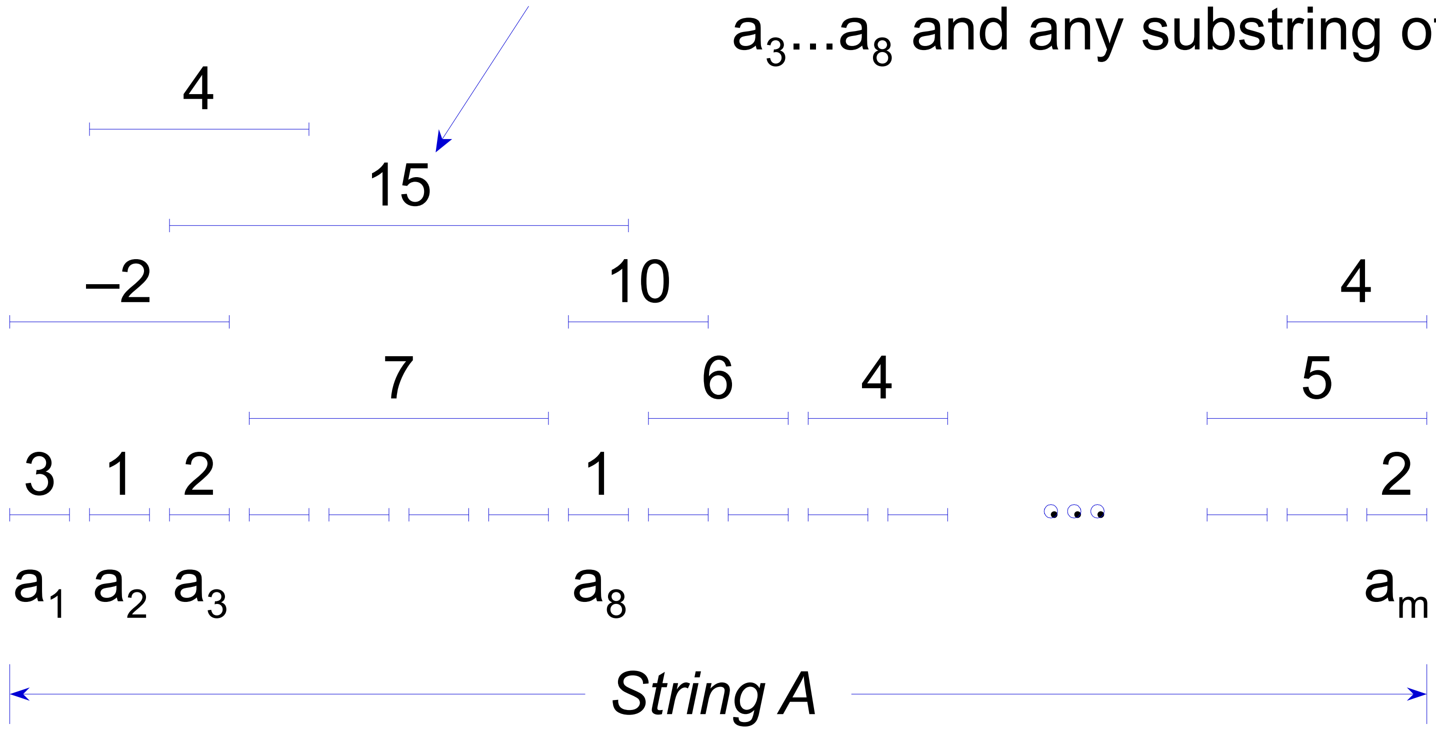
We define matrix W^1 for $1 \leq i \leq j \leq m$ as:

$$W^1(i,j) \equiv \min_{k \leqslant l} \left\{ dist(a_i \ldots a_j, b_k \ldots b_l) \right\}$$

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

$W^1(3,8)$ = cost of best match between $a_3 \ldots a_8$ and any substring of B



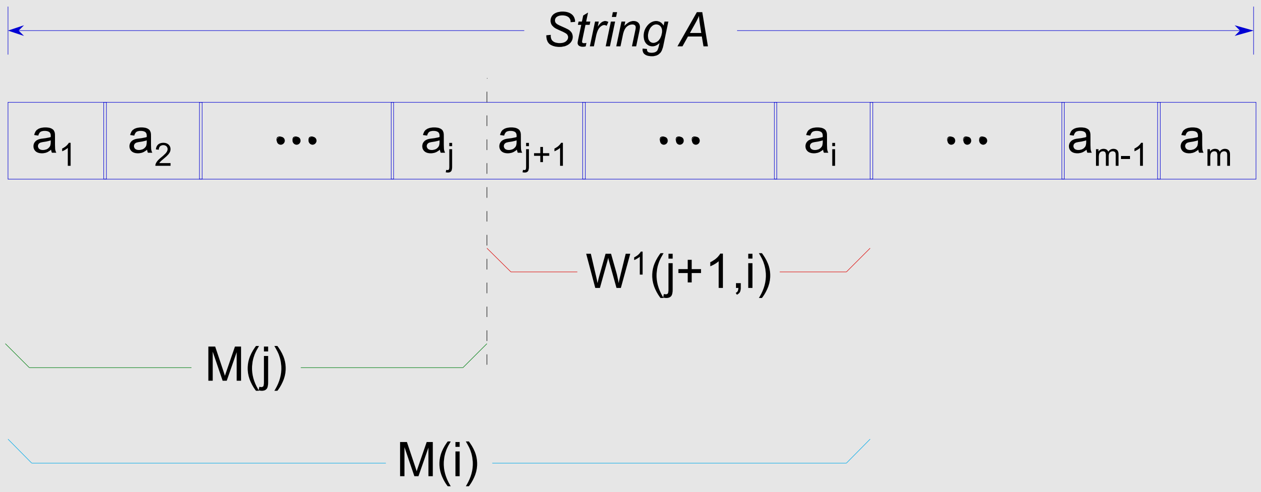
Poly-time algorithm for CD-C'D' block edit distance

$M(i) \equiv$ cost of best block match between $a_1 \ldots a_i$ 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*:

Final block ends there and starts at some index *j*.



Can be solved using dynamic programming:

$$M(i) = \min_{j<i} \{ M(j) + W^1(j+1, i) \}$$

... and takes time $O(m^2) + T(W^1)$.

Computing W

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

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 *dist* is standard edit distance.

Which yields an $O(m^3 n^3)$ algorithm (ugh).

Computing W

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

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

Wrap-up

Readings for next time:

- Section 3.5 in your textbook.
- "Rapid and Sensitive Protein Similarity Searches," D. J. Lipman and W. R. Pearson, *Science*, vol. 227, no. 4693, 1985, pp. 1435-1441.
- "GenBank," D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell and D. L. Wheeler, *Nucleic Acids Research*, 2003, Vol. 31, No. 1, pp. 23-27.

Remember:

- Come to class prepared to discuss what you have read.
- Check Blackboard regularly for updates.