CMPT441: Algorithms in Bioinformatics

Lecture Notes
by
Dr. Alexander Schönhuth

Contents

CHAPTER 1

Molecular Biology Primer

1.1. Life Science Basics

1665: Robert Hooke, originally a professor of geometry, discovered the first cells by means of a microscope.

1830s: Matthias Schleiden and Theodor Schwann found that all organisms were made of cells. Life sciences became cell sciences.
Cells have a life cycle: they are born, eat, replicate and die. Cells are the smallest subunits in an organism that can, given the necessary nutrients, live on their own. They do not necessarily depend on the rest of the organism.

On one hand, cells store all the information necessary for living and, on the other hand, they have the machinery required to manufacture and organize their components, to copy themselves and to kickstart its new offspring.

Despite the great diversity of cells in nature, there seem to be a few principles shared by all organisms. To the best of our actual knowledge, all life on this planet depends on three types of molecules: DNA, RNA and proteins.

1.2. Early Advances in Genetics

1830s: Schleiden and Schwann advanced their studies by discovering threadlike *chromosomes* in cell nucleii. (a *nucleus* is an important subunit in a certain class of cells).

1860s: Experiments of Gregor Mendel, an Augustinian monk, suggested the existence of *genes*. He observed that certain traits of garden peas were conserved over the generations.

1920s: Thomas Morgan discovered that fruit flies inherited certain traits from their ancestors. Moreover, he found that certain combinations of traits were more likely to appear although he also observed them separately. For example, flies having black bodies often appeared to have vestigial wings. There were, however, black-bodied flies having wings of proper size. Alfred Sturtevant, a student of his, constructed a "genetic map" where the distance between two genes was related to the likelihood of observing them simultaneously.

By the early 1940s geneticists understood that

- cells' traits were inherent in its genetic information,
- the genetic information was passed to the offspring and

- genetic information was organized into genes that resided on chromosomes.

Open questions were:

- What are chromosomes made of?
- What, in terms of the matter of chromosomes, characterizes a gene?
- How did the genes give rise to a cell's trait?

1.3. Related Advances in Biochemistry

By the early 1940s, noone observerd that the following facts were closely related to the geneticists' studies.

DNA

1869: Johann Friedrich Miescher isolated DNA (called "nuclein" by him) from the nuclei of white blood cells.

1900s: DNA was known to be a long molecule having a chemical "backbone" of phosphate groups and deoxyribose, a sugar having five carbon atoms. To each one of the phosphate-deoxyribose segments was attached one of four types of bases: adenine (A), cytosine (C), guanine (G) and thymine (T). A DNA molecule has a direction according to whether, at the first end of the chain, there is a free OH-group at the third carbon atom of the deoxyribose and, at the other end of the chain, there is an OH-group at the fifth carbon atom of the deoxyribose, or vice versa. The ends are referred to as the 3' and the 5' end of DNA and, accordingly, DNA is organized either in (5'-3')-direction or in (3'-5'). Hence, for a full description of the DNA molecule at hand, it suffices to indicate the sequence of the bases (e.g. GCAATTTA) and to specify the direction (say 5'-3').

It is just well justified that and this is how computer scientist see it, to perceive a DNA molecule as a word over the alphabet $\{A, C, G, T\}$.

DNA is, from a chemical point of view, extremly stable and inert. This may have been one of the reasons why DNA, though coming from nuclei where chromosomes had been observed (under the microscope, but not chemically specified) were suspected to be highly similar to synthetic polymers where the nucleotides followed a repetitive pattern like, say, TCGATCGATCGATCGA...

Proteins

1820: Henry Braconnot identified the first *amino acid*, glycine.

1820-1900s: Identification of all twenty amino acids as well as their chemical structure.

Early 1900s: Emil Hermann Fischer discovered that amino acids were linked together into linear chains thus forming *proteins*.

In these times, it was postulated that the properties of proteins were given by the composition and the arrangement of the amino acid chain. This is now accepted as the truth.

Unlike DNA, the three-dimensional structure of proteins is highly flexible, see below for the structure of DNA. The structure, in turn, is responsible for a protein's function. It soon became obvious that proteins are major building stones of a cell's machinery.

1.4. The Modern Genomic Era

The open questions from the end of section 1.2 were iteratively answered where first progress was made from the early 1940s on.

1941: George Beadle and Edward Tatum postulated "one gene - one protein" as a result of a famous experiment with the bread mould Neurospora. Neurospora cells were irradiated with x-rays. While Neurospora usually was able to survive in a medium of basic nutrients, irradiated cells could not. However, when vitamin B_6 was added to the medium, also the irradiated mutants went on living. Beadle and Tatum concluded that they had destroyed a gene which was responsible for the internal production of an *enzyme* (a protein that either activates or inhibits a cellular chemical reaction) which in turn triggered the internal production of vitamin B_6.

Despite being perfectly right with regard to a "gene" (still an undefined entity) being responsible for that enzyme, the postulation "one gene - one protein" did not hold true, see below.

1944: Oswald Avery finally established the missing link between genes and DNA. He fed mice with a mixture of both mutant and killed wild type cells of bacteria whose living wild type cells caused severe lung infections. Although neither mutants nor killed wild type alone caused an infection in mice, their combination did. He even detected living wild type cells in these mice. He correctly concluded that some of the genetic information of the killed cells had to be transferred to the mutants thus curing them from their genetic deficiencies. Subsequent experiments showed that only extracting the DNA from the killed wild type cells before injecting the mixture led to no complications for the mice any more. Hence the genetic information had to be carried by DNA molecules.

Two major observation from 1951 finally gave rise to what is referred to today as the beginning of the modern genomic era.

1951: Erwin Chargaff finds that there is a 1-1-ratio between A and T as well as C and G in a DNA molecule.

1951: Maurice Wilkins and Rosalind Franklin obtained sharp x-ray images of DNA that suggested a helical model for the structure of DNA.

However, these people did not come up with a reasonable structural model of DNA. Finally, in 1953, a major breakthrough and maybe the most popular contribution to modern molecular biology, was achieved.

1953: James Watson, an American genius, and Francis Crick, an English PhD student (at the age of 35) proposed the double helical structure model of DNA by means of Tinkertoy rods. By exploiting Chargaff's rule and Rosalind Franklin's images (a controversial issue being that they possibly illegally grabbed them from Rosalind Franklin who probably intended to publish them herself) they concluded that two strands of DNA had to be coiled around each other such that an A resp. C resp. G resp. T in one strand had to be paired with a T resp. G resp. C resp. A in strand number two. The second strand is aligned to the first one in reverse order (3'-5' if strand number one is considered to be 5'-3') and is called the *complementary* strand.

This model immediately suggested a replication procedure by simply unwinding the DNA and generating the complementary strands of the single strands.

However, it remained an open question how the DNA's information was finally translated to the enormous variety of different proteins. Therefore first observe that there are *prokaryotic* and *eukaryotic* cells, the latter being characterized by having a nucleus encapsulating the cell's DNA. It is known that all multicellular organisms are eukaryotic whereas most of the unicellular organisms (such as bacteria) are prokaryotic (there are, however, unicellular eukaryotes, e.g. some amoeba).

It became clear that, in eukaryotes, protein synthesis happened outside the nucleus, in the *cytoplasm*. Hence, the genetic information had to be carried from the nucleus to the cytoplasm.

mid 1950s: Paul Zamecki discovered that the protein synthesis in the cytoplasm happened with the help of ribosomes, large molecular complexes that contained *RNA*.

This suggested RNA as an intermediary agent for the transmission of information from the nucleus to the cytoplasm. RNA can be considered to be DNA where deoxyribose is replaced by ribose and thymine (T) is replaced by uracil (U). Despite the character of these changings being minor modifications, RNA is much more flexible and chemically more active than the inert and stable double helical structure.

1960: Benjamin Hall and Sol Spiegelman demonstrated that RNA forms duplexes with single-stranded DNA.

1960: Jerard Hurwitz and Samuel Weiss identified RNA polymerase, a protein complex which helps *transcribing* DNA to RNA by means of the following mechanism:

(1) DNA unwinds.
(2) RNA polymerase binds to DNA.
(3) DNA is transcribed to RNA nucleotide by nucleotide.

This process is called *transcription*.

It still is not fully understood how RNA polymerase knows where to bind to start transcription. It is known, however, that, in eukaryotes, the RNA transcript in a process called *splicing*. The RNA transcript is "attacked" by a huge molecular complex called *spliceosome*. Several consecutive pieces of the linear chain are cut out and the remaining pieces are put together. This results in RNA which is referred to as *messenger RNA* (or *mRNA* for short). The parts of the DNA which had been transcribed to the parts missing in mRNA are called *introns* whereas the parts which make up the mRNA are called *exons*. In *prokaryotes* the RNA transcript remains unchanged. Thus mRNA fully matches the corresponding DNA chain.

Finally, the definition of a *gene* is a piece of DNA which is transcribed to RNA. As described above, in eukaryotes a gene consists of exons and introns. Neither the genetic function of introns nor the function of segments of DNA which do not participate in the process of transcription at all is not yet fully understood. Therefore, they are sometimes referred to as *junk DNA*.

late 1960s: The *genetic code* was discovered. It means that *codons*, i.e. triplets of consecutive letters of DNA, code for an amino acid, also referred to as *triplet rule*.

See Fig. 3.2 and Table 3.2 in [**JP04**] for examples.

The last step in the production of proteins is called *translation*. In the ribosome, so called *transfer RNA* (*tRNA* for short) performs an elegant job. There are twenty types of tRNA where each one corresponds to an amino acid. They have a binding site to which the corresponding amino acids bind on one hand, and a complementary codon for getting attached to the mRNA on the other hand. The result is a growing polypeptide which in the end is a protein which fully corresponds to the piece of mRNA at hand. This last step from DNA to proteins is called *translation*.

Note that, because of *alternative splicing*, one gene can code for several proteins. Alternative splicing means that the result of splicing the RNA transcript is not unique. Sometimes introns are made mRNA, sometimes not. Alternative splicing also is an active area of research.

Here comes the **Central Dogma** of molecular biology:

DNA *makes* RNA *makes* proteins.

As a good computational biologist, you should at least be able to remember this when asked what you know about biology.

1.5. Evolution

A final question could be that for the definition of a *species*. A species is a maximal set of organisms such that its elements match each other in the sense of mating.

With time passing by, several things can happen to the *genome* (the genome is the entity of genetic information, that usually means the DNA, in an organism, the entity of RNA is also referred to as transcriptome and the proteins are referred to as the proteome) of an organism. Sometimes single nucleotides are exchanged or cut out, sometimes whole segments of DNA are exchanged or removed. Resulting genomes differ from their ancestor genomes but, nevertheless, may code for viable organisms. Resulting mutants may even be adapted better to their actual environment than their ancestors were and so permanent changes occur. If this results in a group of organisms which cannot mate with their ancestors (usually from several generations before, remember that DNA is a very stable molecule), this is called *speciation* as a new species has emerged.

CHAPTER 2

Regulatory Motifs in DNA sequences

2.1. Motivation

Fruit flies have a small set of *immunity genes* that get switched on when the organism gets infected by, say, viruses or bacteria. Turned on, the organism produces proteins that destroy the pathogen, usually curing the infection. The problem is the identification of these genes and to determine what triggers their activation. Imagine the following probable experiment:

(1) Infect the flies.
(2) Grind them up.
(3) Measure (e.g. with a DNA array) which genes had been switched on as an immune response.
(4) Find what could trigger the activation of those genes.

It turned out that many of these genes contained strings similar or equal to the sequence TCGGGGATTTCC (so called NF-κB binding sites). The mechanism is that so called *transcription factors* (a class of proteins) bind to the NF-κB binding sites and thus encourage RNA polymerase to start transcription.

General Problem 2.1 (Motif Finding). From a set of sequences find short subsequences shared by all sequences of the set. These short subsequences do not need to coincide completely.

2.2. The Motif Finding Problem

2.2.1. Problem definition.

Definition 2.2 (Alignment Matrix). Let $s = (s_1, ..., s_t)$ be an array of positions in t sequences of length n, that is, $1 \leq s_i \leq n - l + 1$ for all i and each s_i gives rise to the substring of length l in the i-th sequence which starts at position s_i. The alignment matrix $A(s) \in \{A, C, G, T\}^{t \times l}$ is defined by the (i, j)-th element to be the $s_i + j - 1$-th element of the i-th sequence.

Definition 2.3 (Profile matrix). The profile matrix $P(s) \in \mathbb{N}^{\{A,C,G,T\} \times l}$ is defined by

$$P(s)_{dj} := \text{card } \{i \mid A(s)_{ij} = d\}, d \in \{A, C, G, T\}.$$

DEFINITION 2.4 (Consensus Score). Let $M_{P(s)}(j)$ denote the largest count in column j of $P(s)$. The consensus score $CS(s)$ is defined by

$$CScore(s) := \sum_{j=1}^{l} M_{P(s)}(j).$$

Obviously

$$CScore(s) \in [\frac{lt}{4}, lt] \tag{2.1}$$

as $M_{P(s)}(j) \in [\frac{t}{4}, t]$.

DEFINITION 2.5 (Consensus String). The consensus string $CString(s) \in \{A, C, G, T\}^l$ is defined by

$$CString(s)_j := \operatorname*{argmax}_d P(s)_{dj}.$$

So, the rows of the alignment matrix are simply the subsequences starting at the position specified by s, the profile matrix counts the number of appearance of each nucleotide in a column of the alignment matrix, the consensus score is the sum of the maximal numbers found in each of the columns of the profile matrix and the consensus string is the sequence which refers to the these maximal counts. See Fig. 4.2, Table 4.2 and Fig. 4.3 for an illustration.

Put in a more applicable way, our problem gets:

Motif Finding Problem (MFP)
Given a set of DNA sequences of the same length n, find a set of l-mers (a $\{DNA, RNA, protein\}$-sequence of length l), one from each sequence that maximizes the consensus score.

Input: A $t \times n$-matrix of DNA (representing l sequences of length n), l (length of the pattern, i.e. the length of the subsequences to find)

Output: Array of t starting positions $s^* = (s_1^*, ..., s_t^*)$ such that

$$CScore(s^*) = \max_s CScore(s).$$

2.2.2. Solution. A brute force algorithm requires the examination of $(n - l + 1)^t$ starting positions s. In order to apply branch and bound techniques we organize the starting positions in a tree (see Figs. 4.6, 4.7, [**JP04**]). While traversing the tree in depth-first order we skip proceeding to the bottom if we observe the following.

DEFINITION 2.6 (Partial consensus score). Let $CScore(s, i)$ be the consensus score of the $(i \times l)$ alignment matrix from the first i sequences and $CScore^*(s, i)$ be the consensus score of the $((t - i) \times l)$ alignment matrix from the remaining $t - i$ sequences of the complete set of t sequences

Now we know that

$$CScore(s) = CScore(s,i) + CScore^*(s,i) \overset{(2.1}{\leq} CScore(s,i) + (t-i)l.$$

So, if $CScore(s,i)+(t-i)l$ is lower than or equal to the best score achieved at a leaf so far we can stop going deeper. This is exploited in the following algorithm, see Fig. 4.8, [**JP04**] for an illustration.

BranchAndBoundMedianSearch(DNA,t,n,l)

```
1: s ← (1, ..., 1)
2: bestScore ← 0
3: i ← 1
4: while i > 0 do
5:   if i < l then
6:     optimisticScore ← CScore(s, i) + (t − i)l
7:     if optimisticScore < bestScore then
8:       (s, i) ← next vertex on the same level
9:     else
10:      (s, i) ← next vertex according to depth-first search
11:    end if
12:  else
13:    if CScore(s) > bestScore then
14:      bestScore ← CScore(s)
15:      bestMotif ← s
16:    end if
17:    (s, i) ← next vertex according to depth-first search
18:  end if
19: end while
20: return bestMotif
```

2.3. The Median String Problem

The Motif Finding Problem is equivalent to the so called Median String Problem which is introduced in the following.

2.3.1. Problem definition.

Definition 2.7 ((Total) Hamming Distance). Given two l-mers $v = (v_1, ..., v_l), w = (w_1, ..., w_l)$ their Hamming distance is defined by

$$d_H(v,w) := \text{card } \{i \,|\, v_i \neq w_i\}.$$

Again, let $s = (s_1, ..., s_t)$ be an array of starting positions and $v = (v_1, ..., v_l)$ some l-mer. The total Hamming distance between v and s is defined in two steps. First, let

$$d_H(v,s) := \sum_{i=1}^{t} d_H(v, s_i)$$

where, by slight abuse of notation s_i is supposed to be the l-mer associated with starting position s_i. Then the total Hamming distance of v (with

respect to the given set of sequences) is given by

$$THD(v) := \min_s d_H(v, s).$$

Note that computing the total Hamming distance is simple, as

$$THD(v) = \sum_{i=1}^{t} \min_{s_i} d_H(v, s_i)$$

and $\min_{s_i} d_H(v, s_i)$ is easy to determine.

DEFINITION 2.8 (Median String). The median string v^* is defined by

$$THD(v^*) = \min_v THD(v)$$

that is, as the string minimizing the total Hamming distance.

Median String Problem (MSP)
Given a set of DNA sequences of the same length n, find a median string.

Input: A $t \times n$-matrix of DNA (representing l sequences of length n), l (length of the pattern, i.e. the length of the subsequences to find)

Output: A string $v^* \in \{A, C, G, T\}^l$ such that

$$THD(v^*) = \min_v THD(v).$$

Note that the two problems (MFP and MSP) are computationally equivalent, as

$$d_H(CString(s), s) = lt - Score(s)$$

and further

$$d_H(CS(s), s) = \min_v d_H(v, s).$$

Consequently

$$\underbrace{\min_s \min_v d_H(v, s)}_{MSP} = lt - \underbrace{\max_s CScore(s)}_{MFP}.$$

2.3.2. Solution. In complete analogy to the solution of the Motif Finding Problem, an algorithmic solution of MSP is given by a branch-and-bound technique in a search tree where here the leaves correspond to all 4^l possible l-mers. Internal nodes of the tree can be identified with prefixes of l-mers, see Fig. 4.9, [**JP04**] for an illustration.

Note now that if the total Hamming distance of a full l-mer cannot be smaller than that of its prefixes. Therefore, when finding that the total Hamming distance of an internal vertex is already greater than that of the best l-mer encountered so far one can skip traversing to the bottom of the tree. The insight results in the following algorithm:

BranchAndBoundMedianSearch(DNA,t,n,l)

```
1: s ← (1, ..., 1) #(1,...,1) corresponds to A...A
2: bestDist ← ∞
3: i ← 1
4: while i > 0 do
5:   if i < l then
6:     prefix ← nucleotide string corresponding to (s, i) := (s_1, ..., s_i) ∈
       {1, 2, 3, 4}^i # (s_1, ..., s_i) is an i-mer, i ≤ l
7:     optimisticDist ← THD(prefix)
8:     if optimisticDist > bestDist then
9:       (s, i) ← next vertex on the same level
10:    else
11:      (s, i) ← next vertex according to depth-first search
12:    end if
13:  else
14:    word ← nucleotide string corresponding to (s_1, ..., s_l)
15:    if THD(word) < bestDist then
16:      bestDist ← THD(word)
17:      bestWord ← word
18:    end if
19:    (s, i) ← next vertex according to depth-first search
20:  end if
21: end while
22: return bestWord
```

Finally note that the Median Search algorithm has a running time of $O(4^l nt)$ which usually compares favourably to the $O(ln^t)$ solution of MFP as t, the number of sequences involved often happens to be much larger than that of the length of the motifs (usually $6 \leq l \leq 12$).

CHAPTER 3

Genome Rearrangements

3.1. Motivation

While studying Waardenburg's syndrome, a genetic disorder causing hearing loss and pigmentary abnormalities such as two differently colored eyes, and a related disorder in mice it became clear that there are groups of genes in mice that appear in the same order like in the human genome. These genes are likely to be present in the same order in a common ancestor of humans and mice.

It seems that, in some ways, the human genome is just the mouse genome cut into about 300 large genomic fragements, called *synteny blocks*, that have been pasted together in a different order. Both genomic sequences are just two different shufflings of the ancient mammalian genome. E.g. chromosome 2 in humans is built from fragments of chromosomes 1,2,3,5,6,7,10,11,12,14,17 in mice.

A *genome rearrangement* is a change of gene ordering. It seems that fewer than 250 genome rearrangements have occurred since the divergence of humans and mice 80 million years ago.

General Problem 3.1. Find a series of rearrangements that transform one genome into another. An elementary rearrangement event usually is a *reveral* (or *inversion*), see Fig. 5.1, [**JP04**]. Find the series of events with the smallest number of elementary rearrangements (usually reversals) that explain the difference between two genomes.

This is not an easy problem, as brute force techniques do not apply for more than ten synteny blocks.

