20

MATHEMATICAL BIOLOGY

20.1 Sequence Alignment — *Stephen F. Altschul* and Mihai Pop*

- 20.1.1 Global and Local Pairwise Alignments
- 20.1.2 Pairwise Alignment Scores
- 20.1.3 Path Graphs and Optimal Global Pairwise Alignment
- 20.1.4 Optimal Local Pairwise Alignment
- 20.1.5 Substitution Matrices
- 20.1.6 Affine Gap Scores
- 20.1.7 Sequence Alignment Heuristics
- 20.1.8 Exact String Matching
- 20.1.9 Indexing Methods for String Matching

20.2 Phylogenetics — *Joseph Rusinko*

- 20.2.1 Phylogenetic Trees
- 20.2.2 Tree Comparisons
- 20.2.3 Tree Agreement
- 20.2.4 Tree Reconstruction
- 20.2.5 Tree Distributions
- 20.2.6 Subtrees and Supertrees
- 20.2.7 Applications

20.3 Discrete-Time Dynamical Systems — *Elena Dimitrova*

- 20.3.1 Basic Concepts
- 20.3.2 Boolean Networks
- 20.3.3 Cellular Automata
- 20.3.4 Genetic Algorithms

20.4 Genome Assembly — *Andy Jenkins and Matthew Macauley*

- 20.4.1 Basic Concepts
- 20.4.2 Algorithms for Genome Assembly

20.5 RNA Folding — *Qijun He, Matthew Macauley and Svetlana Poznanović*

- 20.5.1 Basic Concepts
- 20.5.2 Combinatorial Models
- 20.5.3 Minimal Free Energy Folding Algorithms
- 20.5.4 Language-Theoretic Methods

20.6 Combinatorial Neural Codes — *Carina Curto and Vladimir Itskov*

- 20.6.1 Basic Concepts
- 20.6.2 The Code of a Cover
- 20.6.3 The Neural Ring and Ideal
- 20.6.4 Convex Codes
- 20.6.5 Feedforward and Hyperplane Codes

*This author was supported by the Intramural Research Program of the National Institutes of Health, National Library of Medicine.

20.7 Food Webs and Graphs *Margaret Cozzens*
20.7.1 Modeling Predator-Prey Relationships with Food Webs
20.7.2 Trophic Level and Trophic Status
20.7.3 Weighted Food Webs
20.7.4 Competition Graphs
20.7.5 Interval Graphs and Boxicity
20.7.6 Projection Graphs
20.7.7 Open Questions

20.1 SEQUENCE ALIGNMENT

Alignments are a powerful way to compare related DNA or protein sequences. They can be used to capture various facts about the sequences aligned, such as common evolutionary descent or common structural function. We take the general view that the alignment of letters from two or multiple sequences represents the hypothesis that they are descended from a common ancestral sequence.

DNA molecules are composed of chains of nucleotides, and protein molecules are composed of chains of amino acids. The specific order of nucleotides or amino acids within these chains are respectively called DNA and protein sequences. Perhaps chief among the various biological functions of DNA sequences is to encode protein sequences, because proteins are involved in most of the biological functions of living cells.

DNA sequences, and the protein sequences they encode, evolve by mutation followed by natural selection. There are a variety of mechanisms for DNA mutation, but the most common result is the substitution of a single nucleotide for another, or the deletion or insertion of one or several adjacent nucleotides. At the protein level, the most common resulting mutations are the substitution of one amino acid for another, or the insertion or deletion of one or multiple adjacent amino acids. There is no simple biological mechanism for exchanging the order of two letters in a DNA or protein sequence, so an alignment representing the common descent of two DNA or protein sequences is co-linear, with no "crossovers" between corresponding letters.

20.1.1 GLOBAL AND LOCAL PAIRWISE ALIGNMENTS

Definitions:

An ***alphabet*** is a finite set of letters and a ***sequence*** S is a finite string of letters, each chosen from the alphabet.

A ***null character***, generally represented by the symbol "–", is a character not in the alphabet that signifies an absent letter.

Given a sequence S, an ***expanded sequence*** S' is the sequence S with an arbitrary number of null characters placed at its start, its end, or between any two of its characters.

A ***global pairwise alignment*** of sequences S and T is a one-to-one co-linear correspondence of expanded sequences S' and T', such that no nulls from S' and T' correspond.

A ***local pairwise alignment*** of sequences S and T is a one-to-one co-linear correspondence of equal-length segments of the expanded sequences S' and T', such that no nulls

from the segments correspond.

Facts:

1. DNA molecules are composed of two strands, each a string of nucleotides, represented by the four letters A, C, G, T.

2. The strands of a DNA molecule are directional and run in opposite directions. Each nucleotide in one strand of a DNA molecule is paired, through chemical interaction, with a complementary nucleotide in the other strand: A always pairs with T, and C with G. Because of this complementarity, the nucleotide sequence of one strand determines the nucleotide sequence of the other, and thus a DNA molecule is generally represented by the nucleotide sequence of one of its strands, with that of the other implied.

3. Proteins are composed of one, or multiple, chemically interacting amino acid chains, but here we confine our attention to single chains. An amino acid chain is a directional string of amino acids. There are twenty commonly occurring amino acids abbreviated by the letters A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y.

4. A particular amino acid is encoded by three adjacent DNA nucleotides, called a *codon*; for example, the codon ATG represents the amino acid methionine, or M. Some amino acids are encoded by only one codon, whereas others by as many as six. The table describing the correspondence between all $4^3 = 64$ possible codons and the 20 amino acids they represent is called the *genetic code*; except for minor variations, it is universal to all life on earth. The genetic code contains three *stop codons* that represent no amino acid.

Examples:

1. One possible global alignment of the protein sequences VHLTPEEKSAVTALWG and VAFTEKQEALVSSSLEAF is

```
VHLT--PEEKSAV-TALWG-
VAFTEKQEA--LVSSSLEAF
```

2. One possible local alignment of the DNA sequences GTTACTTTGGACCCTCAA and AAATTGATCTTTTAAC is

```
TTTGGACC
T-T-GATC
```

20.1.2 PAIRWISE ALIGNMENT SCORES

In order to select among the many possible global or local alignments of two sequences, it is useful to assign an objective function, or score, to each possible alignment. We then seek an alignment with an optimal score, using the convention that higher scores are better. The simplest way to score an alignment is to specify scores for aligning particular letters to one another, or for aligning letters to nulls, and then to define the score of an alignment as the sum of the scores of all its aligned letters and nulls.

