Crossroads
The ACM Student Magazine

Aligning DNA Sequences Using Dynamic Programming

by ***Eric C. Rouchka***

Biological Sequences

DNA and Protein Sequences

High-throughput biological data come in a variety of forms. The most fundamental data available are the DNA and protein sequences that form the blocks for life. Each cell within our bodies contains the entire genetic blueprint that makes each of us who we are. This blueprint, known as the genome, is composed of deoxyribonucleic acid, or DNA.

DNA can be thought of as a four-letter alphabet with the symbols A, C, G, and T representing the four possible nucleotides: adenine, cytosine, guanine, and thymine. The genome of an organism is composed of a linear sequence of A, C, G, and T that can be decoded from an individual's chromosomes. For the human genome, this is on the order of 3.2 billion bases. Within a genome, special sets of instructions known as genes code for proteins that provide structural and functional properties. Inside of a gene-coding region, every three bases codes for an amino acid. Since there are four possibilities for each position (A, C, G, or T), there are 4 * 4 * 4 = 64 potential three-base patterns, or codons.

One three-base pattern, ATG, is special in the sense that it tells the cellular machinery where to begin translation from DNA to protein sequence. Three other sequences, TAA, TAG, and TGA, indicate stopping points. This leaves 60 codons. There are only twenty amino acids, however, so multiple codons result in the same amino acid, leading to a redundancy in the genetic code (see Figure 1). This process of translating a DNA sequence into an RNA intermediary and then into a protein sequence is known as the central dogma of molecular biology.

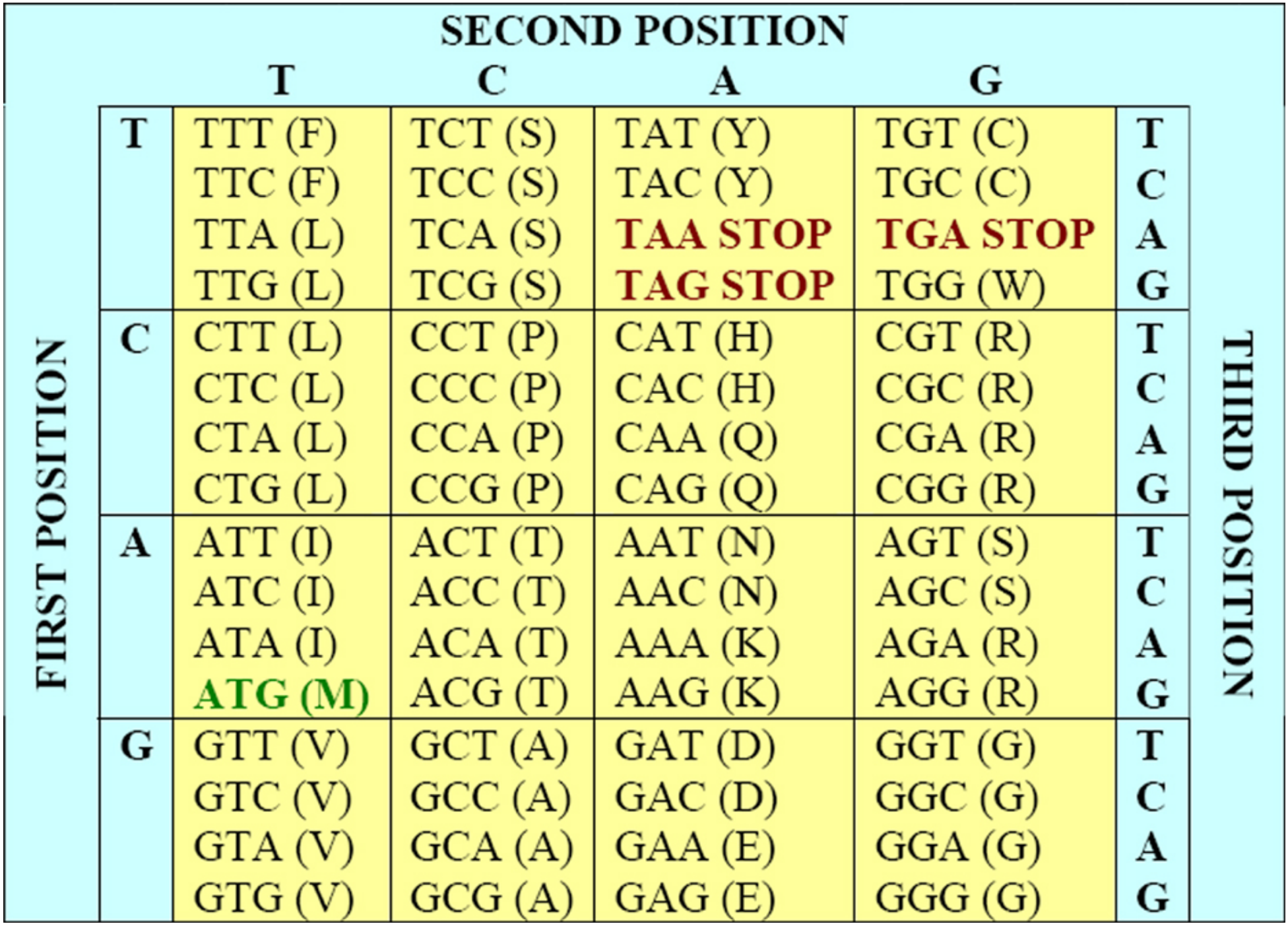


Figure 1: Redundancy of genetic code.

