BMI-203: Biocomputing Algorithms
Lecture 6: Dynamic Programming

Adapted from:

Ajay N. Jain, PhD

Professor
Bioengineering and Therapeutic Sciences

University of California, San Francisco

ajain@jainlab.org

http://www.jainlab.org

Updated Feb 2018, Michael Keiser

Outline

- Dynamic Programming (see Cormen, Chapter 15)
- Sequence alignment and similarity

  Needleman, S.B. and Wunsch, C.D. 1970. A General Method Applicable to the Search for Similarities in Amino Acid Sequence of Two Proteins. JMB, 48: 443-453.

  Smith, T.F. and Waterman, M.S. 1981. Identification of common molecular subsequences, JMB 147: 195-197.

  Altschul, S.F., Gish, W., Miller, W., Myers, E.W., and Lipman, D.J. 1990. Basic local alignment search tool. JMB 215: 403-410.

Dynamic Programming

- Divide and conquer technique
  - Formulate solution as a recurrence relation
- Hallmarks of a DP problem
  - Optimal substructure
    - An optimal solution can be built from optimal subsolutions
  - Overlapping subproblems
    - The same subproblems keep showing up

There are two approaches

- Bottom-up method
  - There must be an evaluation order that solves smaller problems before larger ones
  - By storing and recalling the solutions to smaller problems, the larger ones can be efficiently computed

- Top-down with memoization
  - Follow the usual recursive procedure, but store your results along the way (memo)

- Both have the same asymptotic running time
- DP is always a time-memory trade-off

Example: Fibonacci numbers

- `F(n) = F(n-2) + F(n-1)`
  - `F(0) = 0`
  - `F(1) = 1`
  - `0, 1, 1, 2, 3, 5, 8, 13, …`
- Compute using recursion (not so smart):

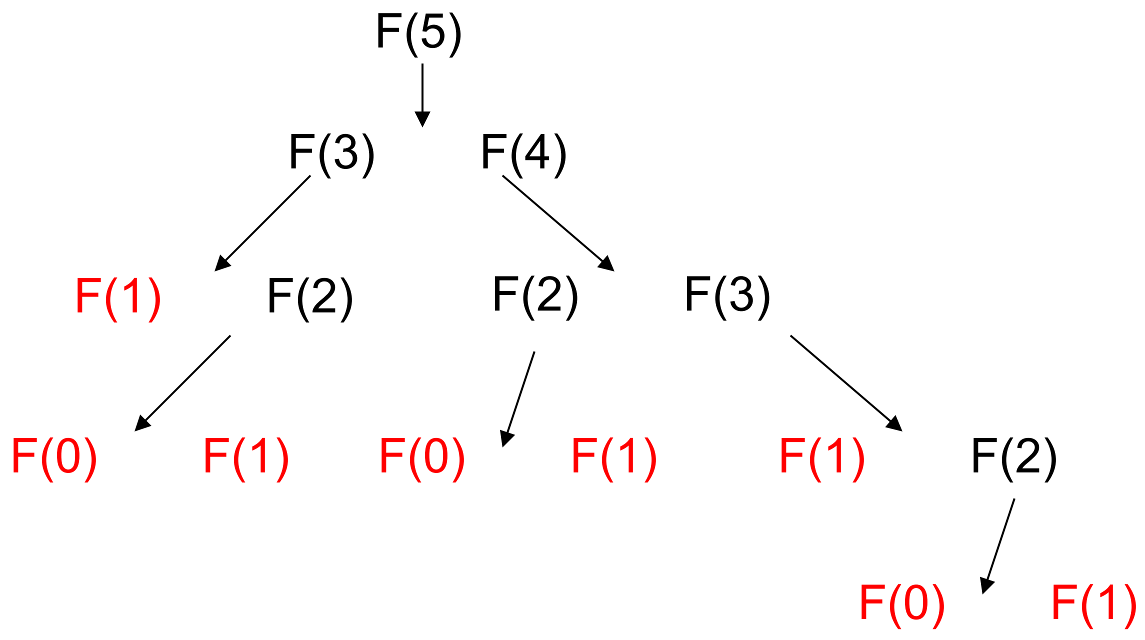
```
F(n)
    If (n < 2) return (n)
    Else
        Return (F(n-2) + F(n-1))
```

Complexity of naive solution

- We will have roughly `N` levels of recurrence
- Each recurrence generates two calls to `F`
- So, we have `O(2`^{N}`)`

- Where is the inefficiency?

We recompute things way too often, see `F(5)` below



The size of the subproblem graph
corresponds to the algorithm's runtime cost

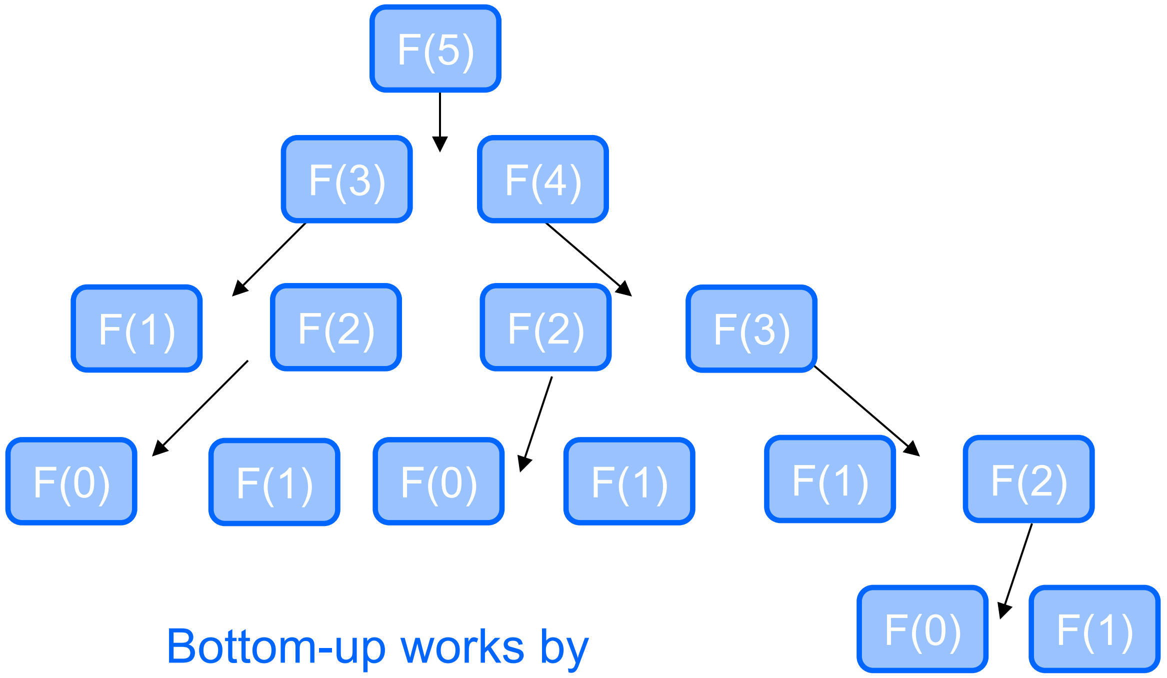
Better way: dynamic programming

- Recurrence: `F(n) = F(n-2) + F(n-1)`
- Evaluation order: small to large
- Number of smaller problems: `n-1`
- So, we should be able to get a good DP solution (in fact, it is linear!)

```
F(n)
   A = [0, 1]
   For k in range (2, n)
      A.append(A[k-1] + A[k-2])
   Return(A(n))
```