Definitions:

A ***column*** of a pairwise alignment is the one-to-one correspondence of a single letter (or null) in one sequence with a single letter (or null) in the other.

A ***substitution*** is a column that aligns two letters. A ***substitution score*** is a score defined for a substitution involving a particular pair of letters.

An ***indel*** is a column that aligns a letter with a null. An ***indel score*** is the score defined for aligning a letter with a null.

A ***gap of length k*** is composed of k adjacent indels, each of which contains a letter from one expanded sequence and a null from the other.

The ***alignment score*** is the sum of the substitution and the indel scores of an alignment's columns.

An ***optimal alignment*** is an alignment with maximum score.

Facts:

1. An alignment of two identical letters is still termed a substitution.

2. The term *indel* represents an abbreviation for "insertion or deletion".

3. Indel scores are usually chosen to be independent of the particular letter aligned with a null, but this restriction is not necessary.

4. The definition of an alignment score will be generalized later (see §20.1.6).

Example:

1. Consider the following alignment of two DNA sequences:

```
TGA-CG
-GTACC
```

Suppose the substitution scores for all columns aligning identical letters are $+5$ and for all columns aligning mismatching letters are -1. Also, suppose all indel scores are -2. Then the alignment score for this DNA alignment is $-2 + 5 - 1 - 2 + 5 - 1 = 4$.

20.1.3 PATH GRAPHS AND OPTIMAL GLOBAL PAIRWISE ALIGNMENT

The most fruitful algorithms for pairwise sequence alignment are based on the concept of a path graph.

Definitions:

Suppose we have two sequences of lengths m and n, respectively. The ***path graph*** consists of a rectangular $(m + 1) \times (n + 1)$ array of ***nodes***, with directed horizontal, vertical, and diagonal ***edges*** between adjacent nodes. The m letters of the first sequence are placed between successive rows of this array and the n letters of the second sequence are placed between successive columns of this array. Diagonal edges of the path graph correspond to substitutions, while horizontal and vertical edges correspond to indels. Scores for the directed edges derive from the corresponding substitution and indel scores.

Facts:

1. The rectangular array underlying the path graph is composed of $m \times n$ *cells*. Each cell is associated with a letter of the first sequence and a letter of the second sequence. The diagonal edge ($\searrow$) in this cell has weight corresponding to the substitution score for these two letters. The lower horizontal edge ($\rightarrow$) in a cell corresponds to a null in the first expanded sequence, and the rightmost vertical edge ($\downarrow$) in a cell corresponds to a null in the second expanded sequence; in each case, the null follows the letter that the diagonal edge signifies aligning.

2. There is a one-to-one correspondence between alignments and directed paths beginning at the upper left node $(0,0)$ of the path graph and ending at the lower right node (m,n). The score of an alignment equals the sum of the edge scores along its corresponding path.

3. The problem of finding an optimal global alignment is then transformed into finding an optimal path through the path graph.

4. Although the number of possible paths through the path graph grows exponentially with m and n, optimal alignments can be found without examining all paths.

5. Algorithm 1 uses a dynamic programming approach to find optimal paths in an efficient way. In brief, there are at most three nodes at locations $(i-1,j),(i,j-1),(i-1,j-1)$ with edges leading into a particular node at location (i,j). Therefore, if one knows the score of an optimal path from the origin to each of these three nodes, one can easily calculate the score of an optimal path through each of these nodes into (i,j). The highest score among these paths is then the score of an optimal path to (i,j). Optimal alignment scores for each node in a path graph may thus be calculated by simply filling in scores for the nodes sequentially, beginning at the origin, and proceeding from left to right along each row in turn [NeWu70], [Sa72], [Se74].

Algorithm 1: **Global pairwise alignment algorithm.**

input: two sequences of lengths m and n, respectively
output: the optimal global alignment score as well as optimal alignments
create the corresponding path graph G with $(m+1)\times(n+1)$ nodes, indexed from 0 to m and from 0 to n
label the upper left node of G with 0
label the rest of the first row and the first column of G, based on the indel scores
for each remaining node in location (i,j) of the array compute
  Vscore := label$(i-1,j)$ + score of edge from $(i-1,j)$ to (i,j)
  Hscore := label$(i,j-1)$ + score of edge from $(i,j-1)$ to (i,j)
  Dscore := label$(i-1,j-1)$ + score of edge from $(i-1,j-1)$ to (i,j)
  label(i,j) := max{Vscore, Hscore, Dscore}; mark any edge achieving this maximum value
label(m,n) represents the optimal global alignment score and the marked edges can be used to retrieve alignments achieving this optimal score

6. In Algorithm 1, the values Vscore, Hscore, and Dscore represent the scores of paths using (respectively) the vertical, horizontal, and diagonal edges entering the node located at (i,j). By tracing back along the marked edges, starting from the lower right corner of G, one can obtain optimal alignments for the given pair of sequences.

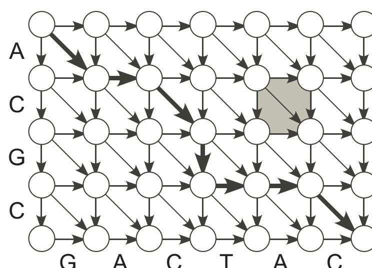
7. To align sequences of lengths m and n, the time and space complexity of Algorithm 1 are each $O(mn)$. The space complexity can be reduced to $O(\min\{m,n\})$ [MyMi88].

8. There are various methods for speeding up the algorithm by avoiding the consideration of nodes through which an optimal alignment cannot possibly pass [Fi84], [Sp89].

9. Optimal alignments are unaltered by multiplying all substitution and indel scores by a positive constant. Optimal alignments are unaltered by adding a constant a to all substitution scores, and $a/2$ to all indel scores.

Examples:

1. We wish to align the sequences $S_1 = \text{ACGC}$ and $S_2 = \text{GACTAC}$. The corresponding path graph G has nodes arranged in an $(m+1) \times (n+1) = 5 \times 7$ rectangular array. The four letters of the first sequence are placed between successive rows of G and the six letters of the second sequence are placed between successive columns of G. The shaded cell corresponds to aligning the first letter C of the first sequence and the second letter A of the second sequence.



The highlighted path (from the upper left node to the lower right node in G) corresponds to the alignment

```
A-CG--C
GAC-TAC
```

2. Suppose that substitution scores are $+1$ for matches and 0 for mismatches, and that indel scores are -1. Then the alignment in Example 1 has score $0-1+1-1-1-1+1 = -2$. This is not, however, an optimal global alignment of S_1 and S_2.