Analogy of the Central Dogma to Compilers

The machinery involved in DNA and protein sequences can be thought of in the same way as a programming language. Noncoding regions are similar to comments, while genes represent functions. The functions that get called in a program depend upon the parameters set at runtime, much in the same way that the genes that get turned on and translated into proteins are dependent upon internal and external factors, or stimuli. Code is taken from one form (such as C++) and translated into assembly language for a particular platform. Biologically, the programming code would be the DNA, while the assembly language would be the protein sequence.

Pairwise Sequence Alignment Using Dynamic Programming

Large-scale genome sequencing projects, such as the Human Genome Project, yield high volumes of sequence data on the order of billions of bases of A,C,G,T for each genome. For humans, this amounts to 3.2 billion bases spread out over 23 chromosomal pairs. Once scientists have these sequences in hand, it is necessary to figure out the meaning of these sequences, particularly in the gene coding regions. Databases such as GenBank at the National Center for Biotechnology Information have been set up to hold all known biological sequences, along with annotation information that might yield insight into a particular sequence's function. To give an idea about how much known sequence data is available, release 154.0 (June 15, 2006) of GenBank contains over 58 million sequences totaling over 63 billion bases.

Once a biologist has a DNA or protein sequence in hand, the first question typically asked concerns the actual function of the sequence. In order to help answer this question, a good starting point is to compare the sequence to a database of known sequences, such as GenBank, since sequences that share similarity at the nucleotide or amino acid level are also likely to share similar functions. Such a search can be performed by pairwise sequence alignment—aligning two sequences at a time.

Suppose that we have a sequence $A = a_1 \ldots a_M$ represented by GGATCGA (in this case $M = 7$), and we want to find its optimal alignment to a sequence $B = b_1 \ldots b_N$ represented by GAATTCAGTTA ($N = 11$). One way this could be performed is by looking at all possible subsets of each sequence and comparing each of those to every single possible subset of the other. Since there are 2^M possible subsets of the first sequence and 2^N of the second, $2^M * 2^N$ or $2^{(M+N)}$ total comparisons must be made. Even with the example sequences, this is 2^{18} comparisons. This is too computationally intense, even for small sequences.

In 1970, an approach using a traditional computer science technique known as dynamic programming was proposed [**8**]. Dynamic programming problems (such as the subset sum and knapsack algorithms) are known to have an optimal and overlapping substructure property such that the optimal solution to a problem builds upon optimal solutions for subsets of the problem space. For pairwise sequence alignment, this greatly reduces the overall search space by finding the optimal global alignment between two sequences of length M and N in $O(M * N)$ time. This approach relies on the recursive property that the best way to align two sequences builds upon the best way to align the prefixes of the two sequences.

Dynamic Programming Take I: Global Sequence Alignment

When aligning sequences, the most fundamental issue is finding the optimal structure for aligning

two sequences across their entirety, which is referred to as global pairwise sequence alignment. The dynamic programming solution creates a two-dimensional score matrix, *C*, with *M* + 1 rows and *N* + 1 columns and proceeds to fill it in using a set of scoring rules for matching bases, mismatching bases, gaps in sequence *A*, or gaps in sequence *B*. Dynamic programming for pairwise sequence alignment requires three steps: initialization, matrix fill, and traceback.

Initialization

The score matrix, *C*, is initialized by placing each character of sequence *B* above each column in the matrix, beginning with the second column, and placing each character of sequence *A* before each row, beginning with the second row. The value at position $C_{i,j}$ will yield the maximum score for aligning the prefix $a_1 \ldots a_i$ of sequence *A* to the prefix $b_1 \ldots b_j$ of sequence *B*. The values for $C_{0,c}$ and $C_{r,0}$ for $c = 1..N$ and $r = 1..M$ are initialized to 0, eliminating the effect of gaps before each sequence. Figure 2 illustrates the dynamic programming matrix *C* after initialization using the two example sequences.

		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0											
G	0											
A	0											
T	0											
C	0											
G	0											
A	0											

Figure 2: Initialization of dynamic programming matrix.