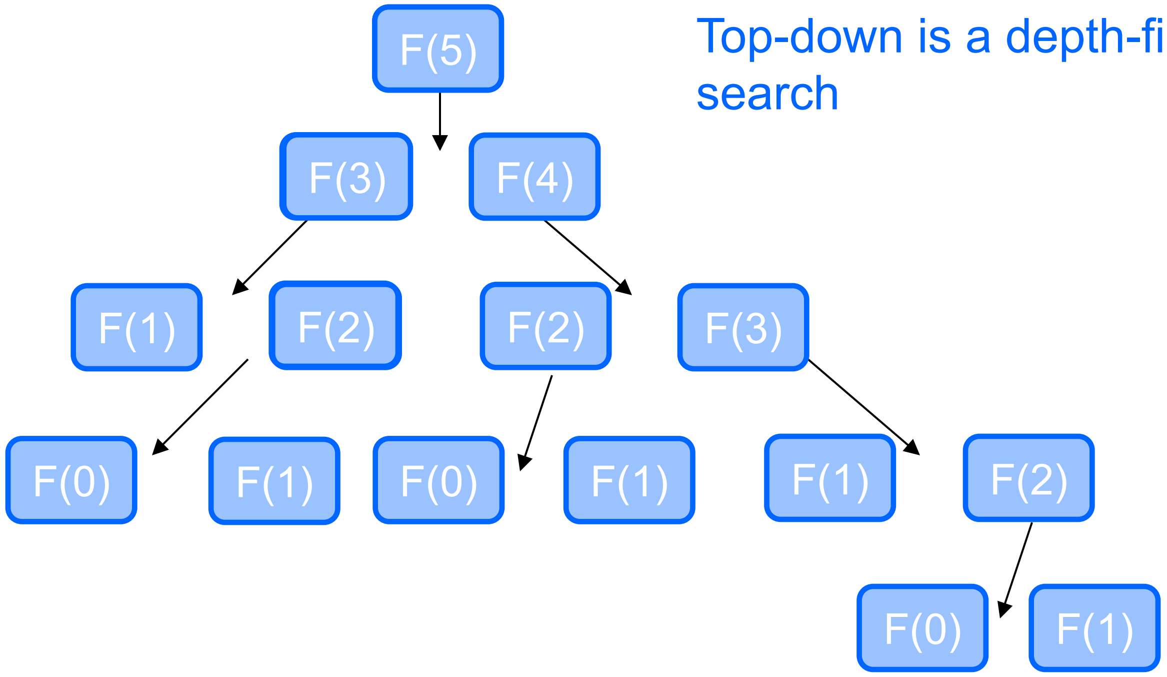
Is this bottom-up or top-down?

How do bottom-up and top-down methods traverse the graph?



Bottom-up works by
reverse topological sort

How do bottom-up and top-down methods traverse the graph?



Top-down is a depth-first search

What are the trade-offs between these two methods?

Dynamic programming in 4 steps

1. Characterize the structure of an optimal solution
2. Recursively define solution’s value
3. Compute the value
4. Construct the solution

Cormen, Chapter 15

1. Characterize

Longest common subsequence (LCS)

Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Cormen, Chapter 15

2. Define recursive solution

Longest common subsequence (LCS)

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases}$$

Cormen, Chapter 15

3. Compute solution value

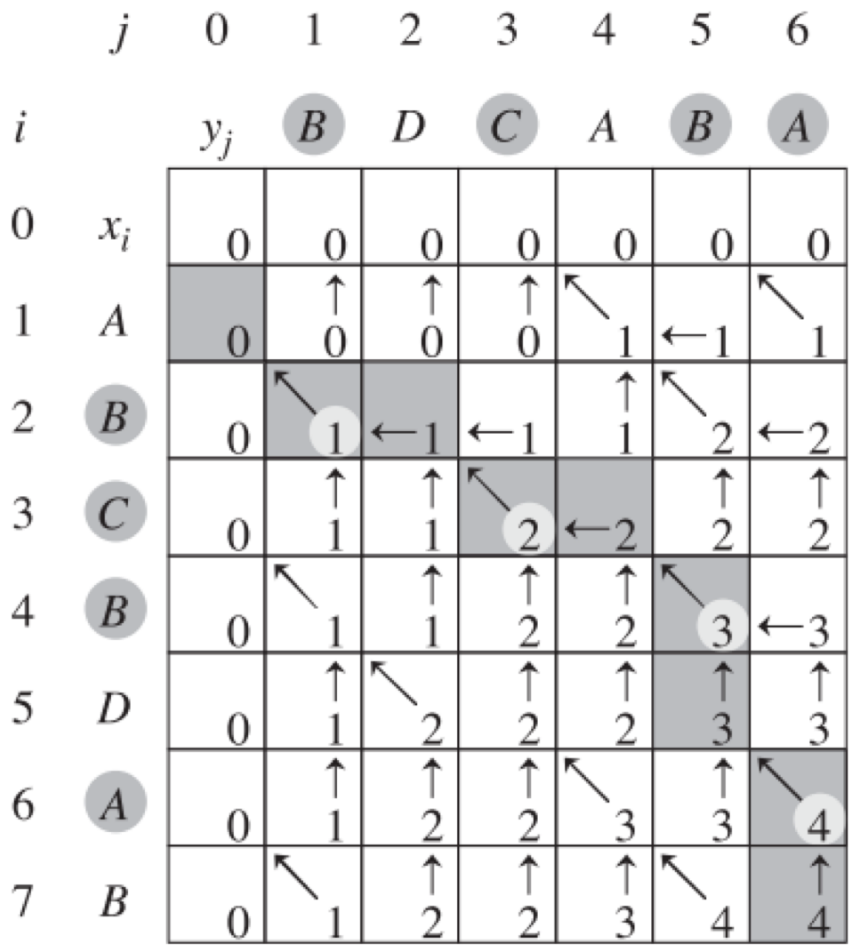
Longest common subsequence (LCS)

```
LCS-LENGTH($X, Y$)
$m = X.length$
$n = Y.length$
let $b[1..m, 1..n]$ and $c[0..m, 0..n]$ be new tables
for $i = 1$ to $m$
    $c[i, 0] = 0$
for $j = 0$ to $n$
    $c[0, j] = 0$
for $i = 1$ to $m$
    for $j = 1$ to $n$
        if $x_i == y_j$
            $c[i, j] = c[i-1, j-1] + 1$
            $b[i, j] =$ "$\nwarrow$"
        elseif $c[i-1, j] \geq c[i, j-1]$
            $c[i, j] = c[i-1, j]$
            $b[i, j] =$ "$\uparrow$"
        else $c[i, j] = c[i, j-1]$
            $b[i, j] =$ "$\leftarrow$"
return $c$ and $b$
```

Cormen, Chapter 15

4. Construct solution

Longest common subsequence (LCS)



i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

Cormen, Chapter 15

Pairwise Sequence Alignment

- What is an alignment, and why might it be significant?
  - An alignment is *a mapping from one sequence to another, identifying elements that are likely to have arisen from a common ancestor*
  - A good alignment (high sequence similarity) is an indication of homology

Similarity vs. Homology Paralogs vs. Orthologs

- *Homology* is an evolutionary relationship that either exists or does not. It cannot be partial.
- An *ortholog* is a homolog with shared function.
- A *paralog* is a homolog that arose through a gene duplication event. Paralogs often have divergent function.
- *Similarity* is a measure of the quality of alignment between two sequences. High similarity is evidence for homology. Similar sequences may be orthologs or paralogs.

How do we compute similarity?

- Similarity can be defined by counting positions that are identical between two sequences

- Gaps (insertions/deletions) can be important

```
abcdef    abcdef    abcdef
||| ||    |         | ||||
abceef    acdef     a-cdef
```

Not all mismatches are the same

- Some amino acids are more substitutable for each other than others.
- We can introduce "mismatch costs" for handling different substitutions.
  - We don't usually use mismatch costs in aligning nucleotide sequences, since no substitution is per se better than any other.
- We will focus on protein sequence alignments

Many possible alignments to consider

- Without gaps, there are roughly `N+M` possible alignments between sequences of length `N` and `M`
- Once we start allowing gaps, there are many possible arrangements to consider:

```
abcbcd      abcbcd     abcbcd
|||  |      |  |||     ||  ||
abc--d      a--bcd     ab--cd
```

- This becomes a very large number when we allow mismatches, since we then need to look at every possible “monotonic” pairing between elements

Avoiding random alignments with a score function

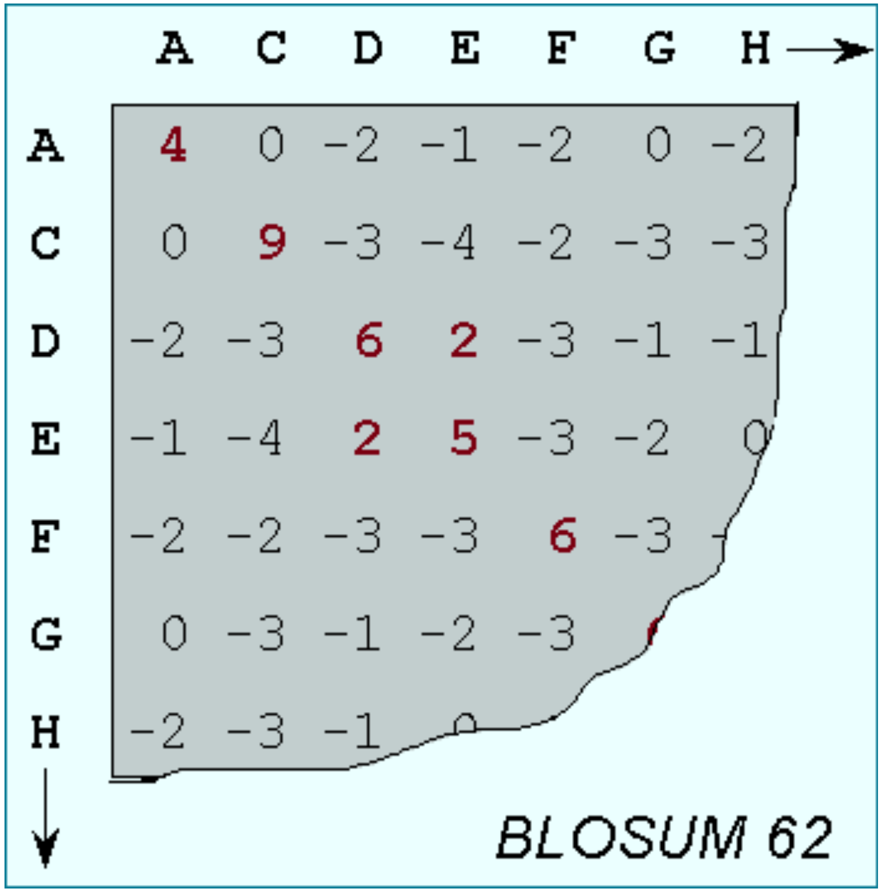
- Not only are there many possible gapped alignments, but introducing too many gaps makes senseless alignments possible
- Need to distinguish between alignments that occur due to homology, and those that could be expected to be seen just by chance.

