Contents

I. APPLICATIONS

21. Bioinformatics

In this chapter at first we present algorithms on sequences, trees and stochastic grammars, then we continue with algorithms of comparison of structures and constructing of evolutionary trees, and finish the chapter with some rarely discussed topics of bioinformatics.

21.1. Algorithms on sequences

In this section, we are going to introduce dynamic programming algorithms working on sequences. Sequences are finite series of characters over a finite alphabet. The basic idea of dynamic programming is that calculations for long sequences can be given via calculations on substrings of the longer sequences.

The algorithms introduced here are the most important ones in bioinformatics, they are the basis of several software packages.

21.1.1. Distances of two sequences using linear gap penalty

DNA contains the information of living cells. Before the duplication of cells, the DNA molecules are doubled, and both daughter cells contain one copy of DNA. The replication of DNA is not perfect, the stored information can be changed by random mutations. Random mutations creates variants in the population, and these variants evolve to new species.

Given two sequences, we can ask the question how much the two species are related, and how many mutations are needed to describe the evolutionary history of the two sequences.

We suppose that mutations are independent from each other, and hence, the probability of a series of mutations is the product of probabilities of the mutations. Each mutation is associated with a weight, mutations with high probability get a smaller weight, mutations with low probability get a greater weight. A reasonable choice might be the logarithm of one over the probability of the mutation. In this case the weight of a series of mutations is the sum of the weights of the individual mutations. We also assume that mutation and its reverse have the same probability, therefore we study how a sequence can be transfered into another instead of evolving two sequences from a common ancestor. Assuming minimum evolution ***minimum evolution,*** we are seeking for the minimum weight series of mutations that transforms one sequence into another. An important question is how we can quickly find such a minimum weight series. The naive algorithm finds all the possible series of mutations and chooses the minimum weight. Since the possible number of series of mutations grows exponentially – as we are going to show it in this chapter –, the naive algorithm is obviously too slow.

We are going to introduce the Sellers' algorithm [69]. Let Σ be a finite set of symbols, and let Σ^* denote the set of finite long sequences over Σ. The n long prefix of $A \in \Sigma^*$ will be denoted by A_n, and a_n denotes the nth character of A. The following transformations can be applied for a sequence:

- Insertion of symbol a before position i, denoted by $a \leftarrow^i -$.
- Deletion of symbol a at position i, denoted by $- \leftarrow^i a$.
- Substitution of symbol a to symbol b at position i, denoted by $b \leftarrow^i a$.

The concatenation of mutations is denoted by the $\circ$ symbol. τ denotes the set of finite long concatenations of the above mutations, and $T(A) = B$ denotes that $T \in \tau$ transforms a sequence A into sequence B. Let $w : \tau \to R^+ \cup \{0\}$ a weight function such that for any T_1, T_2 and S transformations satisfying

$$T_1 \circ T_2 = S \;, \tag{21.1}$$

the

$$w(T_1) + w(T_2) = w(S) \;, \tag{21.2}$$

equation also holds. Furthermore, let $w(a \leftarrow^i b)$ be independent from i. The transformation distance between two sequences, A and B, is the minimum weight of transformations transforming A into B:

$$\delta(A, B) = \min\{w(T)|(T(A) = B\} \;. \tag{21.3}$$

If we assume that w satisfies

$$\begin{aligned} w(a \leftarrow b) &= w(b \leftarrow a) \;, && (21.4)\\ w(a \leftarrow a) &= 0 \;, && (21.5)\\ w(b \leftarrow a) + w(c \leftarrow b) &\geq w(c \leftarrow a) && (21.6) \end{aligned}$$

for any a, b, $c \in \Sigma \cup \{-\}$, then the $\delta(,)$ transformation distance is indeed a metric on Σ^*.

Since $w(,)$ is a metric, it is enough to concern with transformations that change each position of a sequence at most once. Series of transformations are depicted with ***sequence alignments.*** By convention, the sequence at the top is the ancestor and the sequence at the bottom is its descendant. For example, the alignment below shows that there were substitutions at positions three and five, there was an insertion in the first position and a deletion in the eighth position.

```
- A U C G U A C A G
U A G C A U A - A G
```

A pair at a position is called aligned pair. The weight of the series of transformations described by the alignment is the sum of the weights of aligned pairs. Each series of mutations can be described by an alignment, and this description is unique up the permutation of mutations in the series. Since the summation is commutative, the weight of the series of mutations does not depend on the order of mutations.

We are going to show that the number of possible alignments also grows exponentially with the length of the sequences. The alignments that do not contain this pattern

```
# -
- #
```

where # an arbitrary character of Σ gives a subset of possible alignments. The size of this subset is $\binom{|A|+|B|}{|A|}$, since there is a bijection between this set of alignments and the set of coloured sequences that contains the characters of A and B in increasing order, and the characters of A is coloured with one colour, and the characters of B is coloured with the other colour. For example, if $|A| = |B| = n$, then $|A| + |B|\binom{|}{A}| = \Theta(2^{2n}/n^{0.5})$.

An alignment whose weight is minimal called an ***optimal alignment.*** Let the set of optimal alignments of A_i and B_j be denoted by $\alpha^*(A_i, B_j)$, and let $w(\alpha^*(A_i, B_j))$ denote the weights of any alignment in $\alpha^*(A_i, B_j)$.

The key of the fast algorithm for finding an optimal alignment is that if we know $w(\alpha^*(A_{i-1}, B_j))$, $w(\alpha^*(A_i, B_{j-1}))$, and $w(\alpha^*(A_{i-1}, B_{j-1}))$, then we can calculate $w(\alpha^*(A_i, j_m))$ in constant time. Indeed, if we delete the last aligned pair of an optimal alignment of A_i and B_j, we get the optimal alignment of A_{i-1} and B_j, or A_i and B_{j-1} or A_{i-1} and B_{j-1}, depending on the last aligned column depicts a deletion, an insertion, substitution or match, respectively. Hence,

$$\begin{aligned} w(\alpha^*(A_i, B_j)) = \quad & \min\{w(\alpha^*(A_{i-1}, B_j)) + w(- \leftarrow a_i); \\ & w(\alpha^*(A_i, B_{j-1})) + w(b_i \leftarrow -); \\ & w(\alpha^*(A_{i-1}, B_{j-1})) + w(b_i \leftarrow a_i)\}. \end{aligned} \tag{21.7}$$

The weights of optimal alignments are calculated in the so-called ***dynamic programming table,*** D. The $d_{i,j}$ element of D contains $w(\alpha^*(A_i, B_j))$. Comparing of an n and an m long sequence requires the fill-in of an $(n+1)x(m+1)$ table, indexing of rows and columns run from 0 till n and m, respectively. The initial conditions for column 0 and row 0 are

$$d_{0,0} = 0, \tag{21.8}$$

$$d_{i,0} = \sum_{k=1}^{i} w(- \leftarrow a_k) , \tag{21.9}$$

$$d_{0,j} = \sum_{l=1}^{j} w(b_l \leftarrow -) . \tag{21.10}$$

The table can be filled in using Equation (21.7). The time requirement for the fill-in is $\Theta(nm)$. After filling in the dynamic programming table, the set of all optimal alignments can be find in the following way, called ***trace-back.*** We go from the right bottom corner to the left top corner choosing the cell(s) giving the optimal value of the current cell (there might be more than one such cells). Stepping up from position $d_{i,j}$ means a deletion, stepping to the left means an insertion, and the diagonal steps mean either a substitution or a match depending on whether or not $a_i = b_j$. Each step is represented with an oriented edge, in this way, we get an oriented graph, whose vertices are a subset of the cells of the dynamic programming table. The number of optimal alignments might grow exponentially with the length of the sequences, however, the set of optimal alignments can be represented in polynomial time and space. Indeed, each path from $d_{n,m}$ to $d_{0,0}$ on the oriented graph obtained in the trace-back gives an optimal alignment.

21.1.2. Dynamic programming with arbitrary gap function

Since deletions and insertions get the same weight, the common name of them is indel or ***gap,*** and their weights are called ***gap penalty.*** Usually gap penalties do not depend on the deleted or inserted characters. The gap penalties used in the previous section grow linearly with the length of the gap. This means that a long indel is considered as the result of independent insertions or deletions of characters. However, the biological observation is that long indels can be formed in one evolutionary step, and these long indels are penalised too much with the linear gap penalty function. This observation motivated the introduction of more complex gap penalty functions [81]. The algorithm introduced by Waterman *et al.* penalises a k long gap with g_k. For example the weight of this alignment:

```
- - A U C G A C G U A C A G
U A G U C - - - A U A G A G
```

is $g_2 + w(G \leftarrow A) + g_3 + w(A \leftarrow G) + w(G \leftarrow C)$.

We are still seeking for the minimal weight series of transformations transforming one sequence into another or equivalently for an optimal alignment. Since there might be a long indel at the end of the optimal alignment, above knowing $w(\alpha^*(A_{i-1}, B_{j-1}))$, we must know all $w(\alpha^*(A_k, B_j))$, $0 \le k < i$ and $w(\alpha^*(A_i, B_l))$, $0 \le l < j$ to calculate $w(\alpha^*(A_i, B_j))$. The dynamic programming recursion is given by the following equations:

$$w(\alpha^*(A_i, B_j)) = \min\{w(\alpha^*(A_{i-1}, B_{j-1})) + w(b_j \leftarrow a_i) ; \\ \min_{0 \le k < i}\{w(\alpha^*(A_k, B_j)) + g_{i-k}\} ; \\ \min_{0 \le l < j}\{w(\alpha^*(A_i, B_l)) + g_{j-l}\}\} . \tag{21.11}$$

The initial conditions are:

$$d_{0,0} = 0 , \tag{21.12}$$
$$d_{i,0} = g_i , \tag{21.13}$$
$$d_{0,j} = g_j . \tag{21.14}$$

The time requirement for calculating $d_{i,j}$ is $\Theta(i+j)$, hence the running time of the fill-in part to calculate the weight of an optimal alignment is $\Theta(nm(n+m))$. Similarly to the previous algorithm, the set of optimal alignments represented by paths from $d_{n,m}$ to $d_{0,0}$ can be found in the trace-back part.

If $|A| = |B| = n$, then the running time of this algorithm is $\Theta(n^3)$. With restrictions on the gap penalty function, the running time can be decreased. We are going to show two such algorithms in the next two sections.

21.1.3. Gotoh algorithm for affine gap penalty

A gap penalty function is ***affine*** if

$$g_k = uk + v, \quad u \geq 0, \;\; v \geq 0 \;. \tag{21.15}$$

There exists a $\Theta(nm)$ running time algorithm for affine gap penalty [27]. Recall that in the Waterman algorithm,

$$d_{i,j} = \min\{d_{i-1,j-1} + w(b_j \leftarrow a_i);\; p_{i,j};\; q_{i,j}\} \;, \tag{21.16}$$

where

$$p_{i,j} = \min_{0\leq k<i}\{d_{i-k,j} + g_k\} \;, \tag{21.17}$$

$$q_{i,j} = \min_{0\leq l<j}\{d_{i,j-l} + g_l\} \;. \tag{21.18}$$

The key of the Gotoh algorithm is the following reindexing

$$\begin{aligned} p_{i,j} &= \min\{d_{i-1,j} + g_1,\; \min_{1\leq k<i}\{d_{i-k,j} + g_k\}\} \\ &= \min\{d_{i-1,j} + g_1,\; \min_{0\leq k<i-1}\{d_{i-1-k,j} + g_{k+1}\}\} \\ &= \min\{d_{i-1,j} + g_1,\; \min_{0\leq k<i-1}\{d_{i-1-k,j} + g_k\} + u\} \\ &= \min\{d_{i-1,j} + g_1,\; p_{i-1,j} + u\} \;. \end{aligned} \tag{21.19}$$

And similarly

$$q_{i,j} = \min\{d_{i,j-1} + g_1,\; q_{i,j-1} + u\} \;. \tag{21.20}$$

In this way, $p_{i,j}$ és $q_{i,j}$ can be calculated in constant time, hence $d_{i,j}$. Thus, the running time of the algorithm remains $\Theta(nm)$, and the algorithm will be only a constant factor slower than the dynamic programming algorithm for linear gap penalties.

21.1.4. Concave gap penalty

There is no biological justification for the affine gap penalty function [7, 26], its wide-spread use (for example, CLUSTAL-W [75]) is due to its low running time. There is a more realistic gap penalty function for which an algorithm exists whose running time is slightly more than the running time for affine gap penalty, but it is still significantly better than the cubic running time algorithm of Waterman *et al.* [22, 52].

A gap penalty function is concave if for each i, $g_{i+1} - g_i \leq g_i - g_{i-1}$. Namely, the increasement of gap extensions are penalised less and less. It might happen that the function starts decreasing after a given point, to avoid this, it is usually assumed that the function increases monotonously. Based on empirical data [7], if two sequences evolved for d PAM unit [13], the weight of a q long indel is

$$35.03 - 6.88 \log_{10} d + 17.02 \log_{10} q \ , \tag{21.21}$$

which is also a concave function. (One PAM unit is the time span on which 1% of the sequence changed.) There exist an $O(nm(\log n + \log m))$ running time algorithm for concave gap penalty functions. this is a so-called ***forward looking*** algorithm. The Forward-Looking algorithm calculates the ith row of the dynamic programming table in the following way for an arbitrary gap penalty function:

Forward-Looking

```
 1  for $1 \leq j \leq m$
 2      $q_1[i,j] \leftarrow d[i,0] + g[j]$
 3      $b[i,j] \leftarrow 0$
 4  for $1 \leq j \leq m$
 5      $q[i,j] \leftarrow q_1[i,j]$
 6      $d[i,j] \leftarrow \min[q[i,j], p[i,j], d[i-1,j-1] + w(b_j \leftarrow a_i)]$
 7  ▷ At this step, we suppose that $p[i,j]$ and $d[i-1,j-1]$ are already calculated.
 8      for $j < j_1 \leq m$                                        ▷Inner cycle.
 9          if $q1[i,j_1] < d[i,j] + g[j_1 - j]$ then
10                          $q_1[j_1] \leftarrow d[i,j] + g[j_1 - j]$
11                          $b[i,j_1] \leftarrow j$
```

where $g[\,]$ is the gap penalty function and b is a pointer whose role will be described later. In row 6, we assume that we already calculated $p[i,j]$ and $d[i-1,j-1]$. It is easy to show that the forward looking algorithm makes the same comparisons as the traditional, backward looking algorithm, but in a different order. While the backward looking algorithm calculates $q_{i,j}$ at the jth position of the row looking back to the already calculated entries of the dynamic programming table, the Forward-Looking algorithm has already calculated $q_{i,j}$ by arriving to the jth position of the row. On the other hand, it sends forward candidate values for $q[i,j_1]$, $j_1 > j$, and by arriving to cell j_1, all the needed comparisons of candidate values have been made. Therefore, the Forward-Looking algorithm is not faster than the traditional backward looking algorithm, however, the conception helps accelerate the algorithm.

