Algorithms in Bioinformatics: Lectures 03-05 - Sequence Similarity

Lucia Moura

Fall 2011

Lecture(s) Contents: Sequence Similarity (Chapter 2)

Notes: These slides are being developed lecture by lecture.
These slides do not cover the complete lecture contents (use textbook).
These slides are a template for material discussed in the class, blackboard.

Motivation for the study of sequence similarity

“The first fact of biological sequence analysis: In biomolecular sequences (DNA, RNA, or amino acid sequences), high sequence similarity usually implies significant functional or structural similarity.” D. Gusfield, *Algorithms on strings, trees and sequences*

Note that the converse is not true:
“ ... similar sequences yield similar structures, but quite different sequences can produce remarkably similar structures.”
F. E. Cohen (1995)

Bioinformatics Application Examples

- Predicting the biological function of a gene (or a RNA or a protein): Check similarities with genes for which we know their functions.
- Finding the evolution distance (by comparing genomes) Building Phylogenetic trees.
- Helping genome assembly: human genome project used a lot of short DNA sequences, and reconstructed complete genome based on overlapping information.
- Finding a common region in two genomes: two similar or identical genes (from two species), may have come from a common ancestor and have the same function.
- Finding repeats within a genome: The human genome has many repeat substrings, which need to be identified.

Measuring distance between two sequences I

The string edit problem:
Determine the minimum number of operations to transform one string into another, where operations are:

- Replace a symbol with another symbol.
- Insert a symbol into a string.
- Delete a symbol from a string.

$S =$	INTERESTINGLY
6 replacements:	IN**FO**R**MA**TI**CS**LY
3 insertions:	**BIOI**NFORMATICSLY
2 deletions:	BIOINFORMATICS_ _
$T =$	BIOINFORMATICS

This tells us the edit distance is at most 11; it turns out to be 11.

Measuring distance between two sequences II

The (generalized) string edit problem:

Let S and T be two strings over the alphabet Σ.

Determine the minimum **cost** to transform one string into another, where operations are:

- Replace a symbol with another symbol. cost: $\sigma(x, y)$
- Insert a symbol into a string. cost: $\sigma(\_, x)$
- Delete a symbol from a string. cost: $\sigma(x, \_)$

and their costs of operations are given by a **distance matrix** $\sigma(x, y)$ with $x, y \in \Sigma \cup \{\_\}$.

$S =$	INTERESTINGLY	operation cost
6 replacements:	IN**FO**R**MA**TIC**S**LY	$\sigma(x, y) = 1$
3 insertions:	**BIOI**NFORMATICSLY	$\sigma(\_, x) = 2$
2 deletions:	BIOINFORMATICS_ _	$\sigma(x, \_) = 2$
$T =$	BIOINFORMATICS	

Cost : $6 \times 1 + 3 \times 2 + 2 \times 2 = 16.$

Measuring Sequence Similarity

Definition (alignment)

An *alignment* of two strings is formed by inserting spaces in arbitrary locations along the string so that they end up with the same length and there are no spaces at the same position of the two augmented sequences.

Two sequences are similar if their alignment contains:
many positions with the same symbol, while reducing the number of positions they differ.

alignment of two strings:

```
- - I - N T E R E S T I N G L Y
B I O I N F O R M A T I C S - -
```

Measuring Sequence Similarity: Global Alignment

The global alignment problem:
Let S and T be two strings over the alphabet Σ.
Determine an alignment A that **maximizes the similarity score** over all alignments of S and T.
The similarity of a pair of aligned symbols is given by by a **similarity score matrix** $\delta(x, y)$ with $x, y \in \Sigma \cup \{\_\}$.

alignment of two strings:

```
- - I - N T E R E S T I N G L Y
B I O I N F O R M A T I C S - -
```

What would be the score function δ, such that the process using δ would produce the same result for edit distance with previous σ?

Edit Distance is equivalent to Global Alignment

Lemma

Let σ be the cost matrix of the edit distance problem, and δ be the score matrix of the global alignment problem. If $\delta(x,y) = -\sigma(x,y)$ for all $x, y \in \Sigma \cup \{\_\}$, then the solution to the edit distance problem is equivalent to the solution to the string alignment problem.

