

## Turning to Algorithmic Problems

### 2.1 Presentation of Algorithmic Problems

Having introduced a couple of core bioinformatics problems in the first chapter we now transform these problems into algorithmic problems using terminology that is well-known to a computer scientist. Usually, algorithmic bioinformatics problems are *optimization problems*. The following components characterize any optimization problem:

- An alphabet  $\Sigma$  of characters.
- A set  $I$  of admissible character strings that may occur as input instances.
- A solution relation  $S$  defining what is to be found, given admissible input  $x$ .
- An objective function  $c$  evaluating quality of solutions.
- A mode, *min* or *max*.

The role of these components and required properties of components are explained next. For being executable on a computer, all objects occurring within an algorithmic problem must be encoded in computer readable format. This is achieved by encoding data by strings over a suitable alphabet  $\Sigma$ . Usually only a subset of strings represents admissible data. For example,  $\neg(x \vee y)$  encodes an admissible Boolean formula, whereas  $\vee \neg x \neg y$  is not admissible. This is expressed by defining an instance set  $I$  of admissible input strings  $x$ . Of course, it should be possible to efficiently test whether strings  $x$  are members of  $I$ . Given admissible string  $x$  one is interested in strings  $y$  serving as solutions for  $x$ . What exactly solutions are is represented by a binary relation  $S(x, y)$  on pairs of strings. As with instances, it should be possible to efficiently test whether strings are solutions for admissible strings. Furthermore, any solution  $y$  for  $x$  should be bounded in length by a polynomial in the length of  $x$ . This makes sense since otherwise size of solutions would already prevent efficient storing or printing of solutions. What a solution is for a concrete example varies considerably: a solution for an equation or system of equations; a satisfying truth value assignment for the variables of a Boolean formula; a path

in a graph; and many more. Solutions are scored via an objective function  $c(x, y)$  assigning a numerical value to any pair  $(x, y)$  consisting of admissible string  $x$  and solution  $y$  for  $x$ . Of course,  $c(x, y)$  should be efficiently computable. Finally, an optimization problem has a mode, either *max* or *min*. For a minimization problem and admissible instance  $x$  define:

$$c_{\text{opt}}(x) = \begin{cases} \text{'no solution'} & \text{if no solution exists for } x \\ \min_{\text{all solutions } y \text{ for } x} c(x, y) & \text{otherwise.} \end{cases} \quad (2.1)$$

For a maximization problem, *min* is replaced by *max*. Minimization problems are always introduced in the following schematic format:

**NAME**

Given admissible  $x$ , find a solution  $y$  for  $x$  with minimum value  $c(x, y) = c_{\text{opt}}(x)$ .

Sometimes we concentrate on the following simplified problem version that only computes the optimal value of a solution, but does not return an optimal solution.

**NAME**

Given admissible  $x$ , compute value  $c_{\text{opt}}(x)$ .

Usually, an optimal solution  $y$  for  $x$  can be extracted from the computation of minimum value  $c_{\text{opt}}(x)$  with limited extra efforts. Sometimes there is no objective function to be optimized, only the task to simply find a solution provided one exists. This can be seen as a special form of optimization problem with trivial objective function  $c(x, y) = 1$ , for every solution  $y$  for  $x$ .

## 2.2 DNA Mapping

### 2.2.1 Mapping by Hybridization

Usage of sequence tagged sites for hybridization with overlapping fragments of the DNA molecule to be mapped was described in Sect. 1.1. The resulting algorithmic problem is the problem of rearranging columns of a binary matrix with the objective to make all bits '1' appear in consecutive order within every row.

**CONSECUTIVE ONES**

Given a binary matrix with  $n$  rows and  $m$  columns, find all possible rearrangements of columns that lead to a matrix with all bits '1' occurring in consecutive order within each row.

The algorithmic solution that will be presented in Chap. 4 uses the concept of *PQ-trees* introduced by Booth and Lueker (see [13]). To motivate this idea with an example, assume that rows 1 and 2 of a binary matrix contain common bits ‘1’ in columns 1, 2, 3, row 1 contains additional bits ‘1’ in columns 4, 5, 6, and row 2 contains additional bits ‘1’ in columns 7, 8. Let 9, 10, 11 be the remaining columns which thus contain common bits ‘0’ in rows 1, 2. To put bits ‘1’ in columns 1, 2, 3, 4, 5, 6 of row 1, as well as bits ‘1’ in columns 1, 2, 3, 7, 8 of row 2 in consecutive order, we are forced to rearrange columns as follows: take an arbitrary permutation of columns 4, 5, 6, followed by an arbitrary permutation of columns 1, 2, 3, followed by an arbitrary permutation of columns 7, 8. Alternatively, an arbitrary permutation of columns 7, 8, followed by an arbitrary permutation of columns 1, 2, 3, followed by an arbitrary permutation of columns 4, 5, 6 may be performed. In either variant, columns 9, 10, 11 may be arbitrarily arranged besides one of the described arrangements of 1, 2, 3, 4, 5, 6, 7, 8. The reader surely will have noticed that this description is somehow difficult to read. The following concept of a tree with two sorts of nodes, P-nodes and Q-nodes, simplifies descriptions considerably. A *P-node* is used whenever we want to express that an arbitrary permutation of its children is admitted, whereas a *Q-node* is used whenever we want to express that only the indicated fixed or the completely reversed order of its children is admitted. P-nodes are drawn as circles, whereas Q-nodes are drawn as rectangles. Using this notion, the arrangements described above can be visualized as shown in Fig. 2.1.

An algorithm described in Chap. 4 will successively integrate the requirements of consecutiveness, row by row, as long as this is possible. It starts with the PQ-tree consisting of a single P-node as its root and  $n$  leaves that does not constrain order of columns. Ideally, the algorithm ends with a maximally fixed PQ-tree consisting of a single Q-node as root and a permutation of column indices at its leaves. Note that a complete reversal of ordering is always possible in every PQ-tree. Thus, data may determine ordering of columns only up to such a complete reversal. Whenever the algorithm returns a more complex PQ-tree this indicates that the data was not informative enough to uniquely fix (up to a complete reversal) ordering of columns.

### 2.2.2 Mapping by Single Digestion

Given a string of length  $n$  and ascending positions (including start and end positions 1 and  $n$ )  $1 = x_1 < x_2 < \dots < x_m = n$ , define its single digest list as the sorted list of all differences  $x_p - x_q$ , with indices  $p$  and  $q$  ranging from 1 to  $m$  and  $p > q$ . Note that for a list of  $m$  positions, its single digest list consists of a number of  $m(m-1)/2$  entries (duplications allowed). The single digest problem is just the computation of the inverse of the single digest list function.

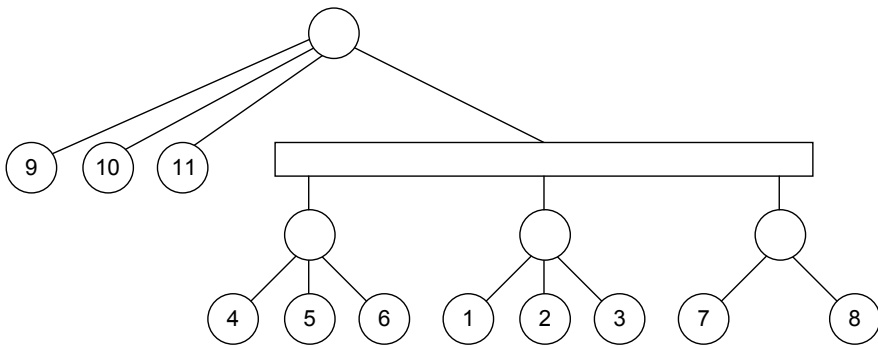


Fig. 2.1. Example PQ-tree

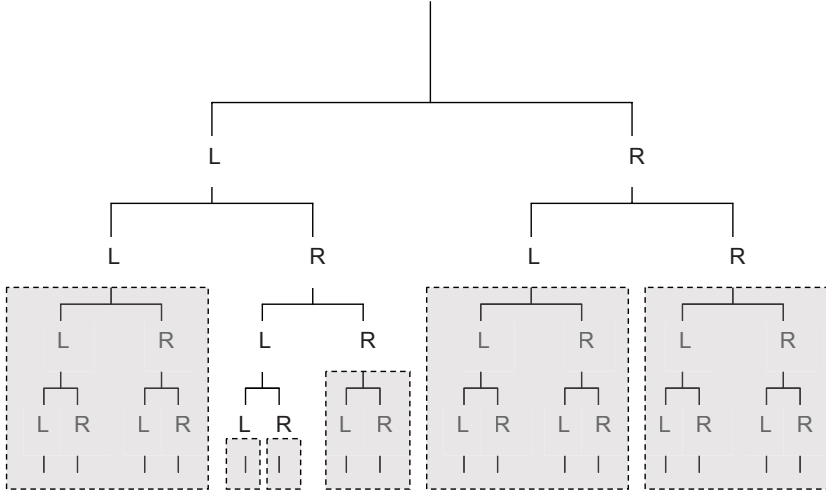
**SINGLE DIGEST**

Given a sorted list  $L$  of  $m$  lengths (duplications allowed) with maximum length  $n$  and minimum length 1, find all possible ascending lists  $P$  of positions ranging from 1 to  $n$ , (always including 1 and  $n$ ) with single digest list of  $P$  identical to  $L$ .

We illustrate an algorithm solving the problem on the basis of a complete backtracking search through all possible position lists  $P$  with a very simple example. After this, we summarize what is known at the moment about the complexity theory status of the problem. As an example, consider the list of lengths  $L = [2, 2, 3, 7, 8, 9, 10, 11, 12, 17, 18, 19, 21, 26, 29]$  consisting of  $1/2 \times 5 \times 6 = 15$  entries. A suitable list of positions  $P = [0, x_1, x_2, x_3, x_4, 29]$  has to be found such that its single digest list coincides with  $L$ , provided such a list exists.

We start with position list  $P = [0, 29]$  that realizes the difference length 29. Lengths 2, 2, 3, 7, 8, 9, 10, 11, 12, 17, 18, 19, 21, 26 remain to be produced by setting further cut positions. From now on, we always concentrate on the greatest number  $z$  amongst the remaining lengths which have not yet been realized as a difference between positions. Obviously,  $z$  can only be realized either relative to the left border as difference  $z = p - 0$  or relative to the right border as difference  $z = 29 - p$  with a further cut position  $p$ , since any other way would result in a number greater than  $z$ . Thus we have to search the binary tree of all options in order to set the next cutting position either relative to the left or the right border. In Fig. 2.2, branches L represent realization  $z = p - 0$  relative to the left border, whereas branches R represent realization  $z = 29 - p$  relative to the right border. This tree is traversed in a depth-first manner for as long as the next cut position can be set consistently with the actual list of remaining lengths. If the setting of the next position is inconsistent with the actual list of remaining lengths, backtracking takes decisions back and proceeds tree traversal at the next available unexplored

branch of the tree. As we will see for the chosen example, lots of subtrees may be pruned, i.e. they must not be visited in the traversal. Such pruned subtrees are indicated in Fig. 2.2 by enclosing them in shaded rectangles.