Matrix Fill

The next step of the dynamic programming algorithm is to fill in the scores for the remainder of the matrix. Let the scoring scheme *d* be defined as follows:

- $d(a_i,b_j)$: The score of matching the character a_i with the character b_j
- $d(a_i,e)$: The score of matching the character a_i with a gap
- $d(e,b_j)$: The score of matching the character b_j with a gap

For DNA sequences, one typical scoring scheme uses the following rules: $d(a_i,b_j) = 5$ if $a_i = b_j$; otherwise, $d(a_i,b_j) = -3$. $d(a_i,e) = d(e,b_j) = -4$, which represents a gap score. Combined, $C_{i,j}$ can be defined as

$$C_{i,j} = Max(C_{i-1,j-1} + d(a_i,b_j), C_{i-1,j} + d(a_i,e), C_{i,j-1} + d(e,b_j))$$

Thus, there are three different ways to arrive at each cell: through a match or mismatch, by adding a gap in sequence *A*, or by adding a gap in sequence *B*.

For the example sequences *A* and *B*, the value in cell $C_{1,1}$ is defined as Max(0 + 5, 0 - 4, 0 - 4) = 5, since a_1 = G and b_1 = G. $C_{1,2}$ is the value Max(0 - 3, 0 - 4, 5 - 4) = 1. For each cell, the path(s) leading to the maximal value should be stored for the traceback step. Figure 3 illustrates the values for the first two cells.

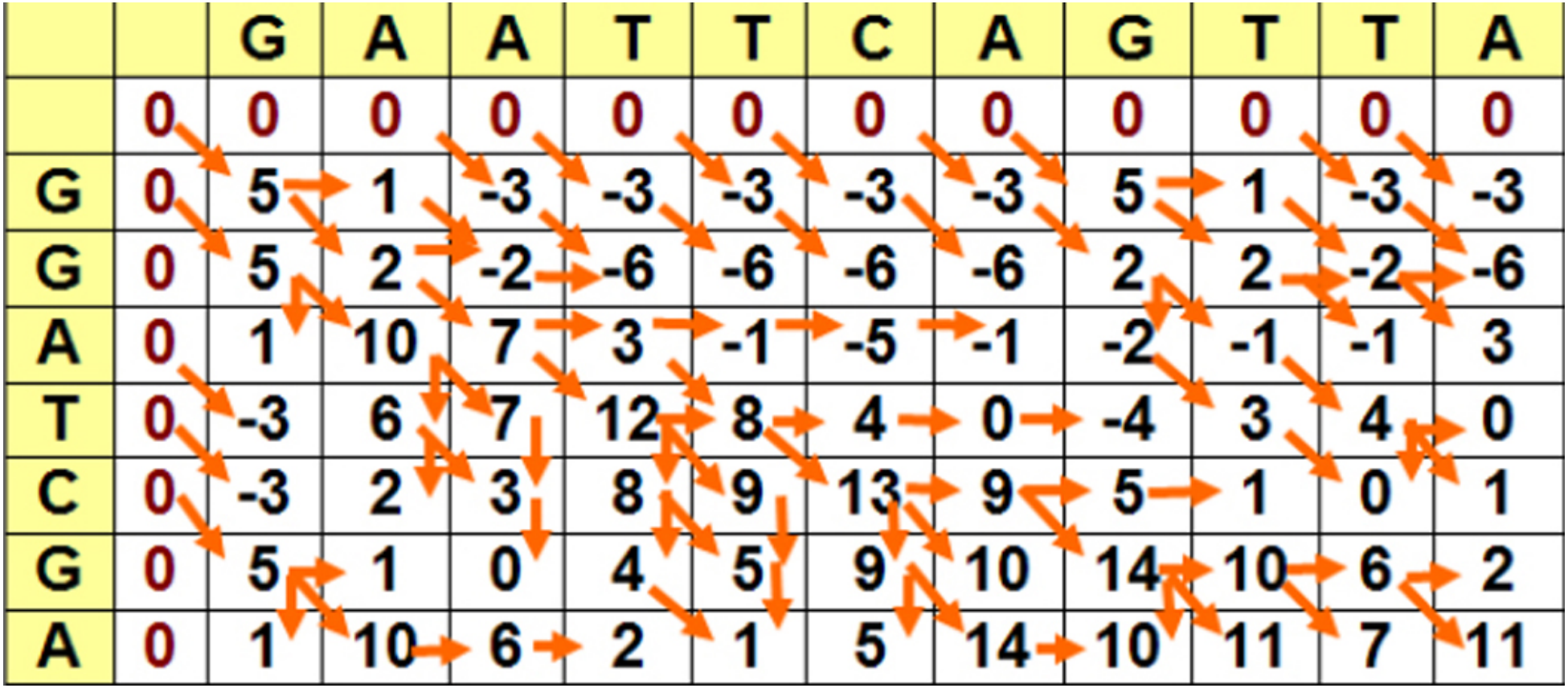
		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	5	1									
G	0											
A	0											
T	0											
C	0											
G	0											
A	0											

Figure 3: Initialization of first two cells.