Proof:
$n_{x,y}$ = number of occurences of operations (x,y).
Then, minimizing the edit distance $\sum_{x\in\Sigma\cup\{\_\}} \sum_{y\in\Sigma\cup\{\_\}} n_{x,y}\sigma(x,y)$ is the same as maximizing
$\sum_{x\in\Sigma\cup\{\_\}} \sum_{y\in\Sigma\cup\{\_\}} n_{x,y}(-\sigma(x,y)) = \sum_{x\in\Sigma\cup\{\_\}} \sum_{y\in\Sigma\cup\{\_\}} n_{x,y}\delta(x,y)$,
which is maximizing the alignment score.

Global Alignment Algorithms

Find a global alignment of two strings of length n and m that maximizes the similarity score.

- Brute force:
  Exponential Time. Why?
- Needleman-Wunsch algorithm (1970): dynamic programming
  Time: $O(nm)$
  Space: $O(nm)$
- banded Needleman-Wunsh algorithm:
  Restricting the number of **indel** (insertions/deletions) to at most d:
  Time: $O((n+m)d)$.
- Addressing space efficiency issue:
  An improvement is possible leading to $O(n+m)$ space.

Needleman-Wunsh Algorithm

Input strings: $S[1..n]$, $T[1,m]$
Output: Optimal alignment $V(n,m)$

Dynamic Programming recurrence relation to calculate $V(i,j)$: score of the optimal alignment between $S[1..i]$ and $T[1..j]$.
When $i=0$ or $j=0$, we need to align with the empty string:

$$\begin{array}{rcl} V(0,0) & = & 0 \\ V(0,j) & = & V(0,j-1)+\delta(\_,T[j]) \\ V(i,0) & = & V(i-1,0)+\delta(S[i],\_) \end{array}$$

When $i>0$ and $j>0$, last character is match/mismatch, delete or insert:

$$V(i,j) = \max \quad \{ \quad V(i-1,j-1)+\delta(S[i],T[j]), \\ V(i-1,j)+\delta(S[i],\_), V(i,j-1)+\delta(\_,T[j]) \quad \}$$

Example Needleman-Wunsh Algorithm

Score matrix δ given next. For all $x, y \in \Sigma$:
$\delta(x,x) = 2$ match, $\delta(x,y) = -1$ mismatch, if $x \neq y$, $\delta(\_,x) = \delta(x,\_) = -1$ indel
Fill out the dynamic programming matrix row by row, for strings: $S =$ACAATCC and $T =$AGCATGC.

$V(i,j)$:

	_	A	G	C	A	T	G	C
_								
A								
C								
A								
A								
T								
C								
C								

Solved in class. See book solution in page 34.

Note: use arrows to show the winner of the **max**, in one of 3 directions:
$\uparrow$ (deletion), $\leftarrow$ (insertion) or $\nwarrow$ (match/mismatch).

Consulting the completed table for Needleman-Wunsh algorithm in the last page (also in page 34 of the textbook), give the optimal global alignment and its score for the following prefixes of S and T, respectively:

1. ACAA and AGC
   score:
   global alignment:
2. ACAA and AGCAT
   score:
   global alignment:
3. ACAATC and AG
   score:
   global alignment:
4. S and T
   score:
   global alignment:

Time and Space Complexity

Time complexity:
To fill each position of the matrix, we use time O(1) and there are $n \cdot m$ matrix positions.
The running time is $O(n \cdot m)$.

Space complexity:
We need two matrices $(n+1) \times (m+1)$:
$V(i,j)$ and a matrix to store the direction chosen by the "max".
There are $O(n \cdot m)$ matrix positions required.

Improving Running Time for Sequence Alignment?

Known lower bounds for the problem:

- Aho, Hirschberg and Ullman (1976): For comparison-based methods that determine if symbols are equal or not, problem requires $\Omega(n \cdot m)$.
- Hirschberg (1978):
  If symbols are ordered and can be compared $(>, <, =)$, problem requires $\Omega(n \log n)$.

Current best method by Masek and Paterson (1980): $O(nm/\log n)$.
Given lower bounds, not much improvement possible on the general problem...
However, improvements can be done if we restrict the number of indel to be at most d:
Only a band of size $(2d+1)$ around the diagonal of the $V(i,j)$ matrix needs to be computed.