The key idea is the following.

Lemma 21.1 *Let j be the actual cell in the row. If*

$$d_{i,j} + g_{j_1 - j} \geq q_1[i,j_1] \ , \tag{21.22}$$

then for all $j_2 > j_1$

$$d_{i,j} + g_{j_2 - j} \geq q_1[i,j_2] \ . \tag{21.23}$$

Proof From the condition it follows that there is a $k < j < j_1 < j_2$ for which

$$d_{i,j} + g_{j_1-j} \geq d_{i,k} + g_{j_1-k} \ . \tag{21.24}$$

Let us add $g_{j_2-k} - g_{j_1-k}$ to the equation:

$$d_{i,j} + g_{j_1-j} + g_{j_2-k} - g_{j_1-k} \geq d_{i,k} + g_{j_2-k} \ . \tag{21.25}$$

For each concave gap penalty function,

$$g_{j_2-j} - g_{j_1-j} \geq g_{j_2-k} - g_{j_1-k} \ , \tag{21.26}$$

rearranging this and using Equation (21.25)

$$d_{i,j} + g_{j_2-j} \geq d_{i,j} + g_{j_1-j} + g_{j_2-k} - g_{j_1-k} \geq d_{i,k} + g_{j_2-k} \geq q[i, j_2] \tag{21.27}$$

■

The idea of the algorithm is to find the position with a binary search from where the actual cell cannot send forward optimal $q_{i,j}$ values. This is still not enough for the desired acceleration since $O(m)$ number of candidate values should be rewritten in the worst case. However, the corollary of the previous lemma leads to the desired acceleration:

Corollary 21.2 *Before the jth cell sends forward candidate values in the inner cycle of the forward looking algorithm, the cells after cell j form blocks, each block having the same pointer, and the pointer values are decreasing by blocks from left to right.*

The pseudocode of the algorithm is the following:

```
Forward-Looking-Binary-Searching($i, m, q, d, g, w, a, b$)

 1  $pn[i] \leftarrow 0$; $p[i,0] \leftarrow m$; $b[i,0] \leftarrow 0$
 2  for $j \leftarrow 1$ to $m$
 3       do $q[i,j] \leftarrow q[i, b[i, pn[i]]] + g[j - b[i, pn[i]]]$
 4          $d[i,j] \leftarrow \min[q[i,j], p[i,j], d[i-1, j-1] + w(b_j \leftarrow a_i)]$
 5  ▷ At this step, we suppose that $p[i,j]$ and $d[i-1,j-1]$ are already calculated.
 6          if $p[i, pn[i]] = j$ then
 7             then $pn[i] - -$
 8          if $j + 1 < m$ and $d[i, b[i,0]] + g[m - b[i,0]] > d[i,j] + g[m-j]$
 9             then $pn[i] \leftarrow 0$; $b[i,0] \leftarrow j$
10             else if $j + 1 < m$
11                     then $Y \leftarrow \max_{0 \leq X \leq pn[i]}\{X | d[i, b[i,X]] + g[p[i,X]$
                              $- b[i,X]] \leq p[i,j] + g[p[i,X] - j]\}$
12             if $d[i, b[i,Y]] + g[p[i,Y] - b[i,Y]] = p[i,j] + g[p[i,X] - j]$
13                then $pn[i] \leftarrow Y$; $b[i,Y] \leftarrow j$
14                else  $E = p[i,Y]$
15                      if $Y < pn[i]$
16                         then $B \leftarrow p[i, Y+1] - 1$
17                         else  $B \leftarrow j + 1$
```

```
18          $pn[i]++$
19          $b[i, pn[i]] \leftarrow j$
20          $p[i, pn[i]] \leftarrow \max_{B \leq X \leq E}\{X | d[i, j] + g[X - j] \leq$
                    $d[i, b[i, Y]] + g[x - b[i, Y]]\}$
```

The algorithm works in the following way: for each row, we maintain a variable storing the number of blocks, a list of positions of block ends, and a list of pointers for each block. For each cell j, the algorithm finds the last position for which the cell gives an optimal value using binary search. There is first a binary search for the blocks then for the positions inside the choosen block. It is enough to rewrite three values after the binary searches: the number of blocks, the end of the last block and its pointer. Therefore the most time consuming part is the binary search, which takes $O(\lg m)$ time for each cell.

We do the same for columns. If the dynamic programming table is filled in row by row, then for each position j in row i, the algorithms uses the block system of column j. Therefore the running time of the algorithm is $O(nm(\lg n + \lg m))$.

21.1.5. Similarity of two sequences, the Smith-Waterman algorithm

We can measure not only the distance but also the similarity of two sequences. For measuring the similarity of two characters, $S(a, b)$, the most frequently used function is the ***log-odds***:

$$S(a, b) = \log\left(\frac{\Pr\{a, b\}}{\Pr\{a\}\Pr\{b\}}\right) , \qquad (21.28)$$

where $\Pr\{a, b\}$ is the joint probability of the two characters (namely, the probability of observing them together in an alignment column), $\Pr\{a\}$ and $\Pr\{b\}$ are the marginal probabilities. The similarity is positive if $\Pr\{a, b\} > \Pr\{a\}\Pr\{b\}$, otherwise negative. Similarities are obtained from empirical data, for aminoacids, the most commonly used similarities are given by the PAM and BLOSUM matrices.

If we penalise gaps with negative numbers then the above described, global alignment algorithms work with similarities by changing minimalisation to maximalisation.

It is possible to define a special problem that works for similarities and does not work for distances. It is the local similarity problem or the local sequence alignment problem [71]. Given two sequences, a similarity and a gap penalty function, the problem is to give two substrings of the sequences whose similarity is maximal. A ***substring*** of a sequence is a consecutive part of the sequence. The biological motivation of the problem is that some parts of the biological sequences evolve slowly while other parts evolve fast. The local alignment finds the most conservated part of the two sequences. Local alignment is widely used for homology searching in databases. The reason why local alignments works well for homology searching is that the local alignment score can separate homologue and non-homologue sequences better since the statistics is not decreased due to the variable regions of the sequences.

The Smith-Waterman algorithm work in the following way. The initial conditions are:

$$d_{0,0} = d_{i,0} = d_{0,j} = 0 \ . \tag{21.29}$$

Considering linear gap penalty, the dynamic programming table is filled in using the following recursions:

$$d_{i,j} = \max\{0;\ d_{i-1,j-1} + S(a_i, b_j), d_{i-1,j} + g;\ d_{i,j-1} + g\} \ . \tag{21.30}$$

Here g, the gap penalty is a negative number. The best local similarity score of the two sequences is the maximal number in the table. The trace-back starts in the cell having the maximal number, and ends when first reaches a 0.

It is easy to prove that the alignment obtained in the trace-back will be locally optimal: if the alignment could be extended at the end with a sub-alignment whose similarity is positive then there would be a greater number in the dynamic programming table. If the alignment could be extended at the beginning with a subalignment having positive similarity then the value at the end of the traceback would not be 0.

21.1.6. Multiple sequence alignment

The multiple sequence alignment problem was introduced by David Sankoff [68], and by today, the multiple sequence alignment has been the central problem in bioinformatics. Dan Gusfield calls it the Holy Grail of bioinformatics. [30]. Multiple alignments are widespread both in searching databases and inferring evolutionary relationships. Using multiple alignments, it is possible to find the conservated parts of a sequence family, the positions that describe the functional properties of the sequence family. AS Arthur Lesk said: [33]: *What two sequences whisper, a multiple sequence alignment shout out loud.*

The columns of a multiple alignment of k sequences is called aligned k-tuples. The dynamic programming for the optimal multiple alignment is the generalisation of the dynamic programming for optimal pairwise alignment. To align k sequences, we have to fill in a k dimensional dynamic programming table. To calculate an entry in this table using linear gap penalty, we have to look back to a k dimensional hypercube. Therefore, the memory requirement in case of k sequence, n long each is $\Theta(n^k)$, and the running time of the algorithm is $\Theta(2^k n^k)$ if we use linear gap penalty, and $\Theta(n^{2k-1})$ with arbitrary gap penalty.

There are two fundamental problems with the multiple sequence alignment. The first is an algorithmic problem: it is proven that the multiple sequence alignment problem is NP-complete [80]. The other problem is methodical: it is not clear how to score a multiple alignment. An objective scoring function could be given only if the evolutionary relationships were known, in this case an aligned k-tuple could be scored according to an evolutionary tree [59].

A heuristic solution for both problems is the ***iterative sequence alignment*** [18],[12],[75]. This method first construct a ***guide-tree*** using pairwise distances (such tree-building methods are described in section 21.5). The guide-tree is used then to construct a multiple alignment. Each leaf is labelled with a sequence, and first the sequences in ”cherry-motives" are aligned into each other, then sequence alignments are aligned to sequences and sequence alignments according to the guide-tree. The iterative sequence alignment method uses the ”once a gap – always gap" rule. This means that gaps already placed into an alignment cannot be modified when aligning the alignment to other alignment or sequence. The only possibility is to insert all-gap columns into an alignment. The aligned sequences are usually described with a ***profile.*** The profile is a $(|\Sigma| + 1) \times l$ table, where l is the length of the alignment. A column of a profile contains the statistics of the corresponding aligned k-tuple, the frequencies of characters and the gap symbol.

The obtained multiple alignment can be used for constructing another guide-tree, that can be used for another iterative sequence alignment, and this procedure can be iterated till convergence. The reason for the iterative alignment heuristic is that the optimal pairwise alignment of closely related sequences will be the same in the optimal multiple alignment. The drawback of the heuristic is that even if the previous assumption is true, there might be several optimal alignments for two sequences, and their number might grow exponentially with the length of the sequences. For example, let us consider the two optimal alignments of the sequences AUCGGUACAG and AUCAUACAG.

```
  A U C G G U A C A G   A U C G G U A C A G
A U C - A U A C A G   A U C A - U A C A G  .
```

We cannot choose between the two alignments, however, in a multiple alignment, only one of them might be optimal. For example, if we align the sequence AUCGAU to the two optimal alignments, we get the following locally optimal alignments:

```
A U C G G U A C A G   A U C G G U A C A G
A U C - A U A C A G   A U C A - U A C A G
A U C G A U - - - -   A U C - G - A U - -
```

The left alignment is globally optimal, however, the right alignment is only locally optimal.

Hence, the iterative alignment method yields only a locally optimal alignment. Another problem of this method is that it does not give an upper bound for the goodness of the approximation. In spite of its drawback, the iterative alignment methods are the most widely used ones for multiple sequence alignments in practice, since it is fast and usually gives biologically reasonable alignments. Recently some approximation methods for multiple sequence alignment have been published with known upper bounds for their goodness [29, 65]. However, the bounds biologically are not reasonable, and in practice, these methods usually give worse results than the heuristic methods.

We must mention a novel greedy method that is not based on dynamic programming. The DiAlign [53, 54, 55] first searches for gap-free homologue substrings by pairwise sequence comparison. The gap-free alignments of the homologous substrings are called diagonals of the dynamic programming name, hence the name of the method: Diagonal Alignment. The diagonals are scored according to their similarity value and diagonals that are not compatible with high-score diagonals get a penalty. Two diagonals are not compatible if they cannot be in the same alignment. After scoring the diagonals, they are aligned together a multiple alignment in a greedy way. First the best diagonal is selected, then the best diagonal that is comparable with the first one, then the third best alignment that is comparable with the first two ones, etc. The multiple alignment is the union of the selected diagonals that might not cover all the characters in the sequence. Those characters that were not in any of the selected diagonals are marked as ”non alignable". The drawback of the method is that sometimes it introduces too many gaps due to not penalising the gaps at all. However, DiAlign has been one of the best heuristic alignment approach and is widely used in the bioinformatics community.

21.1.7. Memory-reduction with the Hirschberg algorithm

If we want to calculate only the distance or similarity between two sequences and we are not interested in an optimal alignment, then in case of linear or affine gap penalties, it is very easy to construct an algorithm that uses only linear memory. Indeed, note that the dynamic programming recursion needs only the previous row (in case of filling in the dynamic table by rows), and the algorithm does not need to store earlier rows. On the other hand, once the dynamic programming table has reached the last row and forgot the earlier rows, it is not possible to trace-back the optimal alignment. If the dynamic programming table is scrolled again and again in linear memory to trace-back the optimal alignment row by row then the running time grows up to $O(n^3)$, where n is the length of the sequences.

However, it is possible to design an algorithm that obtains an optimal alignment in $O(n^2)$ running time and uses only linear memory. This is the ***Hirschberg algorithm*** [32], which we are going to introduce for distance-based alignment with linear gap penalty.

We introduce the suffixes of a sequence, a suffix is a substring ending at the end of the sequence. Let A^k denote the suffix of A starting with character a_{k+1}.

The Hirschberg algorithm first does a dynamic programming algorithm for sequences $A_{[|A|/2]}$ and B using liner memory as described above. Similarly, it does a dynamic programming algorithm for the reverse of the sequences $A^{[|A|/2]}$ and B.

Based on the two dynamic programming procedures, we know what is the score of the optimal alignment of $A_{[|A|/2]}$ and an arbitrary prefix of B, and similarly what is the score of the optimal alignment of $A^{[|A|/2]}$ and an arbitrary suffix of B. >From this we can tell what is the score of the optimal alignment of A and B:

$$\min_j \left\{ w(\alpha^*(A_{[|A|/2]}, B_j)) + w(\alpha^*(A^{[|A|/2]}, B^j)) \right\} , \tag{21.31}$$

and from this calculation it must be clear that in the optimal alignment of A and B, $A_{[|A|/2]}$ is aligned with the prefix B_j for which

$$w(\alpha^*(A_{[|A|/2]}, B_j)) + w(\alpha^*(A^{[|A|/2]}, B^j)) \tag{21.32}$$

is minimal.