Traceback

Once the matrix has been completely filled (see Figure 4), the value at position $C_{M,N}$ contains the highest possible global alignment score given the scoring scheme. In the traceback step, the alignment yielding the maximum score is reconstructed. This is done by looking at the direction in which each cell was reached. Two stacks can be used to help reconstruct the alignment, one for each sequence. If the cell was reached by a match or mismatch, then the character a_i is pushed onto $Stack_A$, and the character b_j is pushed onto $Stack_B$. If the cell was reached by a gap in sequence A, then a '-' character is placed on $Stack_A$ and the character b_j is pushed onto $Stack_B$; otherwise, a '-' character is placed on $Stack_B$ and the character a_i is pushed onto $Stack_A$, indicating a gap in sequence B. When the cell $C_{0,0}$ is reached, the items from each stack can be popped off to reconstruct the alignment. Figure

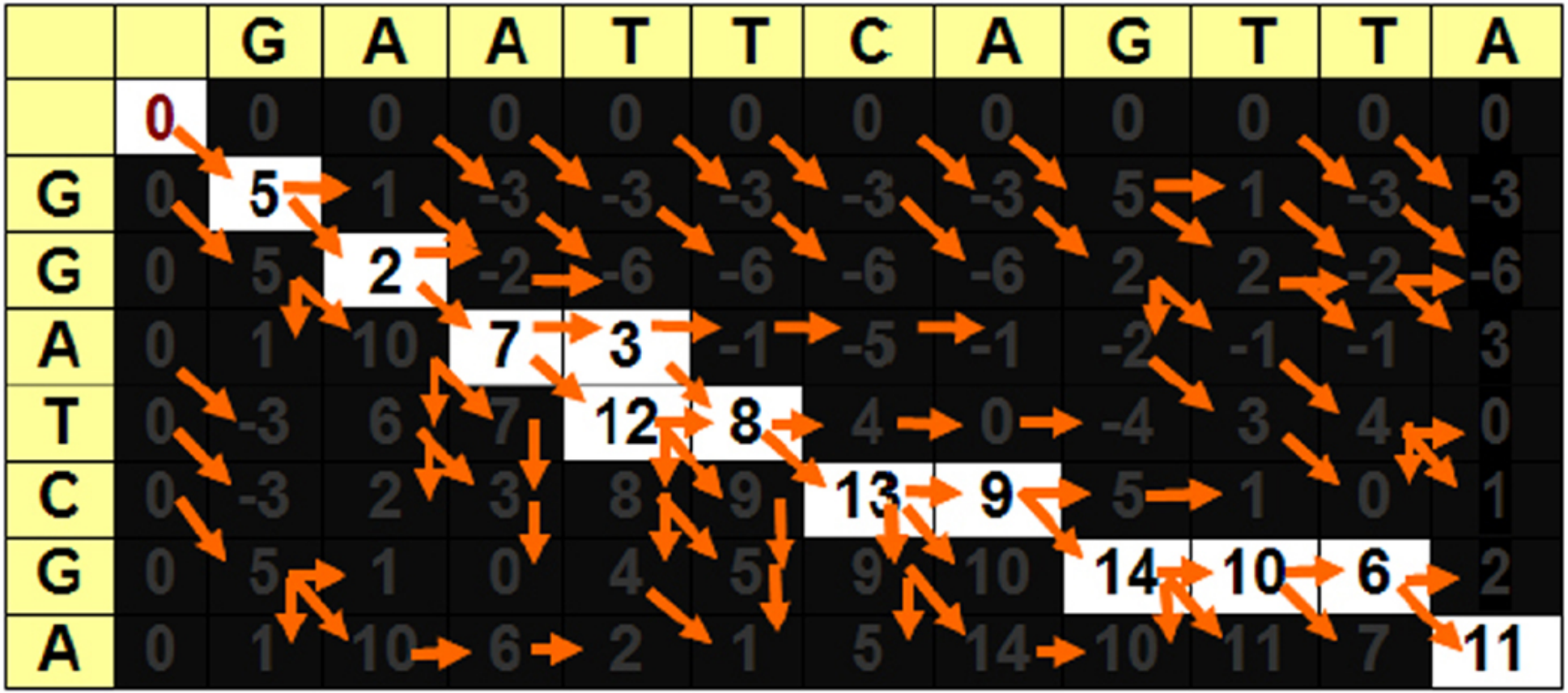
5 indicates the traceback for the two sequences, along with the pairwise global alignment.



		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	5	1	-3	-3	-3	-3	-3	5	1	-3	-3
G	0	5	2	-2	-6	-6	-6	-6	2	2	-2	-6
A	0	1	10	7	3	-1	-5	-1	-2	-1	-1	3
T	0	-3	6	7	12	8	4	0	-4	3	4	0
C	0	-3	2	3	8	9	13	9	5	1	0	1
G	0	5	1	0	4	5	9	10	14	10	6	2
A	0	1	10	6	2	1	5	14	10	11	7	11

Figure 4: Completed score matrix with trace arrows. Shown in orange are the cell locations from which the current value was derived.