Banded Needleman-Wunsh Algorithm

If no more than $d = 3$ indel's (insertions/deletions) are allowed, only the following area of the matrix needs to be completed:

$V(i, j)$:

	_	A	G	C	A	T	G	C
_	*	*	*	*				
A	*	*	*	*	*			
C	*	*	*	*	*	*		
A	*	*	*	*	*	*	*	
A		*	*	*	*	*	*	*
T			*	*	*	*	*	*
C				*	*	*	*	*
C					*	*	*	*

Complete * positions in table as an exercise. See solution in page 35 of the textbook.

Size of $2d + 1$ band is $O((n + m)d)$. So assuming that d is small w.r.t. n and m, the running time got reduced from $O(n \cdot m)$ to $O((n + m) \cdot d)$

Space Efficiency Improvement

For long sequences, using $O(n \cdot m)$ space is a problem. Say, comparing the human and mouse genome would take space proportional to 9×10^{18} matrix positions.

To calculate $V(i,j)$ alone, we can simply store two rows of the matrix at a time (previous and current row), easily reducing space to $O(m)$.
Call such algorithm: Algorithm NWscore (S,T)

To compute the optimal alignment (not only its score), we need to use another method. Hirschberg's algorithm (1975) reduces the space requirement to be $O(n+m)$, which in the given example would yield space reduced to 6×10^9.

Hirschberg's Algor. for $O(n+m)$-space global alignment

```
**Algorithm FindMid**($mid, S[i_1..i_2], T[j_1..j_2]$)
    **NWScore**($S[i_1..mid], T[j_1..j_2]$): returns last row of $V(x,y)$ in $NW[0..m]$
    **NWScore**($(S[mid+1..i_2])^r, (T[j_1,j_2])^r$):
            returns reverse of last row of $V(x,y)$ in $SE[1..m+1]$
    return $j$, $0 \leq j \leq m$, that maximizes $NW[j]+SE[j+1]$.
**Algorithm Alignment**($S[i_1..i_2], T[j_1..j_2]$)
    if $i_1 \geq i_2$ return **NW**($S[i_1..i_2], T[j_1,j_2]$)    regular NW algorithm
    $mid = (i_1+i_2)/2$
    $j =$ **FindMid**($mid, S[i_1..i_2], T[j_1..j_2]$)
    return concatenation of **Alignment**($S[i_1..mid], T[j_1..j]$) and
                            **Alignment**($S[mid+1..i_2], T[j+1..j_2]$)
```

Space: $O(n+m)$
Time: $Time(n,m) = c \cdot nm + Time(n/2, j) + Time(n/2, m-j)$
Solving the recurrence relation: $Time(n,m) \in O(nm)$.
(Algorithm understanding/analysis explained in class)

Example Hirschberg's Algorithm

S =ACTGACCT T =TGTCC
scores: match$= +2$; mismatch/indel$= -1$

Calculate values row by row, only keeping 2 rows at a time:

	-	T	G	T	C	C
-	0	-1	-2	-3	-4	-5
A	-1	-1	-2	-3	-3	-4
C	-2	-2	-2	-3	-1	-1
T	-3	0	-1	0	-1	-2
G	-4	-1	2	1	0	-1
j	0	1	2	3	4	5

	-	C	C	T	G	T
-	0	-1	-2	-3	-4	-5
T	-1	-1	-2	0	-1	-2
C	-2	1	1	0	-1	-2
C	-3	0	3	2	1	0
A	-4	-1	2	2	1	0
j	6	5	4	3	2	1

j	0	1	**2**	3	4	5
$NW[j] + SE[j+1] =$	-4	0	**4**	3	-1	-5

Findmid returns $j = 2$

Another way of looking at the previous step:
NWScore of first half of S and reverse of second half of S, each with T

(mid element of S is maked bold)

				j=2				
		-	T	G	T	C	C	-
	-	0	-1	-2	-3	-4	-5	
	A	-1	-1	-2	-3	-3	-4	
	C	-2	-2	-2	-3	-1	-1	
	T	-3	0	-1	0	-1	-2	
mid=4	**G**	-4	-1	**2**	1	0	-1	
	A		0	1	**2**	2	-1	-4
	C		-2	-2	-1	3	0	-3
	C		-2	-1	-1	0	1	2
	T		-2	-1	0	-2	-2	-1
	-		-5	-4	-3	-2	-1	0

	j=0			
	-	T	G	-
-	0	-1	-2	
A	-1	-1	-2	
C	**-2**	-2	2	
T		**4**	1	-2
G		1	2	-1
-		-2	-1	0

	j=0	
	-	-
-	0	
A	**-1**	
C		**-1**
-		0

		j=1		
	-	T	G	-
-	0	-1	-2	
T	-1	**2**	1	
G		1	**2**	-1
-		-2	-1	-0

NW(A,-)= (A,-)
NW(C,-)= (C,-)
concat: (AC,- -)