- Define a score function that accounts for element **matches**, **mismatches** and a **gap penalty**

Match scores (we will discuss derivation of such matrices later in the course)

- Match scores are often calculated on the basis of the frequency of particular mutations in very similar sequences.

- We can transform substitution frequencies into log odds scores, which can then be added together.



	A	C	D	E	F	G	H
A	4	0	-2	-1	-2	0	-2
C	0	9	-3	-4	-2	-3	-3
D	-2	-3	6	2	-3	-1	-1
E	-1	-4	2	5	-3	-2	0
F	-2	-2	-3	-3	6	-3	-
G	0	-3	-1	-2	-3	[illegible]	
H	-2	-3	-1	0			

BLOSUM 62

Local vs. Global alignments

- A *global alignment* includes all elements of a sequence, and includes gaps (Needleman-Wunsch)
- A *local alignment* includes only subsequences (Smith-Waterman), and sometimes computed without gaps (e.g BLAST)
- Local alignments can find shared domains in divergent proteins and are faster to compute.
- Global alignments are better indicators of homology and take longer to compute.

An alignment score

- An alignment score is the sum of all the match scores of an alignment, with a penalty subtracted for each gap.
- Gap penalties are usually "affine" meaning that the penalty for one long gap is smaller than the penalty for many smaller gaps that add up to the same size.

Match score

Gap start + continuation penalty

Alignment Score

```
a b c - - d
a c c e f d
9 2 7     6 -> 24 - (10 + 2) = 12
```

Finding the optimal alignment

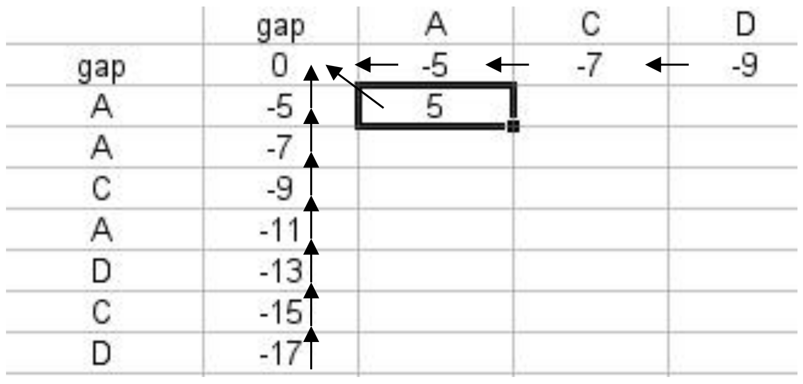
- Given a pair of sequences and a score function, identify the best scoring (optimal) alignment between the sequences.
- Remember: exponential number of possible alignments (most with terrible scores). **(2^{2N})/sqrt(Pi*N)**
- Dynamic programming identifies optimal alignments in time proportional to the product of the lengths of the sequences

Dynamic programming

- The key idea is to start aligning the sequences left to right; once a prefix is optimally aligned, nothing about the remainder of the alignment changes the alignment of the prefix.
- We construct a matrix of possible alignment scores and then "traceback" to find the optimal alignment.
- Needleman-Wunsch or Smith-Waterman depending on the formulation of the recurrence function

Dynamic programming alignment

- Each cell has the score for the best aligned sequence prefix up to that position.
  - If we are to have ***consumed*** all including position `i` of sequence `A` and all including position `j` of sequence `B`, the score in cell `i,j` is the best we can do
- Start by filling in initial gap and first element to first element match score
- Use arrow to indicate path to that alignment



	gap	A	C	D
gap	0	← -5	← -7	← -9
A	↑ -5	↖ 5		
A	↑ -7			
C	↑ -9			
A	↑ -11			
D	↑ -13			
C	↑ -15			
D	↑ -17			

Preexisting scoring matrix:

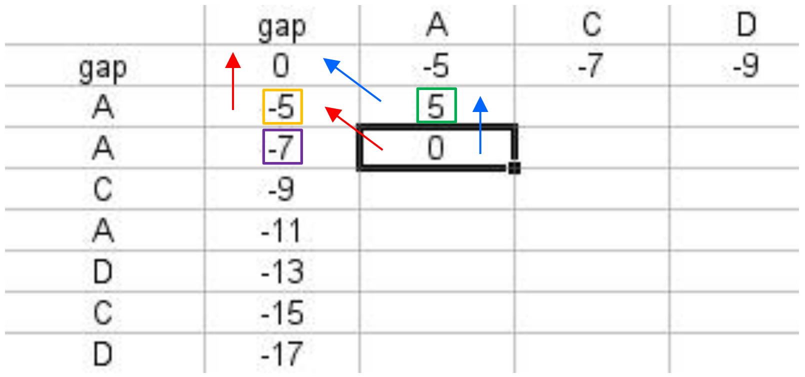
Score matrix		Gap start = -5, gap continue -2		
	A	B	C	D
A	5	3	-1	1
B	3	4	-2	2
C	-1	-2	7	-1
D	1	2	-1	7

Continue filling in optimal path scores

- For each cell, have three choices for how to get there from the last optimal alignment (match, gap sequence 1, gap sequence 2).
- Best score(s) are selected, and arrows indicate route(s).

$\boxed{-5}$ + 5 = 0
$\boxed{5}$ + -5 = 0
$\boxed{-7}$ + -5 = -12

AACADCD
-A
A-

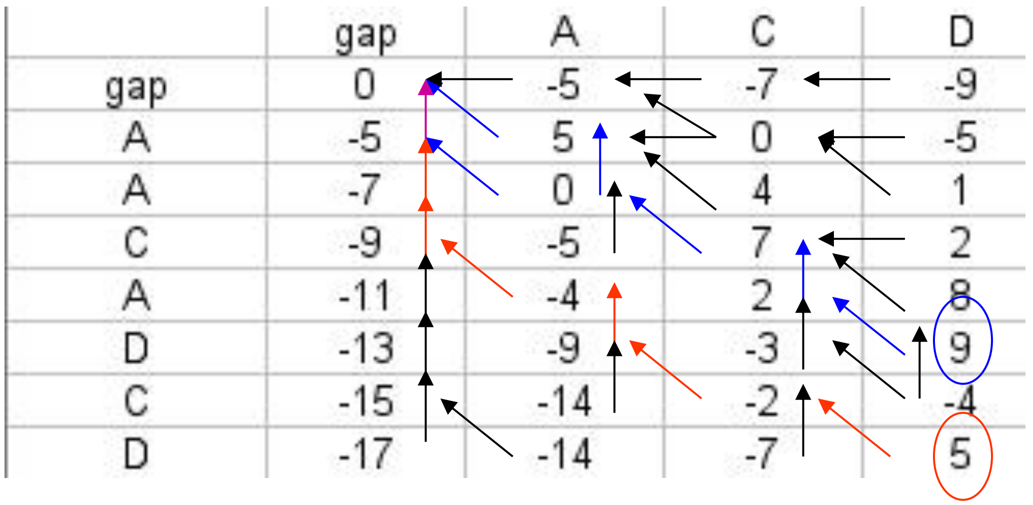


	gap	A	C	D
gap	0	-5	-7	-9
A	-5	5		
A	-7	0		
C	-9			
A	-11			
D	-13			
C	-15			
D	-17			

Optimal alignment by traceback

- We “traceback” a path that gets us the highest score. If we don't have “end gap” penalties, then take any path from the last row or column to the first.
- Otherwise we need to include the top and bottom corners

```
AACADCD
---A-CD
-AC-D
A-C-D
```



	gap	A	C	D
gap	0	-5	-7	-9
A	-5	5	0	-5
A	-7	0	4	1
C	-9	-5	7	2
A	-11	-4	2	8
D	-13	-9	-3	9
C	-15	-14	-2	-4
D	-17	-14	-7	5

Each ***arrow*** is either a match (diagonal) or a gap (vertical is a gap in the bottom sequence, horizontal is a gap in the top sequence).