**Fig. 2.2.** Tree of all settings of cut positions showing pruned sub-trees that represent inconsistent settings

We show which positions are tried out at any stage of the traversal, which lengths remain to be realized at any step, and which subtrees are pruned due to leading to inconsistent lengths.

branch	positions	remaining lengths to be realized
root	0,29	2,2,3,7,8,9,10,11,12,17,18,19,21,26
L	0, <b>2</b> 6,29	2,2,7,8,9,10,11,12,17,18,19,21
LL	0, <b>2</b> 1,26,29	inconsistent due to $26 - 21 = 5$
LR	0, <b>8</b> ,26,29	2,2,7,9,10,11,12,17,19
LRL	0,8, <b>1</b> 9,26,29	2,2,9,12,17
LRLl	0,8, <b>1</b> 7,19,26,29	inconsistent due to $17 - 8 = 9$ and $26 - 17 = 9$
LRLR	0,8, <b>1</b> 2,19,26,29	inconsistent due to $12 - 8 = 4$
LRR	0,8, <b>1</b> 0,26,29	inconsistent due to $26 - 10 = 16$
R	0, <b>3</b> ,29	2,2,7,8,9,10,11,12,17,18,19,21
RL	0,3, <b>2</b> 1,29	inconsistent due to $3 - 0 = 3$
RR	0,3, <b>8</b> ,29	inconsistent due to $8 - 3 = 5$

We observe that list  $L$  cannot be represented as the single digest list of a position list  $P$ . To discover this, considerable parts of the tree had not to be visited. This massive pruning effect can also be observed in implementations. It sheds some light on the first one of the following remarks on the complexity theory status of the problem.

- The backtracking procedure described above usually visits only rather limited parts of the complete search tree to either find a position list  $P$  with single digest list  $L$ , or to find out that such a realization is not possible. Informally stated, average running time is low. There are more precise estimations found in the literature on what is to be expected, on average, as execution time for the backtracking procedure.
- No polynomial time algorithm solving the problem has been found so far.
- The problem has not been shown to be NP-hard so far.

Thus, a lot of work is left concerning the determination of the exact complexity status of the problem.

### 2.2.3 Mapping by Double Digestion

Let three lists  $A, B, C$  (duplications allowed) of lengths be given having identical sum of elements, i.e.

$$\sum_{a \in A} a = \sum_{b \in B} b = \sum_{c \in C} c. \quad (2.2)$$

We say that  $C$  is the *superposition of  $A$  with  $B$*  if drawing all lengths from  $A$  and all lengths from  $B$  in the order given by  $A$  and  $B$  onto a single common line and then taking all cut positions from both lists together, yields just the cut positions of the lengths in  $C$  arranged in the order given by  $C$ .

#### **DOUBLE DIGEST**

Given three lists  $A, B, C$  of lengths, find all permutations  $A^*, B^*, C^*$  of  $A, B, C$  such that  $C^*$  is a superposition of  $A^*$  with  $B^*$ .

Consider the following special case of the DOUBLE DIGEST problem. List  $A$  contains lengths which sum up to an even number  $s$ , list  $B$  contains two identical numbers  $s/2$ , and list  $C$  is the same as list  $A$ . Then permutations  $A^*, B^*, C^*$  exist such that  $C^*$  is a superposition of  $A^*$  with  $B^*$  if and only if the numbers in  $A$  can be separated into two sublists with identical summed lengths  $s/2$ . Only under this condition it is guaranteed that the cut position  $s/2$  inserted from list  $B$  does not introduce a new number into  $C^*$ . As it is well-known in complexity theory, this innocent looking problem of separating a list into two sublists with identical sums is an NP-hard problem, known as the PARTITION problem. Having thus reduced the PARTITION problem to the DOUBLE DIGEST problem shows that the latter one must be NP-hard, too. Arguments like these, i.e. reduction of a problem that is known to be NP-hard to the problem under consideration, will be studied in Chap. 5 in great detail.

## 2.3 Shotgun Sequencing and Shortest Common Superstrings

Shotgun sequencing leads to a set of overlapping fragments of an unknown DNA string  $S$ . Thus  $S$  contains all fragments as substrings. We say that  $S$  is a common superstring for the considered fragments. It seems to be plausible to assume that  $S$  usually is a shortest common superstring for the fragments under consideration. Of course, there are exceptions to this assumption. For example, any string  $S$  that consists of repetitions of the same character, or any string that exhibits strong periodicities leads to fragment sets having considerably shorter common superstrings than  $S$ . For realistic chromosome strings we expect the rule to hold.

Considering fragment sets, we can always delete any fragment that is a substring of another fragment. Having simplified the fragment sets this way we get so-called substring-free fragment sets. Thus we obtain the following algorithmic problem (see [28], [1] and [39]).

**SHORTEST COMMON SUPERSTRING**  
 Given a substring-free set of strings, find a shortest common superstring.

Consider strings AGGT, GGTC, GTGG. Their overlaps and overlaps lengths are shown in Fig. 2.3.

	AGGT	GGTC	GTGG		AGGT	GGTC	GTGG
AGGT	$\varepsilon$	GGT	GT	AGGT	0	3	2
GGTC	$\varepsilon$	$\varepsilon$	$\varepsilon$	GGTC	0	0	0
GTGG	$\varepsilon$	GG	G	GTGG	0	2	1

**Fig. 2.3.** Pairwise overlaps and overlap lengths

The simplest idea for the construction of a common superstring is surely a greedy approach which always replaces two fragments  $S = XY$  and  $T = YZ$  having longest overlap  $Y$  with string  $XYZ$ . Let us see what happens for this example. First, AGGT and GGTC are found having longest overlap of length 3. They are replaced by AGGTC. Now AGGTC and GTGG as well as GTGG and AGGTC have only zero length overlap, thus the final common superstring is AGGTCGTGG having length 9. A “less greedy” strategy works better: first replace AGGT and GTGG having overlap of length 2 with AGGTGG, then replace AGGTGG and GGTC having overlap of length 2 with AGGTGGTC. Thus a shorter common superstring of length 8 is obtained.

Failure of a greedy approach is often an indication of the hardness of a problem. Indeed, in Chap. 5 we show that SHORTEST COMMON SUPERSTRING is an NP-hard problem, thus there is no polynomial time algorithm solving it (unless  $P = NP$ ). In Chap. 6, a polynomial time algorithm is presented that returns a common superstring that is at most four times longer than a shortest common superstring. Such an algorithm is called a 4-approximation algorithm.

Finally, let us compare SHORTEST COMMON SUPERSTRING with the similar looking problem asking for a longest common substring for a given list of strings.

**LONGEST COMMON SUBSTRING**

Given a set of strings, find a longest common substring.

An application would be, for example, to find a longest common substring for two doctoral theses (in order to detect plagiarism). As we will see, this latter problem is efficiently solvable. It is rather obvious why both problems differ so radically from a complexity theory point of view. Whereas the former problem has to search in the unrestricted space of arbitrary superstrings, the latter problem has a search space that is considerably restricted by the given strings: only substrings are admitted. There is not a completely trivial solution for this problem. The data structure of suffix trees will help to develop a particularly efficient algorithm.

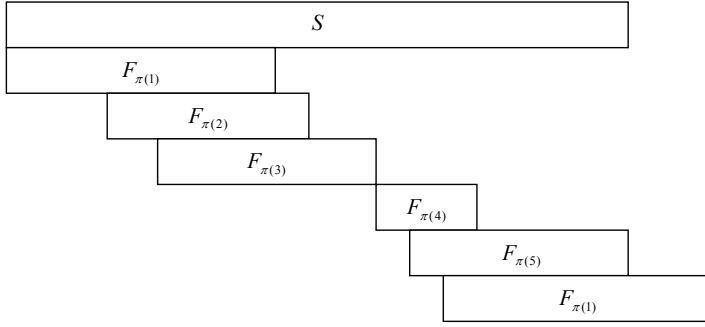
Returning to the former NP-complete problem we show how to embed it into more familiar algorithmic fields, namely Hamiltonian circuit problems, which will also lead us to the announced 4-approximation algorithm in Chap. 6. Let a substring-free list of strings  $F_1, \dots, F_m$  be given. Imagine  $S$  is a shortest common superstring for  $F_1, \dots, F_m$ . We arrange strings  $F_1, \dots, F_m$  as a permutation  $F_{\pi(1)}, F_{\pi(2)}, \dots, F_{\pi(m)}$  with increasing start positions within  $S$  and get a situation that typically looks as in Fig. 2.4 (indicated for  $m = 5$  strings).

Here, the following must be true:

- First string starts at the beginning of string  $S$ .
- Last string ends at the end of string  $S$ .
- Next string starts within previous string or at the end of previous string, i.e. there are no gaps between consecutive strings.
- Next string ends strictly right of the end of previous string (set is substring-free).
- Any two consecutive strings have maximum overlap.

This is true since any deviation from these rules would allow us to construct a shorter common superstring by shifting a couple of consecutive strings to the left. Note that at the end of the permutations of strings the first used string appears again (and ends strictly right of the end of  $S$ ).





**Fig. 2.4.** Arrangement of fragments according to increasing start position within a common superstring  $S$

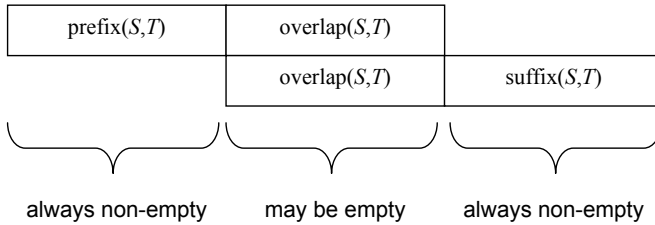
Thus, any shortest superstring of  $F_1, \dots, F_m$  gives rise to a permutation  $F_{\pi(1)}, \dots, F_{\pi(m)}$  of strings, and conversely, any permutation  $F_{\pi(1)}, \dots, F_{\pi(m)}$  of strings gives rise to a common superstring  $S(\pi)$  of  $F_1, \dots, F_m$  (not necessarily of minimal length).

$S(\pi) = P(\pi)O(\pi)$  with

$P(\pi) = \text{prefix}(F_{\pi(1)}, F_{\pi(2)}) \dots \text{prefix}(F_{\pi(m-1)}, F_{\pi(m)})\text{prefix}(F_{\pi(m)}, F_{\pi(1)})$

$O(\pi) = \text{overlap}(F_{\pi(m)}, F_{\pi(1)})$

Here we used the notions of ‘prefix’ and ‘overlap’ defined as follows. Given decompositions of strings  $S = XY$  and  $T = YZ$  such that  $Y$  is the longest suffix of  $S$  that is different from  $S$  and also a prefix of  $T$ , we call  $Y$  the overlap of  $S$  with  $T$  and denote it by  $\text{overlap}(S, T)$ . We refer to  $X$  as  $\text{prefix}(S, T)$  and to  $Z$  as  $\text{suffix}(S, T)$  (see Fig. 2.5). Note that we require strict overlaps, i.e. an overlap is not allowed to cover the complete string. This is automatically true for different strings  $S$  and  $T$  since it was assumed that neither one is a substring of the other, but must be explicitly demanded for the overlap of a string  $S$  with itself. Thus, the overlap of string AAAA with AAAA is AAA, but not AAAA. To define the overlap of a string with itself as a strict overlap will prove to be important later.