NW(T,T)=(T,T)
NW(G,G)=(G,G)
concat: (TG,TG)

concat: (ACTG,- -TG)

			j=2		
	-	T	C	C	
-	0	-1	-2	-3	
A	-1	-1	-2	-3	
C	-2	-2	1	0	
C		0	1	1	-2
T		0	-2	-1	-1
-		-3	-2	-2	0

		j=1		
	-	T	C	-
-	0	-1	-2	
A	-1	**-1**	-2	
C		1	**2**	-1
-		-2	-1	0

		j=1	
	-	C	-
-	0	-1	
C	-1	**2**	
T		-1	**-1**
-		-1	0

NW(A,T)= (A,T)
NW(C,C)= (C,C)
concat: (AC,TC)

NW(C,C)=(C,C)
NW(T,-)=(T,-)
concat: (CT,C-)

concat: (ACCT,TCC-)

concatenating (ACTG,- -TG) and (ACCT,TCC-) leads to final answer:
(ACTGACCT, - -TGTCC-)

Example Hirschberg's Algorithm (cont'd)

(**ACTG**ACCT,**TG**TCC)$_{j=2}$

(**AC**TG,TG)$_{j=0}$ (**AC**CT,TCC)$_{j=2}$

(**A**C,-)$_{j=0}$ (**T**G,TG)$_{j=1}$ (**A**C, TC)$_{j=1}$ (**C**T,C)$_{j=1}$

(A,-)(C,-) (T,T) (G,G) (A,T) (C,C) (C,C) (T,-)

final alignment:

A	C	T	G	A	C	C	T
-	-	T	G	T	C	C	-

Note that the optimal value of 4, found at the first call to **FindMid**, is the score of this alignment.

Sketch: Analysis of Hirschberg's Algorithm

Each call to **FindMid** takes $O(s \cdot m)$ time, where s is the size of string S; say $T_{FindMid}(s,m) \leq c \cdot s \cdot m$, for some constant $c \geq 1$, and $m \geq 1$.

$$\begin{aligned} Time(n,m) &= c'm + d, \quad \text{if } n = 1 \ \ (\text{for some constants } c', d) \\ &= cnm + Time(n/2, j) + Time(n/2, m-j), \text{if } n = 2^k \geq 2 \end{aligned}$$

Theorem

For all $n \geq 1, m \geq 0$, $Time(n,m) \leq 2cnm + c'm + dn = O(nm)$.

Proof: (by induction on $n = 2^k$)
Basis: $n = 1$, trivial.
Ind. Step: Let $N \geq 2$. Assume the inequality holds for every $n = 2^l < N$.
$Time(N,m) = cNm + Time(N/2, j) + Time(N/2, m-j)$
$\leq cNm + (2c(N/2)j + c'j + d(N/2)) + (2c(N/2)(m-j) + c'(m-j) + d(N/2)) =$
$cNm + cNm + c'm + dN.$

More on Global Alignment

Relevant special cases of global alignment:

1. **Longest Common Subsequence (LCS)**:
   Find a subsequence of X and Y which is the longest possible.
   Recall subsequence $\neq$ substring, characters need not be contiguous.
   X :CATPAPLTE and Y : XAPZPLEG yields LCS: APPLE
   Global alignment with:
   score for match $= 1$, score for mismatch $= -\infty$, score for indel $= 0$

2. **Hamming Distance**:
   Number of mismatches between two strings of same length.
   hamdist(TONED,ROSES)$=$3 since they differ in positions 1,3,5.
   Global Alignment with:
   score for match $= 1$, score for mismatch $=$0, score for indel$=-\infty$.

Local Alignment

The local alignment problem:
Let $S[1..n]$ and $T[1..m]$ be two strings over the alphabet Σ.
Determine a pair of substrings A of S and B of T with the highest alignment score.