3.2. Problem setting

Definition 3.2 (Permutation). A *permutation* on a sequence of n numbers is a bijective map

$$\begin{array}{rccc} \pi: & \{1,...,n\} & \longrightarrow & \{1,...,n\} \\ & k & \mapsto & \pi_k \end{array}.$$

Definition 3.3 (Reversal). A *reversal* $\rho(i,j)$ is the permutation defined by

$$\rho(i,j)(k) = \begin{cases} k & k < i, k > j \\ j-k+i & i \leq k \leq j \end{cases}.$$

We write $\pi = \pi_1 ... \pi_n$ for a permutation π. In this sense

$$\rho(i,j) = 1...(i-1)\,\underbrace{j(j-1)...(i+1)i}\,(j+1)...n$$

and

$$\rho(i,j) \circ \pi = \pi_1 ... \pi_{i-1}\pi_j\pi_{j-1}...\pi_{i+1}\pi_i\pi_{j+1}...\pi_n.$$

In simple words, $\rho(i,j)$ reverses the order of the elements between i and j.

Reversal Distance Problem (RDP)
Given two permutations, find a shortest series of reversals that transforms one permutation into another.

Input: Permutations π, σ.
Output: A series of reversals $\rho_1 = \rho(i_1, j_1), ..., \rho_t = \rho(i_t, j_t)$ such that

$$\rho_t \circ ... \circ \rho_1 \circ \pi = \sigma$$

and t is minimal.

Definition 3.4 (Reversal distance). The $d(\pi, \sigma)$ of a shortest series ($= t$ in the problem) is called *reversal distance* between π and σ.

As biology tells nothing about a general overall order of genes, it is up to us to provide one. Hence, in practice, one of the permutations is usually taken to be the identity, leading to the

---

Sorting by Reversals Problem (SRP)
Given two permutations, find a shortest series of reversals that transforms one permutation into another.

Input: A permutation π.
Output: A series of reversals $\rho_1 = \rho(i_1, j_1), ..., \rho_t = \rho(i_t, j_t)$ such that

$$\rho_t \circ ... \circ \rho_1 = \pi$$

and t is minimal.

Analogously, $d(\pi) = d(\pi, id)$ is defined to be the *reversal distance* of π.

3.3. Solution

We explore a first solution strategy which illustrate that, despite the simple formulation, the problem is computationally hard.

Definition 3.5. Let π be a permutation. We define prefix (π) by

$$\text{prefix } (\pi) := \max_i \forall j \leq i : \pi_j = j$$

that is as the largest number up to which π already agrees with the identity.

Example 3.6.

$$\pi = \underbrace{123}_{sorted}\,654 \quad \Longrightarrow \quad \text{prefix } (\pi) = 3.$$

An algorithm which, finding a reversal that increases prefix (π) at each step, will finally terminate with a series of reversals ρ_i such that $\rho_t \circ ... \circ \rho_1 = \pi$. This is what the following greedy (greed in the sense of increasing prefix (π)) algorithm relies on.

SimpleReversalSort(π):

```
1: for i ← 1 to n − 1 do
2:    j ← position of element i in π (i.e. π_j = i)
3:    if j ≠ i then
4:       π ← ρ(i, j) ∘ π
5:       output π
6:    end if
7:    if π is the identity then
8:       return
9:    end if
10: end for
```

Example 3.7. Let $\pi = 612345$.

$$612345 \overset{\pi_2=1,\rho(1,2)}{\longrightarrow} 162345 \overset{\pi_3=2,\rho(2,3)}{\longrightarrow} 126345 \overset{\pi_4=3,\rho(3,4)}{\longrightarrow} 123645$$
$$\overset{\pi_5=4,\rho(4,5)}{\longrightarrow} 123465 \overset{\pi_6=5,\rho(5,6)}{\longrightarrow} 123456$$

However π could have been sorted in just two steps:

$$612345 \overset{\rho(1,6)}{\longrightarrow} 543216 \overset{\rho(1,5)}{\longrightarrow} 123456$$

So, both

$$\begin{aligned} \pi &= \rho(1,6) \circ \rho(1,5) \text{and} \\ \pi &= \rho(5,6) \circ \rho(4,5) \circ \rho(3,4) \circ \rho(2,3) \circ \rho(1,2). \end{aligned}$$

3.3.1. Approximation algorithms. The conclusion from example 3.7 is that SimpleReversalSort is not a correct algorithm. As efficient correct algorithms for the SRP problem are still unknown and brute force techniques are too expensive, we would, for instance, appreciate if the algorithm is efficient and, at least, approximates the solution to an acceptable degree. This is what the theory of approximation algorithms is concerned with.

Definition 3.8 (Approximation ratio). The *approximation ratio* of a minimization algorithm $\mathcal{A}$ on input π is defined to be

$$\max_{|\pi|=n} \frac{\mathcal{A}(\pi)}{OPT(\pi)}$$

where $\mathcal{A}(\pi)$ is the algorithm's solution and $OPT(\pi)$ is the optimal aolution. In case of a maximization problem the approximation ratio is defined to be

$$\min_{|\pi|=n} \frac{\mathcal{A}(\pi)}{OPT(\pi)}.$$

In terms of approximation, SimpleReversalSort is terrible, as the example from above tells that the approximation ratio is at least $\frac{n-1}{2}$ (extend the example to $\pi = n1...n-1$ for general n). Better strategies.are needed where the goal now is to find an approximation algorithm with a better approximation ration. As of the writing of [**JP04**], the best known ration was $\frac{11}{8}$.

We now present an alternative greedy strategy which, at least, leads to a ratio that is independent of the size of the permutation π. Greed here aims at maximizing the so called number of *breakpoints* in each iteration. For technical convenience, we extend every permutation $\pi = \pi_1...\pi_n$ on n numbers by putting $\pi_0 = 0$ and $\pi_{n+1} = n+1$, that is

$$\pi = \pi_0\pi_1...\pi_n\pi_{n+1} = 0\pi_1...\pi_n n+1.$$

Definition 3.9. Neigboring elements π_i, π_{i+1}, $0 \leq i \leq n$ are called *adjacency* if

$$|\pi_i - \pi_{i+1}| = 1$$

and *breakpoints* if not. We define $a(\pi)$ to be the number of adjacencies and $b(\pi)$ to be the number of breakpoints of π.

Example 3.10. Permutation 0213458769 has five adjacencies $(21, 34, 45, 87, 76)$ and four breakpoints $(02, 13, 58, 69)$.

Observe that

$$a(\pi) + b(\pi) = n + 1 \forall \pi$$

and that every reversal $\rho(i, j)$ can eliminate at most 2 breakpoints (one at position i and one at position j). So, if $d(\pi)$ is the reversal distance of π

$$d(\pi) \geq \frac{b(\pi)}{2}.$$

Therefore an immediate approach is

BreakpointReversalSort(π)

1: **while** $b(\pi) > 0$ **do**
2: choose reversal ρ minimizing $b(\rho \circ \pi)$
3: $\pi \longleftarrow \rho \circ \pi$
4: **output** π
5: **end while**
6: **return**

However, it remains unclear whether this algorithm delivers a better approximation ratio than SimpleReversalSort. Even worse, we do not even know whether BreakpointReversalSort terminates as it is unclear whether in each step there will be a reversal ρ such that $b(\rho \circ \pi) < \pi$.

Definition 3.11. A *strip* $\pi_i...\pi_j$ in a permutation is defined by

$$\forall i \leq k \leq j-1 : \quad \pi_i \text{ and } \pi_{i+1} \text{ are adjacent}$$

and

$$\pi_{i-1}, \pi_i \text{ and } \pi_j, \pi_{j+1} \text{ are breakpoints.}$$

In simple words, a strip is defined to be an interval between two consecutive breakpoints. Note that single element strips are allowed.

DEFINITION 3.12. A strip is said to be *increasing* if $\pi_{i+1} = \pi_i + 1$ and *decreasing* if $\pi_{i+1} = \pi - 1$ within the strip. For technical convenience, single element strips π_i are defined to be decreasing if $1 \leq i \leq n$ and increasing if $i \in \{0, n+1\}$.

THEOREM 3.13. *If π contains a decreasing strip then there is a reversal ρ such that*

$$b(\rho \circ \pi) < b(\pi).$$

PROOF. Choose the decreasing strip containing the smallest element k (e.g. in permutation $\underrightarrow{012}\underleftarrow{765}\,\underleftarrow{8}\,\underleftarrow{43}\,\underrightarrow{9}$ we choose the strip 43 as $k = 3$ is minimal in all decreasing strips).By choice of k, $k-1$ cannot belong to a decreasing strip. Moreover, $k-1$ terminates the increasing strip it belongs to (note that $k > 0$, as 0 always is part of an increasing strip). Therefore $k-1$ and k belong to two breakpoints (at the ends of both the increasing strip with $k-1$ and the decreasing strip with k). Reversing the segment between $k-1$ and k makes $k-1$ and k neighbors, hence we have reduced the number of breakpoints by at least one (e.g. by means of exchanging the corresponding segment in the permutation from above we obtain $\underrightarrow{01234}\,\underleftarrow{8}\,\underrightarrow{567}\,\underrightarrow{9}$ diminishing the number of breakpoints from 4 to 3). ◇

We conclude that BREAKPOINTREVERSALSORT could not work only in case that there are no decreasing strips (e.g. check $\underrightarrow{0}\underrightarrow{156}\underrightarrow{7}\underrightarrow{23}\underrightarrow{489}$). Therefore we come up with the following algorithm:

IMPROVEDBREAKPOINTREVERSALSORT(π)

```
1: while b(π) > 0 do
2:    if π has a decreasing strip then
3:       choose reversal ρ minimizing b(ρ ∘ π) # this may differ from the
         reversal of the theorem
4:    else
5:       choose a reversal ρ that flips an increasing strip
6:    end if
7:    π ⟵ ρ ∘ π
8:    output π
9: end while
10: return
```

THEOREM 3.14. IMPROVEDBREAKPOINTREVERSALSORT *(IBRS) has an approximation ratio of at most* 4*, independent of the length of the permutation.*

PROOF. IBRS reduces the number of breakpoints as long as π contains a decreasing strip, that is $b(\rho \circ \pi) < b(\pi)$ in this case. In a step where an increasing strip is flipped we at least have no increase of breakpoints, that is $b(\rho \circ \pi) \leq b(\pi)$. Hence IBRS eliminates at least one breakpoint every two

steps which translates to (we write $IBRS(\pi)$ for the number of reversals needed on π)

$$IBRS(\pi) \leq 2b(\pi). \tag{3.1}$$

Furthermore ($d(\pi)$ is the reversal distance or the optimal value achievable)

$$d(\pi) \geq \frac{b(\pi)}{2} \iff \frac{1}{d(\pi)} \leq \frac{2}{b(\pi)}. \tag{3.2}$$

In sum,

$$\frac{IBRS(\pi)}{d(\pi)} \overset{(3.1)}{\leq} \frac{2b(\pi)}{d(\pi)} \overset{(3.2)}{\leq} \frac{4b(\pi)}{b(\pi)} = 4.$$

$\diamond$

CHAPTER 4

Sequence Alignments

4.1. Motivation

Having found a new gene in the genome does not necessarily mean that one understands its function. The determination of function from sequence information alone is a fragile undertaking; any additional information accessible helps a lot in understanding function. Hence biologists usually search for similarities with genes of known function. A strinking example of this approach was that, in 1984, Russell Doolittle (maybe the first bioinformatician) discovered that a cancer-causing oncogene's sequence matched that of a normal gene. After having detected this similarity biologists became suspicious that cancer might be caused by a normal growth gene switched on at the wrong time. Another example is the advance in understanding cystic fibrosis where the responsible gene's sequence was found to be similar to that of a gene coding for *ATP binding* proteins which in turn shed light of the faulty mechanisms in cystic fibrosis. This chapter will be about the issue of defining quantities which measure the *biological* similarity of two genomic or protein sequences and the algorithms to compute these quantities.

General Problem 4.1. Given two (DNA,RNA,protein) sequences v, w compute a score $SimilarityScore(v, w)$ which accounts for the biological similarity of the two sequences. Biological similarity can be that proteins have similar functions and/or that the genes coding for them (in two different species) have a common ancestor protein from which they both stem.

4.2. The Manhattan Tourist Problem

As an introduction to the class of dynamic programming algorithms usually employed for computing similarity scores between biological sequences, we outline the *Manhattan Tourist Problem* whose strategy of solution is nearly identical to that of the alignment problems encountered later.

Definition 4.2. A *grid* is defined to be a weighted, directed, planar graph which is isomorphic to a grid such that the edges are always directed to the right or to the bottom. The *length* of a path is defined to be the sum of the weights on its edges. The upperleft corner of the grid is called *source* and the lowerright corner is called *sink*.

See Fig. 6.4, [**JP04**] for an example of a grid. Note that $outdegree(sink) = indegree(source) = 0$.

Manhattan Tourist Problem (MTP)
Find a longest path in a grid from source to sink.

Input: A grid G.
Output: A longest path in G from *source* to *sink*.

See Fig. 6.3, [**JP04**] for the eponymous example of tourists in Manhattan striving to maximize the number of sights while proceeding on a shortest way from 59th Street / 8th Avenue to 42nd Street / 3rd Avenue.

A brute force algorithm which examines all paths from source to sink is, already for moderate-sized grids, clearly intractable. Greedy techniques which, at each node of the grid, would opt for the longer of the two possible edges fail in general (Fig. 6.4, [**JP04**] is an example).

Therefore, instead of solving MTP directly, we solve the more general problem of finding the longest path from the source to an arbitrary vertex (i, j) (vertices of a $(m \times n)$-grid are indexed in the obvious way where the source corresponds to $(1, 1)$ and the sink to (m, n)). Albeit seemingly more difficult at first sight, this leads to an efficient dynamic programming strategy for solving the original problem when vertices (i, j) are ordered in an appropriate fashion.

We write s_{ij} for the length of a longest path from source to (i,j), $1 \leq i \leq m, 1 \leq j \leq n$.

First observe that computing the values s_{1j} and s_{i1} for all feasible i, j is easy (let $l(v, w)$ be the weight of the edge between grid nodes v, w):

$$s_{i1} = \sum_{k=2}^{i} l((k-1,1),(k,1)), s_{1j} = \sum_{k=2}^{j} l((1,k-1),(1,k))$$

as proceeding along the borders is the only admissible way. Furthermore observe, and this is the crucial point, that for $i, j \geq 2$

$$s_{ij} = \max \begin{cases} s_{i-1,j} + l((i-1,j),(i,j)) \\ s_{i,j-1} + l((i,j-1),(i,j)) \end{cases}$$

as the longest path to (i, j) has to come from either $(i-1, j)$ or $(i, j-1)$. So, having values $s_{i-1,j}, s_{i,j-1}$ readily at our disposal, it is easy to compute s_{ij}. Implementing this strategy leads to the following algorithm:

ManhattanTourist(m, n, weight function l)
1: $s_{11} \longleftarrow 0$
2: **for** $i \longleftarrow 2$ to m **do**
3: $\quad s_{i1} \longleftarrow s_{i-1,1} + l((i-1,1),(i,1))$
4: **end for**
5: **for** $j \longleftarrow 2$ to n **do**

6: $s_{1j} \longleftarrow s_{1,j-1} + l((1, j-1), (1, j))$
7: **end for**
8: **for** $i \longleftarrow 2$ to m **do**
9: **for** $j \longleftarrow 2$ to n **do**
10:

$$s_{ij} \longleftarrow \max \begin{cases} s_{i-1,j} + l((i-1, j), (i, j)) \\ s_{i,j-1} + l((i, j-1), (i, j)) \end{cases}$$

11: **end for**
12: **end for**
13: **return** $s_{n,m}$

Obviously, ManhattanTourist is of the order $O(mn)$ hence an efficient algorithm.

4.2.1. Longest paths in directed acyclic graphs. Imagine now that, instead of a grid, we have to deal with an arbitrary directed acyclic graph (DAG), that is a directed graph that contains no cycles. One can prove that DAGs have sources (vertices with indegree 0) and sinks (vertices with outdegree 0) where now sources and sinks do not have to be unique (grids are special cases of DAGs with unique source and unique sink). The generalization of the Manhattan Tourist Problem to arbitrary DAGs reads

---

Longest Path in a DAG Problem (LPDAGP)
Given a weighted DAG, a source node and a sink node, find a longest path between source and sink.

Input: A weighted DAG G with source and sink vertices.
Output: A longest path in G from *source* to *sink*.

Again, Manhattan may serve as an example, tourists now not only having the choice of streets and avenues, but also of the Broadway, see Fig. 6.5, [**JP04**].

In the case of general DAGs we write s_v to denote the longest path from the sink to vertex v and further denote u as a predecessor of v if there is a directed edge (u, v) from u to v. Finally, $P(v) \subset G$ is just the set of predecessors of v in the DAG G (see Fig. 6.6, [**JP04**] for an illustration).

In general, LPDAGP is then solved by an algorithm similar to ManhattanTourist, but here exploiting the fact

$$s_v = \max_{u \in P(v)} s_u + l((u, v))$$

where again $l((u, v))$ denotes the length (weight) of the edge from u to v. It is, however, not as easy as in the case of grids to provide a generic method by which the vertices of the DAG should be traversed. Nevertheless it can be assured that the vertices of a DAG can be traversed in an appropriate order (such that we know s_u for all predecessors u of a vertex v before trying

to compute s_v) by establishing *topological orderings* of DAGs.

DEFINITION 4.3 (Topological ordering). An ordering of vertices $\{v_1, ..., v_n\}$ of a DAG $G = (V, E)$ is called *topological* if

$$(v_i, v_j) \in E \quad \leftarrow \quad i < j.$$

It can be shown that topological orderings always exist. See Fig. 6.9, [**JP04**] for different topological orderings of a grid.

4.3. Edit Distance and Alignments

So far we have not yet defined what we mean when talking of "similarity" or "distance" of biological sequences. While being useful for motif analysis, Hamming distance is not an appropriate choice in this case.

If two sequences are similar in terms of evolution then they should not be separated by two many mutations: substitutions (replacing a nucleotide by another), insertions and deletions. For example, strings $ATATATAT$ and $TATATATA$ have maximal Hamming distance while, from an evolutionary vantage point, are rather similar.

Another problem with Hamming distance would be that it does not apply for strings of different length, at least not straightforwardly. However, for the comparison of biological sequences, a distance measure which can be applied to sequences of different length is needed.

In 1966, Vladimir Levenshtein introduced the following notion.

DEFINITION 4.4 (Edit distance). Let $v \in \mathcal{A}^n$, $w \in \mathcal{A}^m$ two strings over an alphabet $\mathcal{A}$ of possibly different lengths n, m. The *edit distance* ED between v, w is defined by

$$ED(v, w) := \min \#\text{edit operations between } v \text{ and } w$$

that is the minimal number of edit operations to transform v into w where edit operations are substitutions, insertions and deletions.

See Figs. 6.10, 6.11, 6.12, [**JP04**] for illustrations.

Levenshtein did not provide an algorithm together with his definition. It was only later that algorithms for that type of problem were discovered, (and rediscovered subsequently), for applications in automated speech recognition and, of course, molecular biology just to name a few of the applications. All algorithms given have in common that they rely on dynamic programming and are closely related to that of solving the problem of finding longest paths in a directed acyclic graph.

DEFINITION 4.5 (Alignment). An *alignment* of two strings $v \in \mathcal{A}^n, w \in \mathcal{A}^m$ of possibly different lengths n, m over an alphabet $\mathcal{A}$ is a two-row matrix such that the first row contains the characters of v and the second row those of w. Both strings can be interspersed.with blanks but two blanks in one column are not allowed.

As two blanks (a blank is usually written as "-") are not allowed to show up in one column, an alignment matrix may have at most $n+m$ columns (and has at least $\max\{m,n\}$ columns). An example of a (biologically realistic) alignment matrix for the two strings ATGTTAT, ATCGTAC is

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

DEFINITION 4.6 (Matches, mismatches, indels). Let $M \in (\mathcal{A} \cup \{-\}^{2\times k}$ be an alignment matrix of two strings $v \in \mathcal{A}^n, w \in \mathcal{A}^m$ (hence $\max\{n,m\} \leq k \leq m+n$). A *match* is a column where both rows contain the same letter, a *mismatch* is a column where letters (not blanks) are different and a column which contains a space is called *indel*. Indels are further divided into *insertions* (the space is in the first row) and *deletions* (space in the second row).

For example, the alignment matrix from above contains 5 matches, no mismatches and 4 indels of which 2 are insertions (columns 3 and 9) and 2 are deletions (columns 6 and 8).