3. Optimal alignments, based on the substitution and indel scores specified in Example 2, can be found using Algorithm 1. After labeling the upper left node with 0, we label nodes in the first row and first column using $-1, -2, -3, \ldots$ since the indel scores are -1. Each remaining node is labeled with the maximum sum of node label plus edge score, taken over the three nodes adjacent to and leading into the given node.

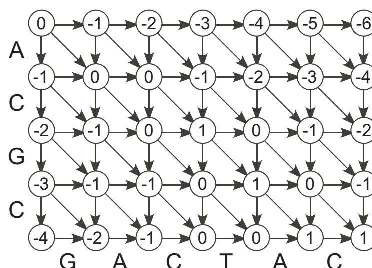
For example, to label the node at location $(2,1)$ we consider its three incident edges:

$(1,1) \to (2,1)$: label of $(1,1)$ + edge score (indel) $= 0 - 1 = -1$

$(2,0) \to (2,1)$: label of $(2,0)$ + edge score (indel) $= -2 - 1 = -3$

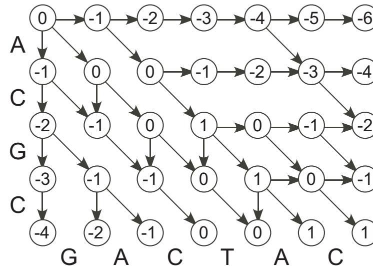
$(1,0) \to (2,1)$: label of $(1,0)$ + edge score (mismatch C, G) $= -1 + 0 = -1$

Node $(2,1)$ thus receives as its label the maximum sum -1. The following figure shows all node labels obtained by following Algorithm 1. Since the label of the lower right node $(4,6)$ is 1, this represents the optimal global alignment score.

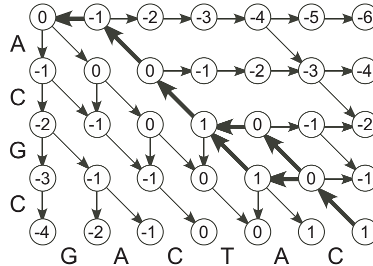


4. In order to reconstruct the alignment or alignments corresponding to this optimal score, the edge or edges corresponding to the best path into each node have been marked.

In Example 3, the edges $(1,1)$ and $(1,0)$ both give the maximum label -1 and so are marked. All edges marked during the execution of Algorithm 1 are shown next.



5. We can then trace back along marked edges from the terminal node $(5,7)$ to the origin node $(1,1)$, giving the highlighted edges shown.



6. From these two paths, we obtain the two optimal alignments of S_1 and S_2, each with (maximum) score $= +1$:

```
-ACG-C        -AC-GC
GACTAC        GACTAC
```

20.1.4 OPTIMAL LOCAL PAIRWISE ALIGNMENT

Many protein or DNA sequences are related only across subsequences, but not over their entire lengths. To recognize these relationships, it is useful to introduce the concept of a local alignment (§20.1.1) and to describe algorithms capable of finding optimal local alignments.

Facts:

1. Algorithm 1 from the previous section can be modified [SmWa81] to find optimal local alignment(s) of two sequences by making a few minor modifications.

- Allow a path to start, with score 0, at any node within the path graph, not just at the origin (top left) node. Thus, if all paths into a given a node have negative score, none of them is chosen as optimal.
- Record which node within the path graph receives the highest score, and start tracing back from there, rather than from the terminal node.
- Terminate the traceback when a node with score 0 is reached.

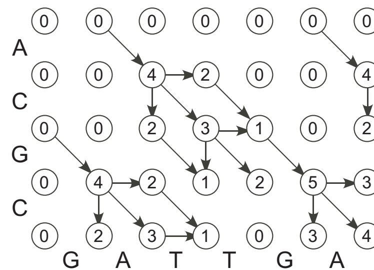
2. To align sequences of lengths m and n, the time and space complexity for this local alignment algorithm are each $O(mn)$. The space complexity can be reduced to $O(\min\{m,n\})$ [MyMi88].

3. Optimal local alignments are unaltered by multiplying all substitution and indel scores by a positive constant.

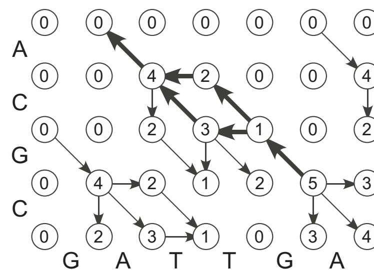
4. In addition to optimal local alignment(s), one may wish to find other local alignments that align regions of the sequences in question that are not implicated in the optimal local alignment. One may define "locally optimal" local alignments as those whose path within the path graph has a score greater than or equal to the score for the path of any local alignment that shares a common edge or node [Se84]. All such locally optimal local alignments can also be found in $O(mn)$ time [AlEr86b].

Examples:

1. We wish to align the sequences S_1 = ACGC and S_2 = GATTGA. We construct a path graph with $(m+1)\times(n+1) = 5\times 7$ nodes. Suppose that substitution scores are $+4$ for matches and -1 for mismatches, and that indel scores are -2. The following figure shows the resulting labels for locally aligning these sequences, with only marked edges shown.



2. The largest score obtained is 5 so this is the optimal local alignment score. Tracing back from this location $(4,6)$ yields the highlighted edges shown next.



As a result, we find two optimal local alignments, each with score 5:

```
AC-G     A-CG
ATTG     ATTG
```

20.1.5 SUBSTITUTION MATRICES

Although the algorithms described in §20.1.3 and §20.1.4 are unaffected by the choice of substitution scores, the quality of the alignments they return depends strongly on which

scores are used. In particular, this subsection discusses the use of certain substitution matrices to specify scoring values.

Definitions:

A ***substitution matrix*** $S = (s_{ij})$ prescribes the score value when letters i and j are aligned.

A ***log-odds matrix*** S is defined by $s_{ij} = \log \frac{q_{ij}}{p_i p_j}$. Here q_{ij} is the ***target frequency*** with which the letters i and j are estimated to appear in columns from accurate alignments of related sequences; p_i and p_j are the ***background probabilities*** with which the letters i and j would appear by chance.

Facts:

1. Many different substitution matrices have been proposed, but theory and practice suggest that the best substitution matrices are log-odds matrices [Al91], [DaScOr78], [KaAl90].