**Fig. 2.5.** Prefix, overlap, suffix

Besides overlaps, prefixes, and suffixes, we also consider corresponding lengths  $o(S, T)$ ,  $p(S, T)$ , and  $s(S, T)$ . These definitions give rise to two graphs, called *overlap graph* and *prefix graph*, for a string list  $F_1, \dots, F_m$ . Both are directed graphs that have  $m$  nodes labelled  $F_1, \dots, F_m$  and directed edges between any two such nodes (thus also from every node back to itself). Furthermore, the edge pointing from node  $F_i$  to node  $F_j$  is weighted by number  $o(F_i, F_j)$  in the overlap graph, and  $p(F_i, F_j)$  in the prefix graph. Shortest common superstrings are closely related to cheapest cycles in prefix graph. Looking at the decomposition of string  $S(\pi)$  above, we infer:

$$\begin{aligned} |S| &= \sum_{i=1}^{m-1} p(F_{\pi(i)}, F_{\pi(i+1)}) + p(F_{\pi(m)}, F_{\pi(1)}) + o(F_{\pi(m)}, F_{\pi(1)}) \\ &= \text{prefixlengths}(\pi) + o(F_{\pi(m)}, F_{\pi(1)}). \end{aligned} \quad (2.3)$$

Here, the term  $\text{prefixlengths}(\pi)$  computes the costs of the cycle through prefix graph defined by permutation  $\pi$ :

$$\text{prefixlengths}(\pi) = \sum_{i=1}^{m-1} p(F_{\pi(i)}, F_{\pi(i+1)}) + p(F_{\pi(m)}, F_{\pi(1)}). \quad (2.4)$$

It is a lower bound for the length of a shortest common superstring. Note that in realistic situations with chromosome length of some 100.000 base pairs and fragments lengths of about 1000 base pairs, prefix costs of a permutation deviate only by a small fraction from the length of the superstring defined by the permutation. Thus, searching for a shortest common superstring might as well be replaced with searching for a cheapest Hamiltonian cycle (closed path visiting each node exactly once) through the prefix graph. Unfortunately, the HAMILTONIAN CYCLE problem is NP-complete, as is shown in Chap. 5. Nevertheless, it opens way for a further, computationally feasible relaxation of the concept of shortest common superstrings. As it is known from algorithm theory, computing a finite set of disjoint cycles (instead of a single cycle) having minimum summed costs and covering every node in a weighted graph is an efficiently solvable problem. Such a finite set of cycles is called a *cycle cover*. We postpone the question of how to compute a cheapest cycle cover.

At the moment let us point to a possible misunderstanding of the concept of cheapest cycle covers. Usually, weighted graphs do not have self-links from a node back to itself, equivalently stated, self-links are usually weighted with a value 0. As a consequence, in such graphs we could always consider the trivial cycle cover consisting of 1-node cycles only. Obviously, summed costs would be 0, thus minimal. Of course, such trivial cycle covers would be of no use for anything. Things change whenever we have self-links that are more costly. It could well be the case that some non-trivial path back to a node  $x$  is cheaper than the 1-node cycle from  $x$  to itself. Indeed, prefix and overlap graphs never have self-links with weight 0 (as overlaps were defined to be proper overlaps). This requirement will later prove to be essential in the construction of an approximation algorithm.

## 2.4 Exact Pattern Matching

### 2.4.1 Naive Pattern Matching

One of the most frequently executed tasks in string processing is searching for a pattern in a text. Let us start with string  $T = T[1 \dots n]$  of length  $n$ , called text, and string  $P = P[1 \dots m]$  of length  $m$ , called pattern. The simplest algorithm compares  $P$  with the substring of  $T$  starting at all possible positions  $i$  with  $1 \leq i \leq n - m + 1$ . Whenever comparison was successful, start position  $i$  is returned. Obviously, this procedure requires  $O(nm)$  comparisons in the worst case.

```

for  $i = 1$  to  $n - m + 1$  do
   $a = 0$ 
  while  $P(a + 1) = T(i + a)$  and  $a < m$  do
     $a = a + 1$ 
  end while
  if  $a = m$  then
    print  $i$ 
  end if
end for

```

### 2.4.2 Knuth-Morris-Pratt Algorithm

There is a more clever algorithm, the so-called *Knuth-Morris-Pratt algorithm* (described in [44]), which achieves running time of  $O(n + m)$  by suitably preprocessing the pattern  $P$ . Preprocessing of  $P$  works as follows. For every  $a = 1, \dots, m$  determine the length  $f(a)$  of the longest proper prefix of  $P[1 \dots a]$  that is also a suffix of  $P[1 \dots a]$ . As an example, all values  $f(a)$  for pattern  $P = \text{'anas'}$  are shown in Fig. 2.6.

$a$	1	2	3	4	5	6
$f(a)$	0	0	1	2	3	0

**Fig. 2.6.** Prefix-suffix match lengths computed for all prefixes of text ‘anas’

With function  $f$  available, pattern searching within text  $T$  may be accelerated as follows: start searching for  $P$  at the beginning of text  $T$ . Whenever comparison of  $P$  with  $T$  fails at some working position  $i$  of  $T$  after having successfully matched the first  $a$  characters of  $P$  with the  $a$  characters of  $T$  left of working position  $i$ , we need not fall back by  $a$  positions to start index  $i - a + 1$  within  $T$  and begin comparison with  $P$  from scratch. Instead we use that  $P[1 \dots f(a)]$  is the longest proper prefix of  $P[1 \dots a]$  that appears left of

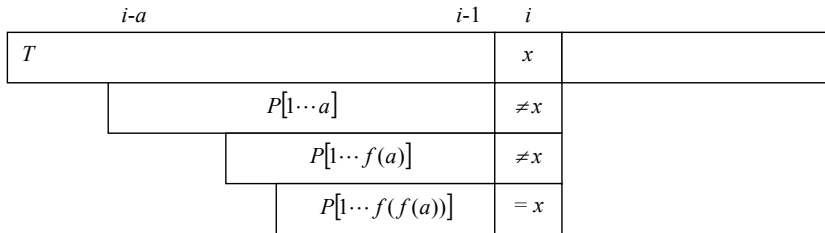
position  $i$  within text  $T$ . Thus we may continue comparison of characters with  $P(f(a)+1)$  and  $T(i)$ . In the pseudo-code below, variables  $i$  and  $a$  maintain the actual working position within  $T$  and the length of the actual prefix of  $P$  that was found as a substring of  $T$  exactly left of working position  $i$ . Both working position  $i$  within  $T$  as well as actual start position  $i - a$  of the matching prefix of  $P$  of length  $a$  monotonically grow from 1 to at most  $n$ . Thus, we find one or all occurrences of  $P$  in  $T$  in  $2n$  steps.

```

a = 0
for i = 1 to n do
  while a > 0 and P(a + 1) ≠ T(i) do
    a = f(a)
  end while
  if P(a + 1) = T(i) then
    a = a + 1
  end if
  if a = m then
    print i - m + 1
    a = f(m)
  end if
end for

```

Figure 2.7 shows a run of the algorithm which requires two shifts to the right of pattern  $P$  to find a match with actual character  $x$ .



**Fig. 2.7.** Shifting prefixes

It remains to be clarified how expensive the computation of all values of function  $f$  is. This is done the same way by shifting prefixes of  $P$  over  $P$ , instead of shifting prefixes of  $P$  over  $T$ . The reader may develop a corresponding diagram as above.

```

f(1) = 0
a = 0
for j = 2 to m do

```

```

while  $a > 0$  and  $P(a + 1) \neq P(j)$  do
     $a = f(a)$ 
end while
if  $P(a + 1) = P(j)$  then
     $a = a + 1$ 
     $f(j) = a$ 
else
     $f(j) = 0$ 
end if
end for

```

As with the former algorithm the whole procedure requires at most  $2m$  moves of start and end index of the actual prefix of  $P$ , thus requires  $O(m)$  steps. All together,  $O(n + m)$  is the complexity of applying the Knuth-Morris-Pratt algorithm to a text of length  $n$  and pattern of length  $m$ .

### 2.4.3 Multi-Text Search

Preprocessing of search patterns is particularly valuable in case of searching for a fixed pattern of length  $m$  in several (say  $p$ ) texts of length  $n$  each. Assume that  $m < n$ . For this case we get a running time of  $O(m + pn) = O(pn)$  instead of  $O(pnm)$  as for the naive algorithm. Thus we have achieved a speed-up factor of  $m$ .

### 2.4.4 Multi-Pattern Search

Unfortunately, in bioinformatics one usually has a fixed text of length  $n$  (say a human genome) and several (say  $p$ ) patterns of length  $m$ . The Knuth-Morris-Pratt algorithm requires  $p$  times a preprocessing of a pattern of length  $m$ , followed by  $p$  times running through text  $T$ . This leads to a complexity of  $O(pm + pn)$ . Using suffix trees this can be done better. Construction of a suffix tree can be performed in time  $O(n)$  (as shown in Chap. 4). Searching for a pattern  $P$  of length  $m$  within a suffix tree can then be done in  $O(m)$  steps by simply navigating into the tree along the characters of  $P$  (for as long as possible). All together, searching for  $p$  patterns of length  $m$  each in a text of length  $n$  can be completed in time  $O(n + pm)$ . Applications of suffix trees are usually straight-forward, whereas efficient construction of suffix trees requires much more effort. A naive algorithm for the construction of the suffix tree for a string  $T[1 \dots n]$  of length  $n$  would first create a single link labelled with suffix  $T[1 \dots n]$ , then integrate the next suffix  $T[2 \dots n]$  into the initial tree by navigating along the characters of  $T[2 \dots n]$  and eventually splitting an edge, then integrating suffix  $T[3 \dots n]$  by the same way, etc. In the worst case,  $n + (n - 1) + (n - 2) + \dots + 2 + 1$  comparisons of characters have to be done leading to a quadratic time algorithm. It requires clever ideas to save a lot of time such that a linear time algorithm results.

### 2.4.5 Formal Definition of Suffix Trees

The reader may use the ‘ananas’ example from Sect. 1.2.2 as an illustration for the formal concepts introduced here (see Fig. 1.7). Given string  $T$  of length  $n$ , a *suffix tree* for  $T$  is a rooted tree with leaves each labelled with a certain number, edges labelled with non-empty substrings of  $T$  in a certain way, additional links between inner nodes called *suffix links*, and a distinguished link to a node called *working position*, such that several properties are fulfilled. To state these properties, we introduce the notion of a *position in the tree* and the *path label* of a position. First, a position is a pointer pointing either to a node of the tree or pointing between two consecutive characters of an edge label. Given a position, its *path label* is the string obtained by concatenating all characters occurring at edges on the path from the root to the position under consideration. As a special case, the *node label* of some node is defined to be the path label of the position pointing to that node. Now, properties of a suffix tree can be defined as follows:

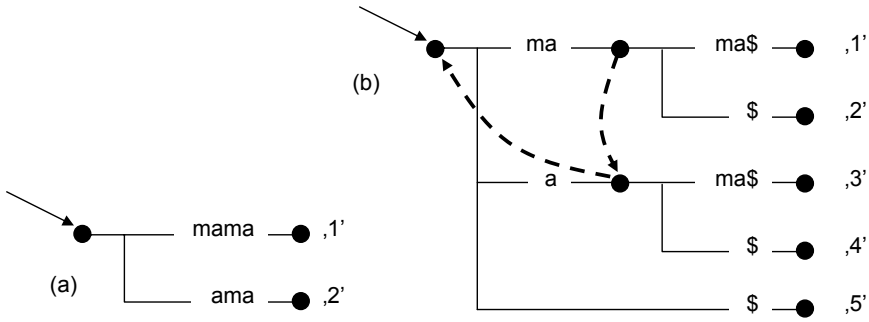
- Every inner node has at least two successor nodes.
- Edges leaving some node are labelled with substrings that have different first characters.
- Every leaf has as node label of some suffix  $T[j \dots n]$  of  $T$  and, in that case, is labelled with ‘ $j$ ’.
- Every suffix appears as a path label of a position in the tree.
- Every inner node with path label  $xw$ , for a character  $x$  and string  $w$ , has a so-called *suffix link* starting at that node and pointing to an inner node with suffix link  $w$  in case that  $w$  is non-empty, and pointing to the root otherwise.

For more extended trees, it is convenient to present them in a horizontal left-to-right manner. The suffix tree shown in Fig. 1.7 for string  $T = \text{‘ananas’}$  is special in the sense that all suffixes of  $T$  appear as path labels of leaves. The following diagram shows a suffix tree<sup>1</sup> for the string ‘mama’ (Fig. 2.8).