Needleman-Wunsch

- Recurrence relation:

What is $S(A_i, B_j)$?

$$
\begin{aligned}
&H_{i,j} = \max(\\
&\quad H_{i-1,j-1} + S(A_i, B_j) && \text{[diag]} \\
&\quad H_{i-1,j} - \text{gap penalty} && \text{[up]} \\
&\quad H_{i,j-1} - \text{gap penalty} && \text{[left]} \\
&)
\end{aligned}
$$

- We trace back from lower right to upper left. This yields a global alignment.

Needleman-Wunsch Length dependent gap penalty

- Recurrence relation:

```
H_{i,j} = max(
    H_{i-1,j-1} + S(A_i,B_j)
    max(H_{i-k,j} - W_k) for k = 1…i
    max(H_{i,j-m} - W_m) for m = 1…j
 )
```

$$H_{i,j} = \max\left(H_{i-1,j-1} + S(A_i,B_j),\ \max(H_{i-k,j} - W_k) \text{ for } k = 1\ldots i,\ \max(H_{i,j-m} - W_m) \text{ for } m = 1\ldots j \right)$$

- We trace back from lower right to upper left.
- Typical gap specification has a high opening penalty (large W_1) and constant extension penalty ($W_{k+1} - W_k$ is constant)

Smith-Waterman
Length dependent gap penalty

- Recurrence relation:

```
$H_{i,j}$ = max(
    $H_{i-1,j-1} + S(A_i,B_j)$
    max($H_{i-k,j} - W_k$) for k = 1…i
    max($H_{i,j-m} - W_m$) for m = 1…j
    0
 )
```

- We trace back from highest score to 0
  - Scores never go negative
  - This finds the highest scoring subsequence (local) alignment

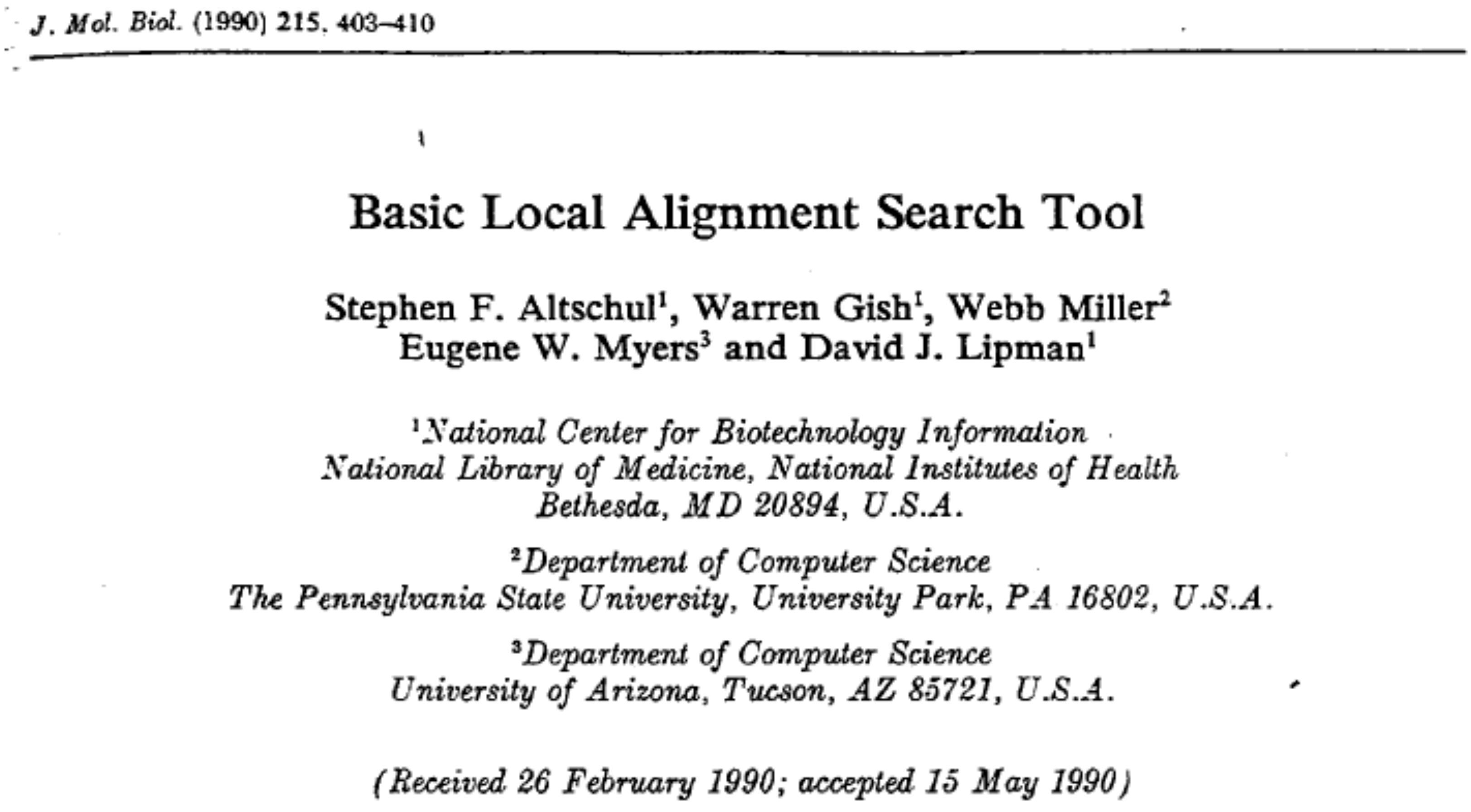
Conclusions

- Dynamic programming can have very substantial complexity benefits in certain types of problems
- The key is that the divide-and-conquer assumption is rigorously true

- Additional wrinkles on sequence alignment
  - Faster search: FASTA and BLAST
  - More refined identification of similar sequences: PSI-BLAST
  - Relationship between sequence and structure alignments

BLAST: Need more speed

- Even though DP approaches for sequence alignment are fast, the databases got very large very quickly



J. Mol. Biol. (1990) 215, 403–410

Basic Local Alignment Search Tool

Stephen F. Altschul[1], Warren Gish[1], Webb Miller[2]
Eugene W. Myers[3] and David J. Lipman[1]

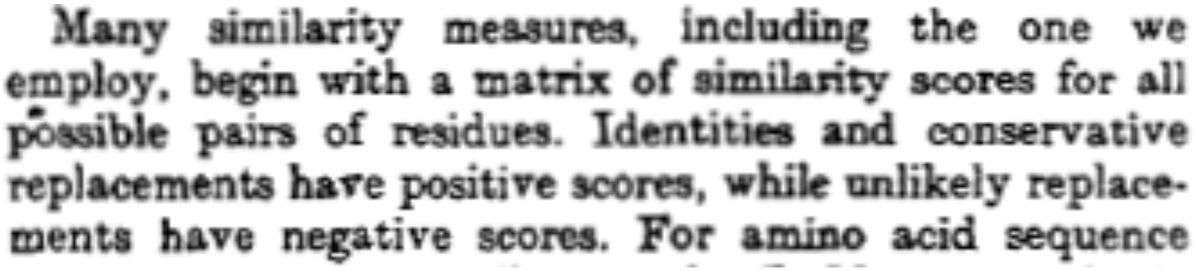
[1]*National Center for Biotechnology Information*
National Library of Medicine, National Institutes of Health
Bethesda, MD 20894, U.S.A.

[2]*Department of Computer Science*
The Pennsylvania State University, University Park, PA 16802, U.S.A.

[3]*Department of Computer Science*
University of Arizona, Tucson, AZ 85721, U.S.A.