2. The target frequencies should depend upon the degree of evolutionary divergence separating the sequences compared, and various methods of estimating these frequencies for proteins have been described, giving rise to the commonly used PAM [ScDa78] and BLOSUM [HeHe92] series of substitution matrices.

20.1.6 AFFINE GAP SCORES

A single mutational event can insert or delete multiple nucleotides into or from a DNA sequence, and this can result in multiple contiguous amino acids being inserted into or deleted from the encoded protein sequence. Thus, an alignment in which the several insertions or deletions are adjacent to one another makes more biological sense than one in which these insertions and deletions are separated. The scoring systems considered earlier, however, do not reflect this biological fact. One remedy is to modify the alignment scoring system by defining appropriate scores for gaps of various lengths in place of scores for individual indels.

Definitions:

Affine gap scores are given by $g(k) = -(a + bk)$, where k is the gap length and a, b are (generally nonnegative) constants.

Facts:

1. It is evident that when $a = 0$, the affine gap score $g(k)$ is equivalent to indel scores of $-b$. However, when $a > 0$, the score for k adjacent indels is better than that for k indels separated by substitution columns.

2. For either global or local alignments, the basic algorithms described in §20.1.3 and in §20.1.4 must be modified to accommodate affine gap costs. Fortunately, it is possible to do this while retaining $O(mn)$ time complexity by remembering not just the score of the best path into each node, but also the best score for a path that arrives through a horizontal, vertical, or diagonal edge [AlEr86a], [Go82]. For example, the incremental score for a path leading out of a node along a horizontal edge is $-b$ for a path that arrived along a horizontal edge, but is $-(a + b)$ for a path that arrived along any other type of edge.

3. It is possible to construct $O(mn)$ algorithms based on $g(k)$ that are more involved than affine gap scores [MiMy88], but these algorithms are fairly complex and in practice almost never used.

4. The elements of sequence alignment considered here been generalized in a great variety of ways. These include rapid heuristic database search algorithms [AlEtal90], [PeLi88]; protein profile construction and comparison [AlEtal97], [GrMcEi87]; and multiple sequence alignment, for which there is a vast and growing literature.

Example:

1. The practical difference in using affine gap costs in place of simple indel costs can be illustrated in the optimal local alignments resulting from the comparison of two protein sequences. Both alignments shown next employ BLOSUM-62 substitution scores [HeHe92], but the first alignment uses indel scores of -6, while the second uses gap scores of $g(k) = -(11 + k)$.

Alignment 1:

```
 49 CERTLKYFLGIAGGKWVVSYFWVTQSIKERKMLNEHDFEVRGDVVNGRNHQGPKRARESQDRK-IFRGLEICCYG   122
    C RT KYFL  A G   VS  WV  S       N            G       R    Q R   F  L
865 C-RTRKYFLCLASGIPCVSHVWVHDSCHANQLQNYRNY-L---LPAGYSLE-EQRILDWQPRENPFQNLKVLLVS   933

123 PFTNMPTDQLEWM-VQLC-GASVVKE-LSS-FT--LGTGVHPIVVVQPDAWTEDNGFHAIGQMCEAPVVTREWVL   191
             L W       GA  VK   SS        GV   VV  P                  PVV  EWV
934 D-QQQNFLEL-WSEILMTGGAASVKQhHSSAHNKDIALGVFDVVVTDPSC-PA-SVLKC-AEALQLPVVSQEWVI  1003
```

Alignment 2:

```
 51 RTLKYFLGIAGGKWVVSYFWVTQSIKERKMLNEHDFEVRGDVVNGRNHQGPKRARESQDRK-IFRGLEICCYG     122
    RT KYFL  A G   VS  WV  S       N                    R    Q R   F  L
866 RTRKYFLCLASGIPCVSHVWVHDSCHANQLQNYRNY-----LLPAGYSLEEQRILDWQPRENPFQNLKVLLVS     933

123 PFTNMPTDQLEWMVQLCGASVVKELSSFT----LGTGVHPIVVVQPDAWTEDNGFHAIGQMCEAPVVTREWVL     191
                     GA  VK   S         GV   VV  P                  PVV  EWV
934 DQQQNFLELWSEILMTGGAASVKQHHSSAHNKDIALGVFDVVVTDPSC---PASVLKCAEALQLPVVSQEWVI    1003
```

Identical letters are echoed on the central lines of these alignments, and the sequence positions of the letters at the start and end of each line are provided. The second alignment contains many fewer gaps, but also a smaller number of identical aligned letters.

20.1.7 SEQUENCE ALIGNMENT HEURISTICS

Sequence alignment, as described so far, is computationally intensive, with the algorithms requiring an execution time quadratic in the size of the sequences being aligned. In the general case this run time is unavoidable. However faster execution times can be obtained when the sequences being aligned have high levels of similarity. Here we discuss banded methods, exclusion methods, and the use of seeds to initiate searches.

Definitions:

Suppose that G is the path graph for the alignment of two sequences. The ***diagonal*** of G consists of all nodes at locations $(0, 0), (1, 1), \ldots, (m, n)$. A ***band of width d*** consists of all nodes within (Manhattan) distance d of the diagonal of G.

A ***k*-mer** is a subsequence of length k found within a given sequence (see §20.4.1).

A ***seed*** is a relatively short matching subsequence that serves as an anchor point for larger matches between sequences. A ***spaced seed*** is a k-mer within which a number t of fixed positions (denoted by $*$) are allowed to match any letter; the spaced seed then has ***width*** k and ***weight*** $k-t$.

Facts:

1. Suppose that two given sequences differ by at most d edits (at most d letters are changed, inserted, or deleted between the two sequences). Then an optimal path in the path graph G cannot pass through nodes that are further than distance d from the diagonal of G as each "move" away from the diagonal implies an additional difference being incorporated in the alignment.

2. By Fact 1 it is sufficient to compute the alignment scores for just a subset of the nodes in G, specifically for nodes in the band of width d. Using this banded alignment algorithm, an optimal alignment can then be determined in $O(\min\{m,n\}d)$ time [ChPeMi92].

3. The value d does not need to be known a priori. Rather, an algorithm that adapts dynamically to the actual similarity between the sequences can be devised by performing a binary search for the band that encompasses the optimal path through the path graph. Specifically, one starts with an initial guess for d, e.g., by setting $d = 1$. The banded alignment algorithm is then executed to find an alignment that occurs within the band of width d. If the resulting alignment contains fewer than d edits, then this is also the optimal alignment. If the alignment contains more than d edits, then d is doubled, and the process is repeated until an optimal alignment is found.