Observe that two of its suffixes (‘ma’ and ‘a’) are represented as path labels of non-leaf positions. The reason for this is that string ‘mama’ contains suffixes that are at the same time prefixes of other suffixes. This is not the case for the string ‘ananas’. By using an additional end marker symbol we can always enforce that suffixes are represented as path labels of leaves. Figure 2.8 shows a suffix tree for string ‘mama’, as well as for ‘mama\$’.

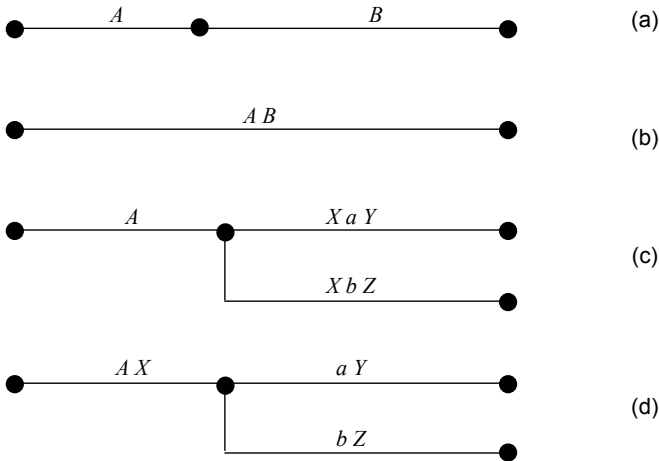
---

<sup>1</sup> In the literature, “suffix tree” for a string  $T$  is sometimes understood as suffix tree (in our sense) for string  $T\$$  with additional end marker symbol, whereas a suffix tree (in our sense) is called “implicit suffix tree”. Sometimes, suffix links and actual working position are not introduced as explicit components of a suffix tree. Since in the description of Ukkonen’s algorithm in Chap. 4 suffix trees for prefixes of a string  $T\$$  play a central role, and such prefixes do not end with end marker \$, we prefer to use the definition of suffix trees as given above.



**Fig. 2.8.** Suffix tree: (a) for string 'mama'; (b) for string 'mama\$'

Suffix trees represent all suffixes of a string  $T$  as labels of positions in a maximally compressed manner. Note that an inner node with only one successor is not allowed to occur within a suffix tree (Fig. 2.9 (a)). Instead of this, substring  $AB$  must be the label of a single edge (Fig. 2.9 (b)). Also note that a node is not allowed to have two or more edges to successors whose labels have a common non-empty maximal prefix  $X$  (Fig. 2.9 (c)). Instead of this, the situation must be represented as in Fig. 2.9 (d).



**Fig. 2.9.** Non-redundant labelling of edges of a suffix tree

As a consequence, a suffix tree for string  $T = T[1 \dots n]$  is, up to the actual working position (that can be chosen at an arbitrary position in the tree), uniquely determined: start with a single edge with label  $T$  between a root node and a leaf marked '1'. Next, navigate along the characters of the suffix  $T = T[2 \dots n]$  into the tree constructed so far, starting at the root. The restrictions on the structure of a suffix tree uniquely determine whether and

where a new node must eventually be introduced. Proceeding this way with all further suffixes of  $T$ , we build up the suffix tree for  $T$  in a unique manner. Besides the nodes and edges constructed so far, the tree does not contain further nodes since every leaf position in the tree must have a suffix of  $T$  as its path label. So far, we have not talked about existence and uniqueness of suffix links. This will soon be done.

As an example, consider string  $T = \text{'abcdabda'}$ . Figure 2.10 (a) shows a suffix tree for  $T$ . Note that whereas the existence of suffix links can be guaranteed for inner nodes, the same is not true for leaves: leaf marked '6' has path label 'bda', but there is no node with path label 'da'.

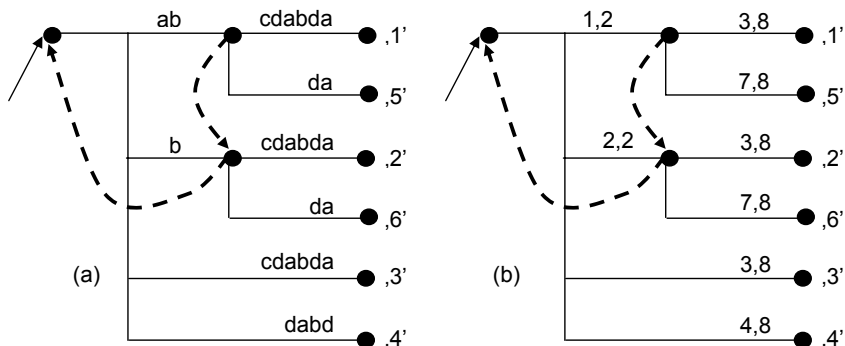


Fig. 2.10. (a) Strings as labels; (b) String limits as labels

### Lemma 2.1.

*For every inner node of a suffix tree with path label  $xw$  (with  $x$  being a single character and  $w$  being a string) there is a unique inner node or root with label  $w$ .*

*Proof.* Let  $xw$  be the path label of inner node  $u$ . Each inner node has at least two successor nodes. So we may choose two leaves below node  $u$ , one with path label  $xwy$ , the other one with label  $xwz$ . We know that strings  $y$  and  $z$  must have different first characters. By definition of a suffix tree, we know that  $xwy$  and  $xwz$  are suffixes of string  $T$ . Hence,  $wy$  and  $wz$  are suffixes of  $T$ , too. By definition of suffix trees, again, there must be positions in the tree with path labels  $wy$  and  $wz$ . This means that on the path from the root down to these positions there must be a node  $v$  with path label  $w$  (at node  $v$ , the path splits into the two considered paths with labels  $wy$  and  $wz$ ). Of course, node  $v$  is uniquely determined.  $\square$

At first sight, it seems to be impossible to have a linear time algorithm for constructing suffix trees, since already the space requirements prevent this: for a text  $T$  of length  $n$ , a suffix tree for  $T\$$  has  $n+1$  leaves with concatenated labels along the paths from the root to its leaves of length  $n+1, n, n-1, \dots, 2, 1$ .



So the explicit insertion of these path labels already requires  $O(n^2)$  time. This problem is not too severe and can be overcome by representing an edge label  $T[a \dots b]$  by its start and end positions  $a$  and  $b$  within  $T$ , instead of explicitly writing the symbols of  $T[a \dots b]$  at the considered edge. In the example above, the suffix tree would be drawn as in Fig. 2.10 (b). Nevertheless, for better readability, in examples we prefer the string representations of edge labels. But even with such sparse representation of edge labels,  $O(n^2)$  time for a naive construction of suffix trees that integrates more and more suffixes into a growing tree seems to be unavoidable since for suffixes of length  $n-1, n-2, \dots$  we must navigate in the growing tree in order to find the correct point of insertion for the actual suffix, with each navigation requiring as many steps as the number of symbols in the actual suffix.

The basic idea that will finally lead to a more efficient construction of the suffix tree of a text  $T = T[1 \dots n]$  is to successively construct suffix trees for growing strings  $T = T[1 \dots i]$ . As it will be shown in Chap. 4, a suffix tree for  $T = T[1 \dots i+1]$  can be simply obtained from a suffix tree for  $T = T[1 \dots i]$  by inserting the next character  $T(i+1)$  at all positions where it has to be placed. Though at first sight it seems to be a procedure that requires  $O(1+2^2+3^2+\dots+n^2) = O(n^3)$  steps (if being implemented in a comparable naive manner as described above), we will show that most of the steps can be saved by a clever analysis of the procedure. A linear time algorithm due to Ukkonnen (see [71]) will be presented in Chap. 4. Previously published, slightly different linear time algorithms constructing suffix-trees are due to Weiner [79] and McCreight [55].

## 2.5 Soft Pattern Matching = Alignment

### 2.5.1 Alignments Restated

As introduced in Chap. 1, an alignment of string  $S$  with string  $T$  is obtained by introducing spacing symbols at certain positions of  $S$  and  $T$  such that two strings  $S^*$  and  $T^*$  of the same length result and no position in  $S^*$  and  $T^*$  is filled with two spacing symbols. As an example,

$$\begin{array}{ccccccccccc} \text{A} & \text{C} & - & \text{G} & \text{A} & - & \text{G} & \text{T} & \text{T} & \text{C} & - & \text{A} & \text{C} & \text{T} \\ - & \text{C} & \text{T} & \text{G} & \text{G} & \text{C} & \text{T} & - & \text{T} & \text{G} & \text{G} & \text{A} & - & \text{T} \end{array}$$

is an alignment of ACGAGTTCACT with CTGGCTTGGAT.

### 2.5.2 Evaluating Alignments by Scoring Functions

Quality of an alignment is measured by a numerical value called the *score* that depends on a *scoring function*  $\sigma$ . The scoring function expresses in terms of numerical values  $\sigma(x, x)$ ,  $\sigma(x, y)$ ,  $\sigma(x, -)$ ,  $\sigma(-, y)$  for different letters  $x$  and  $y$  how strongly matches  $x/x$  are rewarded and mutations  $x/y$ , deletes  $x/-$  and

inserts  $-/y$  are punished. A typical example of a scoring function is shown in Fig. 2.11. Here, matches are rewarded with value  $+2$ , mutations are punished with value  $-1$ , inserts and deletes are even more strongly punished with value  $-2$ .

	A	C	G	T	-
A	2	-1	-1	-1	-2
C	-1	2	-1	-1	-2
G	-1	-1	2	-1	-2
T	-1	-1	-1	2	-2
-	-2	-2	-2	-2	0

**Fig. 2.11.** Scoring matrix

The score of an alignment  $S^*, T^*$  of string  $S$  with string  $T$  is the sum of scores of all pairs that occur at the same position in the alignment and is denoted by  $\sigma^*(S^*, T^*)$ . In case that  $S^*$  and  $T^*$  have common length  $n$  the definition is:

$$\sigma^*(S^*, T^*) = \sum_{i=1}^n \sigma(S^*(i), T^*(i)). \quad (2.5)$$

As discussed in Chap. 1, for practically used scoring functions based on log-odds ratios of the frequencies of occurrence of character pairs and single characters, a greater score represents a more favourable event. Thus the primary goal in aligning strings is to find an alignment  $S^*, T^*$  for strings  $S, T$  having maximum score denoted as in the definition below:

$$\sigma_{\text{opt}}(S, T) = \max_{S^*, T^*} \sigma^*(S^*, T^*). \quad (2.6)$$

Summarizing, the alignment problem is stated as follows:

**ALIGNMENT**

Given strings  $S$  and  $T$ , find an alignment  $S^*, T^*$  with maximum score  $\sigma_{\text{opt}}(S, T)$ .

Having strings  $S$  and  $T$  of length  $n$  and  $m$ , there are more than  $O(2^n)$  alignments since there are already  $O(2^n)$  many choices where to introduce spacing symbols into string  $S$ . The reader may think about a better lower bound for the number of alignments and show that this number is considerably greater than  $2^n$ . The reader may also give an upper bound for the number of alignments of  $S$  and  $T$  using the fact that alignments may be at most  $n + m$  pairs long.

## 2.6 Multiple Alignment

### 2.6.1 Sum-of-Pairs Maximization

Given strings  $S_1, \dots, S_k$ , a multiple alignment consists of strings  $T_1, \dots, T_k$  of the same length that result from  $S_1, \dots, S_k$  by insertion of spacing symbols at certain positions, with the restriction that columns consisting of spacing symbols only are not allowed. Given a multiple alignment  $T_1, \dots, T_k$  of length  $m$  and a scoring function  $\sigma$  for pairs of characters  $\sigma(a, b)$  and pairs of character and spacing symbol  $\sigma(a, -)$  and  $\sigma(-, b)$ , we define its *sum-of-pairs score* as follows:

$$\sigma^*(T_1, \dots, T_k) = \sum_{i < j} \sum_{p=1}^m \sigma(T_i(p), T_j(p)) = \sum_{i < j} \sigma^*(T_i, T_j). \quad (2.7)$$

**SUM-OF-PAIRS MULTIPLE ALIGNMENT**  
 Given a scoring function  $\sigma$  and strings  $S_1, \dots, S_k$ , compute a multiple alignment  $T_1, \dots, T_k$  with maximum sum-of-pairs score.