Since we know the previous rows of the dynamic tables, we can tell if $a_{[|A|/2]}$ and $a_{[|A|/2]+1}$ is aligned with any characters of B or these characters are deleted in the optimal alignment. Similarly, we can tell if any character of B is inserted between

$a_{[|A|/2]}$ and $a_{[|A|/2]+1}$.

In this way, we get at least two columns of the optimal alignment. Then we do the same for $A_{[|A|/2]-1}$ and the remaining part of the prefix of B, and for $A^{[|A|/2]+1}$ and the remaining part of the suffix of B. In this way we get alignment columns at the quarter and the three fourths of sequence A. In the next iteration, we do the same for the for pairs of sequences, etc., and we do the iteration till we get all the alignment columns of the optimal alignment.

Obviously, the memory requirement still only grows linearly with the length of the sequences. We show that the running time is still $\Theta(nm)$, where n and m are the lengths of the sequences. This comes from the fact that the running time of the first iteration is $|A| \times |B|$, the running time of the second iteration is $|A|/2) \times j^* + (|A|/2) \times (|B| - j^*$, where j^* is the position for which we get a minimum distance in Eqn. (21.31). Hence the total running time is:

$$nm \times \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) = \Theta(nm) \ . \tag{21.33}$$

21.1.8. Memory-reduction with corner-cutting

The dynamic programming algorithm reaches the optimal alignment of two sequences with aligning longer and longer prefixes of the two sequences. The algorithm can be accelerated with excluding the bad alignments of prefixes that cannot yield an optimal alignment. Such alignments are given with the ordered paths going from the right top and the left bottom corners to $d_{0,0}$, hence the name of the technique.

Most of the corner-cutting algorithms use a test value. This test value is an upper bound of the evolutionary distance between the two sequences. Corner-cutting algorithms using a test value can obtain the optimal alignment of two sequences only if the test value is indeed smaller then the distance between the two sequences, otherwise the algorithm stops before reaching the right bottom corner or gives a non-optimal alignment. Therefore these algorithms are useful for searching for sequences similar to a given one and we are not interested in sequences that are farther from the query sequence than the test value.

We are going to introduce two algorithms. the Spouge algorithm [72],[73] is a generalisation of the Fickett [19] and the Ukkonen algorithm [78],[79]. The other algorithm was given by Gusfield, and this algorithm is an example for a corner-cutting algorithm that reaches the right bottom corner even if the distance between the two sequence is greater than the test value, but in this case the calculated score is bigger than the test value, indicating that the obtained alignment is not necessary optimal.

The Spouge algorithm calculates only those $d_{i,j}$ for which

$$d_{i,j} + |(n-i) - (m-j)| \times g \leq t \ , \tag{21.34}$$

where t is the test value, g is the gap penalty, n and m are the length of the sequences. The key observation of the algorithm is that any path going from $d_{i,j}$ to $d_{n,m}$ will increase the score of the alignment at least by $|(n-i) - (m-j)| \times g$. Therefore is t is smaller than the distance between the sequences, the Spouge algorithm obtains the optimal alignments, otherwise will stop before reaching the right bottom corner.

This algorithm is a generalisation of the Fickett algorithm and the Ukkonen algorithm. Those algorithms also use a test value, but the inequality in the Fickett algorithm is:

$$d_{i,j} \leq t \ , \tag{21.35}$$

while the inequality in the Ukkonen algorithm is:

$$|i-j| \times g + |(n-i)-(m-j)| \times g \leq t \ . \tag{21.36}$$

Since in both cases, the left hand side of the inequalities are not greater than the left end side of the Spouge inequality, the Fickett and the Ukkonen algorithms will calculate at least as much part of the dynamic programming table than the Spouge algorithm. Empirical results proved that the Spouge algorithm is significantly better [73]. The algorithm can be extended to affine and concave gap penalties, too.

The k-difference global alignment problem [30] asks the following question: Is there an alignment of the sequences whose weight is smaller than k? The algorithm answering the question has $O(kn)$ running time, where n is the length of the longer sequence. The algorithm is based on the observation that any path from $d_{n,m}$ to $d_{0,0}$ having at most score k cannot contain a cell $d_{i,j}$ for which $|i-j| > k/g$. Therefore the algorithm calculates only those $d_{i,j}$ cells for which $(i-j) < k/g$ and disregards the $d_{e,f}$ neighbours of the border elements for which $|e-f| > k/g$. If there exists an alignment with a score smaller or equal than k, then $d_{n,m} < k$ and $d_{n,m}$ is indeed the distance of the two sequences. Otherwise $d_{n,m} > k$, and $d_{n,m} > k$ is not necessary the score of the optimal alignment since there might be an alignment that leaves the band defined by the $|i-j| < k/g$ inequality and still has a smaller score then the best optimal alignment in the defined band.

The corner-cutting technique has been extended to multiple sequence alignments scored by the ***sum-of-pairs*** scoring scheme [9]. The sum-of-pairs score is:

$$SP_l = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} d(c_{i,l}, \, c_{j,l}) \ , \tag{21.37}$$

where SP_l is the lth aligned k-tuple $d(,)$ is the distance function on $\Sigma \cup \{-\}$, k is the number of sequences, $c_{i,j}$ is the character of the multiple alignment in the ith row and jth column. The l-suffix of sequence S is S^l. Let $w_{i,j}(l,m)$ denote the distance of the optimal alignment of the l-suffix and the m-suffix of the ith and the jth sequences. The Carillo and Lipman algorithm calculates only the positions for which

$$d_{i_1,i_2,\ldots i_n} + \sum_{j=1}^{k-1} \sum_{l=j}^{k} w_{j,l}(i_j, i_l) \leq t \ , \tag{21.38}$$

where t is the test value. The goodness of the algorithm follows from the fact that the sum-of-pairs score of the optimal alignment of the not yet aligned suffixes cannot be smaller than the sum of the scores of the optimal pairwise alignments. This corner cutting method can align at most six moderately long sequences [43].

Exercises

21.1-1 Show that the number of possible alignments of an n and an m long sequences is

$$\sum_{i=0}^{\min(n,m)} \frac{(n+m-i)!}{(n-i)!(m-i)!i!} .$$

21.1-2 Give a series of pairs of sequences and a scoring scheme such that the number of optimal alignments grows exponentially with the length of the sequences.
21.1-3 Give the Hirschberg algorithm for multiple alignments.
21.1-4 Give the Hirschberg algorithm for affine gap penalties.
21.1-5 Give the Smith-Waterman algorithm for affine gap penalties.
21.1-6 Give the Spouge algorithm for affine gap penalties.
21.1-7 Construct an example showing that the optimal multiple alignment of three sequences might contain a pairwise alignment that is only suboptimal.

21.2. Algorithms on trees

Algorithms introduced in this section work on rooted trees. The dynamic programming is based on the reduction to rooted subtrees. As we will see, above obtaining optimal cases, we can calculate algebraic expressions in the same running time.

21.2.1. The small parsimony problem

The (weighted) parsimony principle is to describe the changes of biological sequences with the minimum number (minimum weight) of mutations. We will concern only with substitutions, namely, the input sequences has the same length and the problem is to give the evolutionary relationships of sequences using only substitutions and the parsimony principle. We can define the large and the small parsimony problem. For the large parsimony problem, we do not know the topology of the evolutionary tree showing the evolutionary relationships of the sequences, hence the problem is to find both the best topology and an evolutionary history on the tree. The solution is not only locally but globally optimal. It has been proved that the large parsimony problem is NP-complete [21].

The small parsimony problem is to find the most parsimonious evolutionary history on a given tree topology. The solution for the small parsimony problem is only locally optimal, and there is no guarantee for global optimum.

Each position of the sequences is scored independently, therefore it is enough to find a solution for the case where there is only one character at each leaf of the tree. In this case, the evolutionary history can be described with labelling the internal nodes with characters. If two characters at neighbouring vertices are the same, then no mutation happened at the corresponding edge, otherwise one mutation happened. The naive algorithm investigates all possible labelings and selects the most parsimonious solution. Obviously, it is too slow, since the number of possible labelings grows exponentially with the internal nodes of the tree.

The dynamic programming is based on the reduction to smaller subtrees [68]. Here the definition of subtrees is the following: there is a natural partial ordering on the nodes in the rooted subtree such that the root is the greatest node and the leaves are minimal. A subtree is defined by a node, and the subtree contains this node and all nodes that are smaller than the given node. The given node is the root of the subtree. We suppose that for any t child of the node r and any character ω we know the minimum number of mutations that are needed on the tree with root t given that there is ω at node t. Let $m_{t,\omega}$ denote this number. Then

$$m_{r,\omega} = \sum_{t \in D(r)} \min_{\sigma\in\Sigma}\{m_{t,\sigma} + \delta_{\omega,\sigma}\} \ , \tag{21.39}$$

where $D(r)$ is the set of children of r, Σ is the alphabet, and $\delta_{\omega,\sigma}$ is 1 if $\omega = \sigma$ and 0 otherwise.

The minimum number of mutations on the entire tree is $\min_{\omega\in\Sigma} m_{R,\omega}$, where R is the root of the tree. A most parsimonious labelling can be obtained with trace-backing the tree from the root to the leaves, writing to each nodes the character that minimises Eqn. 21.39. To do this, we have to store $m_{r,\omega}$ for all r and ω.

The running time of the algorithm is $\Theta(n|\Sigma|^2)$ for one character, where n is the number of nodes of the tree, and $\Theta(nl|\Sigma|^2)$ for entire sequences, where l is the length of the sequences.

21.2.2. The Felsenstein algorithm

The input of the ***Felsenstein algorithm*** [17] is a multiple alignment of DNA (or RNA or protein) sequences, an evolutionary tree topology and edge lengths, and a model that gives for each pair of characters, σ and ω and time t, what is the probability that σ evolves to ω duting time t. Let $f_{\sigma\omega}(t)$ denote this probability. The equilibrium probability distribution of the characters is denoted by π. The question is what is the likelihood of the tree, namely, what is the probability of observing the sequences at the leaves given the evolutionary parameters consisting of the edge lengths and parameters of the substitution model.

We assume that each position evolves independently, hence the probability of an evolutionary process is the product of the evolutionary probabilities for each position. Therefore it is enough to show how to calculate the likelihood for a sequence position. We show this for an example tree that can be seen on Figure 21.1. s_i will denote the character at node i and v_j is the length of edge j. Since we do not know the characters at the internal nodes, we must sum the probabilities for all possible configurations:

$$\begin{aligned} L &= \sum_{s_0}\sum_{s_6}\sum_{s_7}\sum_{s_8} \pi_{s_0} \times f_{s_0 s_6}(v_6) \times f_{s_6 s_1}(v_1) \times f_{s_6 s_2}(v_2) \\ &\quad \times f_{s_0 s_8}(v_8) \times f_{s_8 s_3}(v_3) \times f_{s_8 s_7}(v_7) \times f_{s_7 s_4}(v_4) \times f_{s_7 s_5}(v_5) . \end{aligned} \tag{21.40}$$

If we consider the four character alphabet of DNA, the summation has 256 members, an in case of n species, it would have 4^{n-1}, namely the computational time grows

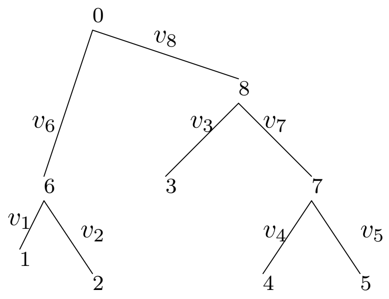

Figure 21.1 The tree on which we introduce the Felsenstein algorithm. Evolutionary times are denoted with vs on the edges of the tree.

exponentially with the number of sequences. However, if we move the expressions not depending on the summation index out of the summation, then we get the following product:

$$L = \sum_{s_0} \pi_{s_0} \left\{ \sum_{s_6} f_{s_0 s_6}(v_6)[f_{s_6 s_1}(v_1)][f_{s_6 s_2}(v_2)] \right\} \times$$
$$\left\{ \sum_{s_8} f_{s_0 s_8}(v_8)[f_{s_8 s_3}(v_3)] \left(\sum_{s_7} f_{s_8 s_7}(v_7)[f_{s_7 s_4}(v_4)][f_{s_7 s_5}(v_5)] \right) \right\} \quad (21.41)$$

which can be calculated in significantly less time. Note that the parenthesis in (21.41) gives the topology of the tree. Each summation can be calculated independently then we multiply the results. Hence the running time of calculating the likelihood for one position decreases to $\Theta(|\Sigma|^2 n)$ and the running time of calculating the likelihood for the multiple alignment is $\Theta(|\Sigma|^2 nl)$ where l is the length of the alignment.

Exercises

21.2-1 Give an algorithm for the weighted small parsimony problem where we want to get minimum weight evolutionary labeling given a tree topology and a set of sequences associated to the leaves of the tree.

21.2-2 The gene content changes in species, a gene that can be found in a genome of a species might be abundant in another genome. In the simplest model an existing gene might be deleted from the genome and an abundant gene might appear. Give the small parsimony algorithm for this gene content evolution model.

21.2-3 Give an algorithm that obtains the Maximum Likelihood labelling on a tree.

21.2-4 Rewrite the small parsimony problem in the form of (21.40) replacing sums with minimalisation, and show that the Sankoff algorithm is based on the same rearrangement as the Felsenstein algorithm.

21.2-5 The Fitch algorithm [20] works in the following way: Each r node is associated with a set of characters, C_r. The leaves are associated with a set containing the character associated to the leaves, and each internal character r has the set:

$$\begin{array}{lll} \cap_{t\in D(r)} C_t & \text{if} & \cap_{t\in D(r)} C_t \neq \emptyset \\ \cup_{t\in D(r)} C_t & & \text{otherwise} \quad , \end{array}$$

After reaching the root, we select an arbitrary character from C_R, where R is the root of the tree, and we choose the same character that we chose at the parent node if the set of the child node has this character, otherwise an arbitrary character from the set of the child node. Show that we get a most parsimonious labelling. What is the running time of this algorithm?

21.2-6 Show that the Sankoff algorithm gives all possible most parsimonious labelling, while there are most parsimonious labellings that cannot be obtained with the Fitch algorithm.

21.3. Algorithms on stochastic grammars

Below we give algorithms on stochastic transformational grammars. Stochastic transformational grammars play a central role in modern bioinformatics. Two types of transformational grammars are widespread, the Hidden Markov Models (HMMs) are used for protein structure prediction and gene finding, while Stochastic Context Free Grammars (SCFGs) are used for RNA secondary structure prediction.