Solving this by brute force:

- Find all substrings A of S and B of T;
- Compute a global alignment of A and B;
- Return the substring pair (A, B) with maximum score.

Time complexity for the brute force method: There are $\binom{n}{2} + n + 1$ choices for A and $\binom{m}{2} + m + 1$ choices for B, time is $O(\binom{n}{2}\binom{m}{2}mn) = O(n^3m^3)$.
Too slow!!!

Local Alignment: Smith-Waterman Algorithm(1981)

For $0 \leq i \leq n$, $0 \leq j \leq m$, define
$V(i,j)$= *maximum score of the global alignment of A and B over all substrings A of S that end at i and all substrings B of T that end at j. (by definition always empty substrings is a valid choice for both)*
When $i = 0$ or $j = 0$, the best alignment aligns empty substrings:

$$\begin{aligned} V(0,j) &= 0 \quad \text{for } 0 \leq j \leq m \\ V(i,0) &= 0 \quad \text{for } 0 \leq i \leq n \end{aligned}$$

When $i > 0$ and $j > 0$, best scenario can be given by both empty substrings (score of 0) or the best alignment where last character (i and j, respectively) is match/mismatch, delete or insert:

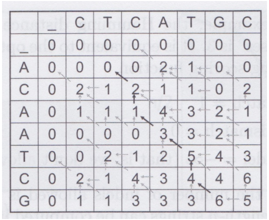
$$V(i,j) = \max \quad \{ \quad 0, V(i-1,j-1) + \delta(S[i],T[j]), \\ V(i-1,j) + \delta(S[i],\_), V(i,j-1) + \delta(\_,T[j]) \quad \}$$

Optimal Local alignment score: $\max_{i,j} V(i,j)$.

Local Alignment Example

S =ACAATCG; T =CTCATGC; scores: match +2; indel/mismatch -1.

Filling row by row, matrix $V(i,j)$: $\nwarrow$(match/mismatch); $\uparrow$ (delete); $\leftarrow$ (insert)



	_	C	T	C	A	T	G	C
_	0	0	0	0	0	0	0	0
A	0	0	0	0	2	1	0	0
C	0	2	1	2	1	1	0	2
A	0	1	1	1	4	3	2	1
A	0	0	0	0	3	3	2	1
T	0	0	2	1	2	5	4	3
C	0	2	1	4	3	4	4	6
G	0	1	1	3	3	3	6	5

Two optimal local alignments (score 6) can be found.

Optimal solution:	Back-tracing from table:
$V(7,6)$	CAATCG C_AT_G
$V(6,7)$	CAAT_C C_ATGC

Time and Space complexity for Smith-Waterman Algorithm

Analysis is identical to the one for optimal global alignment:

Space and Time: $O(m \cdot n)$

A similar space reduction is possible to $O(m + n)$.

Semi-global alignment

It is like a global alignment, but spaces at the beginning and/or end of an alignment are ignored (cost of 0).
Example:
$S =$ ATCCGAACATCCAATCGAAGC
$T =$ AGCATGCAAT

Optimal global alignment: score$= 9 * 2 + 1 * -1 + 11 * -1 = 6$

```
ATCCGAACATCCAATCGAAGC
A___G__CATGCAAT______
```

Optimal semi-global alignment: score $= 8 * 2 + 1 * -1 + 1 * -1 = 14$

```
ATCCGAA_CATCCAATCGAAGC
_______AGCATGCAAT______
```

Semi-global alignment bioinformatics applications

Ignoring flanking (starting or trailing) spacing may be desirable in some situations.

- **Prokaryotes:** aligning genes.
- **Eukaryotes:** aligning an exon to the original gene sequence; spaces adjacent to the exon may be due to an untranslated region or introns.
- **Sequence assembly:** ignore starting spaces of the first sequence and trailing spaces of the second sequence.
  Example:

```
_________ACCTCACGATCCGA
TCAACGATCACCGCA_________
```

  In this case, the semiglobal score can helps us deciding whether the two DNA segments are overlapping.