4. The dynamic algorithm presented in Fact 3 also takes $O(\min\{m,n\}d)$ time, though it can be up to four times slower than an algorithm that starts with an initial correct guess for the value of d.

5. A special case of alignment occurs when one sequence is much longer than the other one, e.g., when aligning one short sequence against an entire genome, or when aligning a sequence against a database. In such situations, a run time proportional to the product of the lengths of the two sequences can be prohibitive. Heuristics have been developed that can focus on the regions of the alignment table that contain high-quality alignments.

6. Suppose we are searching for an alignment of a (query) sequence of length m within a longer sequence (database) of length n, where $m \ll n$. Specifically, we are looking for alignments with up to d differences (indels or substitutions). Then any inexact alignment with d or fewer differences must contain an exact match between the query sequence and the database that is of length at least $\lfloor m/(d+1)\rfloor$. Indeed, the worst-case scenario has the d differences equally separated along the sequence, breaking it up into $d+1$ segments.

7. *Exclusion methods*: These approaches, based on Fact 6, use exact matching to speed up inexact alignment [WuMa92]. First we identify all exact matches of length $\lfloor m/(d+1)\rfloor$ between the two strings, corresponding to diagonal segments within the path graph. Then we perform the banded alignment procedure to find the highest scoring alignment within the neighborhood of the exact match. This procedure is guaranteed to find an optimal alignment with fewer than d edits, if one exists, and is faster than the full alignment algorithm as only a small fraction of the entire path graph needs to be explored. Regions of the path graph that cannot contain good alignments are excluded from consideration.

8. A key factor determining the speed-up we can expect is the length of the exact match being sought: $k = \lfloor m/(d+1)\rfloor$. For small values of k (i.e., when the error being tolerated is high with respect to the size of the query sequence), the expected number of k-mers

that yield a match between the sequence and the database can be very high, making this approach inefficient.

9. The size of k specified in Fact 8 is unnecessarily conservative as it derives from a worst-case scenario where errors are equally distributed throughout the sequence. A random distribution of errors within the sequence is unlikely to be uniform and so contains exact segments that can be much longer than $\lfloor m/(d+1) \rfloor$.

10. The value of k could be determined from theoretical principles, e.g., by setting k to be the expected length of the longest edit-free segment within an alignment of length m that contains d randomly distributed edits. In practice, the exact value of d is usually not known, and k is set empirically to a value that minimizes the number of alignments that need to be performed without missing too many alignments.

11. To increase sensitivity without sacrificing speed, Ma et al. [MaTrLi02] suggested the use of spaced seeds. The intuition behind the use of spaced seeds is that the "don't care" symbols allow the seed to match even if a difference occurs in the middle of the seed.

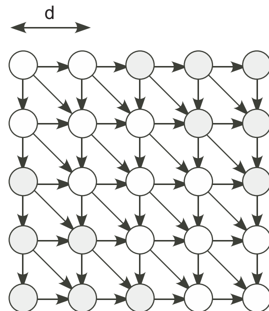
12. Spaced seeds have a higher sensitivity than the exact k-mer seeds of the same weight without a corresponding increase in the number of matches that need to be explored. In practice, algorithms relying on spaced seeds use a combination of multiple seeds in order to maximize sensitivity. Finding the optimal combination of seeds is NP-hard [MaLi07]; however simple heuristic algorithms are highly effective in practice [IlIl07].

13. Spaced seeds can be generalized by extending the original observation underlying all exclusion methods—that any alignment of high enough fidelity must contain a long enough exact match between the query and the database. It can be shown [GhPo09], [KaNa07] that any alignment of high enough fidelity must contain a long enough inexact alignment of higher fidelity than the original one. Specifically, if strings L_1 and L_2 of length ℓ match with an edit distance k, there exist substrings E_1 in L_1 and E_2 in L_2 of length e such that the two substrings match with an edit distance no greater than $k/(\ell - e)$ [GhPo09].

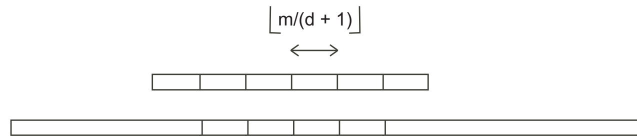
14. Using Fact 13, one could search for inexact seeds that match between a query and a database using a fast inexact alignment algorithm that can tolerate a limited number of edits, and then use the resulting anchors to search for the full alignment. This generalization provides a higher sensitivity than approaches based on spaced seeds.

Examples:

1. The following figure illustrates a path graph with a band of width $d = 1$ indicated. An optimal alignment with at most d edits must occur within a band of width d on either side of the diagonal. Paths traversing the grayed nodes involve more than d edits.



2. In the worst-case scenario, the d edits are equally spaced along the sequence, implying that exact matches between the two strings are at least $\lfloor m/(d+1) \rfloor$ characters long:



3. The spaced seed 111∗111 denotes the set of k-mers of width 7 and weight 6. This seed allows the middle nucleotide to differ between the query sequence and the database sequence. In the following two strings, a spaced seed of width 7 is shared by two strings. Only the characters aligned to 1s in the spaced seed need to match between the two strings.

```
TATT |CGTCTGA| TCAT
     |111*111|
GTCT |CGTATGA| TGCT
```

20.1.8 EXACT STRING MATCHING

Underlying the exclusion methods described in §20.1.7 is the assumption that finding exact matches between two strings is computationally more efficient than finding inexact alignments. Exact string matching is of independent interest as well in many applications (e.g., web search engines largely rely on exact matching of query strings). As such, the problem has been extensively researched and many variants and algorithms have been described in the literature. Here we highlight several key results.

Definitions:

Given a string S, a ***suffix tree*** is a compact representations of all suffixes in S. It is a tree that contains a leaf for each suffix in S, and where each edge is labeled with a string of characters so that the path from the root to each leaf spells the corresponding suffix. The string spelled by the path from the root of a suffix tree to an internal node is termed the ***string label*** of the node.

The suffix tree also contains a collection of ***suffix links***, special edges connecting the node in the tree with string label $c\alpha$ (where c is a single character and α is a string) to the node with string label α.

The ***suffix array*** is a lexicographically sorted array of all suffixes of the string, together with information necessary for speeding up the search process.