We now embed the alignment of two strings $v \in \mathcal{A}^n, w \in \mathcal{A}^m$ into a $(n+1,m+1)$-grid with additional diagonal edges (edges from vertices (i,j) to $(i+1,j+1)$). Therefore we associate strings $N_v, N_w \in \mathbb{N}^k$ (k is the number of columns of the alignment matrix) of natural numbers to the strings v,w where $N_v(i), N_w(i)$ are the number of symbols of v,w present up to column i. For example,

$$N_{ATGTTAT} = 122345677, N_{ATCGTAC} = 123455667$$

relative to the alignment from above. The alignment is then visualized by aligning N_v, N_w position against position:

$$\binom{0}{0}\binom{1}{1}\binom{2}{2}\binom{2}{3}\binom{3}{4}\binom{4}{5}\binom{5}{5}\binom{6}{6}\binom{7}{6}\binom{7}{7} \tag{4.1}$$

where $\binom{0}{0}$ is introduced for technical reasons which become immediately clear in the following. Note first that, this way, an alignment can be perceived as a path through the $(n+1,m+1)$-grid with diagonal edges where each of the $\binom{i}{j}$ is associated to vertex (i,j) and we start counting at 0 auch that the source is identified with $(0,0)$. This is the reason for introducing $\binom{0}{0}$ in (4.1). Letters of $ATGTTAT$ are associated to transitions of rows where the i-th letter corresponds to the transition from row $i-1$ to row i. Accordingly, letters of $ATCGTAC$ correspond to column transitions. See Fig. 6.13 for a detailed illustration.

DEFINITION 4.7 (Alignment grid). The *alignment grid* of two strings $v \in \mathcal{A}^n, w \in \mathcal{A}^m$ is a $(n+1,m+1)$-grid with diagonal edges $(i-1,j-1), 1 \leq i \leq n, 1 \leq j \leq m$ where we start counting at 0 such that the source is identified with vertex $(0,0)$ and the sink with (n,m). Each transition of

rows $i-1 \to i$ corresponds to v_i, the i-th letter of v whereas each transition of columns $j-1 \to j$ corresponds to w_j the j-th letter of w.

Given an alignment grid of two strings v, w, one can identify each of the paths from source to sink with an alignment of the two strings v, w where each diagonal edge $((i-1, j-1), (i, j)$ encountered in the path corresponds to aligning v_i to w_j (resulting in either a match or a mismatch). Each horizontal edge $(i, j-1), (i, j)$ is identified with an insertion whereas each vertical edge $((i-1, j), (i, j))$ is identified with a deletion. See p. 170, [**JP04**] for illustrations.

4.4. Longest Common Subsequences

The simplest measure of similarity of two sequences is the length of the longest common subsequence shared by them where here a subsequence refers to any ordered sequence of characters from both strings (for example, AGCA and ATTA are both subsequences from ATTGCTA). Although this does not yet fully reflect biological similarity of two strings it comes very close to the problem of finding an alignment which, in terms of biology, really makes sense.

Definition 4.8 (Common subsequence). Let $v = v_1...v_n$ and $w = w_1...w_m$ be two sequences. A *common subsequence* is a sequence of positions in v,

$$1 \leq i_1 < i_2 < ... < i_k \leq n$$

and a sequence of positions in v

$$1 \leq j_1 < j_2 < ... < j_k \leq m$$

of the same length k such that

$$v_{i_t} = w_{j_t} \text{ for all } 1 \leq t \leq k.$$

For example, TCTA is a common subsequence of both ATCTGAT and TGCATA.

If $s(v, w)$ is the length of the longest common subsequence of v and w then $d(v, w) = n + m - s(v, w)$ corresponds to the minimal number of insertions and deletions needed to transform v into w which is closely related to the edit distance between the two strings.

Longest Common Subsequence Problem (LCSP)
Find the longest common subsequence of v and w.

Input: Two strings v, w. **Output:** The longest common subsequence of v and w.

A common subsequence of v and w now corresponds to a path in alignment grid which avoids any diagonal edges associated to mismatches. When assigning weight 1 to the remaining diagonal edges and 0 to the vertical

and horizontal ones this translates to finding the longest path from source to sink in the graph resulting from removing mismatch diagonal edges from the alignment grid. This in turn is just an instance of the LPDAGP where the DAG involved is easy to handle due its close similarity to a real grid. A solution to LCSP now becomes obvious. We denote the length of the longest common subsequence between $v_1...v_i$, the i-prefix of v, and $w_1...w_j$, the j-prefix of w, by s_{ij} and obtain

$$s_{ij} = \max \begin{cases} s_{i-1,j}(+0) \\ s_{i,j-1}(+0) \\ s_{i-1,j-1} + 1 \quad \text{if } v_i = w_j \end{cases}$$

where the first case corresponds to that v_i is not in the common subsequence considered (that is an deletion of v_i) and the second case means that w_j is not in the subsequence (an insertion of w_j). to an insertion of w_j. The third case finally means that $v_i = w_j$ which extends the common subsequence under consideration by one letter.

If one additionally attaches backtracking pointers to each node (i, j) of the grid which point to the node from which the maximal value originates (that is, one attaches one of the symbols $\uparrow, \leftarrow$ or $\nwarrow$ to each node corresponding to the the three cases from above) a simple backtracking procedure will finally return the longest common subsequence corresponding to the longest path. These insights result in two algorithms the first one of which employs the usual dynamic programming techniques to compute the length of the longest common subsequence while at the same time storing the backtracking pointers.

LongestCommonSubsequence(v,w)

1: $s_{11} \longleftarrow 0$
2: **for** $i \longleftarrow 0$ to n **do**
3: $\quad s_{i0} \longleftarrow 0$
4: **end for**
5: **for** $j \longleftarrow 1$ to m **do**
6: $\quad s_{0j} \longleftarrow 0$
7: **end for**
8: **for** $i \longleftarrow 1$ to n **do**
9: $\quad$ **for** $j \longleftarrow 1$ to m **do**
10:

$$s_{ij} \longleftarrow \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1 \quad \text{if } v_i = w_j \end{cases}$$

11:

$$b_{ij} \longleftarrow \begin{cases} \text{“}\uparrow\text{”} & s_{ij} = s_{i-1,j} \\ \text{“}\leftarrow\text{”} & s_{ij} = s_{i,j-1} \\ \text{“}\nwarrow\text{”} & s_{ij} = s_{i-1,j-1} + 1 \end{cases}$$

12: $\quad$ **end for**

13: **end for**
14: **return** $s_{n,m}$

The following recursive program then prints out the longest common subsequence which is reconstructed by means of the backtracking pointers stored in $b \in \{\text{“}\uparrow\text{”}, \text{“}\leftarrow\text{”}, \text{“}\nwarrow\text{”}\}^{(n+1)\times(m+1)}$.

PrintLongestCommonSubsequence(v, b, i, j)

1: **if** $i = 0$ or $j = 0$ **then**
2: **return**
3: **end if**
4: **if** $b_{ij} = \text{“}\nwarrow\text{”}$ **then**
5: PrintLongestCommonSubsequence$(v, b, i-1, j-1)$
6: **print** v_i
7: **else if** $b_{ij} = \text{“}\uparrow\text{”}$ **then**
8: PrintLongestCommonSubsequence$(v, b, i-1, j)$
9: **else**
10: PrintLongestCommonSubsequence$(v, b, i, j-1)$
11: **end if**

See Fig. 6.14, [**JP04**] for illustrations of the algorithmic procedure. Finally note that the minimal number of insertions and deletions $d(v, w)$ needed for transforming v into w can be computed according to the initial conditions $d_{i0} = i, d_{0j} = j$ for $1 \le i \le n, 1 \le j \le m$ using the recurrence

$$d_{ij} = \min \begin{cases} d_{i-1,j} + 1 & \\ d_{i,j-1} + 1 & \\ d_{i-1,j-1} & \text{if } v_i = w_j \end{cases}.$$

4.5. Global Sequence Alignment

Rewarding matches with 1 and not penalizing indels does not reflect the biological reality. In order to come up with a more useful model for scoring, we first extend the k-letter alphabet $\mathcal{A}$ (of nucleotides or amino acids) to include the gap character "-". Then we consider an arbitrary $(k+1)\times(k+1)$ *scoring matrix* δ. If, in an aligment of two sequences, letter x from the first sequence is paired with letter y from the other sequence (x and/or y may be "-" for indels in the alignment) we will score the pairing with $\delta(x, y)$ from the scoring matrix.

Definition 4.9 (Alignment score). Let $v \in \mathcal{A}^n, w \in \mathcal{A}^m$ be two sequences over an alphabet $\mathcal{A}$ of size k and $A \in (\mathcal{A} \cup \{\text{“}-\text{”}\}^{2\times l}$ an alignment (remember the definition of an alignment as a $2\times l$-matrix where $\max\{m, n\} \le l \le m+n$ was the length of the alignment). Let further δ a scoring matrix. Then the *alignment score* of the alignment A is defined by

$$\sum_{i=1}^{l} \delta(A_{1l}, A_{2l}).$$

Global Alignment Problem
Find the best (high-score) alignment of two strings under a given scoring matrix.

Input: Strings v, w and a scoring matrix δ.
Output: An alignment of v and w whose alignment score as given by the matrix δ is maximal among all possible alignments of v and w.

This amounts to finding a longest path in the alignment grid where diagonal edges $((i-1, j-1), (i, j))$ are weighted by values $\delta(x, y)$ where x corresponds to the letter from row transition $i - 1 \to i$ and y correspond to the letter from column transition $j-1 \to j$. Horizontal edges $((i, j-1), (i, j))$ are labeled with $\delta(-, y)$ where y is letter according from column transition $(j-1, j)$ wheras vertical edges $((i-1, j), (i, j)$ are labeled with $\delta(x, -)$, x being from row transition $i - 1 \to i$.

The corresponding recurrence for computing scores s_{ij} of optimal alignments between the i-prefix of v and the j-prefix of w now is

$$s_{ij} = \max \begin{cases} s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \\ s_{i-1,j-1} + \delta(v_i, w_j) \end{cases} .$$

4.6. Scoring matrices

While scoring matrices δ for DNA sequence comparison usually are simply defined by having 1's on the diagonal (apart from $\delta(-, -)$, accounting for matches) and having a constant value μ for mismatches and σ for indels ($\delta(-, -)$ is usually set to $-\infty$, as corresponding edges in the alignment graph do not exist), there are more sophisticated matrices for aligning proteins. Common examples of scoring matrices for proteins are *point accepted mutations* (PAM) and *block substitution* (BLOSUM). Entries of these matrices reflect the frequencies with which amino acids are replaced in evolutionarily related sequences.

For example, let a *PAM unit* be the amount of time in which an "average" protein mutates 1% of its amino acids (replacing them by other amino acids, deleting them or introducing insertions). Now take a set of alignments of proteins (determined from true biological knowledge) where, in each of the alignments, one percent of the amino acids has been mutated. Let now $f(i, j)$ be the total number of times amino acids i and j are aligned to each other in these alignments, divided by the total number of aligned positions. Let further $f(i)$ be the frequency of amino acid i in all the proteins from the set of base alignments. Then define $g(i, j) = f(i, j)/f(i)$ which is the probability that an amino acid i mutates into amino acid j within 1 PAM

unit. Entry (i, j) the PAM_1 matrix is then defined as

$$\delta_{PAM_1}(i,j) = \log \frac{f(i,j)}{f(i)f(j)}$$

where $f(i)f(j)$ stands for the frequency of aligning amino acid i with amino acid j simply by chance.
The PAM_n matrix can then be defined as $(PAM_1)^n$, that is as an n-time application of PAM_1.

4.7. Local Sequence Alignments

In biology one is often interested in finding similar subsequences of two sequences, rather than aligning the entire sequences. An example are *homeobox* genes which regulate embryonic development. Homeobox genes are present in a large variety of species and their property is usually given by a *homeodomain* which is responsible for their function. While homeodomains are highly conserved among different species, the entire sequences of homeobox genes are not. More generally, protein researchers are often interested in finding *domains* that are shared by two proteins where a domain usually is a rather short subsequence of a protein determining its function.

Therefore, there are good reasons to seek for a method which detects highly similar subsequences rather than determining the similarity of the sequences as a whole. This corresponds to finding the best *local alignment* that is an alignment of subsequences of two sequences yielding maximal score, see Fig. 6.16, [**JP04**] for an example.

Local Alignment Problem:
Find the best local alignment between two strings.

Input: Strings v, w and a scoring matrix δ.
Output: Substrings of v and w whose global alignment score as given by the matrix δ is maximal among all global alignments of all substrings of v and w.

While it seems, at first sight, to be more complex, the local alignment problem has an astonishingly simple and elegant solution (brought up by Temple Smith and Michael Waterman): we simply add edges from the source to each node (i, j) and from each node (i, j) to the sink and assign weights 0 to them and proceed as always by determining the longest path from source to sink in the resulting directed acyclic graph (see Fig. 6.17, [**JP04**]). This amounts to the recurrence

$$s_{ij} = \max \begin{cases} 0 \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \\ s_{i-1,j-1} + \delta(v_i, w_j) \end{cases} .$$

The largest score s_{ij} found in the alignment grid then coincides with s_{mn} (according to the recurrence for s_{mn} which additionally takes into account all sink edges).

4.8. Alignment with Gap Penalties

Mutations are usually caused by errors in DNA replication. Nature frequently deletes or inserts entire substrings as opposed to deleting or inserting single nucleotides. A *gap* in an alignment is defined to be a contiguous sequence of spaces in one of the rows. Penalizing gaps of length x greater than one simply the same way as you would penalize x gaps of length 1 is too much of a punishment with regard to evolution.
Therefore we define *affine gap penalties* where a gap of length x is scored as $-\rho - \sigma x$ where $-\rho$ punishes the introduction of a gap and σ is a parameter which accounts for punishing the extension of gaps. Typically, ρ is large while σ is small thereby accounting for the facts described above.

The problem of finding optimal alignments with affine gap penalties translates to adding all possible vertical and horizontal edges to the alignment grid graph, labeling these vertical and horizontal edges of length x by the weights $-\rho - \sigma x$ and finding the longest path from source to sink like always. It does not matter whether you want to find global or local alignments; in both cases the modified alignment grid remains a directed acyclic graph. There is one problem remaining: the running time for finding longest paths in DAGs with dynamic programming depends on the amount of edges in it. Having m edges in general results in a running time of $O(m)$. By introducing affine gap penalty edges we increase the number of edges to being of order n^3 where n is the longer of the two string lengths. However, the introduction of the following three recurrences will keep the running time of the problem at the desirable order of $O(n^2)$:

$$s_{ij}^{\downarrow} = \max \begin{cases} s_{i-1,j}^{\downarrow} - \sigma \\ s_{i,j-1} - (\rho + \sigma) \end{cases}$$

$$s_{ij}^{\rightarrow} = \max \begin{cases} s_{i-1,j}^{\rightarrow} - \sigma \\ s_{i,j-1} - (\rho + \sigma) \end{cases}$$

$$s_{ij} = \max \begin{cases} s_{ij}^{\uparrow} \\ s_{ij}^{\rightarrow} \\ s_{i-1,j-1} + \delta(v_i, w_j) \end{cases}$$

The variable $s_{ij}^{\downarrow}$ computes the score of alignment between i-prefix of v and j-prefix of w ending with a deletion (a gap in w) and $s_{ij}^{\rightarrow}$ computes the score for aligning the prefixes ending with an insertion (i.e. a gap in v).

The whole process can be visualized like acting on three different DAGs which are interconnected by edges (see Fig. 6.18, [**JP04**]).

4.9. Multiple Alignments

Simultaneous comparison of biological sequences makes sense as, for example, several proteins can have a common ancestor. Multiple alignments of proteins of the same ancestral origin can yield easy-to-handle sequential explanations of the function they share. The multiple alignments are then subsequently used to construct probabilistic models (such as hidden Markov models, treated in later sessions) which can in turn be used to assign new sequences to "families" of sequences, that is, to sets of sequences sharing the same function and/or having a common ancestor. See Fig. 6.1, [**DEKM98**], Fig. 6.19, [**JP04**] for usual ways of depicting multiple alignments.

Let now $v_1, ..., v_k$ be k strings of length $n_1, ..., n_k$ over an alphabet $\mathcal{A}$. Let $\mathcal{A}' := \mathcal{A} \cup \{-\}$ be the extended alphabet containing the gap character $-$.

DEFINITION 4.10. A *multiple alignment* of $v_1, ..., v_k$ is a $k \times n$-matrix A where $n \geq \max_i n_i$. Each element of the matrix is of $\mathcal{A}'$ that is, $A \in \mathcal{A}'^{k \times n}$ and each row contains the characters of v_i in order (interspersed with $n - n_i$ gap characters). Each column must contain one of the characters of one of the v_i (hence $n \leq \sum_i n_i$).

DEFINITION 4.11. Let A be a multiple alignment. The *consensus string* of A is the string of the most common characters occuring in each column of A.

Let δ be a very general scoring function that assigns a value to each element of $\mathcal{A}'^k$ thus assigning a score to each column of a multiple alignment. The problem of finding a multiple alignment which yields the best score in the usual meaning of summing up scores assigned to columns according to δ amounts to constructing a k-dimensional grid (a graph where vertices are labeled $(m_1, ..., m_k), 1 \leq m_i \leq n_i$ having edges connecting all vertices $(m_1, ..., m_k)$ to vertices $(l_1, ..., l_k)$ such that $l_j - m_j \in \{0, 1\}$) and assigning weights to the edges according to δ, that is, an edge $((m_1, ..., m_k), (l_1, ..., l_k))$ gets assigned the value $\delta(a_1, ..., a_k)$ where $a_i = (v_i)_{l_i}$ if $l_i - m_i = 1$ and $a_i =' -'$ if $l_i = m_i$. A k-dimensional grid is a directed acyclic graph. Therefore, the usual dynamic programming techniques apply. See Figs. 6.20,6.21, [**JP04**] for illustrations.

As an example, we present the case of only three sequences v_1, v_2, v_3 to be aligned, that is, $k = 3$. In the grid, we would like to compute the lengths s_{m_1,m_2,m_3} of the longest paths to vertices (m_1, m_2, m_3) which is done

according to the recurrence relation

$$s_{m_1,m_2,m_3} = \max \begin{cases} s_{m_1-1,m_2,m_3} & +\delta((v_1)_{m_1}, -, -) \\ s_{m_1,m_2-1,m_3} & +\delta(-, (v_2)_{m_2}, -) \\ s_{m_1,m_2,m_3-1} & +\delta(-, -, (v_3)_{m_3}) \\ s_{m_1-1,m_2-1,m_3} & +\delta((v_1)_{m_1}, (v_2)_{m_2}, -) \\ s_{m_1-1,m_2,m_3-1} & +\delta((v_1)_{m_1}, -, (v_3)_{m_3}) \\ s_{m_1,m_2-1,m_3-1} & +\delta(-, (v_2)_{m_2}, (v_3)_{m_3}) \\ s_{m_1-1,m_2-1,m_3} & +\delta((v_1)_{m_1}, (v_2)_{m_2}, (v_3)_{m_3}) \end{cases}$$

Unfortunately, in case of k sequences the running time of a dynamic programming algorithm based on generalized recurrenced like that from above evaluates as $O(n^k)$.

4.9.1. Progressive multiple alignments. A straightforward solution to the runtime problem from above is to first compute all pairwise alignments of the $v_1, ..., v_k$ (there are $\binom{k}{2}$ many) and then, for example, iteratively add sequences to the best alignment found. Algorithms that rely on heuristics like this, do not necessarily find the best alignment possible (see Fig. 6.21, [**JP04**] for examples of sequences which yield incompatible pairwise alignments), but they are fast and efficient, and in many cases the resulting alignments are reasonable. Existing algorithms differ in several ways:

(1) in the way that they choose the order to do the alignment,
(2) in whether the progression involves only alignments of sequences to a single growing alignment or whether subfamilies are built up on a tree structure and, at certain points, alignments are aligned to alignments,
(3) and in the procedure used to align and score sequences or alignments against existing alignments.

One of the most widely used implementation for progressive multiple alignments is CLUSTALW which first computes all pairwise alignment scores between sequences. From the resulting symmetric distance matrix, a guide tree is constructed by a neighbor-joining clustering algorithm. Then, it progressively aligns "nodes" in the tree in order of decreasing similarity. This involves sequence-sequence, sequence-alignment and alignment-alignment alignments. Several heuristics contribute to its accuracy.

4.10. Space-efficient Sequence Alignment

When comparing long DNA fragments, runtime is not as much an issue as space requirements are. Remember that both the runtime of aligning sequences of length n and m and space complexity were of order $O(nm)$. However, computing similarity scores for two sequences is only of runtime order $O(n)$ (where here n could be the length of the shorter sequence) as computing intermediate scores s_{ij} for an alignment of the i-prefix of one sequence against the j-prefix of the other sequence, while traversing the alignment grid columnwise such that the rows are indexed by the shorter sequence, only needs to score values of the preceding column which yields a

maximum requirement of two columns, or $O(2n)$ space.

When one is interested in the longest path (i.e. the alignment), however, one has to store the complete backtracking pointer table, causing the $O(mn)$ requirement. By replacing the backtracking pointer table by another technique we can indeed decrease the space complexity to $O(n)$.