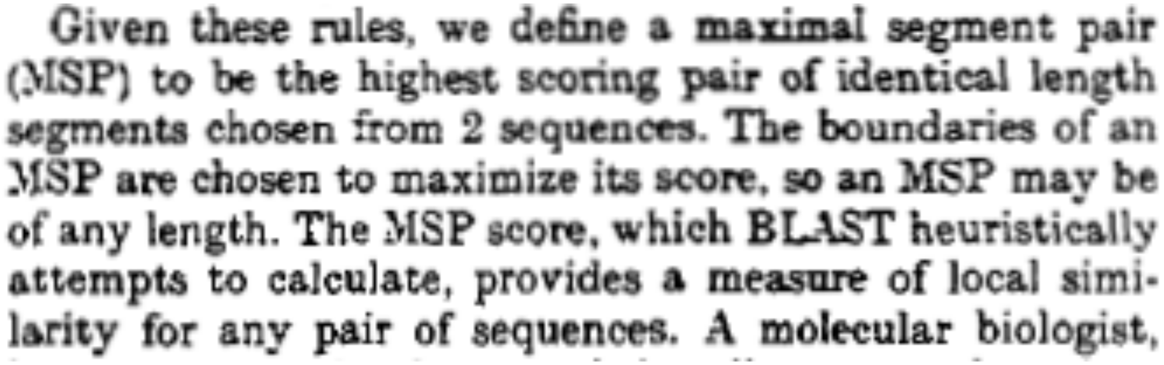
(Received 26 February 1990; accepted 15 May 1990)

Looking for maximal segment pairs



Many similarity measures, including the one we employ, begin with a matrix of similarity scores for all possible pairs of residues. Identities and conservative replacements have positive scores, while unlikely replacements have negative scores. For amino acid sequence

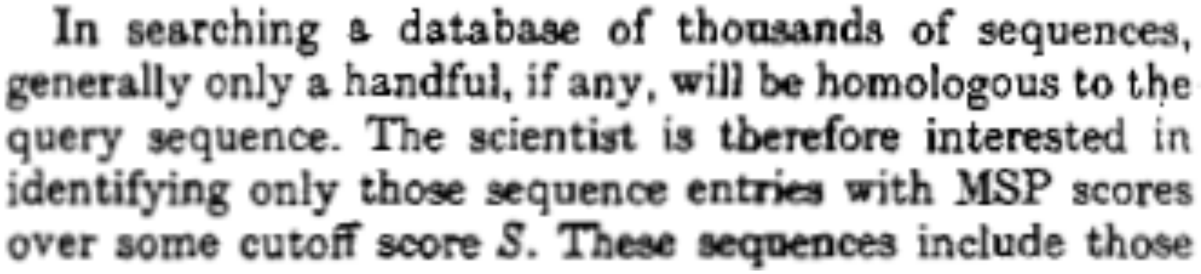
Need a score definition



Given these rules, we define a maximal segment pair (MSP) to be the highest scoring pair of identical length segments chosen from 2 sequences. The boundaries of an MSP are chosen to maximize its score, so an MSP may be of any length. The MSP score, which BLAST heuristically attempts to calculate, provides a measure of local similarity for any pair of sequences. A molecular biologist,

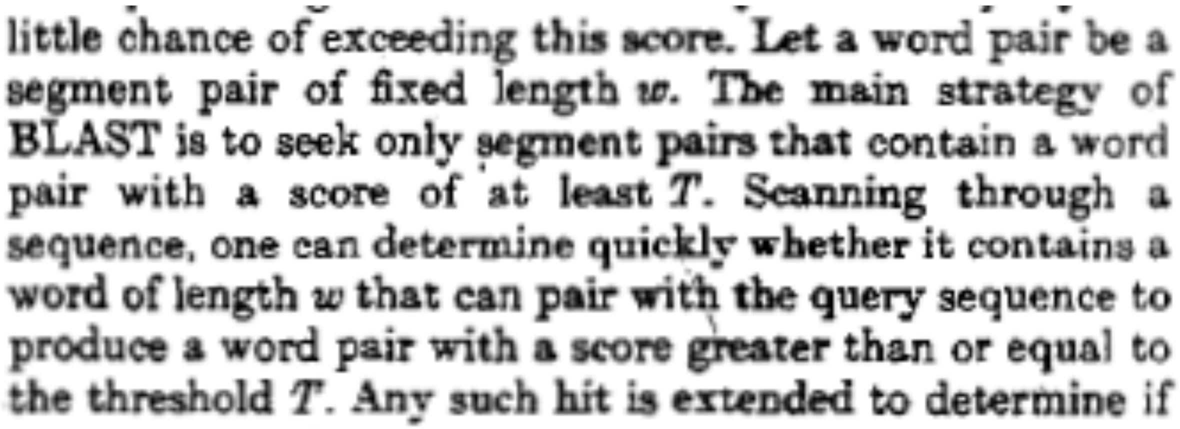
The MSP is the global maximum. BLAST tries to get at these quickly using a heuristic algorithm that is not guaranteed to produce optimal results.

We only really care about high scores



In searching a database of thousands of sequences, generally only a handful, if any, will be homologous to the query sequence. The scientist is therefore interested in identifying only those sequence entries with MSP scores over some cutoff score S. These sequences include those

We are picking cherries here, not trying to rank a whole DB.

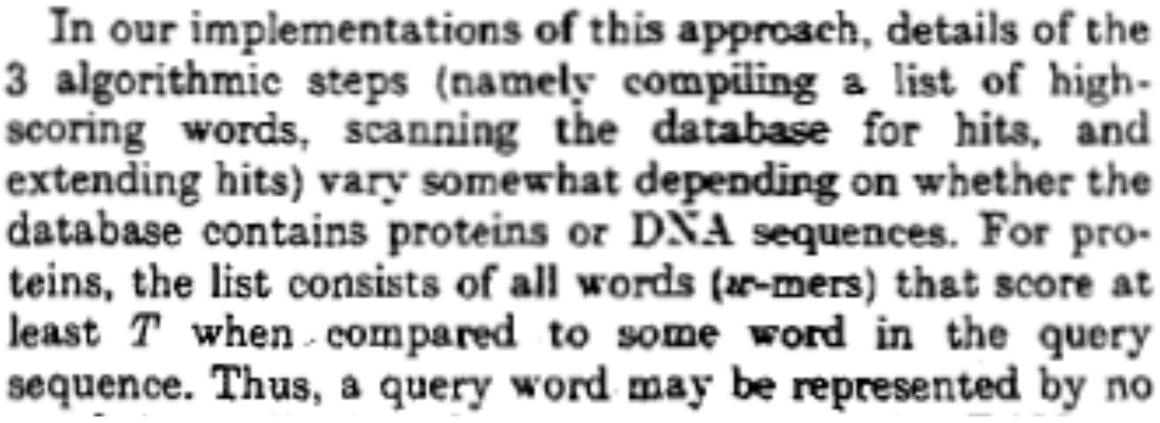


little chance of exceeding this score. Let a word pair be a segment pair of fixed length w. The main strategy of BLAST is to seek only segment pairs that contain a word pair with a score of at least T. Scanning through a sequence, one can determine quickly whether it contains a word of length w that can pair with the query sequence to produce a word pair with a score greater than or equal to the threshold T. Any such hit is extended to determine if

The game is to quickly find starting points between the query and the sequence at hand that exceed some threshold.

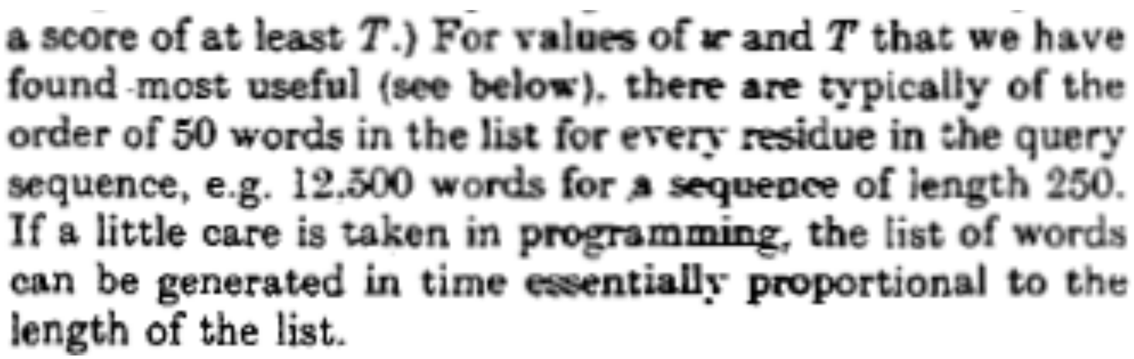
These are expanded to see if they exceed the overall threshold for the MSP score.

We make a list of words that would be good to find



In our implementations of this approach, details of the 3 algorithmic steps (namely compiling a list of high-scoring words, scanning the database for hits, and extending hits) vary somewhat depending on whether the database contains proteins or DNA sequences. For proteins, the list consists of all words (w-mers) that score at least T when compared to some word in the query sequence. Thus, a query word may be represented by no

We end up ignoring words that are common and focus on those with high information content.