5 (A)



5 (B)

GAATTCAGTTA

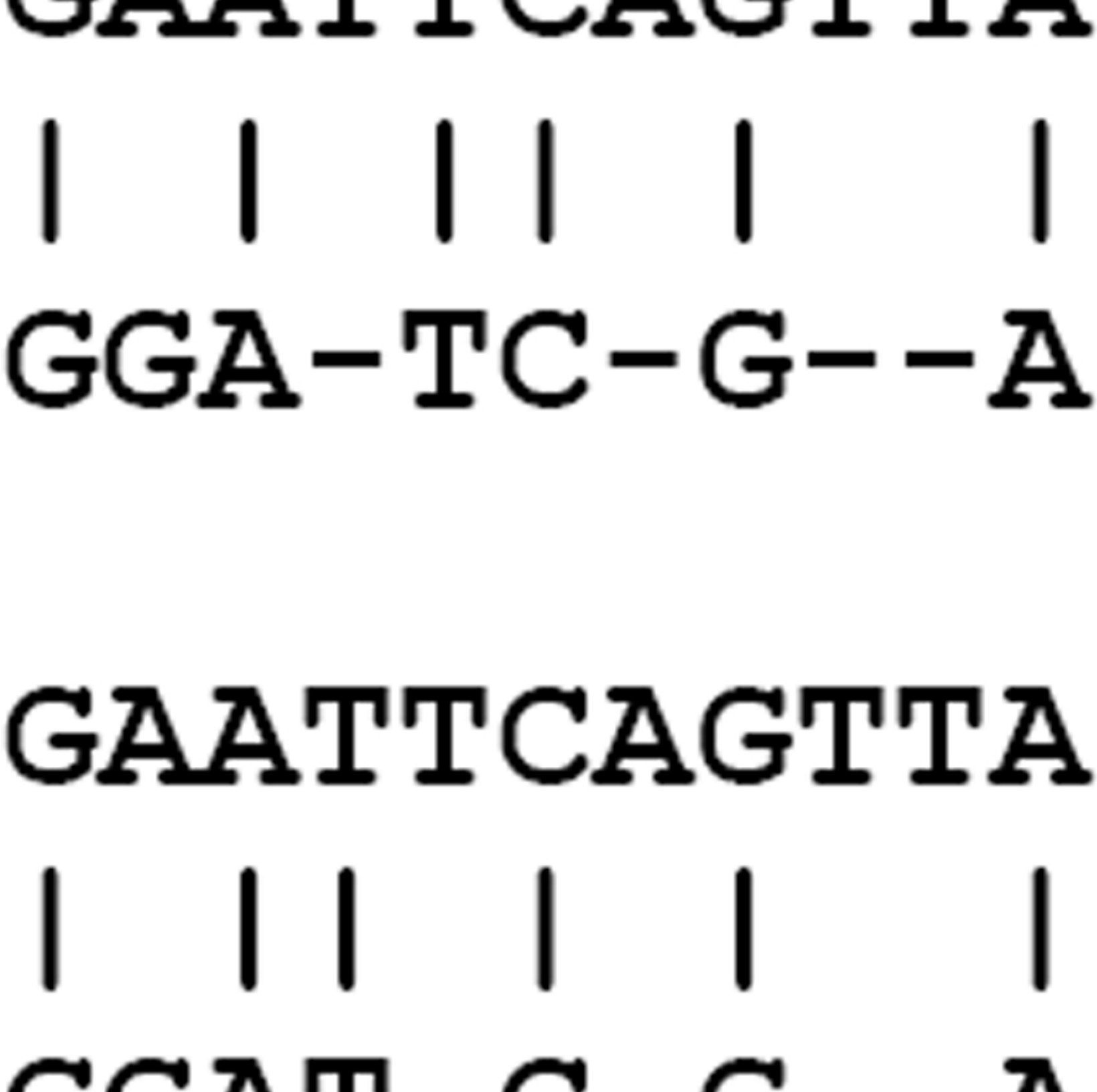

GAATTCAGTTA
GGA-TC-G--A
GAATTCAGTTA

Figure 5: Traceback for multiple alignment. 5(A) shows the traceback for the alternate multiple alignment of sequences A and B, while sequence 5(B) shows the resulting alignments.