DEFINITION 4.12. Given an alignment grid, we define $length(i, j)$ to be the longest path from source to sink passing through vertex (i, j).

A first important observation is that

$$\forall j : \quad s_{mn} = \max_i s_{ij} \tag{4.2}$$

as there must be an i such that the longest path overall (which we are interested in) must pass through a vertex (i, j). We further define s_{ij}^{rev} to be the length of the longest path from vertex (i, j) to the sink. This can be computed as an alignment in the alignment grid where all edges have been reversed. We find that

$$length(i, j) = s_{ij} + s_{ij}^{rev}.$$

Computing all $length(i, j)$ values requires time equal to the size of the left rectangle (from column 1 to column j) for computing values s_{ij} plus time equal to the size of the right rectangle from column j to n, that is time proportional to $(nj + (m - j)n) = mn$. To store the $length(i, j)$ values for one column needs $O(n)$ space (from the computation of the s_{ij} and s_{ij}^{rev}).

See Fig. 7.3, [**JP04**] for an illustration of the following strategy for a space-efficient algorithm for computing the longest path in an alignment grid. We start with column $m/2$ and compute all $length(i, m/2)$ values. We pick the maximum $length(i^*, m/2)$ of these values and have, because of (4.2), found the vertex in column $m/2$ which is traversed on a longest path from source to sink. Now we split the problem into two subproblems in one of which $(i^*, m/2)$ is the sink and $(0, 0)$ remains the source and in the other one of which $(i^*, m/2)$ is the source whereas (m, n) remains the sink. We now, in both of the problems, look for vertices at "half of the problem" where a longest path from the sources to the sinks pass through thereby finding two more vertices which are traversed on a longest path for the original problems. This needs $(O(2n/2) = O(n)$ space and time proportional two the area of two subrectangle referring to the upper left corner and the lower right corner, this being of $O(mn/2)$ size. We iterate this such that we now split up the two subproblems into four even smaller subproblems and do the same thing with them. Running time is always proportional to the area of the rectangles given by the subproblems and this area is halved in each iteration. So, to find all middle vertices (and hence the longest path) for the whole table one needs time proportional to

$$mn + \frac{mn}{2} + \frac{mn}{4} + ... \leq 2 \times mn$$

where the inequality is given by the properties of geometric series.

These facts are summarized in the following recursive algorithm.

PATH(source, sink)
1: **if** source **and** sink are in consecutive columns **then**
2: **output** longest path from source to sink
3: **else**
4: $mid \leftarrow$ middle vertex $(i, \frac{m}{2})$ with largest score $length(i)$
5: PATH(source,mid)
6: PATH(mid,sink)
7: **end if**

4.11. The Four-Russians Speedup

So far, we have seen that there are algorithms that require $O(n^2)$ time for aligning two sequences of length n. As we were feeling comfortable with polynomial algorithms for alignment problems we never asked whether an even faster algorithm could exist. A faster algorithm would possibly be based on techniques different from dynamic programming as the associated tables of size n^2 have to be traversed. It is still an open problem to provide lower bounds on the running time of the alignment problem and an algorithm of time $O(n \log n)$ would certainly revolutionize bioinformatics.

Although no answer has been given yet for the question of a global (or local) alignment, it is possible to solve the Longest Common Subsequence problem in $O(\frac{n^2}{\log n})$ time. Herefore, a technique called the "Four-Russians Speedup" is applied, named after a paper of four russians from 1970. In simple words, the Four-Russians Speedup relies on precomputing values of all alignments between sequences of a sufficiently small length. The values are stored in a lookup table and later used for constructing block alignments, instead of the usual alignments. When summing up the running times of the different steps one ends up with the subquadratic amount from above.

4.11.1. Block Alignments. Let $u = u_1...u_n$ and $v = v_1...v_n$ be two DNA sequences partitioned into blocks of length t:

$$u = |u_1...u_t|\,|u_{t+1}...u_{2t}|...|u_{n-t+1}...u_n|$$
$$v = |v_1...v_t|\,|v_{t+1}...v_{2t}|...|v_{n-t+1}...v_n|$$

. For simplicity we assume that both sequences have the same length and n is divisible by t.

DEFINITION 4.13. A *block alignment* of u and v is an alignment in which every block from u and v is either aligned against an entire block from the other sequence or is deleted resp. inserted as a whole. Within a block, an alignment path can be completely arbitrary.

See Fig. 7.4, [**JP04**] for an illustration where a $n \times n$ grid is partitioned into $t \times t$ grids.

DEFINITION 4.14. A path through a partitioned grid is called a *block path* if it traverses every $t \times t$ square through its corners.

Block alignments correspond to block alignments in the edit graph. When t is on the order of the logarithm of the overall sequence length we can solve the block alignment problem in subquadratic time.

Block Alignment Problem:
Find the longest block path through an edit graph.

Input: Strings v, w, partitioned into blocks of size t and a scoring matrix δ.
Output: The block alignment of u and v having maximum score (i.e. the longest block path through an edit graph).

One now can consider $\frac{n}{t} \times \frac{n}{t}$ blocks and compute the alignment score β_{ij} for each pair of blocks $|u_{(i-1)t+1}...u_{it}|$ and $|v_{(j-1)t+1}...v_{jt}|$. This takes $O(\frac{n}{t} \cdot \frac{n}{t} \cdot t \cdot t) = O(n^2)$ time. If s_{ij} denotes the score of the optimal block alignment between the first i blocks of u and the first j blocks of v, then

$$s_{ij} = \max \begin{cases} s_{i-1,j} - \sigma_{block} \\ s_{i,j-1} - \sigma_{block} \\ s_{i-1,j-1} + \beta_{ij} \end{cases}$$

where σ_{block} denotes the penalty for deleting resp. inserting an entire block. Indices i, j vary from 0 to $\frac{n}{t}$. Therefore the running time of this algorithm is $O(\frac{n^2}{t^2})$ if we do not count precomputing the β_{ij}. This, however, took us $O(n^2)$ time, so no speed reduction so far.

We will achieve a speed reduction by an application of the Four-Russians technique where t is on the oreder of $\log n$. Instead of constructing $\frac{n}{t} \times \frac{n}{t}$ minialignments for all pairs of blocks of u and v we will construct $4^t \times 4^t$ minialignments for all pairs of t-nucleotide strings (t-mers) and store their alignment scores in a lookup table. If $t = \frac{\log n}{4}$ (log is to the base of 2), there are

$$4^t \times 4^t = 2^{2t} \times 2^{2t} = 2^{2\frac{\log n}{4}} \times 2^{2\frac{\log n}{4}} = n^{\frac{1}{2}} \times n^{\frac{1}{2}} = n$$

minialignments to be done, which is much smaller than $\frac{n^2}{t^2}$. So, the resulting lookup table has about n entries and computing all of the entries needs $O(\log n \log n)$ time. This amounts to an overall running time to compute the table of $O(n \cdot (\log n)^2)$. Denoting the entry of the lookup table for two t-mers a, b by $LU(a, b)$, the recurrence from above now reads

$$s_{ij} = \max \begin{cases} s_{i-1,j} - \sigma_{block} \\ s_{i,j-1} - \sigma_{block} \\ s_{i-1,j-1} + LU(i\text{-th block ofv}, j\text{-th block ofu}) \end{cases}$$

resulting in an algorithm of subquadratic time $O(\frac{n^2}{\log n})$ As the precomputation of LU is done in even less time, this is also the running time for the

entire algorithm of both precomputing LU and computing the block alignment with it.

4.11.2. Subquadratic Sequence Alignment. The technique of the last subsection only provided us with a subquadratic algorithm to solve the block alignment problem. Here we will demonstrate how to construct an $O(\frac{n^2}{\log n})$ algorithm for the Longest Common Subsequence problem which is a problem referring to the complete alignment grid and therefore, at first sight, has nothing to do with the block alignment problem. The path referring to the longest common subsequence need not traverse any of the corners of the minialignment grids at all. Therefore, precomputing any $t \times t$ inialignments does not help.

Instead we select all vertices of the original alignment grid along the borders of the blocks and not only the corners of these blocks (see Fig. 7.5, [**JP04**]). There are $\frac{n}{t}$ whole rows and $\frac{n}{t}$ whole columns of vertices which results in a total number of $O(\frac{n^2}{t})$ vertices.

We are now interested in the following problem: given the alignment scores $s_{i,*}$ in the first row and the alignment scores $s_{*,j}$ in the first row and the first column of a $t \times t$ minisquare, compute the alignment scores in the last row and the last column of the minisquare. These values depend on four variables:

(1) the values $s_{i,*}$ of the first row,
(2) the values $s_{*,j}$ of the first column,
(3) the substring of length t corresponding to the transitions of rows of the minisquare and
(4) the substring of length t corresponding to the transitions of columns of the minisquare.

Of course, this could be done by the usual alignment techniques applied to the minialignments given through the minisquares. However, this is too slow here. Instead, we build a lookup table for all possible combinations of the four input variables to the problem: all pairs of t-mers and all pairs of possible scores for the first row $s_{i,*}$ and the first column $s_{*,j}$. For each such quadruple (4-tuple) we store the precomputed scores for the last row and the last column. It is now decisive to see that this can be done in subquadratic time of at most that of the number of vertices of the rows and the columns (which was $O(\frac{n^2}{t})$, see above) such that the entire algorithm has a running time of $O(O(\frac{n^2}{t})$.

A little trick helps us bound the time of the precomputations. Namely, note that the values $(s_{i,j}, s_{i,j+1}, ..., s_{i,j+t-1}, s_{i,j+t})$ along the first row of the minisquare can not be completely arbitrary. Instead it holds that $s_{i,k+1} - s_{i,k} \in \{0, 1\{$, as a longest path ending at $(i, k+1)$ can at most be greater by one than that which ends at (i, k) (due to the formulation of the longest common subsequence problem where horizontal and vertical edges have weight 0 and some diagonal edges have weight 1). Therefore,

values $(s_{i,j}, s_{i,j+1}, ..., s_{i,j+t-1}, s_{i,j+t})$ can be stored as t-dimensional vectors where the entries represent the differences between the $s_{i,k}, s_{i,k+1}$ and each of the entries is either 0 or 1. As an example, the possible sequence of scores $0, 1, 2, 2, 3, 3, 3, 4$ would be encoded as $1, 1, 0, 1, 0, 0, 1$. This amounts to 2^t possible combinations of values in the first row and completely analogous considerations yield 2^t different combinations of values for the first column. Note that the actual value of $s_{i,k}$ is not really relevant as once you know $s_{i,j}$ of the upper left corner, the remaining values are fully determined by the binary vector.

In sum, from $2^t \cdot 2^t$ possible combinations of binary vectors for the first row and the first column and $4^t \cdot 4^t$ pairings of t-mers, there are

$$2^t \cdot 2^t \cdot 4^t \cdot 4^t = 2^{2t} \cdot 2^{4t} = 2^{6t}$$

entries to compute for the lookup table. Setting $t = \frac{\log n}{4}$, the size of the table collapses to

$$2^{6\frac{\log n}{4}} = (2^{\log n})^{\frac{3}{2}} = n^{1.5}$$

which is subquadratic. Hence, the entire algorithm has a running time of $O(\frac{n^2}{\log n})$.

CHAPTER 5

Gene Finding: Similarity-based Approaches

5.1. Motivation

To predict an organism's complexity from the size of its genome alone can be mistaken as there are cases which clearly contradict this assumption. The salamander genome, for example, is ten times larger than the human genome (and the human race actually feels best when considering themselves as the most complex of all beings).

The explanation for this is that large parts of the genome are not transcribed into RNA. Large amounts of the DNA are simply *non-coding* that is, they are not translated to proteins. In both eukaryotes and prokaryotes genes are separated from each other by *intergenic* regions. When sequencing known intergenic regions one finds that they are largely made up of repetitive patterns.

In 1977, Phillip Sharp and Richard Roberts discovered that mRNA transcripts folded back to the DNA strands they came from (building DNA-RNA double strands) forming unexpected structures (see Fig. 6.23, [**JP04**]). The explanation was that, in eukaryotes, transcribed regions are further divided into *exons* and *introns* where introns are the regions which are finally cut out of the RNA transcript by the spliceosome. Hence introns in DNA are not coding for proteins either. The only difference to intergenic regions is that they are localized within *open reading frames* (ORFs). Genes have to be within *open reading frames* (ORFs) which are defined to be bounded by a *start codon*, i.e. the triplet ATG and a *stop codon*, i.e. one of the three triplets TAA, TAG or TGA. However, some ORFs do not refer to genes one of the reasons being that most ORFs are simply too short. The exact definition of an intron is to be a non-coding region within an open reading frame which gives rise to a gene. A *gene* in turn is defined to be the sequence of exons which, in the end, translated into a protein. In prokaryotes, there are no introns. Here, genes are continuous stretches of coding DNA.

Finding genes is not easy even in prokaryotes. In eukaryotic genomes, to find sequences of exons which refer to protein products is even more difficult. However, the increasing knowledge of known genes however gives rise to similarity-based methods which will be discussed in the following. The idea behind these methods is to identify sequences of *putative exons* by comparing them to known sequences of coding regions in related organisms (for example, human and mouse have 99 % of the genes in common).

General Problem 5.1. Find a set of substrings (putative exons) in a genomic sequence (say, mouse) whose concatenation fits a known (say, human) protein.

5.2. Exon Chaining

Given a known protein and a genome to be scanned, a naive brute-force approach would be to find all regions in the genome of high local similarity to the known gene (or protein). The substrings of high local similarity could be connected to each other in a way that some global similarity to the known gene is maximized. Of course, we first have to make sure that regions of high local similarity actually come from coding regions in the genome to be studied, that there is good reason to believe that there are (putative) exons. This is usually done by checking for *splicing signals* indicating either a start (GT) or an end (AG) of an intron (see Fig. 6.24, [**JP04**]).

In order to tackle this problem we will model exons by weighted interval.

Definition 5.2. A *weighted interval* (l, r, w) refers to a substring of the genome to be scanned for genes where l is the left-hand position of sequence, r the right-hand position and w the weight of the subsequence usually given by a local alignment score to the gene in question from the other organism. A *chain* is a set of non-overlapping weighted intervals, the total weight of the chain is the sum of the weights of the participating intervals. A *maximum chain* finally is a chain having maximum total weight.

Having collected a set of *putative exons* we face the following problem.

Exon Chaining Problem (ECP)
Given a set of putative exons, find a maximum set of non-overlapping putative exons.

Input: A set of weighted intervals (modeling putative exons).
Output: A maximum chain of intervals from this set.

The Exon Chaining problem for n intervals is solved by using dynamic programming to find a longest path in a graph G with $2n$ vertices where n vertices represent starting positions and the remaining ones represent ending positions. The set of left and right interval ends is sorted in increasing order into an array of vertices $(v_1, ..., v_{2n})$. Vertices v_i, v_j are connected by directed edges (in increasing order of the vertices) if v_i and v_j are starting and ending position from one putative exon. The weight of the corresponding interval is then assigned to the edge. Furthermore, we have $2n - 1$ additional edges of weight 0 which simply connect adjacent vertices v_i, v_{i+1} (only if v_i, v_{i+1} do not already refer to a weighted edge.. The resulting graph is obviously directed and acyclic, having sink v_1 and source v_{2n}. Finding the longest path from soure to sink solves the ERP. See Fig. 6.26, [**JP04**] for an illustration.

In the following algorithm we write s_i to denote the longest path from source to vertex v_i.

ExonChaining(G, n):
1: **for** $i \leftarrow 1$ to $2n$ **do**
2: $s_i \leftarrow 0$
3: **end for**
4: **for** $i \leftarrow 1$ to $2n$ **do**
5: **if** vertex v_i corresponds to the right end of an interval **then**
6: $j \leftarrow$ index of vertex for left end of the interval
7: $w \leftarrow$ weight of the interval
8: $s_i \leftarrow \max\{s_j + w, s_{i-1}$
9: **else**
10: $s_i \leftarrow s_{i-1}$
11: **end if**
12: **end for**
13: **return** s_{2n}

One shortcoming of this approach is that the endpoints of putative exons are not very well defined. Hence the optimal chain found by ExonChaining may not even refer to a valid alignment.

5.3. Spliced Alignments

In 1996, Mikhail Gelfand and colleagues proposed the *spliced alignment* approach which is very similar to the exon chaining problem. Having been given a set of putative exons from a genome whose genes are to be found one wants to find a linear chain of these exons which maps a target protein from another organism. Note that in this problem the alphabets—DNA sequences on one hand, and amino acid sequences on the other hand—do not match. However, by means of the genetic code (codons coding for amino acids) there are, more or less straightforward, workarounds to overcome this difficulty, so we do not further mention this discrepancy.

Let $U = u_1...u_n$ be the genome sequence to be screened and $T = t_1...t_m$ be the target sequence, i.e. the sequence of a protein from another organism. Let further $\mathcal{B}$ be a set of candidate exons from G, i.e. subsequences of G which refer to putative coding regions. (Assume that these have been carefully filtered such that $\mathcal{B}$ contains all true exons, but, in addition to the true exons, possibly also false ones). The Spliced Alignment problem is now to find a linear chain of exons which maximizes the *global alignment* score (and not, like in the Exon Chaining problem, a linear chain which maximizes the sum of the local similarity scores of the exons). As above, a chain Γ is any sequence of non-overlapping blocks from $\mathcal{B}$. The sequence corresponding to Γ is then denoted by Γ^*.

Spliced Alignment Problem (SAP)
Given a set of putative exons, find a chain that best fits a target sequence.

Input: Genomic sequence U, target sequence T, and a set of candidate exons $\mathcal{B}$.
Output: A chain of candidate exons Γ such that the global alignment score $s(\Gamma^*, T)$ is maximum among all chains of candidate exons from $\mathcal{B}$.

We can cast the problem as finding a path in a directed, acyclic graph G. Vertices in the graph correspond to blocks (candidate exons) and directed edges connect nonoverlapping blocks (see Fig. 6.28, [**JP04**], vertices are denoted by rectangles). A vertex is labeled by the string representing the corresponding block. A path in this graph spells a string by concatenating the strings belonging to the traversed vertices. The weight of a path is defined as the score of the optimal alignment between the concatenated blocks of the path and the target sequence. Note that edges have no weights, so the Spliced Alignment problem is not a problem of finding a longest path in a directed acyclic graph. It can, nevertheless, be solved by dynamic programming techniques.

DEFINITION 5.3. An i-prefix $B(i)$ of U is defined to be the prefix of a candidate exon $B = u_{B,left}...u_i...u_{B,right}$ (a candidate exon containing position i of the genome), that is

$$B(i) = u_{B,left}...u_i$$

where $(B, left)$ and $(B, right)$ are the indices referring to the starting position and the ending position of B in the genome U.

Define, accordingly, $T(j) = t_1...t_j$ to be the j-prefix of the target sequence. Further, if a chain $\Gamma = (B_1, B_2, ..., B)$ ends at block B, define

$$\Gamma^*(i) := u_{B_1,left}...u_{B_1,right}u_{B_2,left}...u_{B_2,right}...u_{B,left}...u_i$$

that is, $\Gamma^*(i)$ is the sequence given by the blocks which ends at position i in the genome S. Finally, let

$$S(i, j, B) := \max_{\text{all chains } \Gamma \text{ ending in } B} s(\Gamma^*(i), T(j)).$$

That is, given i, j and a candidate exon B that covers position i, $S(i, j, B)$ is the optimal spliced alignment between the i-prefix of U and the j-prefix of T under the assumption that the alignment ends in block B.

The following recurrence then leads to a dynamic programming algorithm that solves the problem. For the sake of simplicity assume for the moment that we have linear gap penalties equal to σ for insertions and deletions and that we use the scoring matrix δ for matches and mismatches. The recurrence is split into two cases where the first one refers to i not being the

starting position of block B (that is, $(B, right) \geq i > (B, left)$):

$$S(i,j,B) = \max \begin{cases} S(i-1,j,B) - \sigma \\ S(i,j-1,B) - \sigma \\ S(i-1,j-1,B) + \delta(u_i, t_j) \end{cases}$$

The second case ($(B, left) = i$) is more difficult, as optimal alignments can come from different blocks before B:

$$S(i,j,B) = \max \begin{cases} S(i,j-1,B) - \sigma \\ \max_{\text{all } B' \text{ preceding } B} S((B', right), j, B') - \sigma \\ \max_{\text{all } B' \text{ preceding } B} S((B', right), j-1, B') - \delta(u_i, t_j) \end{cases}$$

After having computed the three-dimensional table $S(i,j,B)$ the score of the optimal spliced alignment can be obtained as

$$\max_B S((B, right), m, B)$$

where the maximum is taken over all possible blocks.