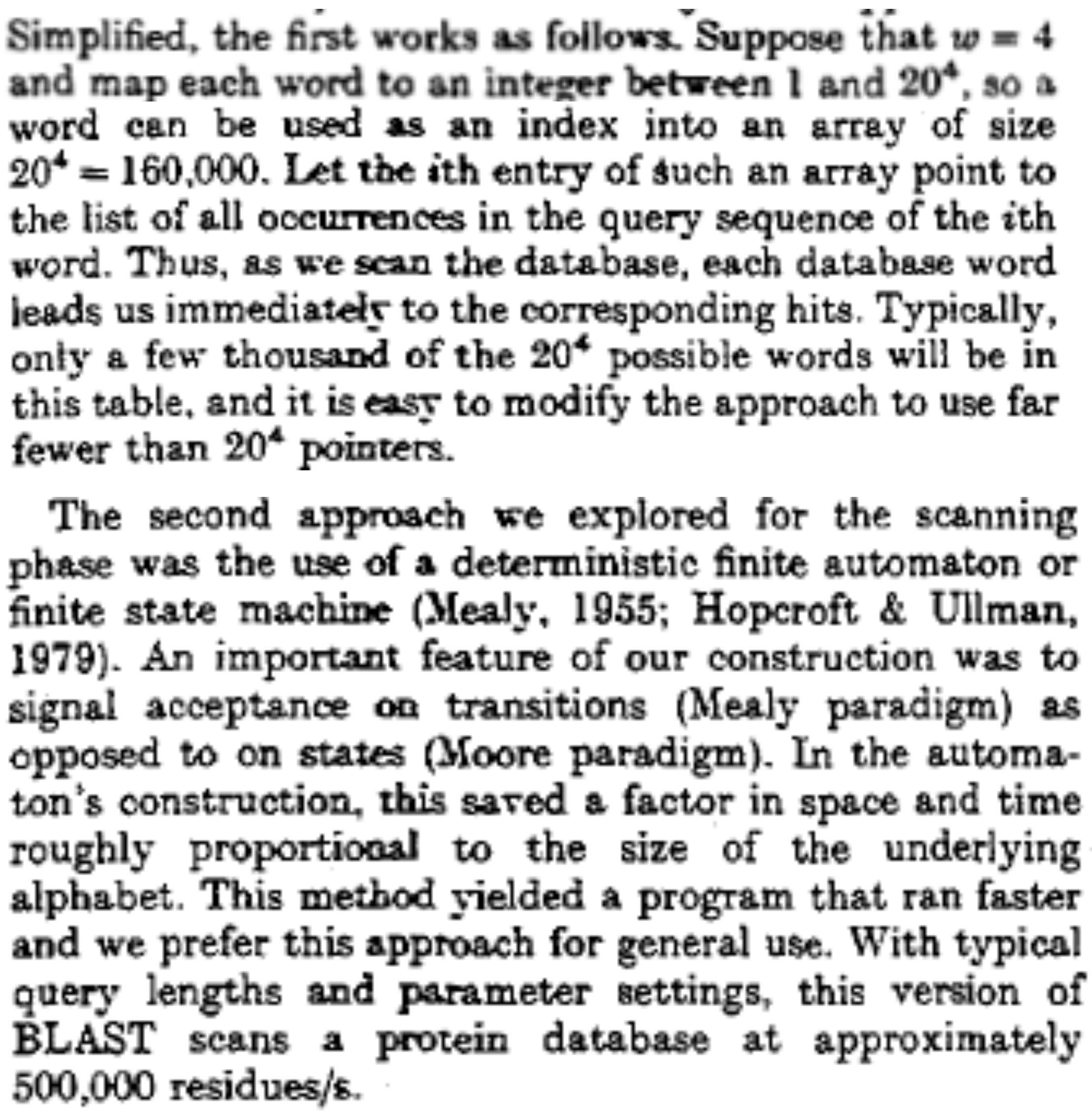


a score of at least T.) For values of w and T that we have found most useful (see below), there are typically of the order of 50 words in the list for every residue in the query sequence, e.g. 12,500 words for a sequence of length 250. If a little care is taken in programming, the list of words can be generated in time essentially proportional to the length of the list.

Generally, this produces about 50 words for each protein residue.

The procedure is linear in length of query sequence.

The scanning phase finds the words in the database seqs



Simplified, the first works as follows. Suppose that $w = 4$ and map each word to an integer between 1 and 20^4, so a word can be used as an index into an array of size $20^4 = 160{,}000$. Let the *i*th entry of such an array point to the list of all occurrences in the query sequence of the *i*th word. Thus, as we scan the database, each database word leads us immediately to the corresponding hits. Typically, only a few thousand of the 20^4 possible words will be in this table, and it is easy to modify the approach to use far fewer than 20^4 pointers.

The second approach we explored for the scanning phase was the use of a deterministic finite automaton or finite state machine (Mealy, 1955; Hopcroft & Ullman, 1979). An important feature of our construction was to signal acceptance on transitions (Mealy paradigm) as opposed to on states (Moore paradigm). In the automaton's construction, this saved a factor in space and time roughly proportional to the size of the underlying alphabet. This method yielded a program that ran faster and we prefer this approach for general use. With typical query lengths and parameter settings, this version of BLAST scans a protein database at approximately 500,000 residues/s.

This can be done by building an index of occurrence of the words in the query. Scanning through a sequence allows us to look up the locations immediately. So we know if and where each word exists.

But we won't tell you how we actually do this in the implementation that we use!!!!

Some modeling to help determine parameters

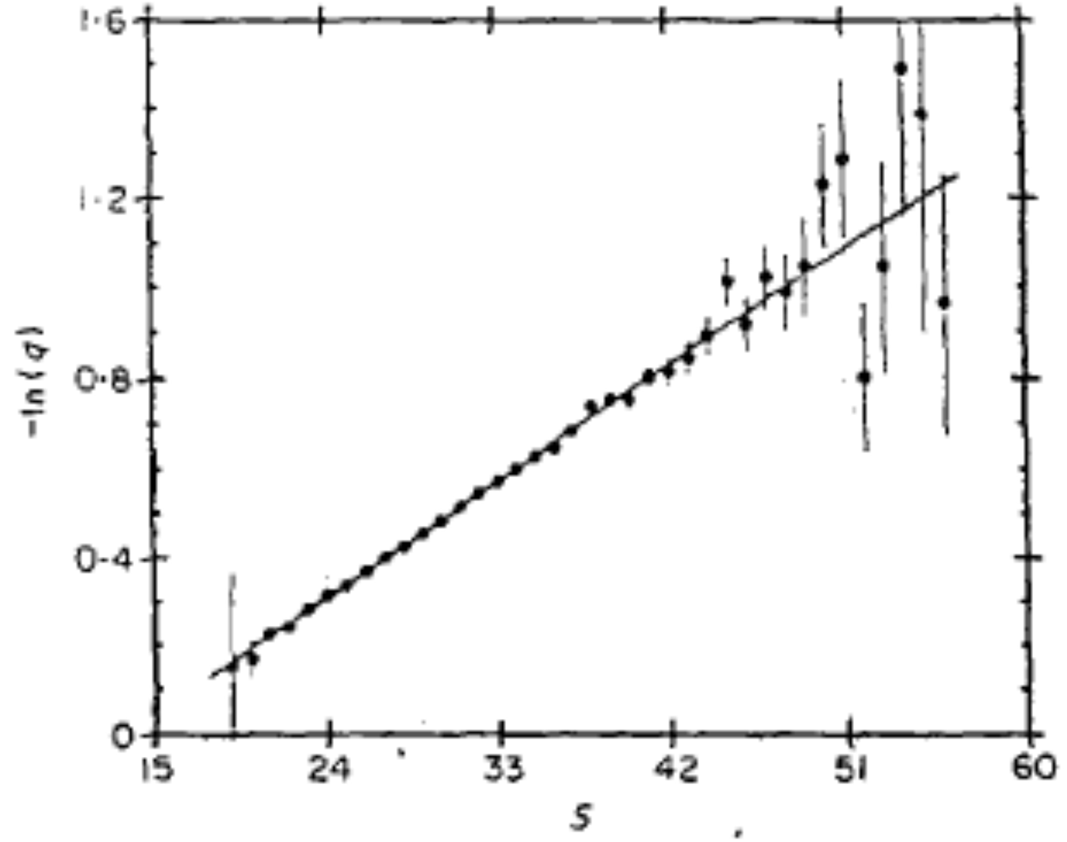


Figure 1. The probability q of BLAST missing a random maximal segment pair as a function of its score S.

The chances of missing an MSP decreases exponentially with increases in the score S.