21.3.1. Hidden Markov Models

We give the formal definition of **H**idden **M**arkov **M**odels (HMM): Let X denote a finite set of states. There are two distinguished states among the states, the start and the end states. The states are divided into two parts, emitting and non-emitting states. We assume that only the start and the end states are non-emitting, we will show that this assumption is not too strict.

The $\mathbf{M}$ transformation matrix contains the transition probabilities, m_{ij}, that the Markov process will jump to state j from state i. Emitting states emit characters form a finite alphabet, Σ. The probability that the state i emits a character ω will be denoted by π^i_ω. The Markov process starts in the start state and ends in the end state, and walks according to the transition probabilities in $\mathbf{M}$. Each emitting state emits a character, the emitted characters form a sequence. The process is hidden since the observer observes only the sequence and does not observe the path that the Markov process walked on. There are three important questions for HMMs that can be answered using dynamic programming algorithms.

The first question is the following: given an HMM and a sequence, what is the most likely path that emits the given sequence? The Viterbi algorithm gives the answer for this question. Recall that A_k is the k-long prefix of sequence A, and a_k is the character in the kth position. The dynamic programming answering the first question is based on that we can calculate the $\Pr_{max}\{A_{k+1}, j\}$ probability, the probability of the most probable path emitting prefix A_{k+1} and being in state j if we already calculated $\Pr_{max}\{\max\}(A_k, i)$ for all possible i, since

$$\Pr_{max}\{A_{k+1},\, j\} = \max_i(\Pr_{max}\{A_k,\, i\}\, m_{i,j}\pi^j_{a_{k+1}}) \; . \tag{21.42}$$

The reason behind the above equation is that the probability of any path is the product of transition and emission probabilities. Among the products having the same last two terms (in our case $m_{i,j}\pi^j_{ak+1}$) the maximal is the one for which the product of the other terms is maximal.

The initialisation of the dynamic programming is

$$\Pr_{max}\{A_0, START\} = 1 \; . \tag{21.43}$$

Since the end state does not emit a character, the termination of the dynamic programming algorithm is

$$\Pr_{max}\{A\} = \Pr_{max}\{A,\, END\} = \max_i(\Pr_{max}\{A,\, i\}\, m_{i,END}) \; , \tag{21.44}$$

where $\Pr_{max}\{A\}$ is the probability of the most likely path emitting the given sequence. One of the most likely paths can be obtained with a trace-back.

The second question is the following: given an HMM and a sequence, what is the probability that the HMM emits the sequence? This probability is the sum of the probabilities of paths that emit the given sequence. Since the number of paths emitting a given sequence might grow exponentially with the length of the sequence, the naive algorithm that finds all the possible emitting paths and sum their probabilities would be too slow.

The dynamic programming algorithm that calculates quickly the probability in question is called the Forward algorithm. It is very similar to the Viterbi algorithm, just there is a sum instead of maximalisation in it:

$$\Pr\{A_{k+1},\, j\} = \sum_i \Pr\{A_k,\, i\}\, m_{i,j}\pi^j_{a_{k+1}} \; . \tag{21.45}$$

Since the *END* state does not emit, the termination is

$$\Pr\{A\} = \Pr\{A,\, END\} = \sum_i \Pr\{A,\, i\}\, m_{i,END} \; . \tag{21.46}$$

where $\Pr\{A\}$ is the probability that the HMM emits sequence A.

The most likely path obtained by the Viterbi algorithm has more and less reliable parts. Therefore we are interested in the probability

$$\Pr\{a_k \text{ is emitted by state } i \mid \text{the HMM emitted sequence } A\} \; .$$

This is the third question that we answer with dynamic programming algorithm. The above mentioned probability is the sum of the probabilities of paths that emit a_k in state i divided by the probability that the HMM emits sequence A. Since the number of such paths might grow exponentially, the naive algorithm that finds all the possible paths and sum their probability is too slow.

To answer the question, first we calculate for each suffix A^k and state i what is the probability that the HMM emits suffix A^k given that state i emits a_k. This can be calculated with the Backward algorithm, which is similar to the Forward algorithm just starts the recursion with the end of the sequence:

$$\Pr\left\{A^k,\, i\right\} = \sum_j (\Pr\left\{A^{k+1},\, j\right\} m_{i,j}\pi^j_{a_{k+1}}) \; . \tag{21.47}$$

Let $\Pr\{a_k = i|A\}$ denote the probability

$$\Pr\{a_k \text{ is emitted by state } i \mid \text{the HMM emitted sequence } A\} \; .$$

Then

$$\Pr\{a_k = i|A\}\Pr\{A\} = \Pr\{A \wedge a_k = i\} = \Pr\{A_k,\, i\}\Pr\left\{A^k,\, i\right\} \; , \tag{21.48}$$

and from this

$$\Pr\{a_k = i|A\} = \frac{\Pr\{A_k,\, i\}\Pr\left\{A^k,\, i\right\}}{\Pr\{A\}} \; , \tag{21.49}$$

which is the needed probability.

21.3.2. Stochastic context-free grammars

It can be shown that every context-free grammar can be rewritten into *Chomsky normal form*. Each rule of a grammar in Chomsky normal form has the form $W_v \to W_y W_z$ or $W_w \to a$, where the Ws are non-terminal symbols, and a is a terminal symbol. In a stochastic grammar, each derivation rule has a probability, a non-negative number such that the probabilities of derivation rules for each non-terminal sum up to 1.

Given a SCFG and a sequence, we can ask the questions analogous to the three questions we asked for HMMs: what is the probability of the most likely derivation, what is the probability of the derivation of the sequence and what is the probability that a sub-string has been derivated starting with W_x non-terminal, given that the SCFG derivated the sequence. The first question can be answered with the CYK (Cocke-Younger-Kasami) algorithm which is the Viterbi-equivalent algorithm for SCFGs. The second question can be answered with the Inside algorithm, this is the Forward-equivalent for SCFGs. The third question can be answered with the combination of the Inside and Outside algorithms, as expected, the OUTSIDE algorithm is analogous to the Backward algorithm. Though the introduced algorithms are equivalent with the algorithms used for HMMs, their running time is significantly greater.

Let $t_v(y, z)$ denote the probability of the rule $W_v \to W_y W_z$, and let $e_v(a)$ denote the probability of the rule $W_v \to a$. The INSIDE algorithm calculates $\alpha(i, j, v)$ for all $i \leq j$ and v, this is the probability that non-terminal W_v derives the substring from a_i till a_j. The initial conditions are:

$$\alpha(i, i, v) = e_v(a_i) \; , \tag{21.50}$$

for all i and v. The main recursion is:

$$\alpha(i,j,v) = \sum_{y=1}^{M}\sum_{z=1}^{M}\sum_{k=i}^{j-1} \alpha(i,k,y)t_v(y,z)\alpha(k+1,j,z) \; , \tag{21.51}$$

where M is the number of non-terminals. The dynamic programming table is an upper triangle matrix for each non-terminal, the filling-in of the table starts with the main diagonal, and is continued with the other diagonals. The derivation probability is $\alpha(1,L,1)$, where L is the length of the sequence, and W_1 is the starting non-terminal. The running time of the algorithm is $\Theta(L^3M^3)$, the memory requirement is $\Theta(L^2M)$.

The Outside algorithm calculates $\beta(i,j,v)$ for all $i \leq j$ and v, this is the part of the derivation probability of deriving sequence A which is "outside" of the derivation of substring from a_i till a_j, starting the derivation from W_v. A more formal definition for $\beta(i,j,v)$ is that this is the sum of derivation probabilities in whom the substring from a_i till a_j is derived from W_v, divided by $\alpha(i,j,v)$. Here we define $0/0$ as 0. The initial conditions are:

$$\beta(1,L,1) = 1 \tag{21.52}$$
$$\beta(1,L,v) = 0 \quad \texttt{ha } v \neq 1 \; . \tag{21.53}$$

The main recursion is:

$$\begin{aligned}\beta(i,j,v) \quad = \quad & \sum_{y=1}^{M}\sum_{z=1}^{M}\sum_{k=1}^{i-1} \alpha(k,i-1,z)t_y(z,v)\beta(k,j,y) + \\ & \sum_{y=1}^{M}\sum_{z=1}^{M}\sum_{k=j+1}^{L} \alpha(j+1,k,z)t_y(z,v)\beta(i,k,y) \; . \end{aligned} \tag{21.54}$$

The reasoning for Eqn. 21.54 is the following. The W_v non-terminal was derivated from a W_y non-terminal together with a W_z non-terminal, and their derivation order could be both W_zW_v and W_vW_z. The outside probability of non-terminal W_v is product of the outside probability of W_y, the derivation probability and the inside probability of W_z. As we can see, inside probabilities are needed to calculate outside probabilities, this is a significant difference from the Backward algorithm that can be used without a Forward algorithm.

The CYK algorithm is very similar to the Inside algorithm, just there are maximalisations instead of summations:

$$\alpha_{\max}(i,j,v) = \max_y \max_z \max_{i \leq k \leq j-1} \alpha_{\max}(i,k,y)t_v(y,z)\alpha_{\max}(k+1,j,z) \; , \tag{21.55}$$

The probability of the most likely derivation is $\alpha_{\max}(1,L,1)$. The most likely derivation can be obtained with a trace-back.

Finally, the probability that the substring from a_i till a_j has been derived by W_v given that the SCFG derived the sequence is:

$$\frac{\alpha(i,j,v)\beta(i,j,v)}{\alpha(1,L,1)} \; . \tag{21.56}$$

Exercises

21.3-1 In a regular grammar, each derivation rule is either in a form $W_v \to aW_y$ or in a form $W_v \to a$. Show that each HMM can be rewritten as a stochastic regular grammar. On the other hand, there are stochastic regular grammars that cannot be described as HMMs.

21.3-2 Give a dynamic programming algorithm that calculate for a stochastic regular grammar and a sequence A

- the most likely derivation,
- the probability of derivation,
- the probability that character a_i is derived by non-terminal W.

21.3-3 An HMM can contain silent states that do not emit any character. Show that any HMM containing silent states other than the start and end states can be rewritten to an HMM that does not contain silent states above the start and end states and emits sequences with the same probabilities.

21.3-4 Pair Hidden Markov models are Markov models in which states can emit characters not only to one but two sequences. Some states emit only into one of the sequences, some states emit into both sequences. The observer sees only the sequences and does not see which state emits which characters and which characters are co-emitted. Give the Viterbi, Forward and Backward algorithms for pair-HMMs.

21.3-5 The Viterbi algorithm does not use that probabilities are probabilities, namely, they are non-negative and sum up to one. Moreover, the Viterbi algorithm works if we replace multiplications to additions (say that we calculate the logarithm of the probabilities). Give a modified HMM, namely, in which "probabilities" not necessary sum up to one, and they might be negative, too, and the Viterbi algorithm with additions are equivalent with the Gotoh algorithm.

21.3-6 Secondary structures of RNA sequences are set of basepairings, in which for all basepairing positions $i\dot{j}$ and $i'\dot{j}'$, $i < i'$ implies that either $i < j < i' < j'$ or $i < i' < j' < i$. The possible basepairings are $A-U$, $U-A$, $C-G$, $G-C$, $G-U$ and $U-G$. Give a dynamic programming algorithm that finds the secondary structure containing the maximum number of basepairings for an RNA sequence. This problem was first solved by Nussionov *et al.* [57].

21.3-7 The derivation rules of the Knudsen-Hein grammar are [38], [39]

$$\begin{aligned} S &\to LS|L \\ F &\to dFd|LS \\ L &\to s|dFd \end{aligned}$$

where s has to be substituted with the possible characters of RNA sequences, and the ds in the dFd expression have to be replaced by possible basepairings. Show that the probability of the derivation of a sequence as well as the most likely derivation can be obtained without rewriting the grammar into Chomsky normal form.

21.4. Comparing structures

In this section, we give dynamic programming algorithms for comparing structures. As we can see, aligning labelled rooted trees is a generalisation of sequence alignment. The recursions in the dynamic programming algorithm for comparing HMMs yields a linear equation system due to circular dependencies. However, we still can call it dynamic programming algorithm.

21.4.1. Aligning labelled, rooted trees

Let Σ be a finite alphabet, and $\Sigma^- = \Sigma \cup \{-\}$, $\Sigma^2 = \Sigma^- \times \Sigma^- \backslash \{-,-\}$. Labelling of tree F is a function that assigns a character of Σ to each node $n \in V_F$. If we delete a node from the tree, then the children of the node will become children of the parental node. If we delete the root of the tree, then the tree becomes a forest. Let A be a rooted tree labelled with characters from Σ^2, and let $c : V_A \to \Sigma^2$ represent the labelling. A is an alignment of trees F and G labelled with characters from Σ if restricting the labeling of A to the first (respectively, second) coordinates and deleting nodes labelled with '$-$' yields tree F (respectively, G). Let $s : \Sigma^2 \to R$ be a similarity function. An optimal alignment of trees F and G is the tree A labelled with Σ^2 for which

$$\sum_{n \in V_A} s(c(n)) \qquad (21.57)$$

is maximal. This tree is denoted by $A_{F,G}$. Note that a sequence can be represented with a unary tree, which has a single leaf. Therefore aligning trees is a generalisation of aligning sequences (with linear gap penalty).

Below we will concern only with trees in which each node has a degree at most 3. The recursion in the dynamic programming algorithm goes on rooted subtrees. A rooted subtree of a tree contains a node n of the tree and all nodes that are smaller than n. The tree obtained by root r is denoted by t_r.

A tree to an empty tree can be aligned only in one way. Two leafes labelled by a and b can be aligned in three different way. The alignment might contain only one node labelled with (a, b) or might contain two nodes, one of them is labelled with $(a, -)$, the other with $(-, b)$. One of the points is the root, the other the leaf.

Similarly, when we align a single leaf to a tree, then in the alignment A either the single character of the node is labelled together with a character of the tree or labelled together with '$-$' in an independent node. This node can be placed in several ways on tree A, however the score of any of them is the same.