Computing a multiple alignment is NP-hard (shown in Chap. 5). We present a 2-approximation algorithm in Chap. 6. In the literature one finds criticism of the adequateness of scoring via sum-of-pairs. For example, from a statistical point of view, it seems quite unnatural to score a column with entries  $a, b, c$  by a sum of pairwise local log-odd values

$$\log \frac{p_{ab}}{p_a p_b} + \log \frac{p_{ac}}{p_a p_c} + \log \frac{p_{bc}}{p_b p_c} \quad (2.8)$$

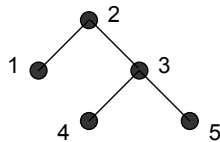
instead of taking a more integrated view expressed by common log-odds value

$$\log \frac{p_{abc}}{p_a p_b p_c}. \quad (2.9)$$

This motivates consideration of various “consensus approaches”.

### 2.6.2 Multiple Alignment Along a Guide Tree

The problem with sum-of-pairs alignment is that the sum of scores between any two strings of an alignment must be maximized. The problem turns into a solvable one if we consider a tree structure on the set of string indices and look for an alignment that maximizes the sum of scores between any two strings with indices connected by a link of the tree. Following the links of the tree in arbitrary order we can incrementally build up an alignment by locally maximizing scores between two linked strings. We discuss an example. Consider the tree in Fig. 2.12.

**Fig. 2.12.** Simple example guide tree

Assume that we have already build up an alignment  $T_1, T_2, T_3, T_4$  with maximum sum  $\sigma(T_1, T_2) + \sigma(T_2, T_3) + \sigma(T_3, T_4)$ . We have to integrate a string  $T_5$  into this alignment and take care to maximize  $\sigma(T_3, T_5)$ , letting the sum obtained at the stage before unchanged. Thus we should simply take a maximum score alignment  $T^3, T_5$  of string  $S_3$  with string  $S_5$ . The only problem is that string  $T^3$  may have spacing symbols at different positions than the formerly computed string  $T_3$ , though both are derived from the same string  $S_3$  by insertion of spacing symbols. A simple adaptation of  $T_3$  and  $T^3$  by insertion of additional spacing symbols (“padding”) solves the problem. We illustrate what has to be done with a simple example:

$T_1$	C G G - T C - - G G T
$T_2$	- G G T T - A A A G T
$T_4$	C - - T T C - A A - G
$T_3$	C G G T - C A A - G -
$T^3$	- C G G T C - A - A G
$S_5$	A - - G T - C A T A -

The padded alignment adapts  $T_3$  and  $T^3$  without changing scores. It looks as follows.

$R_1$	- C G G - T C - - - - G G T
$R_2$	- - G G T T - - A - A A G T
$R_4$	- C - - T T C - - - A A - G
$R_3$	- C G G T - C - A - A - G -
$R^3$	- C G G T - C - A - A - G -
$R_5$	A - - G T - - C A T A - - -

We end with the following alignment:

$R_1$	- C G G - T C - - - - G G T
$R_2$	- - G G T T - - A - A A G T
$R_4$	- C - - T T C - - - A A - G
$R_3$	- C G G T - C - A - A - G -
$R_5$	A - - G T - - C A T A - - -

There are various heuristics that propose a more or less plausible guide tree and then align along the chosen tree. We soon discuss a few of them. Summarizing, having a tree with a distribution of strings  $S_1, \dots, S_k$  to its nodes (every

node is labelled with one of the strings to be aligned) and seeking to maximize the sum of scores between strings linked in the tree makes things easy. Things change towards NP-completeness as soon as tree structures are considered that also have nodes not labelled with any one of the strings  $S_1, \dots, S_k$ . This happens already for the simplest example of a tree with unlabelled root (representing a common unknown ancestor) and strings  $S_1, \dots, S_k$  attached to the leaves. Situations like these will be discussed in the next sections.

### 2.6.3 Consensus Line Optimization

Given a multiple alignment  $T_1, \dots, T_k$  of length  $m$ , its consensus line is the string  $T$  such that the following consensus sum has maximum value:

$$\text{consensus}(T, T_1, \dots, T_k) = \sum_{p=1}^m \sum_{i=1}^k \sigma(T(p), T_i(p)) = \sum_{i=1}^k \sigma^*(T, T_i). \quad (2.10)$$

If there are several choices for a character of  $T$ , fix one in an arbitrary manner.

#### CONSENSUS MULTIPLE ALIGNMENT

Given a scoring function  $\sigma$  and strings  $S_1, \dots, S_k$ , compute a multiple alignment  $T_1, \dots, T_k$  such that its consensus line  $T$  has maximum value  $\text{consensus}(T, T_1, \dots, T_k)$ .

### 2.6.4 Steiner String

An equivalent formulation can be given exclusively in terms of strings. A *Steiner string* for a list of strings  $S_1, \dots, S_k$  is a string  $S$  that maximizes the following *Steiner sum*:

$$\text{steiner}(S, S_1, \dots, S_k) = \sum_{i=1}^k \sigma_{\text{opt}}(S, S_i). \quad (2.11)$$

#### STEINER STRING

Given a scoring function  $\sigma$  and strings  $S_1, \dots, S_k$ , compute a string  $S$  with maximum value  $\text{steiner}(S, S_1, \dots, S_k)$ .

### 2.6.5 Equivalence of Consensus Lines and Steiner Strings

#### Theorem 2.2.

(a) From an arbitrary multiple alignment  $T_1, \dots, T_k$  of  $S_1, \dots, S_k$  with consensus line  $T$  we may construct a string  $S$  with  $\text{steiner}(S, S_1, \dots, S_k) \geq \text{consensus}(T, T_1, \dots, T_k)$ .

(b) From an arbitrary string  $S$  for  $S_1, \dots, S_k$  we may construct a multiple alignment  $T_1, \dots, T_k$  for  $S_1, \dots, S_k$  with consensus line  $T$  and  $\text{consensus}(T, T_1, \dots, T_k) \geq \text{steiner}(S, S_1, \dots, S_k)$ .

(c) If the multiple alignment in (a) has maximum consensus value of its consensus line, we obtain a Steiner string.

(d) If string  $S$  in (b) was a Steiner string, we obtain a multiple alignment with maximum consensus value of its consensus line.

*Proof.* (a) Consider consensus line  $T$ . Delete all occurrences of the spacing symbol from  $T$  obtaining string  $S$ . As  $T, T_i$  is an alignment (not necessarily optimal) of  $S$  with  $S_i$  we conclude that

$$\sigma_{\text{opt}}(S, S_i) \geq \sigma^*(T, T_i).$$

Thus we obtain

$$\begin{aligned} \text{steiner}(S, S_1, \dots, S_k) &= \sum_{i=1}^k \sigma_{\text{opt}}(S, S_i) \\ &\geq \sum_{i=1}^k \sigma^*(T, T_i) \\ &= \text{consensus}(T, T_1, \dots, T_k). \end{aligned}$$

(b) Consider an arbitrary string  $S$ . For each  $i$  take a maximum score alignment  $T^i, T_i$  of  $S$  with  $S_i$ . Note that though each string  $T^i$  is  $S$  with some spacing symbols introduced, spacing symbols may be placed at different positions within different strings  $T^i$ . Nevertheless, we may construct a single alignment  $R_1, \dots, R_k, R$  of  $S_1, \dots, S_k, S$  out of these local alignments by padding strings with additional spacing symbols in such a way that a common string  $R$  instead of  $k$  different strings  $T^1, \dots, T^k$  is used to align  $S$  to the other strings. Scores are not changed by such paddings. By construction, the consensus line  $C$  of alignment  $R_1, \dots, R_k$  has consensus value at least as good as  $R$ , thus

$$\begin{aligned} \text{consensus}(C, R_1, \dots, R_k) &\geq \text{consensus}(R, R_1, \dots, R_k) \\ &= \text{steiner}(S, S_1, \dots, S_k). \end{aligned}$$

(c) Assume that we started in (a) with a multiple alignment  $T_1, \dots, T_k$  of  $S_1, \dots, S_k$  with consensus line  $T$  and maximum value  $\text{consensus}(T, T_1, \dots, T_k)$ . Let  $S$  be obtained by deleting all spacing symbols from  $T$ . We know that

$$\text{steiner}(S, S_1, \dots, S_k) \geq \text{consensus}(T, T_1, \dots, T_k).$$

Now we also consider a Steiner string  $S_{\text{steiner}}$  for  $S_1, \dots, S_k$ . By definition we know that

$$\text{steiner}(S_{\text{steiner}}, S_1, \dots, S_k) \geq \text{steiner}(S, S_1, \dots, S_k).$$

Applying (b) to string  $S_{\text{steiner}}$  we get a further multiple alignment  $R_1, \dots, R_k$  for  $S_1, \dots, S_k$  with consensus line  $R$  and

$$\text{consensus}(R, R_1, \dots, R_k) \geq \text{steiner}(S_{\text{steiner}}, S_1, \dots, S_k).$$

As the initial multiple alignment was optimal we also know that

$$\text{consensus}(T, T_1, \dots, T_k) \geq \text{consensus}(R, R_1, \dots, R_k).$$

Combining all inequalities we obtain

$$\begin{aligned} \text{steiner}(S, S_1, \dots, S_k) &\geq \text{consensus}(T, T_1, \dots, T_k) \\ &\geq \text{consensus}(R, R_1, \dots, R_k) \\ &\geq \text{steiner}(S_{\text{steiner}}, S_1, \dots, S_k) \\ &\geq \text{steiner}(S, S_1, \dots, S_k). \end{aligned}$$

Having equality in all states of the chain above, we conclude that  $S$  indeed is a Steiner string.

(d) is shown in exactly the same manner as (c).  $\square$

### 2.6.6 Generalization to Steiner Trees/Phylogenetic Alignment

Given a list of strings  $S_1, \dots, S_k$ , a phylogenetic tree scheme  $\wp$  for  $S_1, \dots, S_k$  is a rooted tree with  $k$  leaves to which  $S_1, \dots, S_k$  are assigned in a one-to-one manner. Note that in a phylogenetic tree scheme, non-leaf nodes are not labelled, so far, with strings. Stated differently, a phylogenetic tree scheme expresses some knowledge on the evolutionary history of species  $S_1, \dots, S_k$ , but without fixing how ancestors exactly looked like. Given a phylogenetic tree scheme  $\wp$  for strings  $S_1, \dots, S_k$ , a  $\wp$ -alignment consists of an assignment of strings to all non-leaf nodes of  $\wp$ . For every node  $u$  (including leaves) let  $\wp(u)$  denote the string attached to  $u$ . The score of a  $\wp$ -alignment is defined as the sum of all optimal alignment scores taken over set  $E(\wp)$  consisting of all pairs  $u, v$  of nodes with a link between them.

$$\text{score}(\wp) = \sum_{(u,v) \in E(\wp)} \sigma_{\text{opt}}(\wp(u), \wp(v)) \quad (2.12)$$

#### PHYLOGENETIC TREE ALIGNMENT

Given a scoring function  $\sigma$  and a phylogenetic tree scheme  $\wp$  for strings  $S_1, \dots, S_k$ , compute a  $\wp$ -alignment having maximum value  $\text{score}(\wp)$ .