We have a winner! w = 4, T = 17

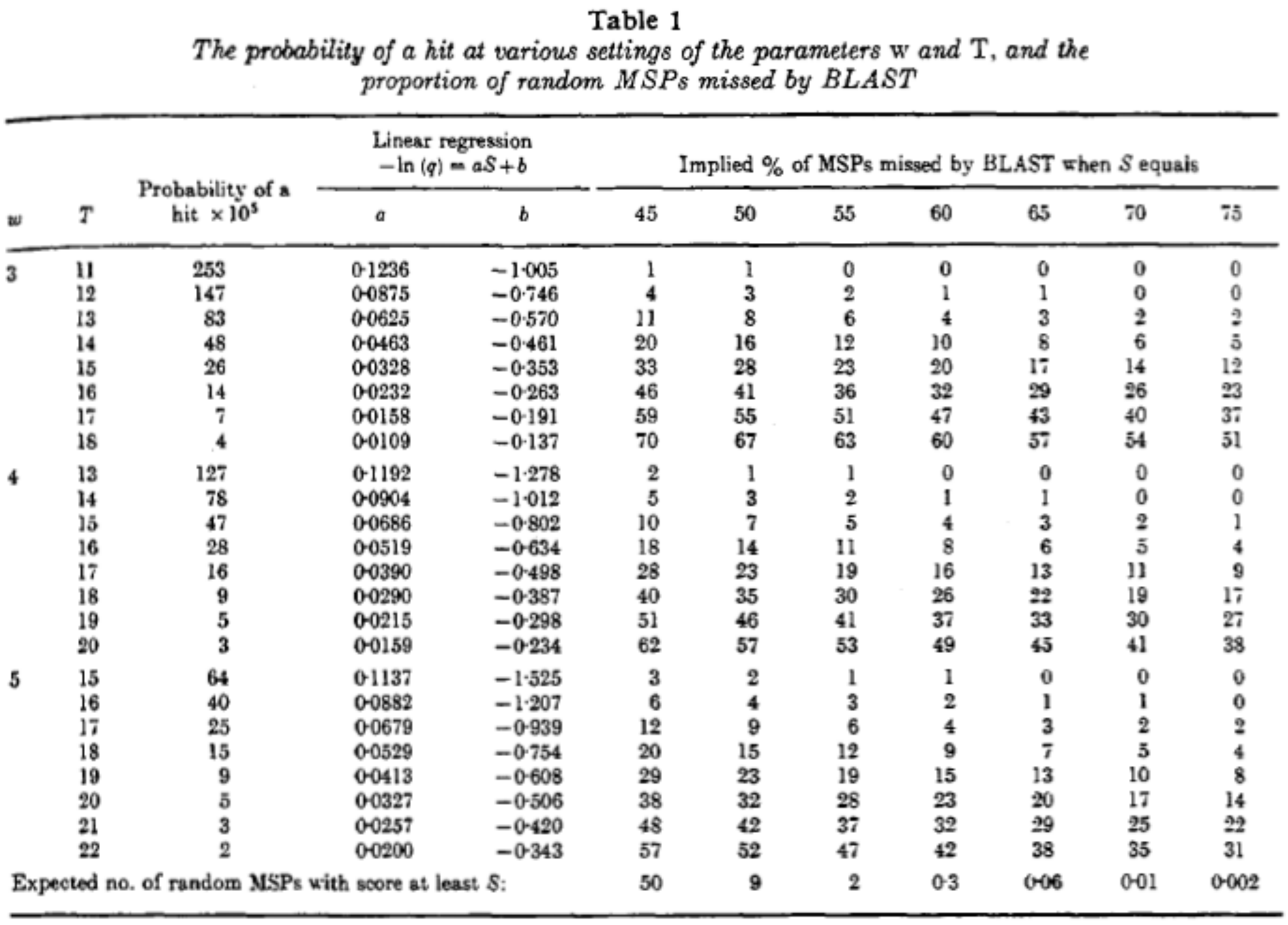


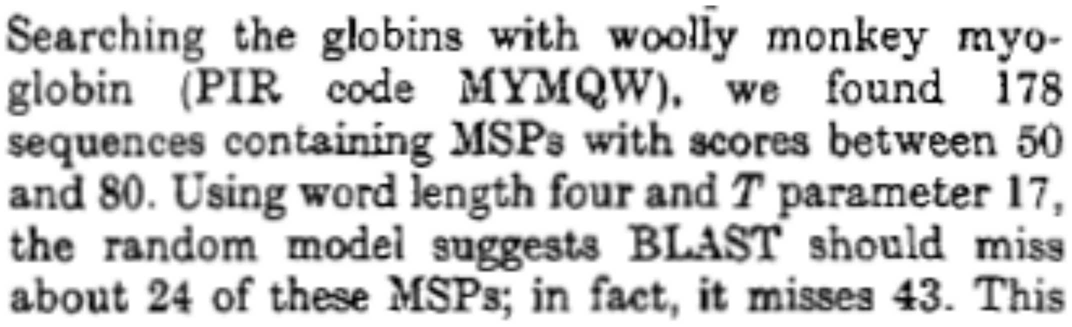
Table 1
The probability of a hit at various settings of the parameters w *and* T, *and the proportion of random MSPs missed by BLAST*

			Linear regression $-\ln(q) = aS + b$		Implied % of MSPs missed by BLAST when S equals						
w	T	Probability of a hit $\times 10^5$	a	b	45	50	55	60	65	70	75
3	11	253	0·1236	−1·005	1	1	0	0	0	0	0
	12	147	0·0875	−0·746	4	3	2	1	1	0	0
	13	83	0·0625	−0·570	11	8	6	4	3	2	2
	14	48	0·0463	−0·461	20	16	12	10	8	6	5
	15	26	0·0328	−0·353	33	28	23	20	17	14	12
	16	14	0·0232	−0·263	46	41	36	32	29	26	23
	17	7	0·0158	−0·191	59	55	51	47	43	40	37
	18	4	0·0109	−0·137	70	67	63	60	57	54	51
4	13	127	0·1192	−1·278	2	1	1	0	0	0	0
	14	78	0·0904	−1·012	5	3	2	1	1	0	0
	15	47	0·0686	−0·802	10	7	5	4	3	2	1
	16	28	0·0519	−0·634	18	14	11	8	6	5	4
	17	16	0·0390	−0·498	28	23	19	16	13	11	9
	18	9	0·0290	−0·387	40	35	30	26	22	19	17
	19	5	0·0215	−0·298	51	46	41	37	33	30	27
	20	3	0·0159	−0·234	62	57	53	49	45	41	38
5	15	64	0·1137	−1·525	3	2	1	1	0	0	0
	16	40	0·0882	−1·207	6	4	3	2	1	1	0
	17	25	0·0679	−0·939	12	9	6	4	3	2	2
	18	15	0·0529	−0·754	20	15	12	9	7	5	4
	19	9	0·0413	−0·608	29	23	19	15	13	10	8
	20	5	0·0327	−0·506	38	32	28	23	20	17	14
	21	3	0·0257	−0·420	48	42	37	32	29	25	22
	22	2	0·0200	−0·343	57	52	47	42	38	35	31
Expected no. of random MSPs with score at least S:					50	9	2	0·3	0·06	0·01	0·002

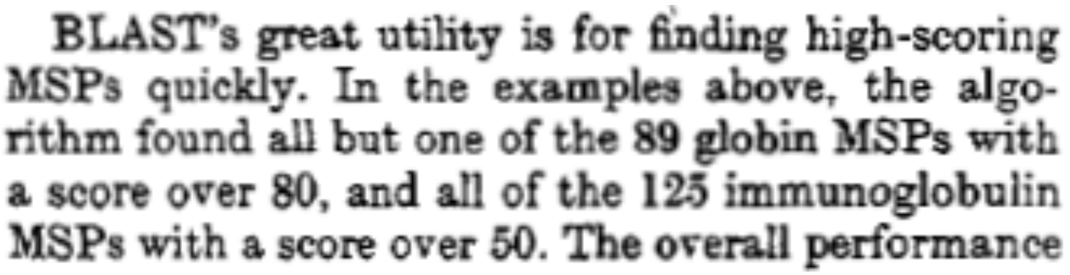
A word size of 4 with a threshold of 17 sensitivity as well as avoiding the memory cost of higher word sizes (20^w).

Selection of T = 17 was done through empirical run-time testing and modeling the complexity.

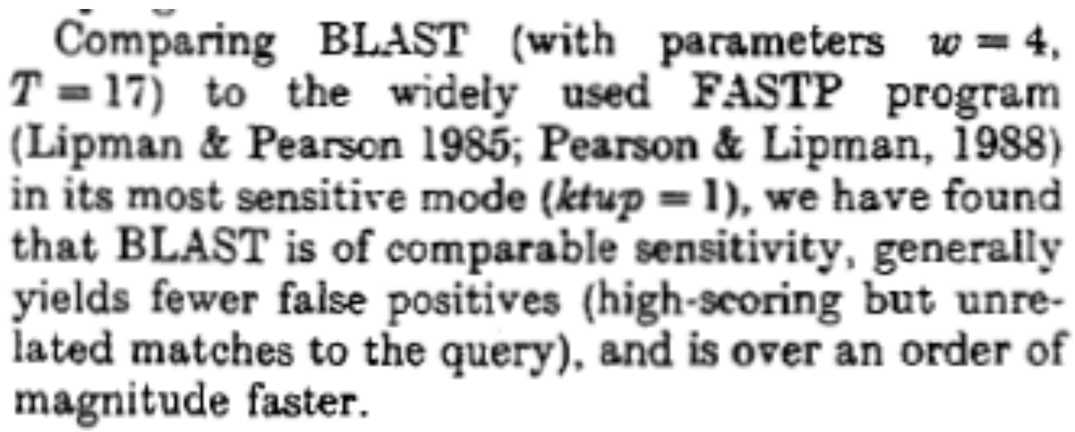
Tests on real data: globins


Searching the globins with woolly monkey myoglobin (PIR code MYMQW), we found 178 sequences containing MSPs with scores between 50 and 80. Using word length four and T parameter 17, the random model suggests BLAST should miss about 24 of these MSPs; in fact, it misses 43. This

BLAST slightly underperformed theory on relatively distantly related proteins.