Dynamic Programming Alignment Take II: Local Sequence Alignment

Often two sequences share only a select region of similarity rather than the entire length. Therefore a locally optimal—rather than globally optimal—solution is desired to find the optimally aligning substrings between the two sequences. Local pairwise sequence alignment can be determined using modifications to the Needleman-Wunsch Algorithm, known as the Smith-Waterman algorithm [**13**]. This approach makes a minor modification to the scoring scheme by resetting the values of a cell to 0 whenever the maximum score becomes negative. The modified scoring scheme is as follows:

$$C_{i,j} = Max(0, C_{i-1,j-1} + d\,(a_i,b_j), C_{i-1,j} + d\,(a_i,e), C_{i,j-1} + d(e,b_j))$$

In order to find the optimal local alignment score, the matrix is searched and the location of the highest value is noted. Reconstruction of the local alignment begins at this cell, tracing back until a zero cell has been reached, such that the zero was caused by a resetting of the alignment scores. For the example sequences, the local alignment matrix and resulting optimal alignments appear in Figure 6.

6 (A)

		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	5	1	0	0	0	0	0	5	1	0	0
G	0	5	2	0	0	0	0	0	5	2	0	0
A	0	1	10	7	3	0	0	5	1	2	0	5
T	0	0	6	7	12	8	4	1	2	6	8	4
C	0	0	2	3	8	9	13	9	5	2	4	5
G	0	5	1	0	4	5	9	10	14	10	6	2
A	0	1	10	6	2	1	5	14	10	11	7	11

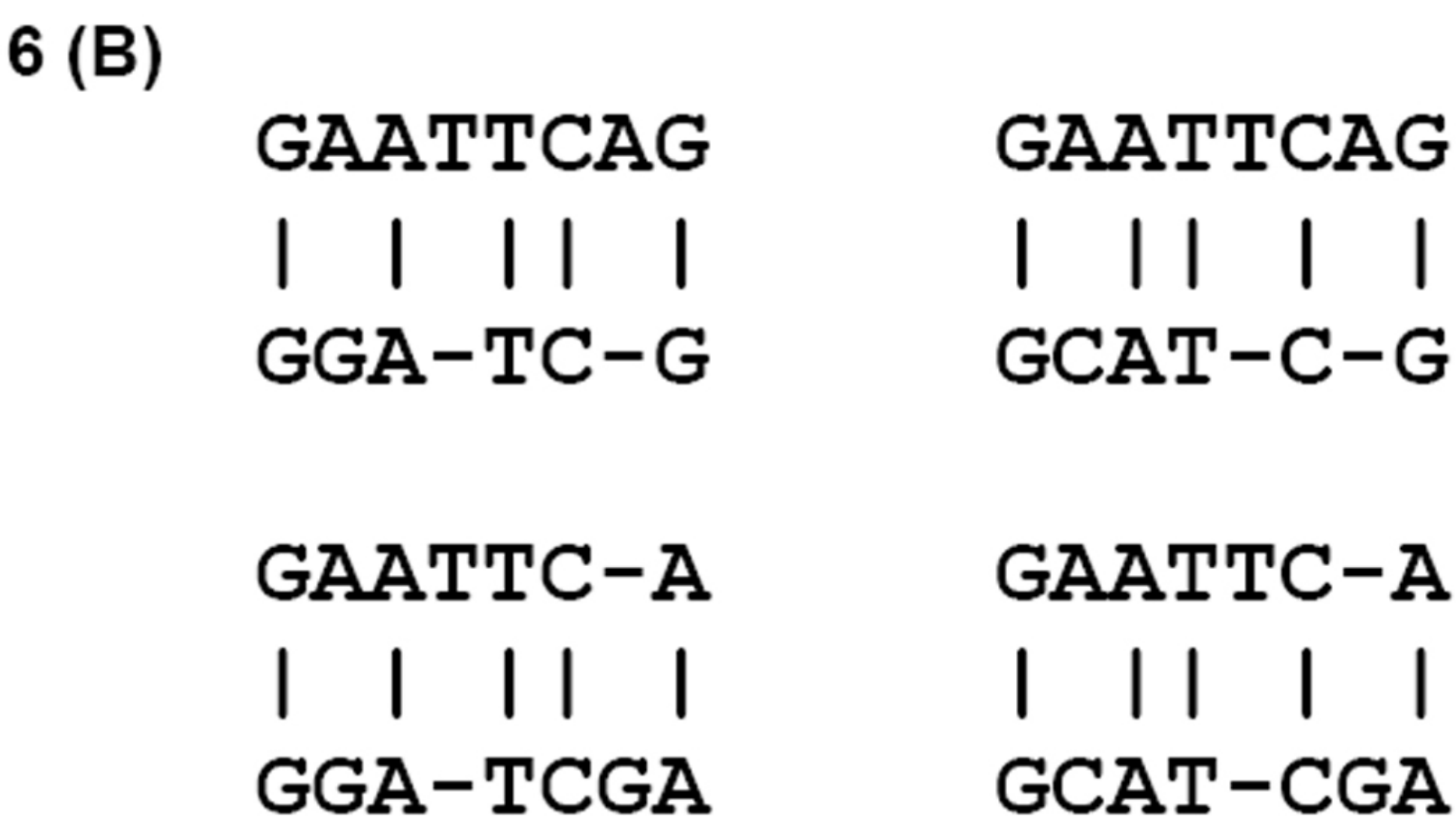


Figure 6: Local sequence alignment. 6(A) shows the local alignment matrix while 6(B) shows the resulting sequence alignments.