After this initialisation, the dynamic programming algorithm aligns greater rooted subtrees using the alignments of smaller rooted subtrees. We assume that we already know the score of the optimal alignments A_{t_r,t_x}, A_{t_r,t_y}, A_{t_u,t_s}, A_{t_v,t_s}, A_{t_u,t_x}, A_{t_u,t_y}, A_{t_v,t_x} and A_{t_v,t_y} when aligning subtrees t_r and t_s, where u and v are the children of r and x and y are the children of s. Should one of the nodes have only one child, the dynamic programming reduces the problem of aligning t_r and t_s to less subproblems. We assume that the algorithm also knows the score of the optimal alignments of t_r to the empty tree and the score of the optimal alignment of t_s to the empty tree. Let the labelling of r be a and the labelling of s be b. We have to consider constant many subproblems for obtaining the score of the optimal alignment of t_r and t_s. If one of the tree is aligned to one of the children's subtree of the other tree, then the other child and the root of the other tree is labelled together with '$-$'. If character of r is co-labelled with the character of s, then the children nodes are aligned together, as well. The last situation is the case when the roots are not aligned in A_{t_r,t_s} but one of the roots is the root of A_{t_r,t_s} and the other root is its only child. The children might or might not be aligned together, this is five possible cases altogether.

Since the number of rooted subtrees equals to the number of nodes of the tree, the optimal alignment can be obtained in $\Theta(|F||G|)$ time, where $|F|$ and $|G|$ are the number of nodes in F and G.

21.4.2. Co-emission probability of two HMMs

Let M_1 and M_2 be Hidden Markov Models. The co-emission probability of the two models is

$$C(M_1, M_2) = \sum_s \Pr_{M_1}\{s\} \Pr_{M_2}\{s\} \ , \tag{21.58}$$

where the summation is over all possible sequences and $\Pr_M\{s\}$ is the probability that model M emitted sequence s. The probability that path p emitted sequence s is denoted by $e(p) = s$, a path from the $START$ state till the x state is denoted by $[x]$. Since state x can be reached on several paths, this definition is not well-defined, however, this will not cause a problem later on. Since the coemission probability is the sum of the product of emission of paths,

$$\begin{aligned} C(M_1, M_2) &= \sum_s \left(\sum_{p_1 \in M_1, e(p_1)=s} \Pr_{M_1}\{p_1\} \right) \left(\sum_{p_2 \in M_2, e(p_2)=s} \Pr_{M_2}\{p_2\} \right) = \\ &= \sum_{p_1 \in M_1, p_2 \in M_2, e(p_1)=e(p_2)} \Pr_{M_1}\{p_1\} \Pr_{M_2}\{p_2\} \ . \end{aligned} \tag{21.59}$$

Let $\bar{p}_1$ denote the path that can be obtained with removing the last state from p_1, and let x_1 be the state before END_1 in path p_1. (We define similarly $\bar{p}_2$ and x_2.) Hence

$$\begin{aligned} C(M_1, M_2) &= \sum_{p_1 \in M_1, p_2 \in M_2, e(p_1)=e(p_2)} m_{x_1,END_1} m_{x_2,END_2} \Pr_{M_1}\{\bar{p}_1\} \Pr_{M_2}\{\bar{p}_2\} = \\ &= \sum_{x_1, x_2} m_{x_1,END_1} m_{x_2,END_2} C(x_1, x_2) \ , \end{aligned} \tag{21.60}$$

where $m_{x,END}$ is the probability of jumping to $END$from x, and

$$C(x_1, x_2) = \sum_{[x_1] \in M_1, [x_2] \in M_2, e([x_1])=e([x_2])} \Pr_{M_1}\{[x_1]\} \Pr_{M_2}\{[x_2]\} \, . \tag{21.61}$$

$C(x_1, x_2)$ can be also obtained with this equation:

$$C(x_1, x_2) = \sum_{y_1, y_2} m_{y_1,x_1} m_{y_2,x_2} C(y_1, y_2) \sum_{\sigma \in \Sigma} \Pr\{\sigma | x_1\} \Pr\{\sigma | x_2\} \ , \tag{21.62}$$

where $\Pr\{\sigma|x_i\}$ is the probability that x_i emitted σ. Equation 21.62 defines a linear equation system for all pairs of emitting states x_1 and x_2 . The initial conditions are:

$$C(START_1, START_2) = 1, \tag{21.63}$$
$$C(START_1, x_2) = 0, \quad x_2 \neq START_2 , \tag{21.64}$$
$$C(x_1, START_2) = 0, \quad x_1 \neq START_1 . \tag{21.65}$$

Unlike the case of traditional dynamic programming, we do not fill in a dynamic programming table, but solve a linear equation system defined by Equation 21.62. Hence, the coemission probability can be calculated in $O\left((n_1n_2)^3\right)$ time, where n_i and M_i are the number of emitting states of the models.

Exercises

21.4-1 Give a dynamic programming algorithm for the local similarities of two trees. This is the score of the most similar subtrees of the two trees. Here subtrees are any consecutive parts of the tree.

21.4-2 Ordered trees are rooted trees in which the children of a node are ordered. The ordered alignment of two ordered trees preserve the orderings in the aligned trees. Give an algorithm that obtains the optimal ordered alignment of two ordered trees and has running time being polynomial with both the maximum number of children and number of nodes.

21.4-3 Consider the infinite Euclidean space whose coordinates are the possible sequences. Each Hidden Markov model is a vector in this space the coordinates of the vector are the emission probabilities of the corresponding sequences. Obtain the angle between two HMMs in this space.

21.4-4 Give an algorithm that calculates the generating function of the length of the emitted sequences of an HMM, that is

$$\sum_{i=0}^{\infty} p_i \xi^i$$

where p_i is the probability that the Markov model emitted a sequence with length i.

21.4-5 Give an algorithm that calculates the generating function of the length of the emitted sequences of a pair-HMM, that is

$$\sum_{i=0}^{\infty}\sum_{j=0}^{\infty} p_{i,j} \xi^i \eta^j$$

where $p_{i,j}$ is the probability that the first emitted sequence has length i, and the second emitted sequence has length j.

21.5. Distance based algorithms for constructing evolutionary trees

In this section, we shell introduce algorithms whose input is a set of objects and distances between objects. The distances might be obtained from pairwise alignments of sequences, however, the introduced algorithms work for any kind of distances. The leaves of the tree are the given objects, and the topology and the lengths of the edges are obtained from the distances. Every weighted tree defines a metric on the leaves of the tree, we define the distance between two leaves as the sum of the weights of edges on the path connecting them. The goodness of algorithms can be measured as the deviation between the input distances and the distances obtained on the tree.

We define two special metrics, the ultrametric and additive metric. The clustering algorithms generate a tree that is always ultrametric. We shell prove that clustering algorithms gives back the ultrametric if the input distances follow a ultrametric, namely, the tree obtained by a clustering algorithm defines exactly the input distances.

Similarly, the Neighbour Joining algorithm creates a tree that represents an additive metric, and whenever the input distances follow an additive metric, the generated tree gives back the input distances.

For both proves, we need the following lemma:

Lemma 21.3 *For any metric, there is at most one tree that represents it and has positive weights.*

Proof The proof is based on induction, the induction starts with three points. For three points, there is exactly one possible topology, a star-tree. Let the lengths of the edges connecting points i, j and k with the internal node of the star three be x, y and z, respectively. The lengths of the edges defined by the

$$x + y = d_{i,j} \tag{21.66}$$
$$x + z = d_{i,k} \tag{21.67}$$
$$y + z = d_{k,l} \tag{21.68}$$

equation system, which has a unique solution since the determinant

$$\begin{vmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{vmatrix} \tag{21.69}$$

is not 0.

For $n > 3$ number of points, let us assume that there are two trees representing the same metric. We find a ***cherry motif*** on the first tree, with cherries i and j. A cherry motif is a motif with two leafes whose connecting path has exactly one internal node. Every tree contains at least two cherry motives, a path on the tree that has the maximal number of internal nodes has cherry motives at both ends.

If there is only one internal node on the path connecting i and j on the other tree, then the length of the corresponding edges in the two cherry motives must be the same, since for any additional point k, we must get the same subtree. We define a new metric by deleting points i and j, and adding a new point u. The distance between u and any point k is $d_{i,k} - d_{i,u}$, where $d_{i,u}$ is the length of the edge connecting i with the internal point in the cherry motif. If we delete nodes i and j, we get a tree that represent this metric and they are the same, according to the induction.

If the path between i and j contains more than one internal node on the other tree, then we find a contradiction. There is a u_1 point on the second tree for which $d_{i,u} \neq d_{i,u_1}$. Consider a k such that the path connecting i and k contains node u. From the first tree

$$d_{i,k} - d_{j,k} = d_{i,u} - d_{j,u} = 2d_{i,u} - d_{i,j} \ , \tag{21.70}$$

while on the second tree

$$d_{i,k} - d_{j,k} = d_{i,u_1} - d_{j,u_1} = 2d_{i,u_1} - d_{i,j} \ , \tag{21.71}$$

which contradicts that $d_{i,u} \neq d_{i,u_1}$. ■

21.5.1. Clustering algorithms

Definition 21.4 *A metric is* ***ultrametric*** *if for any three points, i, j and k*

$$d_{i,j} \leq \max\{d_{i,k}, d_{j,k}\} \tag{21.72}$$

It is easy to prove that the three distances between any three points are all equal or two of them equal and the third is smaller in any ultrametric.

Theorem 21.5 *If the metric on a finite set of points is ultrametric, then there is exactly one tree that represents it. Furthermore, this tree can be rooted such that the distance between a point and the root is the same for all points.*

Proof Based on the Lemma 21.3, it is enough to construct one ultrametric tree for any ultrametric. We represent ultrametric trees as ***dendrograms.*** in this representation, the horizontal edges has length zero. For an example dendrogram, see Figure 21.2. The proof is based on the induction on the number of leaves. Obviously, we can construct a dendrogram for two leaves. After constructing the dendrogram for n leaves, we add leaf $n+1$ to the dendrogram in the following way. We find a leaf i in the dendrogram, for which $d_{i,n+1}$ is minimal. Then we walk up on the dendrogram till we reach the $d_{i,n+1}/2$ distance (we might go upper than the root). The node i is connected to the dendrogram at this point, see Figure 21.3. This dendrogram represents properly the distances between leaf $n+1$ and any other leaf. Indeed, if leaf i that is below the new internal node that bonnets leaf $n+1$, then $d_{i,i'} \leq d_{i,n+1}$ and from the ultrametric property and the minimality of $d_{i,n+1}$ it follows that $d_{i,n+1} = d_{i',n+1}$. On the other hand, if leaf j is not below the new internal point joining leaf $n+1$, then $d_{i,j} > d_{i,n+1}$, and from the ultrametric property it comes that $d_{j,n+1} = d_{i,j}$. ■

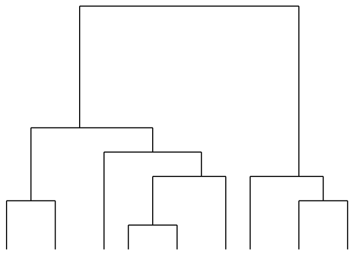

Figure 21.2 A dendrogram.

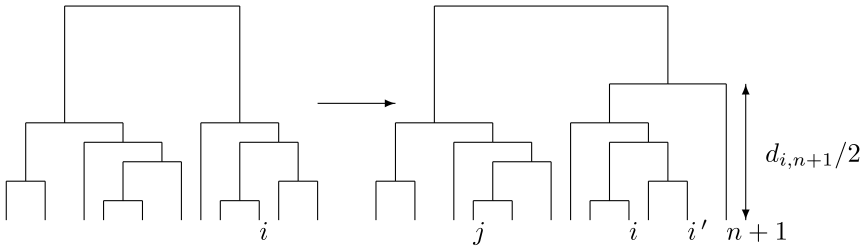

Figure 21.3 Connecting leaf $n+1$ to the dendrogram.

It is easy to see that the construction in the proof needs $O(n^2)$ running time, where n is the number of objects. We shell give another algorithm that finds the pair of objects i and j for which $d_{i,j}$ is minimal. From the ultrametric property, for any $k \neq i, j$, $d_{i,k} = d_{j,k}(\geq d_{i,j})$, hence we can replace the pair of objects i and j to a new object, and the distance between this new object and any other object k is well defined, it is $d_{i,k} = d_{j,k}$. The objects i and j are connected at height $d_{i,j}/2$, and we treat this sub-dendrogram as a single object. We continue the iteration till we have a single object. This algorithm is slower than the previous algorithm, however, this is the basis of the clustering algorithms. The clustering algorithms create a dendrogram even if the input distances do not follow a ultrametric. On the other hand, if the input distances follow a ultrametric, then most of the clustering algorithms gives back this ultrametric.

As we mentioned, the clustering algorithms find the object pair i and j for which $d_{i,j}$ is minimal. The differences come from how the algorithms define the distance between the new object replacing the pair of objects i and j and any other object. If the new object is denoted by u, then the introduced clustering methods define $d_{u,k}$ in the following way:

- Single link: $d_{u,k} = \min\{d_{i,k}, d_{j,k}\}$.
- Complete link: $d_{u,k} = \max\{d_{i,k}, d_{j,k}\}$.
- UPGMA: the new distance is the arithmetic mean of the distances between the elements in u and k : $d_{u,k} = \frac{d_{i,k} \times |i| + d_{j,k} \times |j|}{|i|+|j|}$, where $|i|$ and $|j|$ are the number of

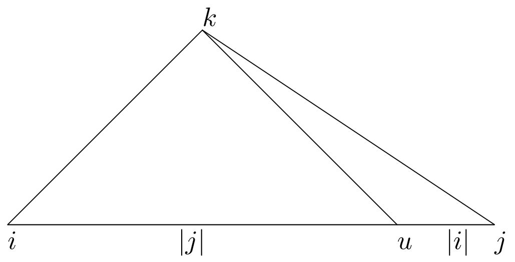

Figure 21.4 Calculating $d_{u,k}$ according to the CENTROID method.

elements in i and j.

- SINGLE AVERAGE: $d_{u,k} = \frac{d_{i,k}+d_{j,k}}{2}$.

- CENTROID: This method is used when the objects can be embedded into a Euclidean space. Then the distance between two objects can be defined as the distance between the centroids of the elements of the objects. It is not necessary to use the coordinates of the Euclidean space since the distance $d_{u,k}$ in question is the distance between point k and the point intersecting the ij edge in $|j| : |i|$ proportion in the triangle obtained by points i, j és k (see Figure 21.4). This length can be calculated using only $d_{i,j}$, $d_{i,k}$ and $d_{j,k}$. Hence the algorithm can be used even if the objects cannot be embedded into a Euclidean space.