CHAPTER 6

DNA Sequencing

6.1. Motivation

DNA sequencing is the problem of reading the complete genomic sequence of an organism. Unfortunately, one cannot read the complete sequence just like a book. Instead, due to the experimental restrictions in a laboratory, all one can do is something which is similar to tearing several copies of the book into pieces and then reading the pieces. Your task is to reconnect the pieces in the correct order. Even more difficult, the words of your book have no meaning such that any more sophisticated methods are not at hand. Moreover, imagine you would have (minor) problems with reading such that you cannot entirely rely on what you have read. All you have is that pieces from different copies of the book may overlap. From these overlaps alone one has to reconstruct the text of the book where, because of your reading problems, it is a bit harder to detect the overlaps as if you had read the pieces properly.

Two DNA sequencing methods were invented independently and simultaneously in Cambridge, England by Fred Sanger (there is now a famous institute in Cambridgeshire, named the Wellcome Trust Sanger Institute, formerly the Sanger Center hosting several genome research projects) and in Cambridge, Massachusetts by Walter Gilbert. Sanger's method makes use of that cells copy their DNA strands (in a reaction called polymerase chain reaction (PCR)) nucleotide by nucleotide. Sanger starved this reaction at a given nucleotide. Like this, he obtained all subsequences of a sequence which stopped at this nucleotide. For example, starving PCR for ACGTAAGCTA at nucleotide T yields the subsequences ACG and ACGTAAGC. Doing this for all nucleotides provides one with a *ladder* of fragments of varying lengths, the so called *Sanger ladder* (and providing Sanger with the Nobel Prize in 1980 as an extra). Based on these elementary reading techniques, computational approaches to fragment assembly (that is, reconnecting the pieces which had been read by Sanger's methods) enabled the sequencing of the 3-billion-nucleotide human genome in 2001. Two different fragment assembly theories were applied by Celera (a company lead by the famous Craig Venter) on one hand and the Human Genome Project on the other hand. Despite the different approaches, both projects can be considered successful.

Nowadays, modern sequencing machines can sequence 500- to 700-nucleotide fragments. These sequencing machines measure the lengths of the fragments in the Sanger ladder, but for doing so, billions of copies of the respective pieces have to be provided, a process which is error-prone

(for multiplying a piece, a technique called *cloning* is applied, see [**JP04**] for further explanations). Thus, the Sanger ladder sequencing does not necessarily produce exact results.

In sum, DNA sequencing is a two stage process including an experimental step for reading fragments and a computational step which is for assembling the reads obtained from the experiments. Both steps are scientifically demanding and require substantial efforts from the participating researchers.

6.2. The Shortest Superstring Problem

Every string, or *read*, came from a much longer genomic string and, as we aim at sequencing the long string as a whole, we are interested in a *superstring* of these reads. The problem is that there are many superstrings which "explain" the reads, i.e. which contain them as substrings. However, not all of the possible superstrings are reasonable explanations for the reads at hand. Simply concatenating the reads instantaneously yields a superstring, hence a putative solution to the problem. However, the concatenation of the reads does not reflect reality at all as you can take it for granted that there are overlaps in the read. Instead, we will search for the *shortest superstring* containing the reads as substrings. With regard to the sampling of the reads, this should be a reasonable guess for the entire sequence of the sequence.

Shortest Superstring Problem:
Given a set of strings, find a shortest string that contains all of them.

Input: Strings $s_1, ..., s_n$.
Output: A string s that contains all strings $s_1, ..., s_n$ as substrings, such that the length of s is as small as possible.

See Fig. 8.14 for two superstrings for the set of all eight three-letter strings in a 0-1 alphabet.

For a solution to the problem define $overlap(s_i, s_j)$ to be the length of the longest prefix of s_j that matches a suffix of s_i. The Shortest Superstring problem can be transformed to a Traveling Salesman problem in a complete directed graph with n vertices for the strings $s_1, ..., s_n$ and edges of length $-overlap(s_i, s_j)$ (note that edges are directed as $overlap(s_i, s_j)$ is not necessarily equal to $overlap(s_j, s_i)$). See Fig. 8.15, [**JP04**] for an *overlap graph* corresponding to Fig. 8.14. Reducing the problem to the Traveling Salesman problem does not lead to efficient algorithm, as it is well-known that the TSP is $\mathcal{NP}$-complete. Moreover, it was shown that the Shortest Superstring problem itself is $\mathcal{NP}$-complete. Therefore, early sequencing approaches used a simple greedy strategy. Here strings were merged iteratively to a growing superstring and greed referred to choosing strings with maximum overlap.

It is an open conjecture that this algorithm has an approximation ratio of 2 (that is, the resulting superstring is at most twice as long as the superstring of minimal length).

At first sight, it looks as if problems of this kind cannot be tackled in a satisfactory way. However, there are subclasses of this problem relevant for addressing related questions arising from DNA sequencing which can be solved in polynomial time. These related questions emerge from different sequencing techniques which are discussed in the subsequent section.

6.3. Sequencing by Hybridization

The problems which were encountered when using the sequencing methods from above were partly overcome by the introduction of a new technique which is called *Sequecing by Hybridization* (SBH). It is based on a device called *DNA array* which allows for finding all l-mers being substrings in a *target string*, which, in our case, is the string to be sequenced.

6.3.1. DNA arrays. DNA arrays (also called *DNA chips*) contain thousands of short DNA framents called *probes*. Each of the probes reveals the occurrence of a known, but short, substring in an unknown *target* sequence.

Given a short probe (usually in the range of 8 to 30 nucleotides) a single-stranded target DNA fragment will *hybridize* to the probe if it contains the sequence which is complementary to the probe as a substring. In 1988, when the first genome sequencing projects were launched, noone believed in an experimental technology of practical use based on attaching probes to a chip. So, one rather relied on conventional techniques like the Sanger ladder method.

1991: Steve Fodor and colleagues presented an approach to manufacturing DNA chips relying on *light-directed polymer synthesis* which drastically diminished the number of reactions needed for the synthesis of the probes which is usually quite large.

1994: *Affymetrix*, a California-based biotechnology company, built the first (64-kb) DNA array. See Fig. 8.16, [**JP04**] for a picture of a DNA array

Today, building 1-Mb or even larger arrays is routine. The *universal* DNA array contains all 4^l probes of length l and is applied as follows (see Fig. 8.17, [**JP04**]):

(1) Attach all possible probes of length l ($l = 8$ in the first papers) to a flat surface. Each probe is at a distinct and known location, usually ordered in the style of an array (hence the name DNA array).
(2) Apply a solution containing the DNA fragment of interest which is fluorescently labeled. The solution should be at a temperature where the DNA is single-stranded (DNA unwinds at temperatures of $60 - 70$ degrees Celsius).

(3) While lowering the temperature, the fragment hybridizes to the probes (winds up with the probes building Watson-Crick-pairs) which are complementary to substrings of length l of the fragment.
(4) Using a spectrocopic detector (measuring intensity levels of fluorescence at the different spots of the array) , determine which probes hybridize to the DNA fragment to obtain the l-mer composition of the target DNA fragment.
(5) Apply a combinatorial algorithm to reconstruct the sequence of the target DNA fragment from the l-mer composition.

6.3.2. SBH as an Eulerian Path Problem. In the following we will present a combinatorial algorithm to solve the problem of fragment assembly when given the collection of substrings of length l of the target sequence.

DEFINITION 6.1. Let s be a string of length n. The *spectrum* of s is the l-mer composition of s given as multiset of $n-l+1$ subsequences of length l and is denoted by $Spectrum(s,l)$.

EXAMPLE 6.2. If $s = TATGCTAT$ then

$$Spectrum(s,3) = \{ATG, CTA, GCT, TAT, TAT, TGC\}$$

in lexicograpical order. This order reflects that we do not know in which order the substrings appear in the target sequence.

Although it is impossible to immediately obtain the order of appearance of the substrings in the spectrum (which would immediately yield the target's sequence) one can obtain the number of appearance in the spectrum of a substring by measuring intensity levels of fluorescence at the array spots.

Sequencing by Hybridization (SBH) Problem:
Reconstruct a string from its l-mer composition.

Input: A multiset $\mathcal{S}$, representing all l-mers from an (unknown) string s.
Output: A string s such that $Spectrum(s,l) = \mathcal{S}$.

The Sequencing by Hybridization problem is very similar to the Shortest Superstring problem. The only, but decisive, difference is that the substrings from which to reconstruct the superstring all have the same length in the Sequencing by Hybridization problem. This already reduces the complexity of the problem such that there is a linear-time algorithm which solves it.

Note first that the SBH problem can be cast as a *Hamiltonian Path* problem in relatively straightforward way, by modeling the elements of $\mathcal{S}$ as vertices and drawing directed edges between them if they overlap by a string of length $l-1$. The Hamiltonian Path problem refers to finding a path in a directed graph which visits each node exactly once. Thus solving the Hamiltonian Path problem solves the SBH problem (see Figs. 8.18, 8.19

[**JP04**]). Alas, "Hamiltonian Path" is $\mathcal{NP}$-complete and therefore delivers no algorithm of practical use.

We are interested in practical solutions of the SBH problem. Therefore, we will reduce the SBH problem to instances of the *Eulerian Path* problem.

Eulerian Path Problem:
Find a path in a (directed) graph G that visits every edge exactly once.

Input: A graph G.
Output: A path in G that visits every edge exactly once.

The reduction of SBH to the Eulerian Path problem is done by the construction of a graph where edges rather than vertices (as in the Hamiltonian Path problem represent l-mers from $Spectrum(s, l)$. Namely, vertices correspond to $l-1$-mers (encountered in the substrings of the spectrum) and an edge is drawn from $l-1$-mer v to w if there is an l-mer u in the spectrum such that v is the $l-1$-prefix of u and w is the $l-1$-suffix of u. If u appears several times in the spectrum then v and w are connected by the corresponding number of directed edges. See Fig. 8.20, [**JP04**] for an example.

For solving the problem, we first consider *Eulerian cycles*, that is, Eulerian paths in which the first and the last vertex coincide.

Definition 6.3. A directed graph is *Eulerian* if it contains an Eulerian cycle. *balanced* if

$$\forall v \in V : \quad indegree(v) = outdegree(v)$$

that is, if, for all vertices, the number of edges entering v equals that of the edges leaving v for all nodes v.

The proof's idea of the following theorem provides a strategy for constructing Eulerian cycles in linear time.

Theorem 6.4. *A connected graph is Eulerian if and only if it is balanced.*

Proof. It is easy to see that a graph is balanced if it is Eulerian.
To construct an Eulerian cycle, we start from an arbitrary vertex v and form an arbitrary path by traversing edges that have not already been used. We stop the path when we encounter a vertex with no way out unless via edges having already been used for the path. In a balanced graph, the only vertex where this can happen is the starting point of our travel since for any other vertex, the balance condition ensures that for every incoming edge there is an outgoing edge that has not yet been used. Therefore the resulting path will end at vertex v and, possibly, we have already constructed an Eulerian cycle. However, if it is not Eulerian, it must contain a vertex w that still has some number of untraversed edges. The cycle we have just constructed is a balanced subgraph (a graph resulting from simply removing edges from

the original graph). The edges that were not traversed in the first cycle also form a balanced subgraph. Hence, by arguing the same way like for the first cycle, there must exist another cycle which starts and ends at w (see Figs. 8.21, 8.22, [**JP04**]). One now combines the two cycles as follows: first traverse the first cycle from v until you reach w. Then do the second cycle completely. Afterwards, complete the first cycle until you reach v again. We are ready if this larger cycle is already Eulerian. If not, repeat the procedure by finding a third vertex x with unused edges attached. We stop if there are no more such vertices. ◇

Note that the SBH problem was transformed into an Eulerian Path problem instead of an Eulerian Cycle problem. However, this problem can easily transformed to an Eulerian Cycle problem the following way.

DEFINITION 6.5. A vertex v in a graph is called *balanced* if

$$indegree(v) = outdegree(v).$$

semibalanced if

$$|indegree(v) - outdegree(v)| = 1.$$

If a graph has an Eulerian path starting at vertex s and ending at vertex t, then all its vertices are balanced, with the possible exception of s and t, which may be semibalanced. The Eulerian path problem can therefore be reduced to an Eulerian cycle problem by adding an edge between two semibalanced vertices. The path is then obtained by removing this additional edge from cycle obtained in the graph with the added edge.

THEOREM 6.6. *A connected graph has an Eulerian path if and only if it contains at most two semibalanced vertices and all other vertices are balanced.*

The construction of Eulerian cycles, however, is done in linear time along the lines of the proof of the first theorem. We have therefore obtained a linear-time algorithm for the SBH problem.

CHAPTER 7

Combinatorial Pattern Matching

The problem of pattern matching is to find exact or approximate occurrences of a given pattern in a long text. Here text usually refers to the sequence of a genome. Although the problem setting is apparently simple difficulties arise when dealing with genomes of large sizes. Here straightforward approaches are simply too slow. This issue is tackled by clever algorithms on one hand and by clever preprocessing on the other hand where text and/or pattern are organized into handy data structures.

7.1. Motivation: Repeat Finding

The *DiGeorge syndrome*, a genetic disease resulting in an impaired immune system and heart defects, is associated with a 3 Mb (Megabases) deletion on human chromosome 22. This deletion removes important genes and thus leads to disease. It is caused by the occurrence of a pair of very similar sequences flanking the deleted region. Pairs of highly similar sequences in the genome are referred to as *repeats*.

The DiGeorge syndrome is only one example for why it is important to find repeats in the genome. In general they hold many evolutionary secrets. Repeats account for about 50 % of the human genome, a phenomenon which is still unexplained.

A straightforward approach to repeat finding would be to construct a table which for a given l contains all of the 4^l possible l-mers and with it the positions of their appearance in the genome. However, biologists are usually interested in *maximal repeats* where respective sequences have length larger than $L = 20$. Constructing tables for l-mers where l varies from 10 to 13 is still feasible. If one is interested in lengths larger than this one could be tempted to start with pairs of sequences of feasible length and trying to extend these. However, only little patterns of length 20 or larger can appear in a genome. So, extending all pairs of matching patterns of feasible size would be, in most cases, a waste of time. To find maximal repeats in a genome, the popular REPuter algorithm, for example, uses *suffix trees*, a data structure which we describe in a later section.

7.2. The Knuth-Morris-Pratt Algorithm

Exact Pattern Matching Problem:
Given a text $T = t_1...t_m$ of length m and a pattern $P = p_1...p_n$ of length n find all positions i in T such that $t_i...t_{i+n-1} = p_1...p_n$.

Input: A text T and a pattern P.
Output: All positions i in T where $t_i...t_{i+n-1} = p_1...p_n$.

A straightforward and naive solution to this problem would be to screen T at each position for the occurrence of P which results in an algorithm of running time $O(nm)$. However, there are more efficient solutions.

The Knuth-Morris-Pratt algorithm is maybe the best known linear-time algorithm for the exact pattern matching problem. It solves the problem in $O(n + m)$ time.

DEFINITION 7.1. For each position i in P define $\Phi_i(P)$ to be the length of the longest proper suffix of $p_1...p_i$ that matches a prefix of P. So, if $\Phi_i(P) = j$ then

$$p_1...p_j = p_{i-j+1}...p_i$$

such that j is maximal.

EXAMPLE 7.2. If $P = abcaeabcabd$ then $\Phi_2(P) = \Phi_3(P) = 0, \Phi_4(P) = 1, \Phi_8(P) = 3$ and $\Phi_{10}(P) = 2$.

Note that, by definition $\Phi_1(P) = 0$ for all P.

DEFINITION 7.3. Let further $\Phi'_i(P)$ be the length j of the longest proper suffix of $p_1...p_i$ such that

$$p_1...p_j = p_{i-j+1}...p_i \quad \text{and} \quad p_{j+1} \neq p_{i+1}$$

Note that $\Phi'_i(P) \leq \Phi_i(P)$ because of the added condition for $\Phi'_i(P)$.

EXAMPLE 7.4. If $P = bbccaebbcabd$, then $\Phi_8(P) = 2$ as $p_1p_2 = p_7p_8 = bb$. However $p_3 = p_9 = c$, so $\Phi'_8(P) < 2$. More exactly, $\Phi'_8(P) = 1$ as $p_8 = p_1 = b$ but $p_9 = c \neq p_2 = b$.

The trick of the Knuth-Morris-Pratt algorithm is to implement the following *shifting rule* (see [**Gus97**], p. 25 for illustrations).

DEFINITION 7.5 (Shifting rule). For any alignment of P and T, if the first mismatch (comparing from left to right) occurs in position $i+1$ in P and position k in T, that is

$$p_1...p_i = t_{k-i}...t_{k-1} \quad \text{but} \quad t_k \neq p_{i+1}$$

shift P exactly $i - \Phi'_i(P)$ places to the right such that $p_1...p_{\Phi'_i(P)}$ aligns with $t_{k-\Phi'_i(P)}...t_{k-1}$. In case that an occurrence of P had been found shift P by $n - \Phi'_n(P)$ places.

By definition of $\Phi'_i(P)$, after a shift according to the shifting rule we have

$$p_1...p_{\Phi'_i(P)} = p_{i-\Phi'_i(P)+1}...p_i = t_{k-\Phi'_i(P)}...t_{k-1}.$$

That is, after such a shift the first $\Phi'_i(P)$ characters of P still match the aligned characters of T. However, it is not that obvious that we could have missed an occurrence of P by shifting P by $i - \Phi'_i(P)$ positions instead of one position only. The following theorem shows why not.

THEOREM 7.6. *For any alignment of P with T, if*

$$p_1...p_i = t_{k-i}...t_{k-1} \quad \textit{but} \quad p_{i+1} \neq t_k \tag{7.1}$$

then

$$p_1...p_n \neq t_{k-l}...t_{k-l+n-1}$$

for all $l = \Phi'_i(P)+1, ..., i$. That is we cannot miss an occurrence of P in T by shifting P by $\Phi'_i(P)$ positions to the right.

PROOF. Assume that not. Choose $l \in \{2, ..., \Phi'_i(P) - 1\}$ such that

$$p_1...p_n = t_{k-l}...t_{k-l+n-1} \tag{7.2}$$

Consider

$$\beta := p_{l-\Phi'_i(P)+1}...p_l = t_{k-\Phi'_i(P)}...t_{k-1} = p_{i-\Phi'_i(P)+1}...p_i$$

where the first equation comes from (7.2) and the second one from (7.1). Consider further

$$\alpha := p_1...p_{l-\Phi'_i(P)}. = t_{k-l}...t_{k-\Phi'_i(P)-1} = p_{i-l+1}...p_{i-\Phi'_i(P)}$$

where the first equation is inferred from (7.2) and the second one is from (7.1). Concatenating α and β leads to

$$\alpha\beta = p_1...p_{l-\Phi'_i(P)}p_{l-\Phi'_i(P)+1}...p_l = p_{i-l+1}...p_{i-\Phi'_i(P)}p_{i-\Phi'_i(P)+1}...p_i$$

which translates to that $p_1...p_l$ is a proper suffix of $p_1...p_i$. Further $p_{i+1} \neq t_k = p_{l+1}$ (by the original assumption p_{i+1} and t_k do not match, but t_k and p_{l+1} do because of (7.2)). This leads to the contradiction that

$$\Phi'_i(P) \geq l > \Phi'_i(P).$$

◇

The running time of shifting P along the shifting rule (already knowing $\Phi'_i(P)$ for all i) turns out be bounded by $2m$ hence is linear in the length of T.

THEOREM 7.7. *In the Knuth-Morris-Pratt algorithm the number of compared characters is at most* $2m$.

PROOF. Divide the algorithm into compare/shift phases where a single phase consists of the comparisons done between successive shifts. After any shift, the comparisons start either with the last character compared in the phase before (in case having found an occurrence of P) or one position to the right of it (in case of not having found an occurrence of P in the phase before). Since P is never shifted to the left, this means that in any phase at most one comparison involves a character already been compared before. In sum, this translates to that any character of T is compared at most twice leading to the bound of the theorem. ◇

Furthermore, the values $\Phi'_i(P)$ can be determined in linear time $O(n)$ (see [**Gus97**], pp. 7-10, 26-27 for explanations). In sum, this results in an algorithm of running time $O(n+m)$ as we had claimed.

Let finally

$$F'(i) := \Phi'_{i-1}(P) + 1$$

for $i = 1, ..., n+1$ where $\Phi'_0(P) := 0$.

KNUTHMORRISPRATT(T, P):

```
1: Preprocess P to find F'(k) for k = 1, ..., n+1.
2: k ⟵ 1
3: j ⟵ 1
4: while k + (n − j) ≤ m do
5:    while p_j = t_k and j ≤ k do
6:       j ⟵ j + 1
7:       k ⟵ k + 1
8:    end while
9:    if j = n + 1 then
10:      report an occurrence of P starting at position k − n of T
11:   end if
12:   if j = 1 then
13:      k ⟵ k + 1
14:   end if
15:   j ⟵ F'(j)
16: end while
```

7.3. Keyword Trees

Multiple Pattern Matching Problem:
Given a text $T = t_1...t_m$ of length m and a set of patterns $P_1 = p_{11}...p_{1n_1}, ..., P_k = p_{k1}...p_{kn_k}$ of lengths $n_1, n_2, ..., n_k$ find all occurrences of any of the patterns in the text.