Dynamic Programming Alignment Take III: Substitution Matrices

The previous examples show a straight match score if the characters a_i and b_j are equal, otherwise a mismatch score is applied. But in many cases, there may be differing degrees to which sequences align. For instance, if you were to take a set of words, there may be some acceptable differences, such as substituting one vowel for another.

Allowable differences show up in DNA as well, since amino acid sequences diverge over evolutionary time. Some mutations are favorable, in the sense that the amino acids have similar properties and thus do not affect the overall structure of the resulting protein. Studies have been performed to find

which substitutions are allowable. Two such matrices are often used when looking at amino acid sequence alignments: point-accepted mutation (PAM) matrices [2] and BLOSUM matrices [4]. These matrices can be used as lookup tables for the score of aligning one amino acid with another. Figure 7 shows the BLOSUM62 matrix given the single letter amino acid code. In addition to the PAM and BLOSUM matrices for amino acid sequences, other lookup tables can be constructed depending on the types of sequences being compared. By using these substitution matrices, the resulting optimal alignments take into account more biologically allowable mutations between two sequences that change the actual characters, but not the overall sequence structure.

	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W	
C	9																				C
S	-1	4																			S
T	-1	1	5																		T
P	-3	-1	-1	7																	P
A	0	1	0	-1	4																A
G	-3	0	-2	-2	0	6															G
N	-3	1	0	-2	-2	0	6														N
D	-3	0	-1	-1	-2	-1	1	6													D
E	-4	0	-1	-1	-1	-2	0	2	5												E
Q	-3	0	-1	-1	-1	-2	0	0	2	5											Q
H	-3	-1	-2	-2	-2	-2	1	-1	0	0	8										H
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	5									R
K	-3	0	-1	-1	-1	-2	0	-1	1	1	-1	2	5								K
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	5							M
I	-1	-2	-1	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	4						I
L	-1	-2	-1	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	4					L
V	-1	-2	0	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	4				V
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	6			F
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	7		Y
W	-2	-3	-2	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	11	W

Figure 7: BLOSUM62 amino acid scoring matrix. Values above zero represent favorable substitutions while values below zero are unfavorable.

Dynamic Programming Alignment Take IV: Concave Gap Penalties

The dynamic programming approaches discussed up to this point treat all gaps the same: The score for *k* gaps of length 1 is the same as the score for one gap of length *k*. Biologically, however, single, large gaps are more likely to occur than many small gaps. In order to account for this, concave gap penalty scores have been introduced [**6**]. Concave gap penalties give preference to the presence of fewer, longer gaps by not penalizing adding onto a gap (gap extension) as much as beginning a new gap (gap opening). In this scheme, the cost of a gap, *w*, can be calculated as a function of the gap length, *k*.

The general form of the concave gap function is $w(k) = a + B * f(k)$ where *a* is the gap opening penalty, and $f(k)$ is the concave gap function. Concave gap scoring schemes add complexity to the dynamic programming algorithm, resulting in a complexity of $O(M * N * (M + N))$. The added complexity comes from the increased possibilities for reaching a particular cell in the scoring matrix. Now a cell can be reached not only from a match/mismatch or a direct insertion or deletion, but also from insertions or deletions that extend a gap that started in any of the positions in the same row or same column. The complexity of concave gap schemes can be reduced by using "minimum envelopes," which are not covered here.

Affine gap models are one specific instance of incorporating concave gaps [**3**]. For the affine gap model, the penalty for a gap is assigned according to the function $w = a + f(k)$. Sequence alignment using an affine gap model can be determined in $O(M * N)$ time by incorporating into each cell in the dynamic programming matrix an additional parameter for the length of the gap resulting in the minimal score up to that point. Incorporation of concave gap penalties, such as the affine gap model, allows the resulting alignment to be more biologically relevant by accounting for fewer yet longer gapped regions.

Discussion

The dynamic programming approach to pairwise sequence alignment is guaranteed to provide the optimal global or local pairwise alignment and score given a particular scoring scheme. The examples here have incorporated varied approaches to the scoring scheme by using substitution models and gap scores based on a function of the gap length. While the recursively overlapping substructure property of the dynamic programming approach allows for a much more efficient algorithm than the brute-force method, it still takes $O(M * N)$ time, where *M* and *N* are the lengths of the sequences to be aligned. *M* and *N* can be very large, especially for an entire chromosome sequence (on the order of 200 million bases or characters in length).