Semi-global Alignment Algorithm:

Modify the global alignment algorithm in the following way:

spaces not charged	implementation:
spaces at the beginning of S	initialize first row with 0's
spaces at the end of S	look for the maximum in the last row
spaces at the beginning of T	initialize first column with 0's
spaces at the end of T	look for the maximum in the last column

Gap Penalty

So far, we penalize l contiguous spaces (1 gap) the same as l "dispersed" spaces. It makes sense to reduce the penalty for contiguous spaces: e.g. mutations may cause the insertion or deletion of a substring which may be as likely as indel of a single base.
Therefore, we will consider a **general gap penalty** $g(q)$ for a gap of length q. Example: match: 2, indels:-1

```
A - C A A C T C G C C T C C
A G C A - - - - - - - T G C
```

$g(q)$	score for above alignment
q	$10-1+(-1-(7))=1$
1	$10-1+(-1-(1))=7$
$1+(1/10)q$	$10-1+(-1-(1.7))=6.3$

General Gap Penalty Model

Using $g(q)$ as gap penalty, we need to slightly modify global alignment:
When $i = 0$ or $j = 0$, we have:

$$\begin{aligned} V(0,0) &= 0 \\ V(0,j) &= -g(j) \\ V(i,0) &= -g(i) \end{aligned}$$

When $i > 0$ and $j > 0$, last character is match/mismatch, end of a deletion gap or end of an insertion gap:

$$V(i,j) = \max \quad \{ \quad V(i-1,j-1) + \delta(S[i],T[j]), \\ \max_{0 \le k \le i-1} \{V(k,j) - g(i-k))\} \quad (\text{delete } S[k+1..i]) \\ \max_{0 \le k \le j-1} \{V(i,k) - g(j-k)\} \quad (\text{insert } T[k+1,j]) \quad \}$$

Complexity: Time $O(nm(n+m))$, Space $O(nm)$.

Affine Gap Penalty Model

$h=$ penalty for initiating the gap
$s=$ penalty proportional to the length of the gap

$$g(q) = h + qs$$

Better time complexity is possible: $O(nm)$.

For every table position, we will keep information for:
$G(i,j)$: score for $S[1..i]$ and $T[1..j]$ with $S[i]$ **aligning with** $T[j]$
$F(i,j)$: ....(same)........... $S[i]$ **matching a space**
$E(i,j)$: ....(same)........... $T[j]$ **matching a space**

Dynamic Programming for Global Alignment with Affine Gap Penalty

$V(0,0) = 0;$
$V(i,0) = F(i,0) = -g(i) = -h - is$
$V(0,j) = E(0,j) = -g(j) = -h - js$
$E(i,0) = F(0,j) = G(i,0) = G(0,j) = -\infty$

When $i > 0$ and $j > 0$:
$V(i,j) = \max\{G(i,j), F(i,j), E(i,j)\}$
$G(i,j) = V(i-1,j-1) + \delta(S[i], T[j])$
$F(i,j) = \max\{F(i-1,j) - s, V(i-1,j) - h - s\}$
$E(i,j) = \max\{E(i,j-1) - s, V(i,j-1) - h - s\}$

Complexity: Time $O(nm)$; Space $O(nm)$.

Scoring Functions

To measure similarity between two sequences need to choose the scoring function $\delta(x, y)$.

• Scoring function for DNA
simpler; positive score for match and negative score for mismatch

- different algorithms use different values:

score:	match	mismatch	best for homol. align. with
NCBI-BLASTN	$+2$	-1	95% identity
WU-BLASTN,FastA	$+5$	-4	65% identity

- some use transition/traversion matrix:
  remember purines are A,G, and pyrimidines are C,T.
  scores: match +1; mismatch in same group -1; mismatch across groups -5.

Scoring Functions (cont'd)

- **Scoring function for protein:** two approaches
  1. similarity score based on chemical/physical properties of amino acids. assume: 2 amino acids w/similar properties are more likely substituted
  2. assign similarity purely based on statistics. a and b are similar if they have a big score $\log \frac{O_{a,b}}{E_{a,b}}$ (log of: observed substitution frequency over expected substitution frequency)
     - Point accepted mutation (PAM) score matrix - (Dayhoff, 1970's) family of matrices: PAM-i (higher i, assume higher evolut. distance) calculated by observing differences in closely related proteins.
     - BLOSUM (BLOck SUbstitution Matrix) - (Hernikoff & Hernikoff, 1992) family of matrices: BLOSUM p (higher p, for smaller evolut. distance) better for evolutionary divergent sequences; calculated using "blocks" of highly conserved sequences found in multiple protein alignments.

  BLOSUM 62 is the default matrix for BLAST 2.0