Input: A text T and a pattern $P_1, ..., P_k$.
Output: All positions $1 \leq i \leq m - \min_l n_l + 1$ in T such that $t_i...t_{i+n_l-1} = p_1...p_{m_l}$ for a $l = 1, ..., k$.

EXAMPLE 7.8. If $T = ATGGTCGGT, P_1 = GGT, P_2 = GGG, P_3 = ATG, P_4 = CG$ then the result of an algorithm that solves the Multiple Pattern Matching problem would be the positions $1, 3, 6$ and 7.

Reducing the Multiple Pattern Matching problem to k individual Pattern Matching problem and applying the Knuth-Morris-Pratt algorithm to each of them would yield an algorithm of running time $O(k(m + N))$ where $N := \max_{1 \leq l \leq k} n_l$. However, one can do even faster by using the concept of a keyword tree.

DEFINITION 7.9. A *keyword tree* for a set of patterns $P_1, ..., P_k$ is a rooted labeled tree satisfying the following conditions, assuming for simplicity that no pattern in the set is a prefix of another pattern.

(i) Each edge of the tree is labeled with a letter of the alphabet.
(ii) Any two edges out of the same vertex have distinct labels.
(iii) Every pattern P_l $(1 \leq l \leq k)$ is spelled on some path from the root to a leaf.
(iv) Every path from root to a leaf correponds to one of the patterns P_l $(1 \leq l \leq k)$.

Because of $[(iv)]$, the keyword tree has at most N vertices where $N = \sum_{l=1}^{k} n_k$. One can construct the keyword tree in $O(N)$ time by progressively extending the keyword tree for the first j patterns into the keyword tree for $j+1$ patterns. See Fig. 9.3, [**JP04**] for the example of a keyword tree.

The keyword tree can be used for finding whether there is a pattern in the set $P_1, ..., P_k$ that matches the text starting at a fixed position i. To do this, we simply traverse the keyword tree using letters $t_i t_{i+1} t_{i+2}...$ of the text to decide where to move at each step (see Fig. 9.4, [**JP04**]). This process either ends at a leaf, or interrupts before arriving at a leaf which refers to either finding the pattern corresponding to the leaf or not finding any of the patterns. If $n := \max_l n_l$ is the maximum of the lengths of the patterns then this algorithm has running time $O(N + nm)$.

The *Aho-Corasick* algorithm reduces the running time further to $O(N + m)$ by generalizing the methods of the Knuth-Morris-Pratt algorithm. For a node v in the keyword tree we define $\Phi(v)$ to be the length of the longest

proper suffix of the string associated with v (given through the edges from the root leading to v) which is a prefix of on of the patterns $P_l, 1 \leq l \leq k$. There is another node $n(v)$ in the keyword tree which refers to this suffix. We traverse the keyword tree until either finding a fully matching pattern or finding a node such that the corresponding string does not further match the text. In either case we shift the text such that it matches the word of node $n(v)$. See [**Gus97**], pp. 52-61 for the details, if interested.

7.4. Suffix Trees

The problem of multiple pattern matching can be tackled from a different side. That is, the text $T = t_1...t_m$ is preprocessed and organized into a *suffix tree*. As a consequence, whether or not an arbitrary pattern of length n occurs in the text, can be decided in running time $O(n)$.

Furthermore, it turns out that the suffix tree for a text of length m can be constructed in $O(m)$ time, which leads immediately to a $O(n+m)$ algorithm for the Pattern Matching problem.

In short words, a *suffix tree* is constructed from a keyword tree of all suffixes $t_m, t_{m-1}t_m, t_{m-2}t_{m-1}t_m, ..., t_1...t_m$ of a text $T = t_1...t_m$ where non-branching stretches of the tree have been collapsed into a single edge which is labeled by the concatenation of the letters along the collapsed edges. See Fig. 9.5, [**JP04**] for a suffix tree and the keyword tree it comes from. More formally:

Definition 7.10. The *suffix tree* for the text $T = t_1...t_m$ is a rooted labeled tree with m leaves (numbered from 1 to m) satisfying the following conditions:

(i) Each edge is labeled with a substring (not necessarily a single letter) of the text.
(ii) Each internal vertex (except possibly the root) has at least 2 children,
(iii) Any two edges out of the same vertex start with a different letter.
(iv) Every suffix of text T is spelled out on a path from the root to some leaf.

Definition 7.11. The *threading* of a pattern P through a suffix tree T is defined as the matching of characters of P along the unique path in T until either all characters of P are matched (a *complete threading*) or until no more matches are possible (an *incomplete threading*). In case of a complete threading the *P-matching leaves* are defined to be the leaves that are descendants of the vertex or edge where the complete threading ends.

See Fig. 9.6, [**JP04**] for the example of a complete threading.

SuffixTreePatternMatching(T, P):
1: Build the suffix tree for text T.
2: Thread pattern P through the suffix tree.

3: **if** threading is complete **then**
4: **output** positions of every P-matching leaf in the tree
5: **else**
6: **output** "pattern does not appear anywhere in the text"
7: **end if**

The idea of a suffix tree can be generalized to contain all suffixes of several texts $T_1, ..., T_s$. In such a tree each of the leaves refers to one of the suffixes of one of the texts and it is constructed in linear time by means of the same ideas yielding the linear-time algorithm to construct a normal suffix tree. With such a *generalized suffix tree* one can, for example, solve the longest common subsequence problem in linear time. First, one constructs the generalized suffix tree of the two sequences and additionally label each internal vertex v with 1 and/or 2 according to which string the leaves of the subtree of v refer. If v is labeled with both 1 and 2 then the string spelled along the edges of the path from the root to v is a common substring of both strings. The vertex referring to the longest such string can be found with standard linear-time tree traversal methods. Hence finding the longest common subsequence can be done in linear time.

See [**Gus97**], p. 122 ff. for a variety of problems which can be efficiently solved by using (generalized) suffix trees.

7.5. Suffix Arrays

DEFINITION 7.12. Let $T = t_1...t_m$ be a text of length m. The *suffix array* $SA(T) \in \mathbb{N}^m$ is defined by $SA(i) = j$ if $t_j...t_m$ is the i-th suffix in a lexicographic ordering of all suffixes.

EXAMPLE 7.13. Let $T = ACAGACAT$. Then the lexicographic ordering of suffixes is $ACAGACAT(j = 1), ACAT(j = 5), AGACAT(j = 3), AT(j = 7), CAGACAT(j = 2), CAT(j = 6), GACAT(j = 4), T(j = 8)$. Hence the suffix array SA for T is

$$SA(1) = 1, SA(2) = 5, SA(3) = 3, SA(4) = 7, SA(5) = 2, SA(6) = 6, SA(7) = 4, SA(8) = 8.$$

It needs $O(n \log n)$ time and $O(n)$ space to construct a suffix array.

Searching for patterns P in a string T is done by a binary search in the suffix array of T.

SuffixArrayPatternMatching($T = t_1...t_m, P = p_1...p_n$):

```
1: Build the suffix array SA for text T.
2: L ⟵ 1
3: R ⟵ m
4: M ⟵ ⌈(L + R)/2⌉
5: for i ⟵ 1 to n do
6:    Perform binary search on p_i:
7:    while p_i ≠ i-th character of SA(M) do
8:       if M = 1 or M = m then
9:          output "pattern doesn't match any substring of the text"
10:         return
11:      end if
12:      if p_i < i-th character of SA(M) then
13:         R ⟵ M
14:         M ⟵ ⌈(L + R)/2⌉
15:      else if p_i > i-th character of SA(M) then
16:         L ⟵ M
17:         M ⟵ ⌈(L + R)/2⌉
18:      end if
19:   end while
20:   continue
21: end for
22: return the found suffix which matches pattern P
```

Running time: $O(\log m)$ per character of the pattern, therefore overall running time $O(n \log m) + O(k)$ where k is the number of occurrences of P in T.

7.6. The Karp-Rabin Fingerprinting Method

The Karp-Rabin method is an algorithm that has worst-case running time $O((m-n+1)n)$ just like the naive pattern matching algorithm but on average needs only $O(m)$ time, independent of the pattern.

In the following, we will treat all characters A, C, G, T as digits $A = 0, C = 1, G = 2, T = 3$. This leads to a radix-4 notation

$$\bar{P} = p_n + 4^1 p_{n-1} + ... + 4^{n-2} p_2 + 4^{n-1} p_1$$

for each DNA pattern string $P = p_1...p_n$.

Example 7.14. If $P = AGACAGT$ then

$$\begin{aligned}\bar{P} &= 0102013 = 3 \cdot 4^0 + 1 \cdot 4^1 + 0 \cdot 4^2 + 2 \cdot 4^3 + 0 \cdot 4^4 + 1 \cdot 4^5 + 0 \cdot 4^6 \\ &= 3 + 4 + 128 + 6144 = 6279 \text{ in radix-10.}\end{aligned}$$

If P matches a substring $T_i := t_i...t_{i+n-1}$ of length n of the text $T = t_1...t_m$ then $\bar{P} = \bar{T}_i$ as well as

$$\bar{P} \equiv \bar{T}_i \mod q$$

for all $q \in \mathbb{N}$. If we choose q small enough such that $\bar{P} \mod q$ fits into a machine then we can do comparisons with reasonable accuracy. Furthermore, we note that computing $\bar{T}_{i+1}$ can be done in constant time $O(1)$ if $\bar{T}_i$ has already been known.

Observation: $\bar{T}_{i+1}$ can computed from $\bar{T}_i$ using Horner's rule. From

$$\begin{aligned}\bar{T}_i &= t_{i+n-1} + 4t_{i+n-2} + 4^2 t_{i+n-3} + ... + 4^{n-1}t_i \\ &= t_{i+n-1} + 4[t_{i+n-2} + 4[t_{i+n-3} + 4[...[t_i]...]\end{aligned}$$

we obtain

$$\begin{aligned}\bar{T}_{i+1} &= t_{i+n} + 4t_{i+n-1} + 4^2 t_{i+n-2} + ... + 4^{n-1}t_{i+1} \\ &= t_{i+n} + 4[t_{i+n-1} + 4[t_{i+n-2} + 4[...[t_{i+1}]...] \\ &= t_{i+n-1} + 4\bar{T}_i - 4^n t_i.\end{aligned}$$

This leads to the following algorithm. Hereby, q is a large prime number in order to minimize the likelihood of mistaken results.

KarpRabinFingerprinting($T = t_1...t_m, P = p_1...p_n$):

1: $h \longleftarrow 4^{n-1} \mod q$
2: $\bar{P} \longleftarrow 0$
3: $\bar{T}_1 \longleftarrow 0$
4: **for** $i \longleftarrow 1$ to n **do**
5: $\quad \bar{P} \longleftarrow (4\bar{P} + p_i) \mod q$
6: $\quad \bar{T}_1 \longleftarrow (4\bar{T}_1 + t_i) \mod q$
7: **end for**
8: **for** $j \longleftarrow 1$ to $m - n + 1$ **do**
9: $\quad$ **if** $\bar{P} = \bar{T}_j$ **then**
10: $\quad\quad$ **output** "match at j"
11: $\quad$ **end if**
12: $\quad$ **if** $j < m - n + 1$ **then**
13: $\quad\quad \bar{T}_{j+1} \longleftarrow [4(\bar{T}_j - t_{j+1}h + t_{j+n}] \mod q$
14: $\quad$ **end if**
15: **end for**

7.7. Heuristic Similarity Search: FASTA

When comparing a short pattern with a large string one often is not only interested in exact but also in approximate matches. Here, *filtration* often is the method of choice. It is based on the observation that good alignments usually include short identical or highly similar fragments. For example, matching 9-mers with one allowable error must contain a completely matching 4-mer. Hence, searching for approximately matching 9-mers (in the sense of maximally one mismatch) can be initiated by searching for matching 4-mers first to filter out regions where approximate matches cannot occur. For filtration-based methods biologists often make use of a *dot matrix*.

DEFINITION 7.15. Let $u = u_1...u_m$ and $v = v_1...v_n$ be two strings. A *dot matrix* $A(u,v) \in \{0,1\}^{m \times n}$ is a binary matrix where a 1 at entry (i,j) indicates some similarity of u and v around u_i and v_j.

The simplest form of a dot matrix is given by setting a 1 when $u_i = v_j$. A more complex form of a dot matrix is given by having a 1 in case of $u_i u_{i+1} = v_j v_{j+1}$, i.e. if the corresponding dinucleotides match (see Fig. 9.7, [**JP04**]).

The *FASTA* algorithm uses dot matrices indicating exact matches of length l. From such a dot matrix it selects diagonals with a high concentration of 1's, and groups these diagonals into longer runs to construct an alignment (see Fig. 9.7, [**JP04**] for an illustration of this idea).

7.8. Approximate Pattern Matching

As before, an approximate match of a pattern $P = p_1...p_n$ with a larger text $T = t_1...t_m$ is defined to be a substring $t_i...t_{i+n-1}$ in the text such that $d_H(t_i...t_{i+n-1}, p_1...p_n) \leq k$ where k is a parameter that meets the requirements of the problem at hand and $d_H(.,.)$ is the Hamming distance.

Approximate Pattern Matching Problem:
Given a text $T = t_1...t_m$ of length m, a pattern $P = p_1...p_n$ and a parameter k, find all approximate occurrences of the pattern in the text.

Input: A text $T = t_1...t_m$, a pattern $P = p_1, ..., p_n$ and a parameter k.
Output: All positions $1 \leq i \leq m - n + 1$ s.t. $d_H(t_i...t_{i+n-1}, p_1...p_n) \leq k$.

The naive brute force algorithm for approximate pattern matching (I recall Ex. 1 from Homework Assignment No. 1) needs $O(nm)$ time.

APPROXIMATEPATTERNMATCHING($T = t_1...t_m, P = p_1...p_n, k$):

```
1: for i ⟵ 1 to m − n + 1 do
2:    dist ⟵ 0
3:    for j ⟵ 1 to n do
4:       if t_{i+j−1} ≠ p_j then
5:          dist ⟵ dist + 1
6:       end if
7:    end for
8:    if dist ≤ k then
9:       output i
10:   end if
11: end for
```

In 1985 Gadi Landau and Udi Vishkin found an algorithm for approximate pattern matching of worst-case running time $O(km)$. It is the algorithm with the best known worst-case performance. However, several filtration-based approaches have better running times in practice, although

their worst-case performances are worse.

7.9. Query Matching

The Query Matching problem further generalizes the Approximate Pattern Matching problem to finding substrings of the pattern which approximately match substrings of the text. That is, given a *query sequence* $Q = q_1...q_l$, a (usually much larger) text $T = t_1...t_m$ and parameters n, k find all occurrences of substrings $Q_j := q_j...q_{j+n-1}$ of length n in Q and substrings $T_i := t_i...t_{i+n-1}$ such that the Hamming distance between Q_j and T_i does not exceed k.

Query Matching Problem:
Find all substrings of the query that approximately match the text.

Input: A text $T = t_1...t_m$, a query $Q = q_1...q_l$ and integers n and k.
Output: All pairs of positions (i, j) where $1 \leq i \leq m - n + 1$ and $1 \leq j \leq l - n + 1$ s.t. $d_H(t_i...t_{i+n-1}, q_j...q_{j+n-1}) \leq k$.

See Fig. 9.8, [**JP04**] for an illustration.

In the following, we will employ the *l-mer filtration technique* to obtain a filtration algorithm. It is again based on the observation that approximate matches of size n must share at least one l-mer for a suitable (not too large) l. A theorem serves as a guide for choosing l given the pattern size n and the number of allowed mismatches k.

THEOREM 7.16. *If the strings $x_1...x_n$ and $y_1...y_n$ match with at most k mismatches, then they share an l-mer for*

$$l = \lfloor \frac{n}{k+1} \rfloor,$$

that is,

$$x_{i+1}...x_{i+l} = y_{i+1}...y_{i+l}$$

for some $1 \leq i \leq n - l + 1$.

PROOF. Partition the set of positions from 1 to n into $k + 1$ groups

$$\{1, ..., l\}, \{l + 1, ..., 2l\}, ..., \{(k - 1)l + 1, ..., kl\}, \{(kl + 1, ..., n\}$$

where the last group may contain more then l positions because of $l = \lfloor n/(k + 1) \rfloor \Rightarrow n \geq (k + 1)l$. Note that k mismatches affect positions in at most k of these groups, so for one of the groups, hence there is a contiguous stretch of at least l positions where x matches y. ◇

The guideline for an algorithm based on the statement of the theorem is as follows:

- *Potential match detection*: Find all matches of l-mers in both the query and the text for $l = \lfloor n/(k + 1) \rfloor$. You can do this either by hashing or a suffix tree approach.

- *Potential match verification*: Verify each potential match by extending it to the left and to the right until the first $k+1$ mismatches are found (or the beginning or end of the query or the text is found).

For most practical values of n and k, the number of potential matches is small which results in a fast algorithm.

7.10. BLAST

When it comes to aligning one sequence against a whole database of sequences (or, say, a whole genome) in terms of high alignment scores the usual alignment procedures are simply too slow. The aforementioned filtration techniques seem to be a practicable idea, but do not directly do their job, as for an alignment we need short regions of high similarity scores instead of l-mers which were needed for determining approximate matches. The problem is that high similarity regions might be frequently interspersed with gaps such that they would not be identified by an approach searching for l-mers with sufficiently large l.

Instead of searching for exactly matching l-mers, BLAST ("basic local alignment search tool") looks for regions yielding high similarity scores. These then serve as a seed for extending alignments.

DEFINITION 7.17. Let $x_1...x_l$ and $y_1...y_l$ be two l-mers and δ be a scoring matrix. Then

$$\sum_{i=1}^{l} \delta(x_i, y_i)$$

is called the *segment score* of x and y. A *segment pair* is just a pair of l-mers, one from each sequence. A *maximal segment pair* is a segment pair with the best score over all segment pairs in the two sequences. A segment pair is *locally maximal* if it cannot be improved either by extending or shortening both segments.

A researcher typically is interested in *statistically significant* locally maximal segment pairs rather than only the highest scoring segment pair. The reason is that there might pairs of which yield high scores only due to their large length. These are not of interest as, in terms of statistics, the corresponding score is easily obtained by pairing random segments of that size.

BLAST strives to find all locally maximal segment pairs from the query and the database sequences the scores of which are statistically significantly high. First, it finds all l-mers in the text (the database sequences) which yield a score above the threshold against some l-mer in the query. The fast algorithm which implements this search is the key ingredient of BLAST. If the threshold is high enough, then the set of all l-mers that have scores above the threshold is not too large. In this case, the database can be searched for all occurrences of these l-mers which amounts to a Multiple Pattern Matching problem. Herefore, the Aho-Corasick algorithm finds the location of any

of these strings in the database.

The choice of the threshold is guided by the Altschul-Dembo-Karlin statistics, which compute probabilities for obtaining similarity scores from random pairings of segments and therefore allow to compute significance values for the scores obtained from segment pairs. BLAST then reports matches to sequences that either have one segment score above the threshold or that have several closely located segment pairs that are statistically significant when combined.

DEFINITION 7.18 (Altschul-Dembo-Karlin statistics). Let λ be a positive root of the equation

$$\sum_{x,y \in \mathcal{A}} p_x p_y e^{\lambda \delta(x,y)} = 1$$

where p_x, p_y are the frequencies of amino acids x, y from the twenty-letter alphabet $\mathcal{A}$ and δ is the scoring matrix. Then the number of matches between sequences of length m and n, according to the *Altschul-Dembo-Karlin* statistics, with scores above θ is approximately Poisson-distributed, with mean

$$E(\theta) = Kmnd^{-\lambda\theta}$$

where K is a constant.

The probability that there is a match of score greater than θ between two "random" sequences of length m and n is computed as $1 - E(\theta)$.

CHAPTER 8

Hidden Markov Models

8.1. Motivation: CG-Islands

The least frequent dimer in many genomes is CG. This is because a C which is followed by a G is more easily mutated (into a T) than any other nucleotide in any other "environment". However, processes involved in mutation of C's followed by G's are suppressed in genetic regions called CG-islands. Locating these CG-islands in a genome is an important problem which is equivalent to the following setting. Assume that there is a (crooked) casino where the dealer has the choice of two coins where one of them is fair (that is, the probability of tossing heads as well as that of tossing tails is 0.5) and a biased coin that gives heads with probability 0.75. For security reasons, the probability that the dealer exchanges these coins between two tosses is assumed to be low (say 0.1). You, as a visitor of this casino, want to find out when series of tosses with the biased coin are going to happen. A naive approach would be to say that the biased coin is in action when long strings of 1's are to be seen. However, this is not a sound approach to the problem.

Fair Bet Casino Problem:
Given a sequence of coin tosses, determine when the dealer used a fair coin and when he used a biased coin.