Facts:

1. A naive search for exact matches between a query string of length m and a database of length n requires $O(mn)$ run time, by checking whether all m letters in the query match, starting at each of the $n - m + 1$ positions within the database.

2. *KMP algorithm*: Knuth, Morris, and Pratt [KnMoPr77] reduced the run time for exact matching to $O(m + n)$ by simple preprocessing of the query string. Specifically, at each position i in the query, the KMP algorithm defines sp_i to be the length of the longest nontrivial prefix of the query that matches a suffix of the string that ends at position i.

3. This information can be computed in $O(m)$ time and can be used to speed up alignment as follows. The query is compared to the database, one letter at a time, until a

mismatch is detected at position $i+1$ in the query. If sp_i is defined, the query is shifted to the right so that the prefix of the query aligns to the corresponding suffix of the string ending at position i prior to the shift. The comparison to the database continues from position $sp_i + 1$, as the properties of the sp_i values guarantee that all prior positions match the database sequence (these were already matched prior to the shift). If $sp_i = 0$, the query is shifted such that its first letter is aligned to the character mismatched prior to the shift, and the comparison between the query and the database starts from the first letter of the pattern.

4. Two observations establish an $O(n)$ run time for the search procedure, giving an $O(m+n)$ run time once the cost of preprocessing is taken into account.

- Once a character in the database is successfully matched to the query it is never again compared to the query, thereby bounding the total number of exact matches performed during the execution of the algorithm to n, the size of the database.
- Every mismatch triggers a shift of the query by at least one position, thus the number of mismatches is also bounded by n, yielding an overall $O(n)$ run time.

5. The KMP algorithm can be used to find full-length matches of a query against a database, as well as matches that involve just a prefix of the query (e.g., finding the longest prefix match in case a full-length match cannot be found). To address more complex exact matching problems, Weiner [We73] and McCreight [Mc76] independently developed suffix trees.

6. The strings represented by the path from the root to any node in the suffix tree are unique. In other words, the shared prefixes of strings represented in the tree are represented by the same path from the root to some internal node in the tree.

7. To ensure that the suffixes of S reach the leaves of the tree (i.e., no suffix is a prefix of some other suffix in the tree), a special character \$ not found in the alphabet is appended to the end of each string stored in the suffix tree.

8. Suffix links are useful in the efficient construction of suffix trees as well as for solving certain exact matching problems, such as identifying the longest substring that matches between the query and the database.

9. When appropriately structured, suffix trees require $O(m)$ space to store all the suffixes of a given string of length m. To achieve such space efficiency, each edge is labeled not with its string label, but with the coordinates within the string where the string label occurs (see Example 2). Each edge in the tree thus requires constant storage space, and the total size of the tree is determined by the number of leaves, equal to the number of suffixes m.

10. If a query is represented within a suffix tree, it can be aligned against a database as follows. Starting with the root of the tree, the matching process follows the edges of the tree that match the database sequence until either reaching a leaf (in which case the corresponding suffix represents a partial match between the query and the database) or a mismatch occurs. The algorithm then follows the suffix link starting from the node immediately above the location of the mismatch and the matching process continues from the same position in the database, starting from the new location in the tree. Following the suffix link implicitly discards the first character in the database and is equivalent to the shift procedure described in the KMP algorithm.

11. The search procedure requires $O(m+n)$ time, matching the time complexity of the KMP algorithm.

12. Despite its asymptotically linear space complexity, the suffix tree data structure requires a substantial amount of memory, estimated at over 20 bytes per letter. Manber and Myers [MaMy93] described an alternative approach for storing all the suffixes of a string in a compact way that uses substantially less space at the cost of a small additional increase in run time.

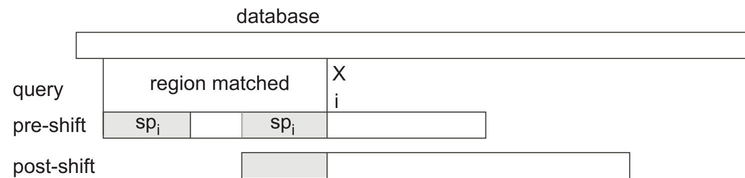
13. In a suffix array, the suffixes themselves are stored as simply an index within the original string, ensuring that the space usage is proportional to the size of the string even though the total length of all suffixes combined is proportional to the length of the string.

14. In addition to the suffixes, the data structure for a suffix array comprises an array that stores the *longest common prefix* (LCP) between adjacent suffixes. This information is necessary to efficiently search within the suffix array using binary search with just an additive overhead of $O(\log m)$. A trivial implementation of binary search within an array of m strings of length $O(m)$ requires $O(m \log m)$ time. Since the LCP values are only used for the binary search process, it is sufficient to record just the longest common prefixes of the pairs of strings that will be compared during the binary search, or $O(m)$ values. Suffix arrays thus occupy $O(m)$ space, and a sequence of length n can be searched against a suffix array in $O(n + \log m)$ time.

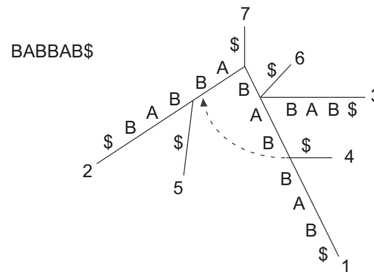
15. Constructing a suffix array involves sorting the set of suffixes of a string, a process that can be performed in $O(m \log m)$ time using an algorithm that takes advantage of the relationship between the suffixes of a same string [MaMy93].

Examples:

1. The following diagram illustrates the sp_i values used by the KMP algorithm and a shift after a mismatch occurs at position i in the query. The shaded regions correspond to identical sequences. After the shift, matching continues from the position marked X.



2. The following figure shows a suffix tree for the string BABBAB. The dashed line is a suffix link connecting the node labeled BAB to the node labeled AB. Edge labels are provided for clarity, but an efficient implementation would compress them to two integers. For example, the edge BAB$ can be represented as $(4, 7)$, the range within the original string that spells this string.



20.1.9 INDEXING METHODS FOR STRING MATCHING

When searching for multiple query sequences against the same database, search efficiency can be increased by preprocessing or indexing the database. For the following, assume we are searching for exact matches between multiple sequences of length m and a database of length n.

Facts:

1. Storing the database in a suffix tree (§20.1.8) enables exact matching queries to be executed in time independent of the database size. For each query sequence, the matching process traverses the suffix tree from the root, following the path that matches the query until either a match is found, or a mismatch indicates the query does not match within the database, yielding a run time of $O(m)$ per query.