- MEDIAN: The centroid of u is the centroid of the centroids of i and j. This method is related to the centroid method as the single average is related to the UPGMA method. It is again not necessary to know the coordinates of the elements, hence this method can be applied to distances that cannot be embedded into a Euclidean space.

It is easy to show that the first four method gives the dendrogram of the input distances whenever the input distances follow a ultrametric since $d_{i,k} = d_{j,k}$ in this case. However, the CENTROID and MEDIAN methods do not give the corresponding dendrogram for ultrametric input distances, since $d_{u,k}$ will be smaller than $d_{i,k}$ (which equals to $d_{j,k}$).

The central problem of the clustering algorithms is that they give a dendrogram that might not be biologically correct. Indeed, the evolutionary tree of biological sequences can be a dendrogram only if the *molecular clock* hypothesis holds. The molecular clock hypothesis says that the sequences evolve with the same tempo at each branches of the tree, namely they collect the same number of mutations at a given time span. However, this is usually not true. Therefore biologists want an algorithm that give a ultrametric tree only if the input distances follow a ultrametric. The most popular such method is the ***Neighbour-Joining*** algorithm.

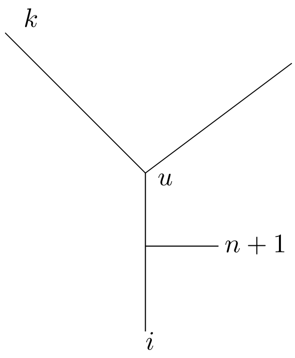

Figure 21.5 Connecting leaf $n+1$ for constructing an additive tree.

21.5.2. Neighbour joining

Definition 21.6 *A metric is called* ***additive*** *or* ***four-point metric,*** *if for any four points i, j, k and l*

$$d_{i,j} + d_{k,l} \leq \max\{d_{i,k} + d_{j,l},\, d_{i,l} + d_{j,k}\} \tag{21.73}$$

Theorem 21.7 *If a metric is additive on a finite set of objects, then there is exactly one tree that represents it.*

Proof Due to Lemma 21.3, there is at most one such tree, therefore it is enough to construct it. First we give the construction then we prove its goodness.

For three objects we can construct a tree according to (21.66)–(21.68). Assume that we constructed the tree for $n \geq 3$ objects, and we want to add leaf $n+1$ to the tree. First we find the topology and then we give the length of the new edge. For obtaining the new topology we start with any leaf i, and let denote u the neighbour of leaf i. There are at least two other edges going out from u, we find two leaves on the paths starting with these two outgoing edges, let k and l denote these leaves, see Figure 21.5. The leaf is connected to the edges between i and u if

$$d_{i,n+1} + d_{k,l} < d_{i,k} + d_{n+1,l} \tag{21.74}$$

Using similar inequalities, we can decide if leaf $n+1$ is before u looking from k or looking from l. If the degree of u is greater than 3, then we find leaves l' on the other paths and we do the same investigations for i, $n+1$, k and l' points. >From the additive property, it follows that inequality can hold at most for one cases. If it holds for i, then we connect leaf $n+1$ to the edge connecting u and i. If the inequality holds for another case, then we derive the maximal subtree of the tree that contains u as a leaf and also contains the leaf for which the inequality holds. We define $d_{u,n+1}$ as $d_{i,n+1} - d_{i,u}$, then renaming u to i we continue the searching for the connection place of leaf $n+1$. If we get equality for all outgoing edges of u, then we connect leaf $n+1$ to u.

After finding the topology, we obtain the length of the new edge. Leaf $n+1$ is

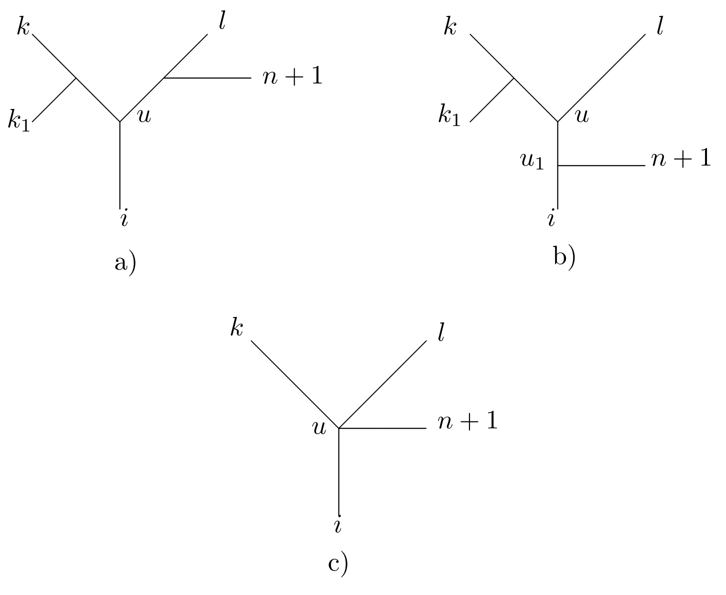

Figure 21.6 Some tree topologies for proving Theorem 21.7.

connected to the edge between i and u, let u_1 denote the new internal point, see Figure 21.6/b. We define $d_{u,n+1}$ as $d_{l,n+1} - d_{l,u}$. then the distances d_{u,u_1}, d_{i,u_1}, and $d_{u_1,n+1}$ can be calculated using (21.66)–(21.68). If the leaf $n+1$ is connected to u, then $d_{u,n+1} = d_{i,n+1} - d_{i,u}$.

Now we prove the correctness of the construction. First we show that $d_{u,n+1}$ is well-defined, namely, for all node j that is not in the new subtree containing leaves $n+1$ and u, $d_{j,n+1} - d_{j,u} = d_{i,n+1} - d_{i,u}$. If the new subtree contains l then for $j = k$ that was used to find the place of leaf $n+1$ will obviously hold (see Figure 21.6/a). Due to the additive metric property and the place of leaf $n+1$

$$d_{k,n+1} + d_{i,l} = d_{i,n+1} + d_{k,l} \ . \tag{21.75}$$

Using inequalities $d_{i,l} = d_{i,u} + d_{u,l}$ és a $d_{k,l} = d_{k,u} + d_{u,l}$, it follows that

$$d_{k,n+1} - d_{k,u} = d_{i,n+1} - d_{i,u}. \tag{21.76}$$

Similarly for all leaves k_1 that are not separated from k by u, it holds that

$$d_{k_1,n+1} + d_{i,l} = d_{i,n+1} + d_{k_1,l} \tag{21.77}$$

It is due to the additive metric and the inequality

$$d_{k,k_1} + d_{l,n+1} < d_{k,n+1} + d_{k_1,l} \tag{21.78}$$

this later inequality comes from these inequalities

$$\begin{aligned} d_{k,k_1} + d_{i,l} &< d_{k_1,l} + d_{k,i} & (21.79)\\ d_{l,n+1} + d_{k,i} &< d_{i,l} + d_{k,n+1} & (21.80)\end{aligned}$$

If the degree of u is greater than 3, then similar inequalities hold.

Due to the way of calculating the new edge lengths, $d_{i,n+1}$ is represented properly on the new tree, hence $d_{j,n+1}$ is represented properly for all j that is separated from leaf $n+1$ by i. Note that i might be an earlier u.

If leaf $n+1$ is connected to the edge between i and u (Figure 21.6/b), then due to the definition $d_{u,n+1}$, $d_{l,n+1}$ is represented properly. From the equation

$$d_{k,n+1} + d_{i,l} = d_{k,i} + d_{l,n+1} \tag{21.81}$$

it follows that

$$d_{k,n+1} = d_{k,u} + d_{u,n+1} \ , \tag{21.82}$$

hence $d_{k,n+1}$ is represented properly. It can be similarly shown that for all points j that are separated from $n+1$ by u, the $d_{j,n+1}$ is represented properly on the tree.

If leaf $n+1$ is connected to node u (Figure 21.6/c), then from the equations

$$d_{i,n+1} + d_{k,l} = d_{k,i} + d_{l,n+1} = d_{k,n+1} + d_{j,i} \tag{21.83}$$

it comes that both $d_{k,n+1}$ and $d_{l,n+1}$ are represents properly on the new tree, and with similar reasoning, it is easy to show that actually for all nodes j that is separated from $n+1$ by u, $d_{j,n+1}$ is represented properly on the tree.

Hence we construct a tree containing leaf $n+1$ from the tree containing the first n leaves, thus proving Theorem 21.7. ■

It is easy to show that the above algorithm that constructs the tree representing an additive metric takes $O(n^2)$ running time. However, it works only if the input distances follow an additive metric, other wise inequality (21.74) might hold several times, hence we cannot decide where to join leaf $n+1$ to. We shell introduce an algorithm that has $\Theta(n^3)$ running time and gives back the additive tree whenever the input distances follow an additive metric, moreover it generates an additive tree that approximates the input distances if those are not follow an additive metric.

The **Neighbour-Joining** algorithm works in the following way: Given a set with n points and a distance function d on the points. First we calculate the for each point i the sum of the distances from the other points:

$$v_i = \sum_{j=1}^{n} d_{i,j} \ . \tag{21.84}$$

Then we find the pair of points for which

$$s_{i,j} = (n-2)d_{i,j} - v_i - v_j \tag{21.85}$$

is minimal. The length of the edges from points i and j to the new point u are

$$e_{i,u} = \frac{d_{i,j}}{2} - \frac{v_i - v_j}{2n-4} \tag{21.86}$$

and

$$e_{j,u} = \frac{d_{i,j}}{2} - e_{i,u} \tag{21.87}$$

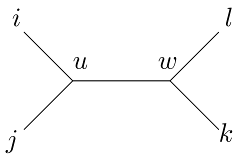

Figure 21.7 The configuration of nodes i, j, k and l if i and j follows a cherry motif.

Then we recalculate distances. We drop points i and j, and add point u. The distance between u and any other point k is defined as

$$d_{k,u} = \frac{d_{k,i} + d_{k,j} - d_{i,j}}{2} \; . \tag{21.88}$$

Theorem 21.8 *If d follows an additive metric, then the* Neighbour-Joining *algorithm generates a tree that gives back d.*

Proof From Theorem 21.7 there is exactly one tree that represents the distances. It is enough to prove that the Neighbour-Joining algorithm always pick a cherry motif on this tree, since a straightforward calculation shows that in this case the calculated edge lengths are proper.

First we prove if i and j follows a cherry motif then for all k, $s_{i,j} < s_{i,k}$ és $s_{i,j} < s_{k,j}$. Indeed, rearranging s, we have to prove that

$$\sum_{l \neq i,j} (d_{i,j} - d_{i,l} - d_{j,l}) - 2d_{i,j} - \sum_{m \neq j,k} (d_{j,k} - d_{j,m} - d_{k,m}) + 2d_{j,k} < 0 \tag{21.89}$$

If $l = m \neq i, j, k$, then we get that

$$(d_{i,j} - d_{i,l} - d_{j,l}) - d_{j,k} + d_{j,l} + d_{k,l} = 2d_{w,l} - 2d_{u,l} < 0 \; , \tag{21.90}$$

(see also Figure 21.7). $2d_{j,k} - 2d_{i,j}$ and the cases $l = k$ and $m = i$ inside the sums cancel each other, hence we prove that the (21.89) inequality holds.

Now we prove the Theorem 21.8 in an indirect way. Suppose that i and j does not follow a cherry motif, however, $s_{i,j}$ is minimal. From the previous lemma, neither i nor j are in a cherry motif with other leaves. We find a cherry motif with leaves k and l and internal node w. Let v denote the last common node of paths going from w to i and to j. Since $s_{i,j}$ is minimal,

$$s_{k,l} - s_{i,j} > 0 \; . \tag{21.91}$$

Rearranging this we get that

$$\sum_{m_1 \neq k,l} (d_{k,l} - d_{m_1,k} - d_{m_1,l}) - 2d_{k,l} - \sum_{m_2 \neq i,j} (d_{i,j} - d_{m_2,i} - d_{m_2,k}) + 2d_{i,j} > 0 \; . \tag{21.92}$$

$2d_{i,j} - 2d_{k,l}$ and cases $m_1 = k$, $m_1 = l$, $m_2 = i$ and $m_2 = j$ inside the sum cancel

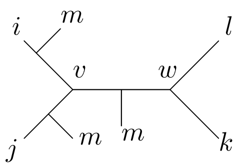

Figure 21.8 The possible places for node m on the tree.

each other. For the other $m = m_1 = m_2 \neq i, j, k, l$, the left hand side is

$$d_{k,l} - d_{m,k} - d_{m,l} - d_{i,j} + d_{m,i} + d_{m,k} \ . \tag{21.93}$$

If m joins to the tree via the path connecting i and j, then the expression 21.93 will be always negative, see also Figure 21.8. Let these cases be called I. class cases. If m joins to the tree via the path between v and w, then the expression 21.93 might be positive. Let these cases called II. class cases. To avoid contradiction, the sum of absolute values from I. class cases must be less than the sum from the II. class cases.

There is another v' node on the path connecting i and j, and we can find a cherry motif after node v' with leaves k' and l' and internal node w'. Here again the II. class cases have to be more than the I. class cases, but this contradict to the situation with the first cherry motif. Hence i and j form a cherry motif and we prove Theorem 21.8. ■

Exercises

21.5-1 Show that in a ultrametric, three distances coming from three points are all equal or two of them equal and the third is smaller. Prove the similar claim for the three sum of distances coming from four points in an additive metric.
21.5-2 Show that a ultrametric is always an additive metric.
21.5-3 Give an example for a metric that is not additive.
21.5-4 Is it true that all additive metric is a Euclidean metric?
21.5-5 Give the formula that calculates $d_{u,k}$ from $d_{i,j}$, $d_{i,k}$ and $d_{j,k}$ for the centroid method.
21.5-6 Give algorithms that decide in $O(n^2)$ whether or not a metric is

- additive
- ultrametric

(n is the number of points.)

21.6. Miscellaneous topics

In this section, we cover topics that are usually not mentioned in bioinformatics books. We only mention the main results in a nutshell and do not prove theorems.

21.6.1. Genome rearrangement

The genome of an organism consists of several genes. For each gene, only one strand of the double stranded DNA contains meaningful information, the other strand is the reverse complement. Since the DNA is chemically oriented, we can talk about the direction of a gene. If each gene has one copy in the genome then we can describe the order and directions of genes as a signed permutation, where the signs give the directions of genes.