Input: A sequence $x = x_1...x_n$ of coin tosses (either $x_i = H$ or $x_i = T$ for all i) made by two possible coins (F or B).
Output: A sequence $\pi = \pi_1...\pi_n$ where $\pi_i = F$ or $\pi_i = B$ indicating which coin was used for each toss x_i.

This problem is ill-defined. For example, both $\pi = FF...F$ and $\pi = BB...B$ are valid answers as any other combination of F's and B's is. We need a method to grade different coin sequences as being better than other ones. The solution to this is to provide a statistical model for the fair bet casino problem. In this vein, different coin sequences are graded as being better the more likely they are according to the model at hand.

Assume for the moment that the dealer never changes coins within a sequence. Seeing a sequence $x = x_1...x_n$ the question is which of the two coins he used for generating the sequence. For sake of simplicity let 0 denote tails and 1 denote heads. We write $p^+(0) = p^+(1) = 0.5$ for the probabilities of the fair coin and $p^-(0) = 0.25, p^-(1) = 0.75$ for that of the biased coin.

We can compute probabilities that x was generated by the fair coin

$$P(x|\text{fair coin}) = \prod_{i=1} p^+(x_i) = \frac{1}{2^n}$$

and the according probability of the biased coin where k is the number of heads in x

$$P(x|\text{biased coin}) = \prod_{i=1} p^-(x_i) = \prod_{x_i=0} \frac{1}{4} \prod_{x_i=1} \frac{3}{4} = \frac{1}{4^{n-k}} \frac{3}{4}^k = \frac{3^k}{4^n}.$$

From

$$\frac{1}{2^n} = \frac{3^k}{4^n} \quad \Leftrightarrow \quad k = \frac{n}{\log_2 3}$$

we would come to the conclusion that if $k > \frac{n}{\log_2 3}$ the dealer used a biased coin as $P(x|\text{biased coin})$ was greater in that case. If $k < \frac{n}{\log_2 3}$ we would conclude that the dealer was using the fair coin. In a more formal manner we would compute the *log-odds ratio*

$$\log_2 \frac{P(x|\text{fair coin})}{P(x|\text{biased coin})} = \sum_{i=1}^{n} \log_2 \frac{p^+(x_i)}{p^-(x_i)} = n - k \log_2 3.$$

Then we would make a decision according to whether the log-odds ratio was positive or negative.

However, we know that the dealer changes coins within sequences. A naive approach would be to shift a window over the sequence and, by computing log-odds ratios, to determine the coin which was used more probable. Similarly, a naive approach to finding CG-islands would be to shift a window over the genome and to determine, according to dimer probabilities, whether the subsequence under consideration was an island or not. However, lengths of CG-islands can be highly variable so that the naive approach is of little use. A solution to the problem are hidden Markov models. With them variable length sequences can be separated into regions where one of the two submodels (fair or biased coin, CG-island or not) was more likely to be in effect.

8.2. Hidden Markov Models

DEFINITION 8.1. A *probability distribution* $P = (P_1, ..., P_m)$ is a set of numbers where

$$\forall i = 1, ..., n : \ \mathbb{P}_i \geq 0 \quad \text{and} \quad \sum_{i=1}^{n} P_i = 1.$$

Each P_i refers to an event i which happens with probability P_i.

DEFINITION 8.2. A *hidden Markov model* $\mathcal{M} = (\Sigma, Q, A, E)$ (HMM) consists of the following elements:

- An alphabet of symbols $\Sigma = \{b_1, ..., b_m\}$,
- A set of *hidden states* (hence the name hidden Markov model) $Q = \{q_1, ..., q_n\}$,

- a *transition matrix* $A = (a_{q_k q_l})_{(k,l)\in\{1,...,n\}^2}$ whose rows and columns are indexed by the hidden states, and where each row is a probability distribution over the hidden states and
- an *emission matrix* $E = (e_{q_k b_j})_{(k,j)\in\{1,...,n\}\times\{1,...,m\}}$ where each row is a probability distribution over the symbols from Σ.

The way an HMM $\mathcal{M}$ is interpreted is as a machine which generates sequences of symbols from Σ. At each step, the HMM "is in a hidden state". Having generated the sequence $x_1...x_t$ at periods $1, ..., t$ let k be the hidden state $\mathcal{M}$ is in at period t. In order to produce the next symbol, the HMM chooses a new hidden state according to the probability distribution given by the the k-row of A, that is, the new hidden state q_l is chosen with probability $a_{q_k q_l}$. Having entered the chosen q_l, symbol b_j is generated with probability $e_{q_l b_j}$, that is, according to the *emission probability distribution* attached to the hidden state q_l given by the values $e_{q_l b_j}, b_j \in \Sigma$. Having generated the symbol $x_{t+1} \in \Sigma$ this procedure is repeated.

As all that remains from the generation process is the sequence of the observable symbols, it becomes obvious that the elements of Q are referred to as the *hidden* states. From observing a sequence of symbols alone one cannot determine the underlying sequence of hidden states as, in general, different sequences of hidden states can account for the same sequence of observables. However, and this is one of the reasons for the popularity of HMMs, one can efficiently determine the most probable underlying sequence of hidden states, even for long runs of the HMM.

8.2.1. The Fair Bet Casino as an HMM. The Fair Bet Casino is modeled the following way. The observable symbols are 0 and 1 for tails and heads, respectively. Hidden states refer to the coin which is in use. Transition probabilities refer to the probabilities that the dealer changes coins and emission probabilities refer to probabilities of tossing heads and tails with the different coins. More formally,

- $\Sigma = \{0, 1\}$ for 0 and 1 denoting tails or heads,
- $Q = \{F, B\}$ for F referring to the fair and B to the biased coin,
- $a_{BB} = a_{FF} = 0.9, a_{FB} = a_{BF} = 0.1$ for modeling that the dealer changes coins with probability 0.1 and
- $e_{F1} = e_{F0} = 0.5, e_{B1} = 0.75, e_{B0} = 0.25$ for modeling the probabilities of the outcomes of different coin tosses.

See Fig. 11.1, [**JP04**] for an illustration.

DEFINITION 8.3. A *path* $\pi = \pi_1...\pi_t \in Q^t$ in the HMM $\mathcal{M} = (\Sigma, Q, A, E)$ is a sequence of hidden states.

We write $P(x_i|\pi_i)$ for the probability that x_i was emitted from state π_i, that is, $P(x_i|\pi_i) = e_{\pi_i x_i}$ and $P(\pi_i \to \pi_{i+1} = a_{\pi_i \pi_{i+1}}$ for the probability of the transition from state π_i to pi_{i+1}. For choosing an initial state we have

probabilities $P(\pi_0 \to \pi_1) = P_{start}(\pi_1)$. For example, for modeling that the dealer initially chooses both the fair and the biased coin with probability 0.5 we set $P(\pi_0 \to B) = P(\pi_0 \to F) = 0.5$.

The path corresponding to that the dealer used the fair coin for the first three and for the last three and the biased coin for five tosses in between would be $FFFBBBBBFFF \in Q^{11}$. If the sequence of tosses was 01011101001, the corresponding sequence of emission probabilities $P(x_i|\pi_i)$ would be

$$\underbrace{\frac{1}{2}}_{P(x_1|\pi_1)} \quad \frac{1}{2} \quad \frac{1}{2} \quad \frac{3}{4} \quad \frac{3}{4} \quad \frac{3}{4} \quad \frac{1}{4} \quad \frac{3}{4} \quad \frac{1}{2} \quad \frac{1}{2} \quad \underbrace{\frac{1}{2}}_{P(x_{11}|\pi_{11})}$$

and the sequence of transition probabilities would be

$$\underbrace{\frac{1}{2}}_{P(\pi_0\to\pi_1)} \quad \underbrace{\frac{9}{10}}_{P(\pi_1\to\pi_2)} \quad \frac{9}{10} \quad \frac{1}{10} \quad \frac{9}{10} \quad \frac{9}{10} \quad \frac{9}{10} \quad \frac{9}{10} \quad \frac{1}{10} \quad \frac{9}{10} \quad \underbrace{\frac{9}{10}}_{P(\pi_{10}\to\pi_{11})}.$$

The probability $P(x, \pi)$ that this combination of a path π of hidden states and and a sequence x of coin tosses is generated is

$$\begin{aligned} P(x,\pi) &= \prod_{i=1}^{11} P(\pi_{i-1} \to \pi_i) P(x_i|\pi_i) \\ &= \underbrace{(\frac{1}{2} \cdot \frac{1}{2})}_{i=1} \times \underbrace{(\frac{9}{10} \cdot \frac{1}{2})}_{i=2} \times \cdots \times \underbrace{(\frac{9}{10} \cdot \frac{1}{2})}_{i=10} \underbrace{(\frac{9}{10} \cdot \frac{1}{2})}_{i=11} \\ &\approx 2.66 \times 10^{-6}. \end{aligned}$$

We are interested in the best explanation of the sequence x of coin tosses in terms of a path π which maximizes $P(x, \pi)$. That is, we would like to find

$$\pi^* := \operatorname*{argmax}_{\pi \in \{F,B\}^{11}} P(x,\pi) = \prod_{i=1}^{11} P(\pi_{i-1} \to \pi_i) P(x_i|\pi_i).$$

It turns out that the path from above is not optimal in this sense. The path π^* yielding maximal probability $P(\pi^*, 01011101001)$ is $\pi^* = FFFBBBFFFFF$ with the slightly better $P(FFFBBBFFFFF, 01011101001) \approx 3.54 \times 10^{-6}$.

For formal convenience we introduce fictitious initial and terminal states π_0 and π_{t+1} to compute probabilities

$$P(x,\pi) = a_{\pi_0,\pi_1} \prod_{i=1}^{t} e_{\pi_i x_i} a_{\pi_i \pi_{i+1}}$$

for that a combination of hidden states and symbol sequences comes up in a run of length t of the HMM $\mathcal{M} = (\Sigma, Q, A, E)$. We are now in position to reformulate the problem from before in a well defined manner.

Decoding Problem:
Find an optimal hidden path of states given a sequence of observations.

Input: A sequence $x = x_1...x_n$ of observations generated by an HMM $\mathcal{M} = (\Sigma, Q, A, E)$.

Output: A path $\pi = \pi_1...\pi_n$ that maximizes $P(x, \pi)$ over all possible paths π.

Although the decoding problem looks intractable at first sight there is an efficient dynamic programming algorithm.

8.3. Decoding Algorithm

The Decoding Problem is solved by the *Viterbi algorithm*. It is a dynamic programming algorithm which is based on an analog of the Manhattan grid and searches for the longest path throught it. It needs $O(|Q|^2 n)$ time where $|Q|$ is the number of hidden states. In more detail, the underlying directed, acyclic graph can be described as a $|Q| \times n$-grid where column i can be identified with symbol x_i of the sequence $x = x_1...x_n$ at hand. There are edges from (k, i) to $(l, i + 1)$ for each choice of k and l from $1, ..., |Q|$. So there is an edge between two vertices in the grid if and only if their column index differs by exactly 1. This results in picture slightly different from those of the alignment problems. Edges $(k, i) \to (l, i + 1)$ are labeled by the probability that, in the HMM, one makes a jump from hidden state q_k to hidden state q_l and subsequently emits symbol x_{i+1}. That is, the length $l((k, i), (l, i + 1))$ is given by

$$l((k, i), (l, i + 1)) = a_{q_k q_l} \cdot e_{q_l x_i}.$$

Hence probabilities $P(x, \pi)$ that sequence x is emitted along a hidden path $\pi = \pi_1...\pi_n$ is just the product of the edge weights along the path corresponding to the path π. See Fig. 11.2, [**JP04**] for an illustration. Now define p_{ki} to be the probability that a path ends in vertex (k, i) and note that

$$\begin{aligned} p_{l,i+1} &= \prod_{j=1}^{i+1} a_{\pi_{j-1}\pi_j} \cdot e_{\pi_j x_j} \\ &= (\prod_{j=1}^{i} a_{\pi_{j-1}\pi_j} \cdot e_{\pi_j x_j}) a_{\pi_i \pi_{i+1}} e_{\pi_{i+1} x_{i+1}} \\ &= p_{ki} a_{\pi_i \pi_{i+1}} e_{\pi_{i+1} x_{i+1}} = p_{ki} l((k, i), (l, i + 1)). \end{aligned}$$

This leads to the following (multiplicative) recurrence which is then the basis of the aforementioned dynamic programming algorithm. The idea behind it is that the optimal path for the $(i + 1)$-prefix $x_1...x_{i+1}$ uses a path for $x_1...x_i$ that is optimal among the paths ending in some unknown state π_i.

Accordingly define s_{ki} to be the probability of the most likely path for the prefix $x_1...x_i$ that ends in state k. Then

$$\begin{aligned} s_{l,i+1} &= \max_{q_k \in Q}\{s_{ki} \cdot l((k,i),(l,i+1)) \\ &= \max_{q_k \in Q}\{s_{ki} \cdot a_{kl} \cdot e_{q_l x_{i+1}}\} \\ &= e_{q_l x_{i+1}} \cdot \max_{q_k \in Q}\{s_{ki} \cdot a_{kl}\}. \end{aligned}$$

We initialize $s_{begin,0} = 1$ and $s_{k0} = 0$ for $k \neq begin$. If π^* is an optimal path, then

$$P(x|\pi^*) = \max_{q_k \in Q}\{s_{kn}\}.$$

The running time of dynamic programming algorithms for finding longest path in DAGs is proportional to the number of edges in the DAG. Hence we obtain a running time of $O(|Q|^2 n)$ as claimed in the introduction of the section.

In an implementation of this recurrence the high amount of multiplications of small numbers is an issue as resulting numbers may be too small and lead to overflow in the computer. Therefore the recurrence is transformed into an additional one by taking logarithms of all described quantities. Namely

$$S_{ki} := \log s_{ki}$$

and the recurrence

$$S_{l,i+1} = \log e_{q_l x_{i+1}} + \max_{q_k \in Q}\{S_{ki} + \log a_{q_k q_l}\}$$

and a final

$$P(x|\pi^*) = \max_{q_k \in Q}\{\exp S_{kn}\}$$

do the job.

8.4. Forward and Backward Algorithm

We can also ask a different, but equally interesting question: Given a sequence $x = x_1...x_n \in \Sigma^n$ and the HMM (Σ, Q, A, E) what is the probability $P(\pi_i = q_k|x)$ that the HMM was in state k at time i? In the casino we would ask what is the probability that the dealer used the biased coin at time i upon seeing the sequence of coin tosses x. We will see that, by computing these probabilities efficiently, we obtain, as a byproduct, an efficient method to compute probabilities $P(x)$ for sequences to be emitted by the HMM at all.

For the following note that probabilities $P(A,B) := P(A \cap B)$ of the common occurrence of events A, B are computed as

$$P(A,B) = P(A \,|\, B) \cdot P(B). \tag{8.1}$$

That is, $P(A,B)$ is the product of probabilities $P(B)$ of the occurrence of event B and $P(A|B)$, the probability of occurrence of event A given that we know that B is to occur. In the text we have A is that x is emitted by

the HMM and B that in the HMM a path is generated which, at position i has hidden state q_k. Note further that if an event C can be partitioned into non-overlapping events C_k which together cover the event C, then

$$P(C) = \sum_k P(C_k). \tag{8.2}$$

For example, in the following C will be the event that x is generated by the HMM and C_k will be the event that x is generated by a path which traverses hidden state q_k at time i.

We compute

$$P(x) \overset{(8.2)}{=} \sum_{q_k \in Q} P(x, \pi_i = q_k)$$

and

$$P(x, \pi_i = q_k) \overset{(8.2)}{=} \sum_{\pi, \pi_i = q_k} P(x, \pi).$$

Consequently,

$$P(\pi_i = q_k \,|\, x) \overset{(8.1)}{=} \frac{P(x, \pi_i = q_k)}{P(x)}$$

which is what we are interested in.

For an efficient computation of the $P(x, \pi_i = q_k)$ we recursively define *forward probabilities*

$$f_{ki} := e_{q_k x_i} \cdot \sum_{q_l \in Q} f_{l,i-1} a_{q_l q_k}$$

which turn out to be the probabilities of emitting prefix $x_1 ... x_i$ on a path ending in state q_k. This recurrence is used by the *forward algorithm* whose only difference to the Viterbi algorithm is that "max" has become a "$\sum$". With the forward algorithm one can efficiently compute probabilities $P(x)$ by observing that

$$P(x) = \sum_{q_k \in Q} f_{kn}. \tag{8.3}$$

However, the quantity $P(\pi_i = q_k \,|\, x)$ is not only affected by forward probabilities f_{ki} as symbols $x_{i+1} ... x_n$ have an influence on it. Therefore we come up with a recursive definition of *backward probabilities*

$$b_{ki} := \sum_{q_l \in Q} b_{l,i+1} \cdot a_{q_k q_l} \cdot e_{q_l x_{i+1}}$$

which are the probabilities that the HMM is at state q_k and subsequently emits the suffix $x_{i+1} ... x_n$. The *backward algorithm* makes use of the above recurrence by using dynamic programming. In sum,

$$P(\pi_i = q_k \,|\, x) = \frac{P(x, \pi_i = q_k)}{P(x)} = \frac{f_{ki} \cdot b_{ki}}{P(x)}.$$

Note that, in a more general fashion than (8.3),

$$P(x) = \sum_{q_k \in Q} \{f_{ki} \cdot bki\} \tag{8.4}$$

for all $i = 1, ..., n$ where $b_{kn} := 1$ for all $q_k \in Q$.

8.5. HMM Parameter Estimation

This section is concerned about the following problem: given a set of hidden states, the set of output symbols and a set of sequences which have been generated by the HMM acting on the hidden states and the symbols what is the set of parameters (i. e. the transition probabilities and emissionn probabilities) that, in terms of the HMM, explains the generated sequences best? The question arises very naturally. Imagine, for example, you have visited the aforementioned casino several times and you have observed several sequences of coin tosses. The only difference to before is that now you do know nothing about how often the dealer changes coins and with what probabilities heads and tails are emitted by the coins in use. More formally:

Parameter Estimation Problem:
Given observed symbol strings $x^1, ..., x^m$ what is the set of parameters $\Theta = (A, E)$ such that the corresponding HMM (Σ, Q, A, E) has, most likely among all HMMs acting on Q and Σ, generated the symbol strings.

Input: *Training* sequences $x^1, ..., x^m$, a set of hidden states Q and an output alphabet Σ such that x^j is a string over Σ.

Output: A matrix of transition probabilities A and a matrix of emission probabilities E such that

$$\prod_{j=1}^{m} P(x^j \mid \Theta)$$

is maximized where Θ corresponds to the parameters of the HMM (Σ, Q, A, E).

As $\prod_{j=1}^{m} P(x^j \mid \Theta)$ is usually very small and computers could run into overflow problems the logarithm of it is maximized, that is one searches for parameters Θ that maximize $\sum_{j=1}^{m} \log P(x^j \mid Theta)$. This turns out to be a difficult optimization problem in a high-dimensional real vector space (of dimension of the number of entries of the trasition and emission matrix) and there is no strategy for finding the global optimum. Therefore, heuristics are used.

To address the problem first assume that the paths along which the sequences were generated are known. We can then count the number of transitions A_{kl} from any state k to another state l and obtain

$$a_{kl} = \frac{A_{kl}}{\sum q \in Q A_{kq}} \tag{8.5}$$

as a reasonable estimator for the transition probability a_{kl}. Correspondingly, a reasonable estimator for the emission probability e_{kb} of emitting symbol b from hidden state k would be

$$e_{kb} = \frac{E_{kb}}{\sum c \in \Sigma E_{kc}} \tag{8.6}$$

where E_{kb} is the number of times symbol b was generated from hidden state k. In case of that the paths of hidden states are unknown (which they usually are) we have to come up with more involved heuristics.

8.5.1. The Baum-Welch algorithm. Usually, paths of hidden states are unknown. To overcome the problem, the simple idea of the Baum-Welch strategy is to start with an arbitrary set of parameters, to compute most probable paths of hidden states for the sequences and to use the estimators from above where transition and emission counts refer to the optimal paths computed. Having found better values a_{kl}, ekb for all combinations $k, l \in Q, b \in \Sigma$ the procedure is repeated. It can be shown that $\sum_{j=1}^{m} \log P(x^j \,|\, Theta)$ is improved in each iteration. Hence a local optimum will be reached. However, due to the complexity of the search space, a global optimum cannot be guaranteed.