Approximations can be made to the dynamic programming approach to reduce the space and time complexity. These improvements build upon the observation that if two sequences share similarity, they are much more likely to occur within a band around the main diagonal in the scoring matrix. If not, there will be large regions of gaps in both sequences to eventually return to position *M,N* in the matrix. Myers and Miller [**7**] propose one such method that reduces optimal sequence alignment to a linear search space in $O(M + N)$ while still allowing for affine gaps. Spouge [**14**] added improvements in search time by constructing the scoring matrix starting at both position 0,0 and *M,N*, and Ukkonen

[15] offers memory reductions by using linked nodes within a bounded band surrounding the main diagonal rather than storing the complete scoring matrix.

Dynamic programming is a commonly used tool for shorter sequences, but the computational complexity and memory requirements of larger sequences limit its effectiveness. In order to deal with larger searches, a number of approximation algorithms (which are not guaranteed to give the optimal pairwise alignment) have been implemented. Among these are BLAST [1] and FASTA [11], which operate by finding a list of words of a minimum length common between the two sequences. Other tools such as BLAT [5] and SSAHA [9] use hash tables and suffix trees to find common sequences of very high similarity. Such approximation algorithms work very effectively when the sequences have a high degree of similarity, but all of these fail at some level when the sequence conservation is slight.

A number of general purpose uses of dynamic programming exist in bioinformatics outside of the realm of sequence alignment. Scoring schemes may be devised such that a match is based upon an observed distribution. Two example uses of these would be to align two sets of restriction digest fragments, whose molecular weights can be determined within a certain error [12], or to map the observed fragments of a proteomics experiment with a database of known protein weights and properties. In addition, RNA secondary structures have been predicted using dynamic programming techniques [10]. As more high-throughput data become available that need to be mined, more potential uses of dynamic programming in bioinformatics will become clear.

References

1

Althschul,S.F., Gish, W., Miller, W., Meyers, E.W., and Lipman, D.J. Basic local alignment search tool. *Journal of Molecular Biology*, 215, (1990), 403-410.

2

Dayhoff, M.O., and Eck, R.V. A model of evolutionary change in proteins. In *Atlas of protein sequence and structure 1967-1968*, National Biomedical Research Foundation, Silver Spring, MD. (1968), pp. 33-45.

3

Fitch, W., and Smith T.F. Optimal sequence alignments. *Proceedings of the National Academy of Sciences of the United States of America*, 80, (1983) 1382-1386.

4

Heinikoff, S., and Heinikoff, J.G. Amino acid substitution matrices from protein blocks.*Proceedings of the National Academy of Sciences of the United States of America* 89, (1992), 10915-10919.

5

Kent, W.J. BLAT: The BLAST-like alignment tool. *Genome Research*, 12, 4, (2002), 656-664.

6

Knight, J.R., and Myers, E.W. Approximate regular expression pattern matching with concave gap penalties. In *Proceedings of the third symposium on combinatorial pattern matching*, (1992) pp. 67-78.

7

Myers, E.W., and Miller, W. Optimal alignments in linear space. *Computer applications in the biosciences*, 4, 1, (Mar. 1988), 11-17.

8

Needleman, S.B., and Wunsch, C.D. A general method applicable to the search for similarities in the

amino acid sequence of two proteins. *Journal of Molecular Biology*, 48, (1970), 443-453.

9

Ning, Z., Cox, A.J., and Mullikin, J.C. SSAHA: a fast search method for large DNA databases. *Genome Research*, 11, 10,(2001), 1725-1729.

10

Nussinov, R., and Jacobson, A.B. Fast algorithm for predicting the secondary structure of single-stranded RNA. *Proceedings of the National Academy of Sciences of the United States of America*, 77, 11, (1980), 6309-6313.

11

Pearson, W.R. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods in Enzymology*, 210, (1990), 575-601.

12

Rouchka, E.C., and States, D.J. Sequence assembly validation by multiple restriction digest fragment coverage analysis. In *Proceedings of the International Conference of Intelligent Systems in Molecular Biology*, 6, (1998), 140-147.

13

Smith, T.E., and Waterman, M.S. Identification of common molecular subsequences. *Journal of Molecular Biology* 147, (1981), 195-197.

14

Spouge, J.L. Improving sequence-matching algorithms by working from both ends. *Journal of Molecular Biology*, 181, 1, (Jan. 1985), 137-138.

15

Ukkonen, E. Algorithms for approximate string matching. *Information and Control*, 64, (1985), 100-118.

Biography

Dr. Rouchka's interests in bioinformatics and computational biology date back to the mid-1990s while working on creating model-specific parameters for motif detection using Gibbs Sampling algorithms with Chip Lawrence at the Wadsworth Laboratories in Albany, New York. He received his Doctorate of Science (DSc) degree in Computer Science from Washington University in St. Louis in 2002, where his research concentrated on analysis of human genome data. He is currently an assistant professor in the Computer Engineering and Computer Science Department at the University of Louisville, where he is director of the Bioinformatics Laboratory.