Given two genomes with the same gene content, represented as a signed permutation then the problem is to give the minimal series of mutations that transform one genome into another. We consider three types of mutations:

- **Reversal** A reversal acts on a consecutive part of the signed permutation. It reverse the order of genes on the given part as well as the signs of the genes.
- **Transposition** A transposition swaps two consecutive block of genes.
- **Reverted transposition** It swaps two consecutive blocks and one of the blocks is reverted. As for reversals, the signs in the reverted block also change.

If we assume that only mutations happened, then we can give an $O(n^2)$ running time algorithm that obtains a shortest series of mutations transforming one genome into another, where n is the number of genes.

If we consider other types of mutations, then the complexity of problems is unknown. For transpositions, the best approximation is an 1.375 approximation [14], if we consider all possible types of mutations, then the best approximation is a 2-approximation [28]. For a wide range of and biologically meaningful weights, the weighted sorting problem for all types of mutations has a 1.5-approximation [6].

If we do not know the signs, then the problem is proved to be NP-complete [8]. Similarly, the optimal reversal median problem even for three genomes and signed permutations is NP-complete [?]. The optimal reversal median is a genome that minimises the sum of distances from a set of genomes.

Below we describe the Hannenhalli-Pevzner theorem for the reversal distance of two genomes. Instead of transforming permutation π_1 into π_2, we transform $\pi_2^{-1}\pi_1$ into the identical permutation. Based on elementary group theory, it is easy to show that the two problems are equivalent. We assume that we already calculated $\pi_2^{-1}\pi_1$, and we will denote it simply by π.

We transform an n long signed permutation into a $2n$ long unsigned permutation by replacing $+i$ to $2i-1, 2i$ and $-i$ to $2i, 2i-1$. Additionally, we frame the unsigned permutation into 0 and $2n+1$. The vertexes of the so-called graph of desire and reality are the numbers of the unsigned permutation together with 0 and $2n+1$. Starting with 0, we connect every other number in the graph, these are the reality edges. Starting also with 0, we connect $2i$ with $2i+1$ with an arc, these are the desire edges. An example graph can be seen on Figure 21.9. Since each vertex in the graph of desire and reality has a degree of two, the graph can be unequivocally decomposed into cycles. We call a cycle a directed cycle if on a walk on the cycle, we go at least once from left to right on a reality cycle and we go at least once from right to left on a reality cycle. Other cycles are unoriented cycles.

The span of a desire edge is the interval between its left and right vertexes. Two

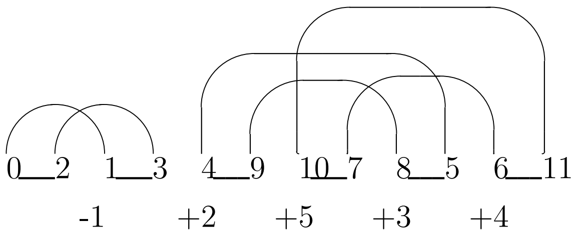

Figure 21.9 Representation of the -1, $+2$, $+5$, $+3$, $+4$ signed permutation with an unsigned permutation, and its graph of desire and reality.

cycles overlap if there are two reality edges in the two cycles whose spans intersect. The vertexes of the overlap graph of a signed permutation are the cycles in its graph of desire and reality, two nodes are connected if the two cycles overlap. The overlap graph can be decomposed into components. A component is directed if it contains a directed cycle, otherwise it is unoriented. The span of a component is the interval between its leftmost and rightmost nodes in the graph of desire and reality. An unoriented component is a hurdle if its span does not contain any unoriented component or it contains all unoriented component. Other components are called protected non-hurdles.

A super-hurdle is hurdle for which it is true that if we delete this hurdle then one of the protected non-hurdles becomes a hurdle. A fortress is a permutation in which all hurdles are super-hurdles and their number is odd.

The Hannenhalli-Pevzner theorem is the following:

Theorem 21.9 *Given a signed permutation π. The minimum number of mutations sorting this permutation into the identical permutation is*

$$n + 1 - c_\pi + h_\pi + f_\pi \qquad (21.94)$$

where n is the length of the permutation, c_π is the number of cycles, h_π is the number of hurdles, and $f_\pi = 1$ if the permutation is a fortress, otherwise 0

The proof of the theorem can be found in the book due to Pevzner.

The reversal distance was calculated in $\Theta(n)$ time by Bader et al.. It is very easy to obtain c_π in $\Theta(n)$ time. The hard part is to calculate h_π and f_π. The source of the problem is that the overlap graph might contain $\Omega(n^2)$ edges. Therefore the fast algorithm does not obtain the entire overlap graph, only a spanning tree on each component of it.

21.6.2. Shotgun sequencing

A genome of an organism usually contain significantly more than one million nucleic acids. Using a special biochemical technology, the order of nucleic acids can be obtained, however, the uncertainty grows with the length of the DNA, and becomes absolutely unreliable after about 500 nucleic acids.

A possible solution for overcoming this problem is the following: several copies are made from the original DNA sequence and they are fragmented into small parts that can be sequenced in the above described way. Then the original sequence must be reconstructed from its overlapping fragments. This technique is called ***shotgun sequencing.***

The mathematical definition of the problem is that we want to find the shortest common super-sequence of a set of sequences. Sequence B is a super-sequence of A if A is subsequence of B. Recall that a subsequence is not necessarily a consecutive part of the sequence. Maier proved that the shortest common super-sequence problem is NP-complete is the size of the alphabet is at least 5 and conjectured that it is the case if the size is at least 3. Later on it has been proved that the problem is NP-complete for all non-trivial alphabet [64].

Similar problem is the shortest common super-string problem, that is also an NP-complete problem [23]. This later has biological relevance, since we are looking for overlapping substrings. Several approximation algorithms have been published for the shortest common super-string problem. A greedy algorithm finds for each pair of strings the maximal possible overlap, then it tries to find a shortest common super-string by merging the overlapping strings in a greedy way [74]. The running time of the algorithm is $O(Nm)$, where N is the number of sequences and m is the total length of the sequences. This greedy method is proved to be a 4-approximation [**?**]. A modified version being a 3-approximation also exist, and the conjecture is that the modified version is a 2-approximation [**?**].

The sequencing of DNA is not perfect, insertions, deletions and substitutions might happen during sequencing. Therefore Jiang and Li suggested the shortest k-approximative common super-string problem [**?**]. Kececioglu and Myers worked out a software package including several heuristic algorithm for the problem [**?**]. Later on Myers worked for Celera, which played a very important role in sequencing the human genome. A review paper on the topic can be found in [82].

Exercises

21.6-1 Show that a fortress contains at least three super-hurdle.

21.6-2 At least how long is a fortress?

Problems

21-1 Concave Smith–Waterman

G ive the Smith–Waterman-algorithm for concave gap penalty.

21-2 Concave Spouge

G ive Spouge-algorithm for concave gap penalty.

21-3 Serving at a petrol station

T here are two rows at a petrol station. Each car needs either petrol or diesel oil. At most two cars can be served at the same time, but only if they need different type of fuel, and the two cars are the first ones in the two rows or the first two in the same row. The serving time is the same not depending on whether two cars are being served or only one. Give a pair-HMM for which the Viterbi-algorithmqprindexViterbi-algorithm provides a shortest serving scenario.

21-4 Moments of an HMM

G iven an HMM and a sequence. Obtain the mean, variance, kth moment of the probabilities of paths emitting the given sequence.

21-5 Moments of a SCFG

G iven a SCFG and a sequence. Obtain the mean, variance, kth moment of the probabilities of derivations of the given sequence.

21-6 Co-emission probability of two HMMs

C an this probability be calculated in $O((n_1n_2)^2)$ time where n_1 and n_2 are the number of states in the HMMs?

21-7 Sorting reversals

A sorting reversal is a reversal that decreases the reversal distance of a signed permutation. How can a sorting reversal change the number of cycles and hurdles?

Chapter Notes

The first dynamic programming algorithm for aligning biological sequences was given by Needleman and Wunch in 1970 [56]. Though the concave gap penalty function is biologically more relevant, the affine gap penalty has been the standard soring scheme for aligning biological sequences. For example, one of the most popular multiple alignment program, CLUSTAL-W uses affine gap penalty and iterative sequence alignment [75]. The edit distance of two strings can calculated faster than $\Theta(l^2)$ time, that is the famous ”Four Russians’ speedup"qindexFour Russians speedup [4]. The running time of the algorithm is $O(n^2/\log(n))$, however, it has such a big constant in the running time that it is not worth using it for sequence lengths appear in biological applications. The longest common subsequence problem can be solved using a dynamic programming algorithm similar to the dynamic programming algorithm for aligning sequences. Unlike that algorithm, the algorithm of Hunt and Szymanski creates a graph whose points are the characters of the sequences A and B, and a_i is connected to b_j iff $a_i = b_j$. Using this graph, the longest common subsequence can be find in $\Theta((r + n)\log(n))$ time, where r is the number of edges in the graph and n is the number of nodes [35]. Although the running time of this algorithm is $O(n^2 \lg(n))$, since the number of edges might be $O(n^2)$, in many cases the number of edges is only $O(n)$, and in this cases the running time is only $O(n \lg(n))$. A very sophisticated version of the corner-cutting method is the diagonal extension technique, which fills in the dynamic programming table by diagonals and does not need a test value. An example for such an algorithm is the algorithm of Wu at al. [83]. the `diff` command in the Unix operating system is also based on diagonal extension [51], having a running time $O(n + m + d_e^2)$, where n and m are the lengths of the sequences and d_e is the edit distance between the two sequences. The Knuth-Morris-Pratt string-searching algorithm searches a small pattern P in a long string M. Its running time is $\Theta(p+m)$, where p and m are the length of the sequences [40]. Landau and Vishkin modified this algorithm such that the modified version can find a pattern in M that differs at most in k position [42]. The running time of the algorithm is $\Theta(k(p \log(p) + m))$, the memory requirement is $\Theta(k(p + m))$. Although dynamic programming algorithms are the most frequently used techniques for aligning sequences, it is also possible to attack the problem with integer linear programming. KececiogluqnevindexKececioglu, John D. and his colleges gave the first linear programming algorithm for aligning sequences [37]. Their method has been extended to arbitrary gap penalty functions [3]. LanciaqnevindexLancia, G. wrote a review paper on the topic [41] and Pachter and Sturmfels showed the relationship between the dynamic programming and integer linear programming approach in their book *Algebraic Statistics for Computational Biology* [58]. The structural alignment considers only the 3D structure of sequences. The optimal structural alignment problem is to find an alignment where we penalise gaps, however, the aligned characters scored not by their similarity but by how close their are in the superposed 3D structures. Several algorithms have been developed for the problem, one of them is the combinatorial extension (CE) algorithm [70]. For a given topology it is possible to find the Maximum Likelihood labeling [63]. This algorithm has been integrated into PAML, which is one of the most popular software package for phylogenetic analysis (http://abacus.gene.ucl.ac.uk/software/paml.html). The Maximum Likelihood tree problem is to find for a substitution model and a set of sequences the tree topology and edge lengths for which the likelihood is maximal. Surprisingly, it has only recently been proved that the problem is NP-complete [10, 67]. The similar problem, the Ancestral Maximum Likelihood problem has been showed to be NP-complete also only recently [1]. The AML problem is to find the tree topology, edge lengths and labellings for which the likelihood of a set of sequences is maximal in a given substitution model. The two most popular sequence alignment algorithms based on HMMs are the SAM [34] and the HMMER (http://hmmer.wustl.edu/) packages. An example for HMM for genome annotation is the work of Pedersen and Hein [60]. Comparative genome annotation can be done with pair-HMMs like the DoubleScan [49], (http://www.sanger.ac.uk/Software/analysis/doublescan/) and the Projector [50], (http://www.sanger.ac.uk/Software/analysis/projector/) programs. Goldman, Thorne and Jones were the first who published an HMM in which the emission probabilities are calculated from evolutionary informations [25]. It was used for protein secondary structure prediction. The HMM emits alignment columns, the emission probabilities can be calculated with the Felsenstein algorithm. The Knudsen-Hein grammar is used in the PFold program, which is for predicting RNA secondary structures [39]. This SCFG generates RNA multiple alignments, where the terminal symbols are alignment columns. The derivation probabilities can be calculated with the Felsenstein algorithm, the corresponding substitution model is a single nucleotide or a dinucleotide model, according to the derivation rules. The running time of the Forward algorithm grows squarely with the number of states in the HMM. However, this is not always the fastest algorithm. For a biologically important HMM, it is possible to reduce the $\Theta(5^n L^n)$ running time of the Forward algorithm to $\Theta(2^n L^n)$ with a more sophisticated algorithm [44, 45]. However, it is unknown whether or not similar acceleration exist for the Viterbi algorithm. The Zuker-Tinoco model [76] defines free energies for RNA secondary structure elements, and the free energy of an RNA structure is the sum of free energies of the elements. The Zuker-Sankoff algorithm calculates in $\Theta(l^4)$ time the minimum free energy structure, using $\Theta(l^2)$

memory, where l is the length of the RNA sequence. It is also possible to calculate the partition function of the Boltzmann distribution with the same running time and memory requirement [48]. For a special case of free energies, both the optimal structure and the partition function can be calculated in $\Theta(l^3)$ time, using still only $\Theta(l^2)$ memory [47]. Two base-pairings, $i \cdot j$ and $i' \cdot j'$ forms a pseudo-knot if $i < i' < j < j'$. Predicting the optimal RNA secondary structure in which arbitrary pseudo-knots are allowed is NP-complete [46]. For special types of pseudo-knots, polynomial running time algorithms exist [2, 46, 66, 77]. RNA secondary structures can be compared with aligning ordered forests [**?**]. Atteson gave a mathematical definition for the goodness of tree-constructing methods, and showed that the Neighbor-Joining algorithm is the best one for some definitions [5]. Elias and Lagergren recently published an improved algorithm for Neighbor-Joining that has only $O(n^2)$ running time [15]. There are three possible tree topologies for four species that are called quartets. If we know all the quartets of the tree, it is possible to reconstruct it. It is proved that it is enough to know only the short quartets of a tree that are the quartets of closely related species [16]. A genome might contain more than one DNA sequences, the DNA sequences are called chromosomes. A genome rearrangement might happen between chromosomes, too, such mutations are called translocations. Hannenhalli gave a $\Theta(n^3)$ running time algorithm for calculating the translocation and reversal distance [31]. Pisanti and Sagot generalised the problem and gave results for the translocation diameter [62]. The generalisation of sorting permutations is the problem of finding the minimum length generating word for an element of a group. The problem is known to be NP-complete [36]. Above the reversal distance and translocation distance problem, only for the block interchange distance exists a polynomial running time algorithm [11]. We mention that Bill Gates, the owner of Microsoft worked also on sorting permutations, actually, with prefix reversals [24].