### 2.6.7 Profile Alignment

All of the problems associated with multiple alignments are NP-hard as will be shown later (Chap. 5). For each of them a 2-approximation algorithm will be

presented (Chap. 6). There are other approaches to multiple alignments that admit efficient algorithms. These are based on a statistical view to multiple alignments. The simplest such approach works as follows. Given a multiple alignment  $T_1, \dots, T_k$  of length  $m$ , define its profile  $\pi$  as the string  $\pi_1, \dots, \pi_m$  of length  $m$  consisting of the following frequency vectors.

$$\pi_p(c) = \frac{\text{number of occurrences of } c \text{ in column } p}{m} \quad (2.13)$$

Thus, “character”  $p$  of the profile of a multiple alignment is the probability distribution of characters within column  $p$  of the multiple alignment. Being a string of vector characters, the notion of an alignment of a profile with a string  $S$  is well-defined. What has to be clarified is how such an alignment is scored. This is done as follows, for all  $p$  and characters  $x$ :

$$\begin{aligned} \sigma_{\text{profile,string}}(\pi_p, x) &= \sum_{\text{all characters } y} \pi_p(y) \sigma(y, x) \\ \sigma_{\text{profile,string}}(\pi_p, -) &= \sum_{\text{all characters } y} \pi_p(y) \sigma(y, -) \\ \sigma_{\text{profile,string}}(-, x) &= \sigma(-, x). \end{aligned} \quad (2.14)$$

#### STRING TO PROFILE ALIGNMENT

Given a scoring function  $\sigma$ , profile  $\pi$ , and string  $S$ , compute a maximum score string-to-profile alignment of  $\pi$  with  $S$ .

We can also align a profile  $\pi$  to another profile  $\rho$ . Scores must obviously be defined as follows.

$$\begin{aligned} \sigma_{\text{profile,profile}}(\pi_p, \rho_q) &= \sum_{\text{all characters } y, z} \pi_p(y) \rho_q(z) \sigma(y, z) \\ \sigma_{\text{profile,profile}}(\pi_p, -) &= \sum_{\text{all characters } y} \pi_p(y) \sigma(y, -) \\ \sigma_{\text{profile,profile}}(-, \rho_q) &= \sum_{\text{all characters } z} \rho_q(z) \sigma(-, z). \end{aligned} \quad (2.15)$$

#### PROFILE TO PROFILE ALIGNMENT

Given a scoring function  $\sigma$  and profiles  $\pi$ ,  $\rho$ , compute a maximum score profile-to-profile alignment of  $\pi$  with  $\rho$ .

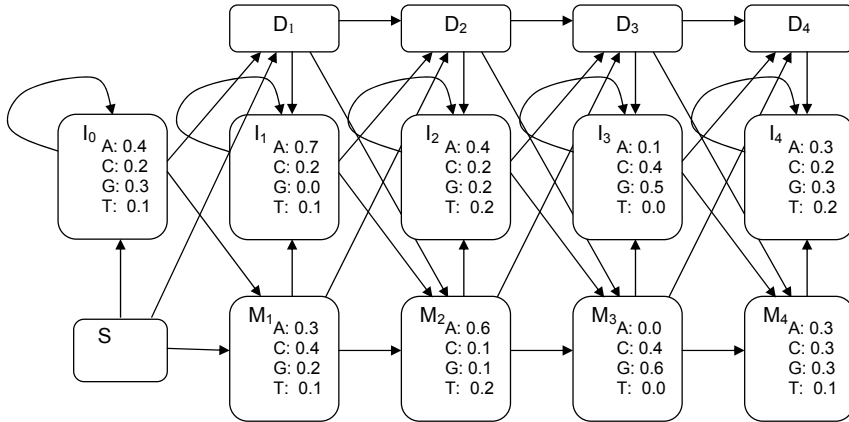
### 2.6.8 Hidden Markov Multiple Alignment

A profile of a multiple alignment can be seen as a statistical model that describes for each position (column of the alignment) the emission probabilities



for all characters (including spacing symbol). Emission probabilities at different positions are independent from each other. A major generalization is to consider statistical models with local interdependencies between adjacent positions. This gives considerably more flexibility to model evolutionary rules behind the generation of multiple alignments. Best suited are Hidden Markov models. Local interdependencies between adjacent positions are modelled by using internal states with transition probabilities between them and emission probabilities for characters associated with each internal state. Thus separating an internal behaviour of a Hidden Markov model from its observable outer behaviour is the source of increased flexibility, and also opens way to fit Hidden Markov models to data. Formal definitions of Hidden Markov models will be presented in Chaps. 3 and 7.

Figure 2.13 presents a model used for the generation of multiple alignments. Only state transitions having probability greater than zero are depicted. Example values for emission probabilities are indicated within each node of the graph. Node  $S$  indicates the start state.



**Fig. 2.13.** Profile Hidden Markov model for the generation of multiple alignments

Having in mind an unknown common ancestor string of known length  $n$  ( $n = 4$  in the example in Fig. 2.13) for all strings that are to be aligned (remember that an alignment partially expresses the evolutionary history of each string), the intended Hidden Markov model generates such “evolution protocols” by walking through a sequence of internal states. Available states are “match/mutate” states  $M_1, \dots, M_n$ , “insert” states  $I_0, I_1, \dots, I_n$ , and “delete” states  $D_1, \dots, D_n$ . Walking through state  $M_k$  means to match or mutate (via character emission) the  $k^{\text{th}}$  ancestor character, walking through state  $I_k$  means to insert fresh characters (via character emission) right of the position of the  $k^{\text{th}}$  ancestor symbol, walking through state  $D_k$  means to delete the  $k^{\text{th}}$  ancestor character (here no character is emitted, i.e. delete states are “mute”).

We illustrate how the model works with an example. Assume there are a couple of strings to be aligned. The number of characters in the unknown common ancestor string is usually estimated to be the arithmetical mean of the lengths of the considered strings. In our example, let this be 4. Further assume that for some of the strings there already exists a plausible multiple alignment (obtained by one of the methods described before or proposed by biologists on basis of plausibility). As an example, consider the following multiple alignment.

1		2	3	4			
-	C	-	-	G	-	A	-
-	-	-	-	G	C	A	A
-	C	-	C	-	A	-	G
-	T	-	-	T	C	C	T
A	C	G	G	G	C	-	C

Here, we have decided to interpret columns with a majority of characters over spacing symbols to be derived from one of the conjectured four characters of the unknown ancestor by either match, mutate, or delete events. On the other hand, columns with a majority of spacing symbols are assumed to be the result of insert events. On basis of these decisions we next introduce at all positions of the multiple alignment the correct state that led to the entry.

1		2	3	4				
-	M <sub>1</sub>	-	-	M <sub>2</sub>	D <sub>3</sub>	-	M <sub>4</sub>	-
-	D <sub>1</sub>	-	-	M <sub>2</sub>	M <sub>3</sub>	I <sub>3</sub>	M <sub>4</sub>	I <sub>4</sub>
-	M <sub>1</sub>	-	I <sub>1</sub>	D <sub>2</sub>	M <sub>3</sub>	-	M <sub>4</sub>	-
-	M <sub>1</sub>	-	-	M <sub>2</sub>	M <sub>3</sub>	I <sub>3</sub>	M <sub>4</sub>	-
I <sub>0</sub>	M <sub>1</sub>	I <sub>1</sub>	I <sub>1</sub>	M <sub>2</sub>	M <sub>3</sub>	-	M <sub>4</sub>	-

Now we estimate transition and emission probabilities of the Hidden Markov model as relative frequencies. For example, there are two transitions from state M<sub>1</sub> to state M<sub>2</sub>, one transition to state D<sub>2</sub>, and one transition to state I<sub>1</sub>. Thus, transition probabilities from state M<sub>1</sub> to other states are estimated as follows: 50% to M<sub>2</sub>, 25% to D<sub>2</sub>, 25% to I<sub>1</sub>. Emission probabilities are estimated similarly, for example, for state M<sub>4</sub> emission probability is estimated for character A as 40%, for C as 20%, for G as 20%, and for T as 20%.

So far, parameters of the model have been fixed on basis of available aligned strings. The general algorithmic problem behind parameter choice is training a model or fitting a model to available data. Parameters are chosen in such a way that the resulting model  $M$  has maximum probability of generating the training data. This is called maximization of model likelihood. The required formal concepts to make this precise are the probability that model  $M$  generates data  $D$ , briefly denoted  $P_M(D)$ , as well as the likelihood of model  $M$  given data  $D_1, \dots, D_k$  defined by

$$L(M|D_1, \dots, D_k) = \prod_{i=1}^k P_M(D_k). \quad (2.16)$$

Thus the following general problem is associated with every sort of adaptive system with parameters that may be optimally fitted to training data.

**HIDDEN MARKOV MODEL TRAINING**

Given the structure of a Hidden Markov model (i.e. number of states and connection structure, but without having fixed parameters) and observed data  $D_1, \dots, D_k$ , compute model parameters such that the resulting model has maximum likelihood of generating the data.

In Chap. 3 we will discuss the *Baum-Welch algorithm* which achieves a local maximization of model likelihood. Global maximization of model likelihood is a hard to solve problem.

Proceeding with our example, assume now that a further string CACGCTC has to be integrated into the alignment above. Referring to the chosen model, a most probable state sequence can be computed by the so-called *Viterbi algorithm*.

**HIDDEN MARKOV MODEL DECODING**

Given a Hidden Markov model and observed data  $D$ , compute a most probable state sequence that emits  $D$ .

Details of this algorithm are postponed to Chap. 3 on dynamic programming.

Assume here that as the most probable state sequence the algorithm proposes  $M_1I_1I_1M_2M_3I_3D_4I_4$ . Note that there are seven match/insert states corresponding to the seven characters of the string, and four match/delete states corresponding to the four characters of the unknown ancestor. The correct extension of the alignment thus looks as follows.

	1		2	3		4	
-	C	-	-	G	-	-	A
-	-	-	-	G	C	A	A
-	C	-	C	-	A	-	G
-	T	-	-	T	C	C	T
A	C	G	G	G	C	-	C
-	C	A	C	G	C	T	-

One says that a further string “has been aligned to the Hidden Markov model”.

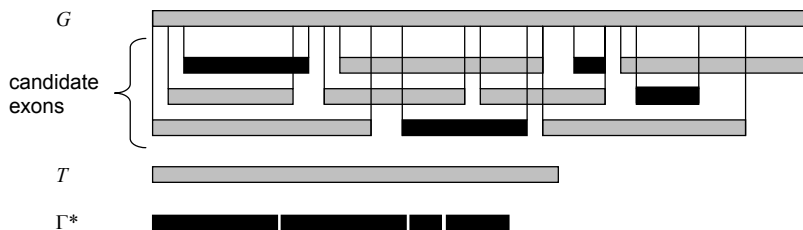
### 2.6.9 Example Heuristic for Obtaining Multiple Alignments

The following incremental construction of multiple alignments is plausible and widely used. Given strings  $S_1, \dots, S_k$  we first compute for every pair of indices  $i < j$  value  $\sigma_{\text{opt}}(S_i, S_j)$ . Then we use these values to apply a standard clustering procedure, as for example CLUSTALW, to group the most similar strings  $S_a$  and  $S_b$  into a group  $S_{ab}$ . We treat this new group  $S_{ab}$  as a single

new object, define scores  $\sigma_{\text{opt}}(S_i, S_{ab})$  as arithmetical mean of  $\sigma_{\text{opt}}(S_i, S_a)$  and  $\sigma_{\text{opt}}(S_i, S_b)$ , thus obtaining a tree with strings  $S_1, \dots, S_k$  attached to its leaves. This is a rather plausible guide tree which can be used to efficiently construct a multiple alignment.