2. As discussed in §20.1.8, the main drawback of this strategy is the large memory necessary to store the suffix tree.

3. *Inverted index*: Finding short exact matches of length k between one or more query sequences and a database, a component of the exclusion methods described in §20.1.7, can be performed with a simple (inverted) index linking all strings of length k within the database to their location in the database. Lookups within the index can be performed efficiently if the entire index is stored in memory; however, the memory space necessary to store all k-mers within a database can be very large. Furthermore, k-mers that occur frequently within the database impact both memory usage and the time needed to search the index, and are usually excluded from the index in practical implementations.

4. *Burrows-Wheeler transform (BWT)*: A reversible permutation of the indexed string can be used to construct a memory-efficient indexing structure called the *FM index* [FeMa00]. The Burrows-Wheeler transform [BuWh94] of a string S can be constructed using Algorithm 2. It is assumed here that the added character $ is lexicographically smallest.

Algorithm 2: **Burrows-Wheeler transform.**

input: string S

output: Burrows-Wheeler transform of S

augment string S by appending to its end a character $ not found in the alphabet

construct a table T comprising all circular rotations of S as rows

sort the rows of T into lexicographic order

return the last column in T as the Burrows-Wheeler transform of S

5. The BWT and the suffix array of S are conceptually linked. The BWT represents the characters that precede the lexicographically ordered suffixes of S, and the BWT can be trivially constructed from the suffix array of S. See Example 2.

6. The BWT can be reversed, obtaining the original string, with the help of a simple observation: the order in which multiple instances of the same character occur in the first column of the sorted BWT table is the same as the order in which these characters occur in the last column of the table. Specifically, the first A in the last column corresponds to the same character in the original string as the first A in the first column, the third C in the last column corresponds to the same character in the original string as the third C in the first column, etc. This *last-to-first* (LF) mapping guides the reversal of the BWT; see Algorithm 3.

Algorithm 3: **BWT reversal.**

input: BWT table T for S

output: original string S

start with the first row of the BWT table and output the character c in the last column; this is the last character in the original string

repeat until the entire string is reconstructed

  use the LF property to identify the corresponding instance of c in the first column

  prepend the character at the end of the respective row to the string reconstructed so far

7. In Algorithm 3 only the first and the last columns of the BWT matrix are necessary to reverse the transformation. The last column is the BWT itself, while the first column is simply the lexicographically sorted list of characters in the original string.

8. To efficiently perform the LF mapping it is necessary to know the index of each character within the last column. That is, for a character c within the ith row of the matrix we need to know the number of characters equal to c that occur within the rows above. Storing this information for each row of the matrix allows the LF lookup to occur in constant time; however, it requires a large amount of memory $O(n \log n)$ for a string of length n. A trade-off between run time and memory usage can be obtained by storing the index information at every bth row, yielding a memory usage of $O((n \log n)/b)$. Each LF mapping, however, requires $O(b)$ time as the index values need to be computed on the fly within the block between the selected rows. See Example 5.

9. The same procedure used for reversing the BWT can be used to match a query Q against a database stored in the BWT. This process is similar to a binary search in that the transitions between the first and last columns of the BWT matrix repeatedly shrink the range within which the query string may be found. See Algorithm 4.

Algorithm 4: **BWT query match.**

input: query Q, BWT matrix T

output: matching string in the database

identify the block of rows in T starting with the last character in Q

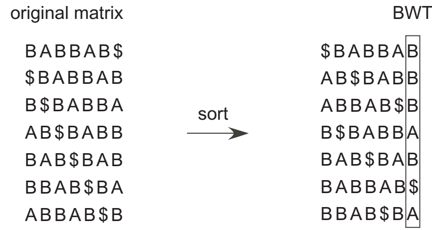
repeat until the query is fully matched

  within the last column of these rows, find those rows containing the rightmost unmatched character in Q; if no rows end in this character, no match has been found

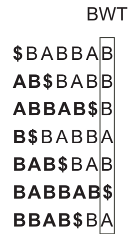
  within the first columns of the BWT matrix, identify the rows starting with the characters found above

Examples:

1. We illustrate the Burrows-Wheeler transform of the string BABBAB. The character $ is added to this string, giving the first row of the table shown on the left; the rows of this table represent successive circular rotations. After lexicographically sorting the rows, we obtain the BWT matrix on the right, whose last column is the Burrows-Wheeler transform of the original string.

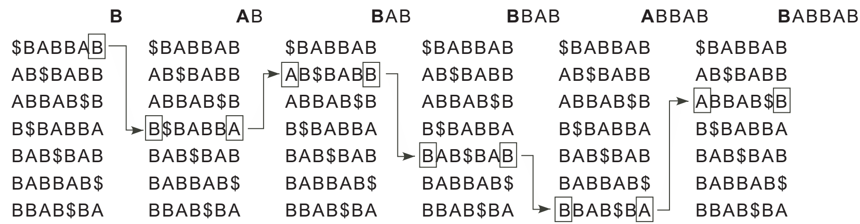


2. The suffix array of the string in Example 1 is shown highlighted in bold within the Burrows-Wheeler matrix.

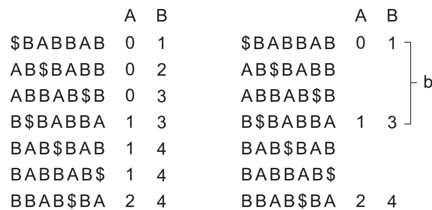


3. We illustrate the LF mapping for the BWT matrix shown in Example 1. The first B in the last column is the fourth B occurring in the original string BABBAB; the first B in the first column also corresponds to the fourth B in the original string. The second B in the last column is the third B occurring in the original string BABBAB; the second B in the first column also corresponds to the third B in the original string.

4. Here we show how to reverse the BWT in Example 1 to generate the original string BABBAB, starting from the rightmost character. The arrows indicate the LF mapping. The string being reconstructed is shown above the table with the latest character added shown in bold. Note that only the first and last columns of the table are needed for this operation.

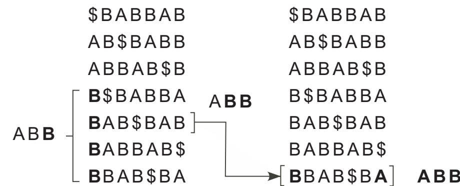


5. A character index is needed for performing the LF mapping. The full index is shown on the left. On the right is a sparse index that stores the information every b rows.