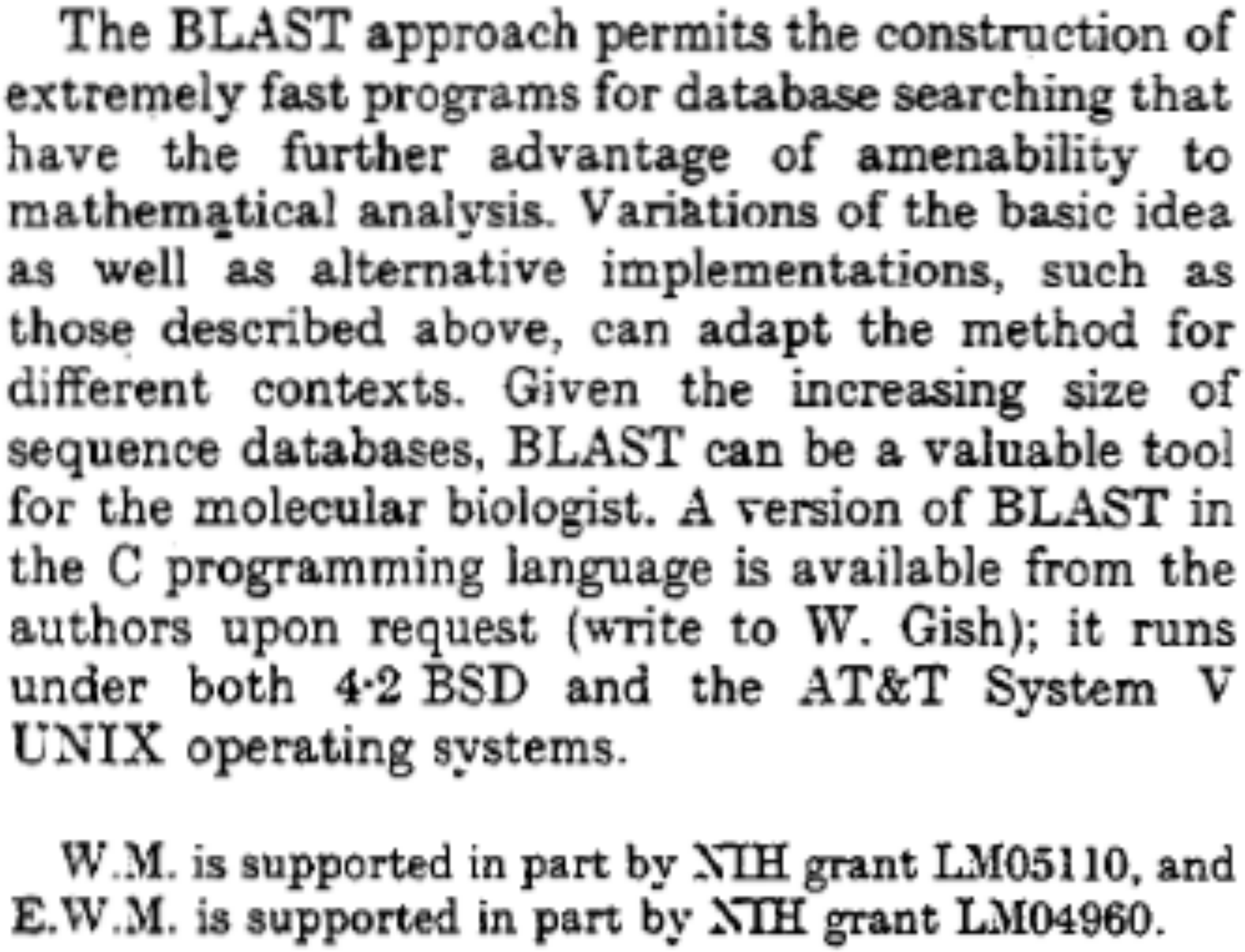

BLAST's great utility is for finding high-scoring MSPs quickly. In the examples above, the algorithm found all but one of the 89 globin MSPs with a score over 80, and all of the 125 immunoglobulin MSPs with a score over 50. The overall performance

However, for high-scoring MSPs, it is both fast and effective.


Comparing BLAST (with parameters $w = 4$, $T = 17$) to the widely used FASTP program (Lipman & Pearson 1985; Pearson & Lipman, 1988) in its most sensitive mode ($ktup = 1$), we have found that BLAST is of comparable sensitivity, generally yields fewer false positives (high-scoring but unrelated matches to the query), and is over an order of magnitude faster.

It was more than ten-fold faster than FASTP.

Open source via snail-mail!



The BLAST approach permits the construction of extremely fast programs for database searching that have the further advantage of amenability to mathematical analysis. Variations of the basic idea as well as alternative implementations, such as those described above, can adapt the method for different contexts. Given the increasing size of sequence databases, BLAST can be a valuable tool for the molecular biologist. A version of BLAST in the C programming language is available from the authors upon request (write to W. Gish); it runs under both 4·2 BSD and the AT&T System V UNIX operating systems.

W.M. is supported in part by NIH grant LM05110, and E.W.M. is supported in part by NIH grant LM04960.

Lest you think DP outdated…

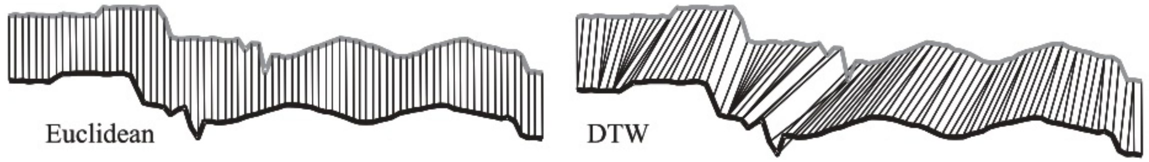


Fig. 1. Note that, while the two time series have an overall similar shape, they are not aligned in the time axis. Euclidean distance, which assumes the i^{th} point in one sequence is aligned with the i^{th} point in the other, will produce a pessimistic dissimilarity measure. The nonlinear dynamic time warped alignment allows a more intuitive distance measure to be calculated

Gee wouldn't it be nice to compare and predict time series?

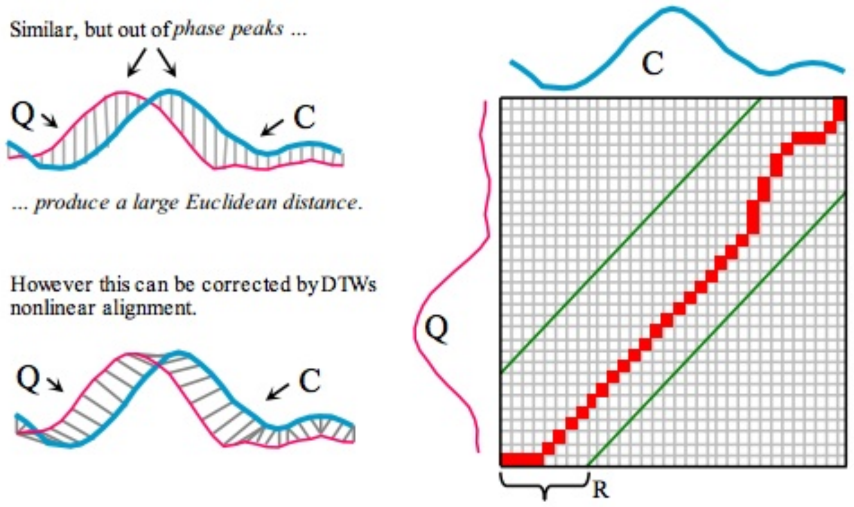

Similar, but out of phase peaks ...
Q
C
... produce a large Euclidean distance.
However this can be corrected by DTWs nonlinear alignment.
Q
C
C
Q
R

Figure 3: left) Two time series which are similar but out of phase. right) To align the sequences we construct a warping matrix, and search for the optimal warping path (red/solid squares). Note that Sakoe-Chiba Band with width R is used to constrain the warping path

https://practicalquant.blogspot.com/2012/10/mining-time-series-with-trillions-of.html