## 2.7 Gene Detection

Having sequenced DNA strings the next important task is to identify functional units, e.g. genes or regulatory regions. We concentrate on gene detection. Biology contributes several indications for a substring to be part of an exon that belongs to a gene, e.g. absence of a stop codon, certain short strings that are characteristic for beginning or ending of an exon, or a higher frequency of CpG pairs in exon regions when compared to non-exon regions. Known frequencies of CpG pairs in exon versus non-exon regions, for example, can be used to probabilistically classify on a likelihood basis strings into exons and versus non-exons. Such approaches, as also the more sophisticated Hidden Markov model approaches, for predicting CpG-islands are based on standard methods of probability theory. However, they do not require sophisticated algorithmic know-how and we do not further discuss them in this book. Instead we concentrate on a different approach of *gene assembly* that will later be solved by dynamic programming. In gene assembly one assumes that a new strand  $G$  of DNA has been sequenced that is suspected to contain the exons of a gene producing a certain protein  $T$  (for simplicity assuming that it is given by its DNA code). We assume that well-known heuristics for predicting exons give us an ensemble of substrings  $E_1, \dots, E_b$  of  $G$  that are suspected to be valid exons. We assume that indeed all true exons are covered by the list of candidate exons  $E_1, \dots, E_b$ , whereas also non-exons may occur among this list. The task arises to select an ascending chain  $\Gamma$  of non-overlapping candidate exons (black shaded areas) whose concatenation  $\Gamma^*$  gives greater or equal optimal alignment score with  $T$  than all other selections of ascending chains (Fig. 2.14).



**Fig. 2.14.** Optimal alignment of  $\Gamma^*$  with  $T$  gives best value among all possible chains.

Letting  $T$  range over a database of known DNA codes of proteins, we may find one string  $T$  which can best be aligned with a concatenation of an ascending chain of candidate exons. Thus, with some confidence, we find out which protein is probably synthesized by the expected gene within string  $G$  and moreover, which substrings of  $G$  are the putative exons within that gene. Of course, simply trying out all ascending chains one by one is no good idea for obtaining an efficient algorithm. We will later see that clever usage of dynamic programming leads to an efficient solution (Chap. 3).

## 2.8 Genome Rearrangement

In Chap. 1 we discussed as a particular case of genome reorganization the operation of directed reversal. Encoding a sequence of  $n$  genes distributed on both strands of a chromosome as a permutation of signed numbers, i.e. numbers equipped with a sign (+ or -), a directed reversal takes a segment of the sequence, inverts ordering of numbers, as well as inverts sign for each number. As an example, signed permutation -4 +2 +6 -1 -3 +5 might be transformed by a single signed reversal into -4 +3 +1 -6 -2 +5, but also into +4 +2 +6 -1 -3 +5. Given two signed permutations of  $n$  numbers, one is interested in computing the least number of directed reversals required to transform one signed permutation into the other. This minimum number is called the (directed) *reversal distance* between two signed permutations. As can be easily verified, it defines a metric on signed permutations of  $n$  numbers. By renumbering genes and exchanging signs one can always assume that the target signed permutation is the sorted one +1 +2 ... + $n$ . Thus the problem simplifies to the task to compute the least number of directed reversals required to transform a given signed permutation into the sorted one. The problem is therefore also called the problem of sorting a signed permutation.

### SIGNED PERMUTATION SORTING

Given a signed permutation  $\pi$  of  $n$  numbers, compute the least number  $d(\pi)$  of directed reversals required to sort permutation  $\pi$ .

A similar problem arises for unsigned permutations, i.e. permutations of  $n$  numbers without + or - sign attached to them.

### UNSIGNED PERMUTATION SORTING

Given an unsigned permutation  $\pi$  of  $n$  numbers, compute the least number  $d(\pi)$  of undirected reversals required to sort permutation  $\pi$ .

It will turn out that these problems, though looking quite similar, are radically different from a complexity point of view. One of them admits an efficient

algorithm, the other one is NP-hard (see the series of papers [41, 42, 43] and also [15]). To give the reader a first impression of how complex the problem of sorting permutations is, sorting of a signed permutation of 36 signed numbers as shown in Fig. 2.15 requires 26 directed reversals. In Chap. 4 we will see that 26 is indeed the least number of directed reversals required for sorting the presented signed permutation.

-12	+31	+34	-28	-26	+17	+29	+04	+09	-36	-18	+35	+19	+01	-16	+14	+32	+33	+22	+15
-11	-27	-05	-20	+13	-30	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+20	+05	+27	+11	-15	-22	-33	-32	-14	+16	-01	-19	-35	+18	+36	-09	-04	-29	-17	+26
+28	-34	-31	+12	+13	-30	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	-16	+14	+32	+33	+22	+15	-11	-27	-05	-20	-19	-35	+18	+36	-09	-04	-29	-17	+26
+28	-34	-31	+12	+13	-30	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	-16	-15	-22	-33	-32	-14	-11	-27	-05	-20	-19	-35	+18	+36	-09	-04	-29	-17	+26
+28	-34	-31	+12	+13	-30	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	-16	-15	-36	-18	+35	+19	+20	+05	+27	+11	+14	+32	+33	+22	-09	-04	-29	-17	+26
+28	-34	-31	+12	+13	-30	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	-16	-15	-14	-11	-27	-05	-20	-19	-35	+18	+36	+32	+33	+22	-09	-04	-29	-17	+26
+28	-34	-31	+12	+13	-30	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	-16	-15	-14	+31	+34	-28	-26	+17	+29	+04	+09	-22	-33	-32	-36	-18	+35	+19	+20
+05	+27	+11	+12	+13	-30	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	+26	+28	-34	-31	+14	+15	+16	+17	+29	+04	+09	-22	-33	-32	-36	-18	+35	+19	+20
+05	+27	+11	+12	+13	-30	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	+26	+28	+18	+36	+32	+33	+22	-09	-04	-29	-17	-16	-15	-14	+31	+34	+35	+19	+20
+05	+27	+11	+12	+13	-30	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	+26	+28	+29	+04	+09	-22	-33	-32	-36	-18	-17	-16	-15	-14	+31	+34	+35	+19	+20
+05	+27	+11	+12	+13	-30	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	+26	+28	+29	+30	-13	-12	-11	-05	-20	-19	-35	-34	-31	+14	+15	+16	+17	+18	
+36	+32	+33	+22	-09	-04	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	+26	+11	+12	+13	-30	-29	-28	-27	-05	-20	-19	-35	-34	-31	+14	+15	+16	+17	+18
+36	+32	+33	+22	-09	-04	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	+26	+27	+28	+29	+30	-13	-12	-11	-05	-20	-19	-35	-34	-31	+14	+15	+16	+17	
+18	+36	+32	+33	+22	-09	-04	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07			
+01	+26	+27	+28	+29	+30	+31	+34	+35	+19	+20	+05	+11	+12	+13	+14	+15	+16	+17	+18
+36	+32	+33	+22	-09	-04	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	+26	+27	+28	+29	+30	+31	+34	+35	+19	+20	+09	-22	-33	-32	-36	-18	-17	-16	-15
-14	-13	-12	-11	-05	-04	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	+26	+27	+28	+29	+30	+31	+22	-09	-20	-19	-35	-34	-33	-32	-36	-18	-17	-16	-15
-14	-13	-12	-11	-05	-04	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	+26	+27	+28	+29	+30	+31	+32	+33	+34	+35	+19	+20	+09	-22	-36	-18	-17	-16	-15
-14	-13	-12	-11	-05	-04	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	+26	+27	+28	+29	+30	+31	+32	+33	+34	+35	+36	+22	-09	-20	-19	-18	-17	-16	-15
-14	-13	-12	-11	-05	-04	-23	+10	+06	+03	+24	+21	+08	+25	+02	+07				
+01	+26	+27	+28	+29	+30	+31	+32	+33	+34	+35	+36	+22	-09	-24	-03	-06	-10	+23	+04
+05	+11	+12	+13	+14	+15	+16	+17	+18	+19	+20	+21	+08	+25	+02	+07				
+01	+26	+27	+28	+29	+30	+31	+32	+33	+34	+35	+36	+22	-09	-08	-21	-20	-19	-18	-17
-16	-15	-14	-13	-12	-11	-05	-04	-23	+10	+06	+03	+24	+25	+02	+07				
+01	+26	+27	+28	+29	+30	+31	+32	+33	+34	+35	+36	+08	+09	-22	-21	-20	-19	-18	-17
-16	-15	-14	-13	-12	-11	-05	-04	-23	+10	+06	+03	+24	+25	+02	+07				
+01	+26	+27	+28	+29	+30	+31	+32	+33	+34	+35	+36	+08	+09	-22	-21	-20	-19	-18	-17
-16	-15	-14	-13	-12	-11	-05	-04	-03	-06	-10	+23	+24	+25	+02	+07				
+01	+26	+27	+28	+29	+30	+31	+32	+33	+34	+35	+36	+08	+09	-22	-21	-20	-19	-18	-17
-16	-15	-14	-13	-12	-11	-05	-04	-03	-02	-25	-24	-23	+10	+06	+07				
+01	+02	+03	+04	+05	+11	+12	+13	+14	+15	+16	+17	+18	+19	+20	+21	+22	-09	-08	-36
-35	-34	-33	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	+10	+06	+07				
+01	+02	+03	+04	+05	+11	+12	+13	+14	+15	+16	+17	+18	+19	+20	+21	+22	-09	-08	-07
-06	-10	+23	+24	+25	+26	+27	+28	+29	+30	+31	+32	+33	+34	+35	+36				
+01	+02	+03	+04	+05	+06	+07	+08	+09	-22	-21	-20	-19	-18	-17	-16	-15	-14	-13	-12
-11	-10	+23	+24	+25	+26	+27	+28	+29	+30	+31	+32	+33	+34	+35	+36				
+01	+02	+03	+04	+05	+06	+07	+08	+09	+10	+11	+12	+13	+14	+15	+16	+17	+18	+19	+20
+21	+22	+23	+24	+25	+26	+27	+28	+29	+30	+31	+32	+33	+34	+35	+36				

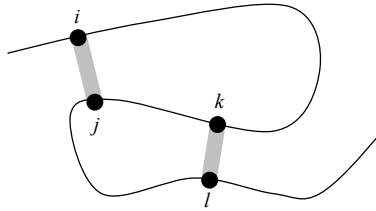
Fig. 2.15. Sorting an extended signed permutation

## 2.9 RNA Structure Prediction

RNA macromolecules are versatile and have the ability to fold into diverse structures. The common scaffold comprises base pairing between complementary bases. Formally speaking, a *structure*  $R$  over an RNA sequence  $S = S[1 \dots n]$  is defined as a set of base pairs:

$$R = \{(i, j) \mid 1 \leq i < j \leq n \wedge i \text{ and } j \text{ form an admissible base pair}\} \\ \text{such that } \forall (i, j), (i', j') \in R : i = i' \Leftrightarrow j = j'.$$

This definition demands that in an RNA structure  $R$ , a base can form a pair with at most one other base. Additionally, one commonly asks for every base pair  $(i, j)$  in  $R$  to be separated by at least one base, i.e.  $j - i \geq 2$ . In more advanced prediction methods, an even longer distance between paired bases  $i$  and  $j$  is demanded to serve for naturally occurring RNA structures. Structure  $R$  is called *free of pseudoknots* if there are no two pairs  $(i, j)$  and  $(k, l)$  in  $P$  such that  $i < k < j < l$ . Allowing overlapping base pairs results in a pseudoknot structure (Fig. 2.16).



**Fig. 2.16.** Pseudoknot structure with overlapping base pairs

A simple approach for RNA structure prediction from sequence is based on maximizing the number of base pairs.