6. We illustrate the search process using a BWT/FM-index. The search range is iteratively refined as more characters in the query string ABB are being matched, starting

with the rightmost character. The matched characters are shown in bold.



REFERENCES

Printed Resources:

[Al91] S. F. Altschul, "Amino acid substitution matrices from an information theoretic perspective", *Journal of Molecular Biology* 219 (1991), 555–565.

[AlEr86a] S. F. Altschul and B. W. Erickson, "Optimal sequence alignment using affine gap costs", *Bulletin of Mathematical Biology* 48 (1986), 603–616.

[AlEr86b] S. F. Altschul and B. W. Erickson, "Locally optimal subalignments using nonlinear similarity functions", *Bulletin of Mathematical Biology* 48 (1986), 633–660.

[AlEtal90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool", *Journal of Molecular Biology* 215 (1990), 403–410.

[AlEtal97] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs", *Nucleic Acids Research* 25 (1997), 3389–3402.

[BuWh94] M. Burrows and D. Wheeler, "A block sorting lossless data compression algorithm", *Technical Report* 124, Digital Equipment Corporation, 1994.

[ChPeMi92] K-M. Chao, W. R. Pearson, and W. Miller, "Aligning two sequences within a specified diagonal band", *Computer Applications in the Biosciences* 8 (1992), 481–487.

[DaScOr78] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt, "A model of evolutionary change in proteins", in *Atlas of Protein Sequence and Structure*, Volume 5, Supplement 3, M. O. Dayhoff (ed.), National Biomedical Research Foundation, Washington, DC, 1978, pp. 345–352.

[FeMa00] P. Ferragina and G. Manzini, "Opportunistic data structures with applications", *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, 2000, 390–398.

[Fi84] J. W. Fickett, "Fast optimal alignment", *Nucleic Acids Research* 12 (1984), 175–180.

[GhPo09] M. Ghodsi and M. Pop, "Inexact local alignment search over suffix arrays", *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine (BIBM '09)*, 2009, 83–87.

[Go82] O. Gotoh, "An improved algorithm for matching biological sequences", *Journal of Molecular Biology* 162 (1982), 705–708.

[GrMcEi87] M. Gribskov, A. D. McLachlin, and D. Eisenberg, "Profile analysis: detection of distantly related proteins", *Proceedings of the National Academy of Sciences USA* 84 (1987), 4355–4358.

[HeHe92] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks", *Proceedings of the National Academy of Sciences USA* 89 (1992), 10915–10919.

[IlIl07] L. Ilie and S. Ilie, "Multiple spaced seeds for homology search", *Bioinformatics* 23 (2007), 2969–2977.

[KäNa07] J. Kärkkäinen and J. C. Na, "Faster filters for approximate string matching", *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX' 07)*, 2007, 84–90.

[KaAl90] S. Karlin and S. F. Altschul, "Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes", *Proceedings of the National Academy of Sciences USA* 87 (1990), 2264–2268.

[KnMoPr77] D. Knuth, J. H. Morris, and V. Pratt, "Fast pattern matching in strings", *SIAM Journal on Computing* 6 (1977), 323–350.

[MaLi07] B. Ma and M. Li, "On the complexity of the spaced seeds", *Journal of Computer and System Sciences* 73 (2007), 1024–1034.

[MaTrLi02] B. Ma, J. Tromp, and M. Li, "PatternHunter: faster and more sensitive homology search", *Bioinformatics* 18 (2002), 440–445.

[MaMy93] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches", *SIAM Journal on Computing* 22 (1993), 935–948.

[Mc76] E. M. McCreight, "A space-economical suffix tree construction algorithm", *Journal of the ACM* 23 (1976), 262–272.

[MiMy88] W. Miller and E. W. Myers, "Sequence comparison with concave weighting functions", *Bulletin of Mathematical Biology* 50 (1988), 97–120.

[MyMi88] E. W. Myers and W. Miller, "Optimal alignments in linear space", *CABIOS* 4 (1988), 11–17.

[NeWu70] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequences of two proteins", *Journal of Molecular Biology* 48 (1970), 443–453.

[PeLi88] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison", *Proceedings of the National Academy of Sciences USA* 85 (1988), 2444–2448.

[Sa72] D. Sankoff, "Matching sequences under deletion-insertion constraints", *Proceedings of the National Academy of Sciences USA* 69 (1972), 4–6.

[ScDa78] R. M. Schwartz and M. O. Dayhoff, "Matrices for detecting distant relationships", in *Atlas of Protein Sequence and Structure*, Volume 5, Supplement 3, M. O. Dayhoff (ed.), National Biomedical Research Foundation, Washington, DC, 1978, pp. 353–358.

[Se74] P. H. Sellers, "On the theory and computation of evolutionary distances", *SIAM Journal on Applied Mathematics* 26 (1974), 787–793.

[Se84] P. H. Sellers, "Pattern recognition in genetic sequences by mismatch density", *Bulletin of Mathematical Biology* 46 (1984), 501–514.

[SmWa81] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences", *Journal of Molecular Biology* 147 (1981), 195–197.

[Sp89] J. L. Spouge, "Speeding up dynamic programming algorithms for finding optimal lattice paths", *SIAM Journal on Applied Mathematics* 49 (1989), 1552–1566.

[We73] P. Weiner, "Linear pattern matching algorithms", *14th Annual IEEE Symposium on Switching and Automata Theory*, 1973, 1–11.

[WuMa92] S. Wu and U. Manber, "Fast text searching: allowing errors", *Communications of the ACM* 35 (1992), 83–91.

Web Resources:

`http://mbcf149.dfci.harvard.edu/cmsmbr/biotools/biotools16.html` (List of sequence alignment servers and databases.)

`http://rosalind.info/problems/locations/` (Online platform for learning bioinformatics algorithms through coding. Includes an extensive collection of exercises and problems related to sequence alignment.)

`http://www.ebi.ac.uk/Tools/emboss/align/` (Local and global tools for pairwise sequence alignment.)

`http://www-igm.univ-mlv.fr/~lecroq/string/` (Exact string matching algorithms in C.)

`http://www.langmead-lab.org/teaching-materials/` (Videos and lecture slides for string matching algorithms from Ben Langmead, including Python code.)

`https://blast.ncbi.nlm.nih.gov/Blast.cgi/` (Web server for running BLAST.)