Description of many algorithms of bioinformatics can be found in the book of Pevzner and Jones [61]. We wrote only about the most important topics of bioinformatics, and we did not cover several topics like recombination, pedigree analysis, character-based tree reconstructing methods, partial digesting, protein threading methods, DNA chip analysis, knowledge representation, biochemical pathways, scale-free networks, etc. We close the chapter with the words of Donald Knuth: ”It is hard for me to say confidently that, after fifty more years of explosive growth of computer science, there will still be a lot of fascinating unsolved problems at peoples’ fingertips, that it won’t be pretty much working on refinements of well-explored things. Maybe all of the simple stuff and the really great stuff has been discovered. It may not be true, but I can’t predict an unending growth. I can’t be as confident about computer science as I can about biology. Biology easily has 500 years of exciting problems to work on, it’s at that level."

Bibliography

[1] L. Addario-Berry, B. Chor, M. Hallett, J. Lagergren, A. Panconesi, T. Wareham. Ancestral maximum likelihood of phylogenetic trees is hard. *Lecture Notes in Bioinformatics*, 2812:202–215, 2003. 1011

[2] T. Akutsu. Dynamic programming algorithms for RNA secondary prediction with pseudoknots. *Discrete Applied Mathematics*, 104:45–62, 2000. 1012

[3] E. Althaus, A. Caprara, H. Lenhof, K. Reinert. Multiple sequence alignment with arbitrary gap costs: Computing an optimal solution using polyhedral combinatorics. *Bioinformatics*, 18:S4–S16, 2002. 1011

[4] V. Arlazanov. A. Dinic, M. Kronrod, I. Faradzev. On economic construction of the transitive closure of a directed graph. *Doklady Academii Nauk SSSR*, 194:487–488, 1970. 1010

[5] K. Atteson. The performance of the neighbor-joining method of phylogeny reconstruction. *Algorithmica*, 25(2/3):251–278, 1999. 1012

[6] M. Bader, E. Ohlebusch. Sorting by weighted reversals, transpositions and inverted transpsitions. *Lecture Notes in Bioinformatics*, 3909:563–577, 2006. 1007

[7] S. A. Benner, M. A. Cohen, H. G. H. Gonnet. Empirical and structural models for insertions and deletions in the divergent evolution of proteins. *Journal of Molecular Biology*, 229(4):1065–1082, 1993. 978, 979

[8] A. Caprara. Sorting permutations by reversals and eulerian cycle decompositions. *SIAM Journal on Discrete Mathematics*, 12(1):91–110, 1999. 1007

[9] H. Carillo, D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48:1073–1082, 1988. 986

[10] B. Chor, T. Tuller. Maximum likelihood of evolutionary trees: hardness and approximation. *Bioinformatics*, 21:i97–i106, 2005. 1011

[11] D. Christie. Sorting permutations by block-interchanges. *Information Processing Letters*, 60(4):165–169, 1996. 1012

[12] F. Corpet. Multiple sequence alignment with hierarchical clustering. *Nucleic Acids Research*, 16:10881–10890, 1988. 982

[13] M. O. Dayhoff, R. M. Schwartz, B. Orcutt. A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5:345–352, 1978. 979

[14] I. Elias, T. Hartman. A 1.375 approximation algorithm for sorting by transpositions. *Lecture Notes in Bioinformatics*, 3692:204–215, 2005. 1007

[15] I. Elias, J. Lagergren. Fast neighbor joining. *Lecture Notes in Computer Science*, 3580:1263–1274, 2005. 1012

[16] P. L. Erdős, M. Steel, L. Székely, T. Warnow. Local quartet splits of a binary tree infer all quartet splits via one dyadic inference rule. *Computers and Artificial Intelligence*, 16(2):217–227, 1997. 1012

[17] J. Felsenstein. Evolutionary trees from DNA sequences: a maximum likelihood approach. *Journal of Molecular Evolution*, 17:368–376, 1981. 988

[18] D. Feng, R. F. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25:351–360, 1987. 982

[19] J. W. Fickett. Fast optimal alignment. *Nucleid Acids Research*, 12:175–180, 1984. 985

[20] W. M. Fitch. Toward defining the course of evolution: minimum change for a specified tree topology. *Systematic Zoology*, 20:406–416, 1971. 989

[21] L. Foulds, R. L. Graham. The Steiner problem in phylogeny is NP-complete. *Advances in Applied Mathematics*, 3:43–49, 1982. 987

[22] Z. Galil, R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64:107–118, 1989. 978

[23] J. Gallant, D. Maier, J. Storer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20(1):50–58, 1980. 1009

[24] W. H. Gates, C. H. Papadimitriouhttp://www.cs.berkeley.edu/ christos/. Bounds for sorting by prefix reversals. *Discrete Mathematics*, 27:47–57, 1979. 1012

[25] N. Goldman, J. Thorne, D. Jones. Using evolutionary trees in protein secondary structure prediction and other comparative sequence analyses. *Journal of Molecular Biology*, 263(2):196–208, 1996. 1011

[26] H. G. H. Gonnet, M. A. Cohen, S. A. Benner. Exhaustive matching of the entire protein sequence database. *Science*, 256:1443–1445, 1992. 978

[27] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982. 978

[28] Q-P. Gu, S. Peng, H. Sudborough. A 2-approximation algorithm for genome rearrangements by reversals and transpositions. *Theoretical Computer Science*, 210(2):327–339, 1999. 1007

[29] D. M. Gusfield. Efficient methods for multiple sequence alignment with guaranteed error bounds. *Bulletin of Mathematical Biology*, 55:141–154, 1993. 983

[30] D. M. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997. 982, 986

[31] S. Hannenhalli. Polynomial-time algorithm for computing translocation distance between genomes. *Discrete Applied Mathematics*, 71:137–151, 1996. 1012

[32] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18:341–343, 1975. 984

[33] T. J. P. Hubbard, A. M. Lesk, A. Tramontano. Gathering them into the fold. *Nature Structural Biology*, 4:313, 1996. 982

[34] R. Hughey, A. Krogh. Hidden markov models for sequence analysis: Extension and analysis of the basic method. *CABIOS*, 12(2):95–107, 1996. 1011

[35] J. Hunt, T. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977. 1010

[36] M. R. Jerrum. The complexity of finding minimum-length generator sequences. *Theoretical Computer Science*, 36:265–289, 1986. 1012

[37] J. Kececioglu, H. Lenhof, K. Mehlhorn, P. Mutzel, K. Reinert, M. Vingron. A polyhedral approach to sequence alignment problems. *Discrete Applied Mathematics*, 104((1-3)):143–186, 2000. 1011

[38] B. Knudsen, J. Hein. RNA secondary structure prediction using stochastic context free grammars and evolutionary history. *Bioinformatics*, 15(6):446–454, 1999. 994

[39] B. Knudsen, J. Hein. Pfold: RNA secondary structure prediction using stochastic context-free grammars. *Nucleic Acids Researchs*, 31(13):3423–3428, 2003. 994, 1011

[40] D. E. Knuth, J. Morris, V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. 1010

[41] G. Lancia. Integer programming models for computational biology problems. *Journal of Computer Science and Technology*, 19(1):60–77, 2004. 1011

[42] G. Landau, U. Vishkin. Eficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986. 1010

[43] D. Lipman, S. J. Altshuland, J. Kecioglu. A tool for multiple sequence alignment. *Proc. Natl. Academy Science*, 86:4412–4415, 1989. 986

[44] G. Lunter, I. Miklós, A. Drummond, J. L. Jensen, J. Hein. Bayesian phylogenetic inference under a statistical indel model. *Lecture Notes in Bioinformatics*, 2812:228–244, 2003. 1011

[45] G. Lunter, I. Miklós, Y., J. Hein. An efficient algorithm for statistical multiple alignment on arbitrary phylogenetic trees. *Journal of Computational Biology*, 10(6):869–889, 2003. 1011

[46] R. Lyngso, C. N. S. Pedersen. RNA pseudoknot prediction in energy based models. *Journal of Computational Biology*, 7(3/4):409–428, 2000. 1012

[47] R. Lyngso, M. Zuker, C. Pedersen. Fast evaluation of internal loops in RNA secondary structure prediction. *Bioinformatics*, 15(6):440–445, 1999. 1012

[48] J. S. McCaskill. The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers*, 29:1105–1119, 1990. 1012

[49] I. M. Meyer, R. Durbin. Comparative ab initio prediction of gene structures using pair HMMs. *Bioinformatics*, 18(10):1309–1318, 2002. 1011

[50] I. M. Meyer, R. Durbin. Gene structure conservation aids similarity based gene prediction. *Nucleic Acids Research*, 32(2):776–783, 2004. 1011

[51] W. Miller, E. Myers. A file comparison program. *Software – Practice and Experience*, 15(11):1025–1040, 1985. 1010

[52] W. Miller, E. W. Myers. Sequence comparison with concave weighting functions. *Bulletin of Mathematical Biology*, 50:97–120, 1988. 978

[53] B. Morgenstern. DIALIGN 2: improvement of the segment-to-segment approach to multiple sequence alignment. *Bioinformatics*, 15:211–218, 1999. 983

[54] B. Morgenstern, A. Dress, T. Werner. Multiple DNA and protein sequence alignment based on segment-to-segment comparison. *Proc. Natl. Academy Science*, 93:12098–12103, 1996. 983

[55] B. Morgenstern, K. Frech, A. Dress, T. Werner. DIALIGN: Finding local similarities by multiple sequence alignment. *Bioinformatics*, 14:290–294, 1998. 983

[56] S. N. Needleman, C. Wunch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970. 1010

[57] R. Nussinov, G. Pieczenk, J. Griggs, D. Kleitman. Algorithms for loop matching. *SIAM Journal of Applied Mathematics*, 35:68–82, 1978. 994

[58] L. Pachter, B. Sturmfels (Eds.). *Algebraic Statistics for Computational Biology*. Cambridge University Press, 2005. 1011

[59] R. Page, E. Holmes. *Molecular Evolution: a Phylogenetic Approach*. Blackwell, 1998. 982

[60] J. S. Pedersen, J. Hein. Gene finding with a hidden Markov model of genome structure and evolution. *Bioinformatics*, 19(2):219–227, 2003. 1011

[61] P. A. Pevzner, N. Jones. *Bioinformatics Algorithms*. The MIT Press, 2004. 1012

[62] N. Pisanti, M. Sagot. Further thoughts on the syntenic distance between genomes. *Algorithmica*, 34(2):157–180, 2002. 1012

[63] T. Pupko, I. Peer, R. Shamir, D. Graur. A fast algorithm for joint reconstruction of ancestral amino acid sequences. *Molecular Biology and Evolution*, 17:890–896, 2000. 1011

[64] K. Räihä, E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16:187–198, 1981. 1009

[65] R. Ravi, J. D. Kececioglu. Approximation algorithms for multiple sequence alignment under a fixed evolutionary tree. *Discrete Applied Mathematics*, 88(1–3):355–366, 1998. 983

[66] E. Rivas, S. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *Journal of Molecular Biology*, 285(5):2053–2068, 1999. 1012

[67] S. Roch. A short proof that phylogenetic tree reconstruction by maximum likelihood is hard. *EEE Transactions on Computational Biology and Bioinformatics*, 3(1):92–94, 2006. 1011

[68] D. Sankoff. Minimal mutation trees of sequences. *SIAM Journal of Applied Mathematics*, 28:35–42, 1975. 982, 988

[69] P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal of Applied Mathematics*, 26:787–793, 1974. 975

[70] I. Shindyalov, P. Bourne. Protein structure alignment by incremental combinatorial extension (CE) of the optimal path. *Protein Engineering*, 11(9):739–747, 1998. 1011

[71] T. F. Smith, M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981. 981

[72] J. L. Spouge. Speeding up dynamic programming algorithms for finding optimal lattice paths. *SIAM Journal of Applied Mathematics*, 49:1552–1566, 1989. 985

[73] J. L. Spouge. Fast optimal alignment. *CABIOS*, 7:1–7, 1991. 985, 986

[74] J. Tarhio, J. E. Ukkonen A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science*, 57:131–145, 1988. 1009

[75] J. D. Thompson, D. G. Higgins, T. J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994. 978, 982, 1010

[76] I. Tinoco, O., Uhlenbeck M. Levine. Estimation of secondary structure in ribonucleic acids. *Nature*, 230:362–367, 1971. 1011

[77] Y. Uemura, A. Hasegawa, Y. Kobayashi, T. Yokomori. Tree adjoining grammars for RNA structure prediction. *Theoretical Computer Science*, 210:277–303, 1999. 1012

[78] E. Ukkonen. On approximate string matching. *Lecture Notes in Computer Science*, 158:487–495, 1984. 985

[79] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985. 985

[80] L. Wang, T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1:337–348, 1994. 982

[81] M. S. Waterman, T. F. Smithand, W. A. Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20:367–387, 1976. 977

[82] J. W. Weber, E. Myers. Human whole genome shotgun sequencing. *Genome Research*, 7:401–409, 1997. 1009

[83] S. Wu, E.. W. Myers, U. Manber, W. Miller. An $O(NP)$ sequence comparison algorithm. *Information Processing Letters*, 35(6):317–323, 1990. 1010

This bibliography is made by HBibTEX. First key of the sorting is the name of the authors (first author, second author etc.), second key is the year of publication, third key is the title of the document.

Underlying shows that the electronic version of the bibliography on the homepage of the book contains a link to the corresponding address.

Index

This index uses the following conventions. Numbers are alphabetised as if spelled out; for example, “2-3-4-tree" is indexed as if were “two-three-four-tree". When an entry refers to a place other than the main text, the page number is followed by a tag: *ex* for exercise, *exa* for example, *fig* for figure, *pr* for problem and *fn* for footnote.

The numbers of pages containing a definition are printed in *italic* font, e.g.

time complexity, *583*.

Name Index

This index uses the following conventions. If we know the full name of a cited person, then we print it. If the cited person is not living, and we know the correct data, then we print also the year of her/his birth and death.