**BASE PAIR MAXIMIZATION**

Given an RNA sequence  $S$ , compute a structure  $R$  with maximum number of base pairs.

More sophisticated optimization methods attempt to minimize the so-called free energy of a fold. Out of the exponential number of possible structures, an RNA molecule will fold into the one with minimum free energy. The state of *minimum free energy* is always determined by both enthalpic and entropic forces. It is clear that by considering only stable stems, RNA folding is very much oversimplified. In RNA, enthalpic terms arise from base pairing (stabilizing forces) and entropic terms from unstructured regions (destabilizing forces). RNA comprises various *structure elements*, e.g. stems, hairpin loops, bulges, internal loops, and multiloops, which we introduced in Sect. 1.6 (a formal definition is postponed to Sect. 3.6). All these motifs contribute to the

overall free energy and much experimental work has been done to determine their free (positive or negative) energy parameters. Stems act as stabilizing elements in folds, thus add a negative value to the free energy. Loop regions act as destabilizing elements in folds, expressed by a positive contribution to free energy. The result is a refined variant of the RNA structure prediction problem with the goal to minimize the free energy.

**FREE ENERGY MINIMIZATION**

Given an RNA sequence  $S$ , compute a structure  $R$  with minimum overall free energy.

Note that in a pseudoknot-free fold secondary structure elements occur either nested or side by side, but not interleaving. If we ignore pseudoknots, an RNA structure with maximum number of base pairs or minimum free energy can be computed in polynomial time using dynamic programming. We will introduce these successful and elegant approaches in Sect. 3.6. However, including more complicated motifs such as pseudoknots dampens the optimism of solving the RNA structure prediction problem. The general pseudoknot structure prediction problem is NP-complete and the proof for this based on [52] will be delivered in Sect. 5.5.6.

## 2.10 Protein Structure Prediction

### 2.10.1 Comparative Approaches Using Neural Networks

Predicting whether an amino acid belongs to an alpha-helix, beta-sheet or coil (loop) structure is a classical task for *feedforward neural networks* (also called *multi-layer perceptrons*). These are supplied with a certain number of amino acids surrounding the amino acid whose secondary structure class is to be predicted. Having enough such windows with known class of its centre amino acid, there is a good chance that a properly designed feedforward network may be trained to these data - and also generalise well on so far unseen data. Moving from secondary structures towards tertiary structure of proteins, an intermediate problem is contact map prediction, i.e. to predict for every pair of amino acids whether they are nearby in the natural fold of the protein. Whenever there is a contact predicted between amino acids  $A_i$  and  $A_j$ , it is reasonable to assume that both these amino acids belong to a core structure which is usually highly conserved throughout all members of a family  $\wp$  of homologous proteins. This should be observable in a high positive covariance with respect to mutations at sites  $i$  and  $j$  in protein family  $\wp$ . Moreover, characteristic covariance patterns should also be observable between all pairs of sites in the surroundings of sites  $i$  and  $j$ . This motivates an approach that presents to a neural network, among other parameters as the above discussed secondary structure class predictions, lots of standard covariance coefficients,



taken from a multiple alignment of a protein family  $\wp$ . Having now considerably more parameters as an input for a neural network than in the example before, taking care of the notorious problem of overfitting in neural learning becomes more urgent. Support vector machines are particularly well suited to solve this problem. Indeed, there are successful approaches to contact map prediction using support vector machines.

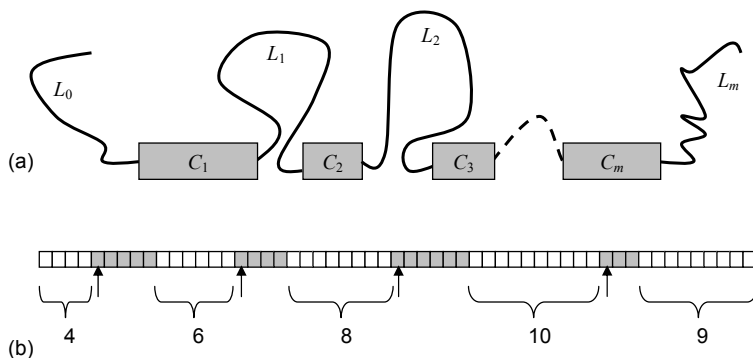
As the emphasis of this book lies more on traditional (exact) combinatorial algorithms, we do not further develop neural network techniques in the main chapters of this book, but postpone this theme to Chap. 7 on metaheuristics, and now switch back to a more classical combinatorial problem.

### 2.10.2 Protein Threading

Let  $S$  be a string of  $n$  amino acids describing the primary structure of a novel protein. Let a reference protein  $T$  of  $k$  amino acids be given with known tertiary structure consisting of a sequence of core segments (alpha or beta)  $C_1, \dots, C_m$  of fixed lengths  $c_1, \dots, c_m$  that are separated by loop segments  $L_0, L_1, \dots, L_{m-1}, L_m$  of lengths  $l_0, l_1, \dots, l_{m-1}, l_m$ . Thus, reference protein  $T$  is segmented into a concatenation of substrings

$$T = L_0 C_1 L_1 C_2 \dots C_m L_m \quad (2.17)$$

with loop and core segments of known lengths occurring in alternation as shown in Fig. 2.17.



**Fig. 2.17.** (a) Core structure of reference protein; (b) Threading into reference protein

Assume that length  $c_i$  of core segment  $C_i$  is strongly conserved in evolution, whereas length  $l_i$  of loop segment  $L_i$  varies within an interval  $[\lambda_i \dots \Lambda_i]$ . The task is to identify corresponding core segments  $C^1, \dots, C^m$  of lengths  $c_1, \dots, c_m$  within the novel protein  $S$  such that a certain distance measure

between  $C_1, \dots, C_m$  and  $C^1, \dots, C^m$  is minimized and, of course, constraints on the lengths of loop segments are respected. Let core segments  $C^1, \dots, C^m$  of fixed lengths  $c_1, \dots, c_m$  within protein  $S$  be defined by their start positions  $t_1, \dots, t_m$  within reference protein  $T$ . Such a list of start positions is called a *threading* of  $S$  into  $T$ . As an example, let core lengths 5, 4, 6, 3 and loop length intervals  $[3 \dots 7], [3 \dots 6], [7 \dots 9], [6 \dots 12], [7 \dots 11]$  be given (Fig. 2.17). Then, start positions 5, 16, 28, 44 of the shaded core segments define an admissible threading of protein  $S$  into  $T$ . Admissibility of a threading may be expressed by the following inequalities ( $i = 2, \dots, m-1$ ):

$$\begin{aligned} 1 + \lambda_0 &\leq t_1 \leq 1 + \Lambda_0 \\ t_i + c_i + \lambda_i &\leq t_{i+1} \leq t_i + c_i + \Lambda_i \\ t_m + c_m + \lambda_m &\leq n + 1 \leq t_m + c_m + \Lambda_m. \end{aligned} \quad (2.18)$$

Using this notation, the quality of a threading is measured by a function  $f(t_1, \dots, t_m)$ .

$$f(t_1, \dots, t_m) = \sum_{i=1}^m g(t_i) + \sum_{i=1}^{m-1} \sum_{j=i+1}^m h(t_i, t_j) \quad (2.19)$$

Here, term  $g(t_i)$  is an abbreviation for some function depending on core segment  $C_i$  of  $T$  and substring  $C^i(t_i)$  of  $S$  with length  $c_i$  that starts at position  $t_i$ . Furthermore,  $h(t_i, t_j)$  is an abbreviation for some function depending on corresponding substrings  $C_i$ ,  $C^i(t_i)$ ,  $C_j$ , and  $C^j(t_j)$ . *Local term*  $g(t_i)$  may be used as a measure of how good core segment  $C^i$  of  $S$  resembles core segment  $C_i$  of  $T$ , whereas *coupling term*  $h(t_i, t_j)$  may be used to express interactions between core segments. An example might be

$$g(t_i) = d_{\text{opt}}(C_i, C^i(t_i)) \text{ with } C^i(t_i) = S[t_i \dots t_i + c_i - 1]$$

with a distance function  $d$ . The protein threading problem now attempts to compute a threading  $t_1, \dots, t_m$  of protein  $S$  into the loop and core structure of reference protein  $T$  such that  $f(t_1, \dots, t_m)$  is minimized.

### PROTEIN THREADING

We refer to a fixed scoring  $f(t)$  for any threading  $t$  as described above. Given a novel protein  $S$  of length  $n$  and a reference protein  $T$  of length  $k$  with known core-loop structure, and also given intervals for admissible core lengths, determine an admissible threading  $t$  of  $S$  into the structure of  $T$  with minimum value  $f(t)$ .

In Chap. 3 we will see that this problem is efficiently solvable by a dynamic programming approach provided the evaluation function  $f(t)$  does not contain any coupling terms, whereas in Chap. 5 we show that the presence of simplest coupling terms in  $f(t)$  immediately leads to NP-hardness of the problem (see [50]).

## 2.11 Bi-Clustering

We start with a matrix of binary entries with  $n$  rows corresponding, for example, to  $n$  different microarray measurements, each involving the expression of  $m$  genes (columns of the matrix). We assume that measurements are taken in the context of a disease exhibiting a few different variants, each of them characterized by a different gene expression background. What we would like to have is a prediction algorithm telling us to which variant each measurement (row) belongs and which gene expression pattern characterizes measurements of a certain variant. Thus, this is on the one hand a problem of clustering measurements and on the other hand, of finding gene expression rules that characterize clusters. One can imagine various sorts of rules for the characterization of a cluster of measurements:

- Expression rule: characterize a cluster by a fixed subset of genes that must be expressed in measurements of the cluster.
- Expression/suppression rule: characterize a cluster by a fixed subset of genes that must be expressed and a fixed subset of genes that may not be expressed in measurements of the cluster.
- Arbitrary Boolean rule: characterize a cluster by validity of a Boolean formula on the expression/suppression states of the genes in a fixed set of genes in all measurements of the cluster.

It is expected that number of clusters as well as size of each cluster have to be restricted to make the result of clustering and characterization of clusters meaningful, for example by expecting  $k$  clusters with at least  $K$  elements within each cluster for suitably chosen numbers  $k, K$ . Also, admitted rules should be somehow restricted. Admitting, for example, every Boolean formula is surely no good idea, since it allows characterization of each cluster in an arbitrary clustering of measurements by simply using as a rule characterizing a considered cluster the Boolean formula in disjunctive normal form that simply enumerates all measurements in the cluster. We thus fall back, as an example, to the simplest sort of cluster rules by gene expression. Note that we look at the same time for clusters and gene expression rules for each cluster. This is different from the task to find, for example, for a partitioning into two classes a decision tree that separates classes. For such a situation with known clusters one could apply standard techniques of machine learning (e.g. version space procedure or ID3) for the inference of decision trees. Here, the different problem of simultaneously finding clusters and gene expression characterizations for each cluster has to be solved. As a first problem variant (the simplest one among lots of further ones) we discuss the following.

### MAXIMUM SIZE ONLY-ONES SUBMATRIX

Given binary matrix  $M$  and lower bound  $b$ , is there submatrix of size at least  $b$  (size of a matrix = number of entries) that contains only entries '1'.

As will be shown in Chap. 5, already this simply looking problem is NP-complete (shown in [63]).

## 2.12 Bibliographic Remarks

The most comprehensive treatment of algorithmic problems resulting from bioinformatics problems is surely Gusfield [31]. Various more special, and in parts rather intricate problems are treated in Pevzner [64].

Bioinformatics

Problem Solving Paradigms

Sperschneider, V.

2008, XVIII, 290 p. 208 illus., Hardcover

ISBN: 978-3-540-78505-7