Furthermore, the Baum-Welch algorithm exploits that the counts A_{kl} and E_{kb} can be efficiently computed as the *expected* number of times each transition or emission is used, given the training sequences. Therefore, first, probabilities that a state transition from k to l was used at position i in sequence x^j is computed as

$$P(\pi_i = k, \pi_{k+1} = l \,|\, x, \Theta) = \frac{f_{ki} a_{kl} e_{l x_{i+1}} b_{l,i+1}}{P(x)}$$

where f_{ki} and b_{ki} are the forward and backward probabilities from section 8.4. The expected number of times transition $k \to l$ was used can then be computed as

$$A_{kl} = \sum_{j=1}^{m} \frac{1}{P(x^j)} \sum_{i=1}^{l_j} f_{ki}^j a_{kl} e_{l x_{i+1}} b_{l,i+1}^j,$$

where l_j is the length of sequence x^j, f_{ki}^j is the forward variable calculated for sequence x^j and b_{li}^j is the corresponding backward variable. Similarly, we can find the expected number of times that letter b appears in state k,

$$E_{kb} = \sum_{j=1}^{m} \frac{1}{P(x^j)} \sum_{\{i | x_i^j = b\}} f_{ki}^j b_{ki}^j,$$

where the inner sum is only over positions i for which the symbol emitted is b.

Having calculated these expectations new model parameters are computed according to (8.5) and (8.6). Having obtained new parameters the procedure is repeated until a stopping criterion is met. This is usually given by a threshold specifying the minimum amount of change in log likelihood between iterations.

BaumWelchTraining$(\Sigma, Q, x^j, j = 1, ..., m)$:

Initialization: Pick arbitrary model parameters $\Theta = (a_{kl}, e_{kb}, k, l \in Q, b \in \Sigma)$.
$LL \leftarrow -\infty$
while $\sum_{j=1}^{m} \log P(x^j \,|\, Theta) - LL \geq \epsilon$ **do**
  $LL \leftarrow \sum_{j=1}^{m} \log P(x^j \,|\, Theta)$
  $\forall k, l \in Q, b \in \Sigma : A_{kl} \leftarrow 0, E_{kb} \leftarrow 0$
  for $j \leftarrow 1$ to m **do**
    calculate f_{ki} using the forward algorithm
    calculate b_{ki} using the backward algorithm
    $A_{kl} \leftarrow A_{kl} + \frac{1}{P(x^j)} \sum_{i=1}^{l_j} f_{ki}^j a_{kl} e_{lx_{i+1}} b_{l,i+1}^j$
    $E_{kb} \leftarrow E_{kb} + \frac{1}{P(x^j)} \sum_{\{i|x_i^j=b\}} f_{ki}^j b_{ki}^j$
  end for
  $a_{kl} \leftarrow \frac{A_{kl}}{\sum q \in Q A_{kq}}$
  $e_{kb} \leftarrow \frac{E_{kb}}{\sum c \in \Sigma E_{kc}}$
  $\Theta \leftarrow (a_{kl}, e_{kb}, k, l \in Q, b \in \Sigma)$
end while
return Θ

8.6. Profile HMMs

Because of the course of evolution, proteins can be classified according to *family membership*. A family consists of proteins that are functionally related with each other and/or have a common ancestor protein from which they originate. Given a new protein of unknown function it is natural to ask whether the protein at hand can be assigned to a well known family. However, pairwise alignment procedures do not necessarily reveal their membership as the protein could share weak, insignifcant similarities with all proteins. As the overall amount of similarity could be meaningful though one has to watch out for a novel approach to this problem.

Usually, a family of related proteins is given by a multiple alignment and the corresponding profile. HMMs are particularly useful to align a sequence against a profile and we will describe the underlying class of HMMs, *profile HMMs* in the following.

A profile HMM consists of n sequentially linked match states $M_1, ..., M_n$ with emission probabilities e_{ia} taken from the profile. That is, e_{ia} is the probability that a appears at position i in the profile. See Fig. 11.4, [**JP04**] to recall the idea of a profile. The probability of a string $x_1...x_n$ given the profile P then is

$$\prod_{i=1}^{n} e_{ix_i}.$$

To further model insertions and deletions we add *insertion* states $I_0, I_1, ..., I_n$ and *deletion* states $D_1, ..., D_n$ to the HMM. We assume that $e_{I_j a}$, that is the

probability that symbol a is emitted from insertion state I_j is given by

$$e_{I_j a} = p(a)$$

where $p(a)$ is the frequency of occurrence of symbol a in all sequences giving rise to the profile. Deletion states emit symbol "-" with probability 1 (alternatively they can be considered as *silent states* which do not emit a symbol at all). See Fig. 11.5, [**JP04**] for a profile HMM. As for the sequences giving rise to the profile paths through the profile HMM are known, transition probabilities can be determined by formulas (8.5), (8.6) from section 8.5.

When one is given an unknown protein $x_1...x_n$ that is potentially related to the profile one computes its alignment with the profile by computing the Viterbi path through the profile HMM. Therefore let $v_j^M(i)$ be the logarithmic likelihood score of the best path of matching $x_1...x_i$ to the profile ending with symbol x_i emitted by match state M_j. Define $v_j^D(i)$ and $v_j^I(i)$ similarly. The resulting programming recurrence then looks like

$$v_j^M(i) = \log \frac{e_{M_j x_i}}{p(x_i)} + \max \begin{cases} v_{j-1}^M(i-1) + \log a_{M_{j-1}M_j} \\ v_{j-1}^I(i-1) + \log a_{I_{j-1}M_j} \\ v_{j-1}^D(i-1) + \log a_{D_{j-1}M_j} \end{cases}.$$

In a similar fashion values $v_j^D(i), v_j^I(i)$ are computed as

$$v_j^I(i) = \log \frac{e_{I_j x_i}}{p(x_i)} + \max \begin{cases} v_j^M(i-1) + \log a_{M_{j-1}I_j} \\ v_j^I(i-1) + \log a_{I_{j-1}I_j} \\ v_j^D(i-1) + \log a_{D_{j-1}I_j} \end{cases},$$

$$v_j^D(i) = \max \begin{cases} v_{j-1}^M(i) + \log a_{M_{j-1}D_j} \\ v_{j-1}^I(i) + \log a_{I_{j-1}D_j} \\ v_{j-1}^D(i) + \log a_{D_{j-1}D_j} \end{cases}.$$

See Fig. 11.6, [**JP04**] for an edit graph with a path that is related to the path traversing the corresponding profile HMM.

CHAPTER 9

Clustering

9.1. Motivation: Gene Expression Analysis

A common problem in biology is to partition a set of experimental data into groups (clusters) such that data points within a cluster are similar, preferably according to some biological feature of interest. A good example of a, in these times ubiquitous, problem comes from the large-scale analysis of expression levels of genes under certain conditions. Therefore a DNA array (see section **??**) that allows to analyze all genes in an organism simultaneously is applied. The organism's cells are subject to different conditions and for each of the conditions a DNA experiment is performed. This results in a set of expression levels (real numbers being identified with fluorescence intensity levels), one for each gene and condition. Seen for a single gene this amounts to having a real-valued vector, a so called *expression pattern*, of dimension the number of conditions that were tested.

The objective of such an analysis is to infer groups of genes which look similar from the point of view of their expression patterns. This translates to grouping a set of real-valued vectors (number of genes many) in a space of dimension of the number of conditions tested. From this arises the need for a method which groups sets of real-valued vectors into clusters such that clusters contain *similar* vectors. Similarity can be expressed acccording to a wide variety of similarity measures the most obvious of which is the euclidean distance between two vectors.

Having grouped the genes one can now reasonably assume that the genes' functions within a cluster are similar or interact in a common cellular mechanism.

In its most general form, the input to a clustering problem is a set of data points $v_i, i = 1, ..., n$ together with *distance matrix* $D = (d_{ij})_{i,j=1,...,n}$ where d_{ij} is the distance between points i and j.

Clustering Problem:
Given a set of data points $V = \{v_i, i = 1, ..., n\}$ together with a distance matrix (d_{ij}) find a partition $V = V_1 \dot{cup} ... \dot{\cup} V_K$ such that the "overall distance within a cluster" V_k is minimal and "overall distances between clusters" are maximal.

Input: A dataset $V = \{v_i, i = 1, ..., n\}$ and a distance matrix (d_{ij}).
Output: A partition $V = V_1 \dot{cup} ... \dot{\cup} V_K$ with minimal *intra-cluster distance*

(*homogeneity criterion*) and maximal *inter-cluster distance* (*separation criterion*).

See Figs. 10.1, 10.2, [**JP04**] for illustrations. In gene expression experiments usual numbers of genes are in the range of several thousands whereas the number of conditions is between 4 and 30, sometimes up to 100 conditions are tested. One immediately sees that there really is a need for good clustering methods as handcrafting appropriate clusterings is beyond a human being's capability.

In general there are a lot of questions arising from the clustering problem to be answered in practice. The most important are:

- How many clusters does one want to have which translates to choosing the right K?
- The choice of an appropriate distance.
- Based on the distance how to measure inner- and intra-cluster distance.

9.2. Hierarchical Clustering

The basic idea of hierarchical clustering is that clusters themselves can be partitioned into subclusters. Seen it the other way round, one starts with a huge number of small clusters which are iteratively merged into larger clusters. The corresponding procedure can be displayed as an organization of the data points into a tree where the data points themselves correspond to the leaves. Vertices in the tree correspond to clusters that consist of all the leaves hanging off the vertex. See Figs. 10.3, 10.4, [**JP04**] for illustrations. Edges are labeled such that the distance between two data points is just the sum of all labels on the (unique) path from one data point to another. Drawing a horizontal line through the tree translates to choosing a clustering. The horizontal line crosses a certain number of (vertically displayed) edges which are translated to clusters by choosing the vertices which come next when going down along the edges. The number of clusters obviously corresponds to the number of edges crossed.

The Hierarchical Clustering algorithm can be seen as a procedure which constructs the related clustering trees. It starts by choosing the pair of leaves which are closest among all pairs of data points and connects them. Then it proceeds by iteratively connecting vertices in the tree and producing the corresponding inner vertices. As distances are given for pairs of leaves only, it remains to come up with a method for computing distances between arbitrary vertices in the tree.

HierarchicalClustering($D = (d_{ij})_{i,j=1,...,n}, K$):

1: Form n clusters, each with one element.
2: Construct a graph T by assigning an isolated vertex to each cluster.
3: **while** there is more than K clusters **do**
4: Find the two closest clusters C_1, C_2.
5: Merge C_1 and C_2 into a new cluster $C = C_1 \cup C_2$.
6: Compute the distance between C and all other clusters.
7: Add a new vertex C to T and connect to vertices C_1 and C_2.
8: Remove rows and columns of D corresponding to C_1 and C_2.
9: Add a row and column to D for the new cluster C.
10: **end while**
11: **return** T

As mentioned earlier, one has to address the computation of the distance between arbitrary clusters (see line 6 in the algorithm). Some prevalent choices are

$$d_{min}(C^*, C) = \min_{x_i \in C^*, x_j \in C} d_{ij}$$

which is the smallest distance between any pair of elements from the two clusters or

$$d_{avg}(C^*, C) = \frac{1}{|C^*||C|} \sum_{x_i \in C^*, x_j \in C} d_{ij}.$$

It is a hierarchical clustering procedure which was first employed to group gene expression patterns (from experiments where human genes were tested for growth response when starving them). Though by far not being the gold standard in clustering gene expression patterns (until now there has not been established a standard) pictures of trees resulting from this clustering techniques are ubiquitous and most biologists would apply this method as a fist choice certainly because of the underlying intuitive tree representation. However, when trying to couple corresponding clusterings to biological entities, results are relatively poor.

9.3. K-means clustering

If data points are elements of some multidimensional space there is an alternative popular method for clustering. Given a set of n data points in form of m-dimensional real-valued vectors $x_i = (x_{i1}, ..., x_{im}), i = 1, ..., n$ and a predetermined number of clusters K the problem is to find K *centers* in m-dimensional space that minimize the *squared error distortion* defined below. Centers are points $y_1, ..., y_K$ simply are points in m-dimensional space themselves. We define the distance $d(x_i, Y)$ of a data point x_i to the centers as

$$d(x_i, Y) := \min_{k \in \{1,...,K\}} d(x_i, y_k)$$

where $d(x_i, y_k)$ is a common choice of a distance in multidimensional space. Here, the most popular choice is the Euclidean distance

$$d(x_i, y_k) = \sqrt{\sum_{j=1}^{m} (x_{ij} - y_{kj})^2}.$$

The squared error distortion $d(X, Y)$ for the data points $X = (x_1, ..., x_n)$ and a choice of centers $Y = (y_1, ..., y_K)$ is then calculated as

$$d(X, Y) := \frac{1}{n} \sum_{i=1}^{n} d(x_i, Y)^2.$$

K-Means Clustering Problem:
Given n data points $X = (x_1, ..., x_n)$ in multidimensional space find K center points $Y = (y_1, ..., y_K)$ which minimize the squared error distortion.

Input: A dataset $X = \{x_1, ..., x_n\}$ and a parameter K.
Output: A set $Y = (y_1, ..., y_K)$ of K points (centers) that minimize the squared error distortion $d(X, Y)$.

Having found a set of centers that minimize the squared error distortion one obtains a clustering of the data points by assigning them to the closest center and collecting the points of a center into a cluster.

Although the K-Means Clustering problem looks simple, there is no efficient algorithm for finding a *global* optimum. The *Lloyd* K-Means Clustering Algorithm is one of the most popular heuristics. At least it finds a local optimum to the optimization problem inherent to K-Means clustering. It proceeds according to the following steps:

(1) Choose a set of centers randomly.
(2) Assign each data point to the closest center.
(3) Recompute the center as the center of gravity of the data points assigned to it.
(4) Exit and return the corresponding centers if this results in no improvement in the squared error distortion. Otherwise go back to (2).

As the Lloyd algorithm finds a local optimum the iteration will finally be stopped. Using other criteria than the squared error distortion lead to similar problems where $\sum_{i=1}^{n} d(x_i, Y)$ (*K-Median Problem)* or $\max_{1 \leq i \leq n} d(x_i, Y)$ are popular alternative choices.

It must be emphasized that all of these methods are only concerned with obtaining clusters of minimal intra-cluster distance (optimal homogeneity) rather than maximal inter-cluster distance (optimal separation). Indeed, clusterings from the K-Means algorithm can be arbitrarily bad with respect to inter-cluster distance for unlucky choices of data points and initial centers.

In another approach, in each iteration only one of the data points is chosen for reassignment. To put it in its most general fashion, assume that one has chosen a cost function *cost* which measures the quality of the clustering (the squared error distortion is a particular choice). Given a partition P of the data points into clusters, define $P_{x_i \to C}$ to be the partition resulting from P by reassigning x_i to cluster C. The overall clustering cost $cost(P)$ is improved if

$$\Delta(x_i \to C) := cost(P) - cost(P_{x_i \to C}) > 0.$$

Choosing in each iteration a data point x_i and a cluster C such that $\Delta(x_i \to C)$ is maximized results in the following algorithm.

PROGRESSIVEGREEDYK-MEANS($X = (x_1, ..., x_n), K$):

```
1: Select an arbitrary partition of X into K clusters.
2: while forever do
3:   bestChange ← 0
4:   for every cluster C do
5:     for every data point x_i do
6:       if Δ(x_i → C) > bestChange then
7:         bestChange ← Δ(x_i → C)
8:         i* ← i
9:         C* ← C
10:      end if
11:    end for
12:  end for
13:  if bestChange > 0 then
14:    Change partition P into P_{i*→C*}.
15:  else
16:    return P
17:  end if
18: end while
```

9.4. Corrupted Cliques

In expression analysis, the distance matrix (d_{ij}) of the distances between all gene expression patterns x_i, x_j is often transformed into a distance graph $G = G(\theta)$. Its vertices are the gene expression patterns and an edge is drawn between two patterns if the distance is lower than a threshold θ, that is $d_{ij} < \theta$. Graph-based clustering procedures are then applied to the distance graph in order to obtain sets of clusters.

DEFINITION 9.1. A *complete graph*, written K_n, is an undirected graph on n vertices with every two vertices connected by an edge. A *clique graph* is a graph in which every connected component is a complete graph.

Fig. 10.5, [**JP04**] shows a clique graph consisting of three connected components, K_3, K_5 and K_6. The idea of clustering the vertices of a graph is that of transforming the graph into a clique graph by adding or removing only a small number of edges and subsequently identifying the connected

components with clusters.

DEFINITION 9.2. A subset $V' \subset V$ of vertices in a graph $G = (V, E)$ is a *complete subgraph* if the *induced subgraph* (that is the graph given by the vertices V' and precisely the edges from E between them) is complete. A *clique* is a complete subgraph which is maximal in the sense of that adding another vertex and its corresponding edges is not a complete subgraph any more.

See Fig. 10.5, [**JP04**] for examples of cliques and complete subgraphs which are not cliques.

Based on these definitions, we can formalize the problem from above and apply its solution to distance graphs.

Corrupted Cliques Problem:
Determine the smallest number of edges that need to be removed or added to transform a graph into a clique graph.
Input: A graph G.
Output: The smallest number of additions and removals of edges that will transform G into a clique graph.

It turns out the Corrupted Cliques problem is $\mathcal{NP}$-hard. We will two heuristics to solve it one of which is theoretically sound, but time-consuming whereas for the other one the opposite qualities apply.

It must be noted that there is a general problem with distance graphs made from gene expression patterns. Gene expression data is very noisy in general and a mistaken choice of a threshold will remove substantial edges or keep noisy artificial ones. Noise is a general problem in the analysis of gene expression which explains that statistical procedures have outperformed methods which are rely on the data as is. All methods presented so far are non-statistical in nature.

9.4.1. Parallel Classification with Cores (PCC). Suppose we attempt to cluster the whole set of vertices V and that we are given a correct clustering for a subset V', that is we know that $V' = C_1 \dot{\cup} ... \dot{\cup} C_K$ for clusters $C_k, k = 1, ..., K$. It would be a natural idea to extend the clustering of V' into one of V by assigning each vertex $v_i \in S \setminus S'$ to the cluster C_k which maximizes the *affinity*

$$\frac{N(x_i, C_k)}{|C_k|}$$

where $N(x_i, C_k)$ is the number of edges connecting x_i with vertices from C_k. In this way we would iteratively add vertices v_i to clusters C_k such that $\frac{N(x_i,C_k)}{|C_k|}$ is maximized.

However, a correct clustering of a subset of vertices is not known in general. The PCC algorithm overcomes this problem by starting the procedure from above for every choice of clustering on a random selection V'. Having obtained a clustering of all vertices it checks for how many edges have to be removed or added to turn the partition into a clique graph. This number is denoted by $score(P)$ where P is the partition. The output is the best partition obtained by this procedure. Note that the computation of number of edges to be removed or added is efficient in case that the partition is known.

PCC(G, K):

1: $V \leftarrow$ set of vertices in the distance graph
2: $n \leftarrow |V|$
3: $bestScore \leftarrow \infty$
4: Randomly select a "very small" set $V' \subset V$ where $|V'| = \log\log n$.
5: Randomly select a "small" set $V' \subset V$ where $|V''| = \log n$.
6: **for** every partition P' of V' into K clusters **do**
7: Extend P' into a partition P'' of V''.
8: Extend P'' into a partition P of V.
9: **if** $score(P) > bestScore$ **then**
10: $bestScore \leftarrow score(P)$
11: $bestPartition \leftarrow P$
12: **end if**
13: **end for**
14: **return** $bestPartition$

The number of iterations that PCC requires is on the order of the number of possible partitions of the set V', that is

$$K^{|V'|} = O(K^{\log\log n}) = O((\log n)^{\log K}) = O((\log n)^C).$$

In each iteration, the amount of work is $O(n^2)$ resulting in $O(n^2(\log n)^C$ overall. However, this is too slow for most applications to gene expression patterns.

9.4.2. Cluster Affinity Search Technique (CAST). Define the distance $d(x_i, C_k)$ between vertex v_i and cluster of vertices C_k as

$$d(x_i, C_k) := \frac{\sum_{x_j \in C_k} d_{ij}}{|C_k|}.$$

Given a threshold θ, a vertex x_i is *close* to cluster C_k if $d(x_i, C_k) < \theta$ and *distant* otherwise. The rough idea of the CAST algorithm is to iteratively construct partitions by finding clusters C such that no vertex $v_i \notin C$ is close to C and no vertex $v_i \in C$ is distant.

CAST(G, θ):

1: $V \leftarrow$ set of vertices in the distance graph G
2: $P \leftarrow \emptyset$
3: **while** $V \neq \emptyset$ **do**
4: $v \leftarrow$ vertex of maximal degree in the distance graph G
5: $C \leftarrow \{v\}$

```
    while there exists a close x_i ∉ C or a distant x_i ∈ C do
      Find the closest vertex x_i ∉ C and add it to C.
      Find the farthest distant vertex x_i ∈ C and remove it from C.
    end while
    Add cluster C to partition P.
    V ← V \ C
    Remove vertices of cluster C from the distance graph G.
end while
return P
```

Although CAST comes with no performance guarantee at all it works remarkably well with gene expression data in general.

Bibliography

[DEKM98] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[JP04] Neil C. Jones and Pavel Pevzner. *An Introduction to Bioinformatics Algorithms*. MIT press, 2004.