13 Markov chains and Hidden Markov Models

We will discuss:

- Markov chains
- Hidden Markov Models (HMMs)
- Profile HMMs

This chapter is based on: S. Durbin, S. Eddy, A. Krogh and G. Mitchison, Biological Sequence Analysis, Cambridge, 1998

13.1 Markov chains

Example: finding CpG-islands in the human genome.

Double stranded DNA:

```
...| | | | | | | | | | | | | | | | | | | | | | | | | | | | | ...
```

The C in a CpG pair is often modified by methylation (that is, an H-atom is replaced by a CH$_3$-group). There is a relatively high chance that the methyl-C will mutate to a T. Hence, CpG pairs are under-represented in the human genome.

Upstream of a gene, the methylation process is suppressed in short regions of the genome of length 100-5000. These areas are called CpG-islands and they are characterized by the fact that we see more CpG-pairs in them then elsewhere.

13.2 CpG-islands

CpG-islands are useful marks for genes in organisms whose genomes contain 5-methyl-cytosine.

CpG-islands in the promoter-regions of genes play an important role in the deactivation of a copy of the X-chromosome in females, in imprinting and in the deactivation of intra-genomic parasites.

Classical definition: DNA sequence of length 200 with a C + G content of 50% and a ratio of observed-to-expected number of CpG’s that is above 0.6. (Gardiner-Garden & Frommer, 1987)

According to a recent study, human chromosomes 21 and 22 contain about 1100 CpG-islands and about 750 genes. (Comprehensive analysis of CpG islands in human chromosomes 21 and 22, D. Takai & P. A. Jones, PNAS, March 19, 2002)
13.3 Questions

1. Given a short segment of genomic sequence. How to decide whether this segment comes from a CpG-island or not?

2. Given a long segment of genomic sequence. How to find all contained CpG-islands?

13.4 Markov chains

Our goal is to set up a probabilistic model for CpG-islands. Because pairs of consecutive nucleotides are important in this context, we need a model in which the probability of one symbol depends on the probability of its predecessor. This leads us to a Markov chain.

Example:

Circles = states, e.g. with names A, C, G and T.
Arrows = possible transitions, each labeled with a transition probability \( a_{st} = P(x_i = t \mid x_{i-1} = s) \).

Definition A (time-homogeneous) Markov chain (of order 1) is a system \((S, A)\) consisting of a finite set of states \( S = \{s_1, s_2, \ldots, s_n\} \) and a transition matrix \( A = \{a_{st}\} \) with \( \sum_{t \in S} a_{st} = 1 \) for all \( s \in S \), that determines the probability of the transition \( s \to t \) as follows:

\[
P(x_{i+1} = t \mid x_i = s) = a_{st}.
\]

(At any time \( i \) the chain is in a specific state \( x_i \) and at the tick of a clock the chain changes to state \( x_{i+1} \) according to the given transition probabilities).

Example Weather in Tübingen, daily at midday: Possible states are rain, sun, clouds or tornado.

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>S</th>
<th>C</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>.5</td>
<td>1</td>
<td>.4</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>.2</td>
<td>.6</td>
<td>.2</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>.3</td>
<td>.3</td>
<td>.4</td>
<td>0</td>
</tr>
</tbody>
</table>

Weather: ...rrrrrcsssssssssscccccrrrcssss...

Given a sequence of states \( x_1, x_2, x_3, \ldots, x_L \). What is the probability that a Markov chain will step through precisely this sequence of states?
\[ P(x) = P(x_L, x_{L-1}, \ldots, x_1) \]
\[ = P(x_L \mid x_{L-1}, \ldots, x_1)P(x_{L-1} \mid x_{L-2}, \ldots, x_1) \ldots P(x_1), \]
(by repeated application of \( P(X, Y) = P(X \mid Y)P(Y) \))
\[ = P(x_L \mid x_{L-1})P(x_{L-1} \mid x_{L-2}) \ldots P(x_2 \mid x_1)P(x_1) \]
\[ = P(x_1) \prod_{i=2}^{L} a_{x_{i-1} x_i}, \]

because \( P(x_i \mid x_{i-1}, \ldots, x_1) = P(x_i \mid x_{i-1}) = a_{x_{i-1} x_i} \), the Markov chain property!

### 13.5 Modeling the begin and end states

In the previous discussion we overlooked the fact that a Markov chain starts in some state \( x_1 \), with initial probability of \( P(x_1) \).

We add a \textit{begin state} to the model that is labeled \( 'b' \). We will always assume that \( x_0 = b \) holds. Then:

\[ P(x_1 = s) = a_{bs} = P(s), \]

where \( P(s) \) denotes the background probability of symbol \( s \).

Similarly, we explicitly model the end of the sequence of states using an \textit{end state} \( 'e' \). Thus, the probability that we end in state \( t \) is

\[ P(x_L = t) = a_{x_L e}. \]

### 13.6 Extension of the model

Example:

# Markov chain that generates CpG islands
(Source: DEKM98, p 50)
Number of states:
6
State labels:
A C G T * +
13.7 Determining the transition matrix

The transition matrix $A^+$ for DNA that comes from a CpG-island, is determined as follows:

$$ a_{st}^+ = \frac{c_{st}^+}{\sum_{t'} c_{st'}^+}, $$

where $c_{st}$ is the number of positions in a training set of CpG-islands at which state $s$ is followed by state $t$.

We obtain $A^-$ empirically in a similar way.

13.8 Two examples of Markov chains

<table>
<thead>
<tr>
<th># Markov chain for CpG islands</th>
<th># Markov chain for non-CpG islands</th>
</tr>
</thead>
<tbody>
<tr>
<td># (Source: DEKM98, p 50)</td>
<td># (Source: DEKM98, p 50)</td>
</tr>
<tr>
<td># Number of states: 6</td>
<td># Number of states: 6</td>
</tr>
<tr>
<td># State labels: A C G T * +</td>
<td># State labels: A C G T * +</td>
</tr>
<tr>
<td># Transition matrix:</td>
<td># Transition matrix:</td>
</tr>
<tr>
<td>.1795 .2735 .4255 .1195 0 0.002</td>
<td>.2995 .2045 .2845 .2095 0 .002</td>
</tr>
<tr>
<td>.1705 .3665 .2735 .1875 0 0.002</td>
<td>.3215 .2975 .0775 .0775 0 .002</td>
</tr>
<tr>
<td>.1605 .3385 .3745 .1245 0 0.002</td>
<td>.2475 .2455 .2975 .2075 0 .002</td>
</tr>
<tr>
<td>.0785 .3545 .3835 .1815 0 0.002</td>
<td>.1765 .2385 .2915 .2915 0 .002</td>
</tr>
<tr>
<td>.2495 .2495 .2495 .2495 0 .002</td>
<td>.2495 .2495 .2495 .2495 0 .002</td>
</tr>
<tr>
<td>.0000 .0000 .0000 .0000 0 1.000</td>
<td>.0000 .0000 .0000 .0000 0 1.000</td>
</tr>
</tbody>
</table>

13.9 Answering question 1

Given a short sequence $x = (x_1, x_2, \ldots, x_L)$. Does it come from a CpG-island (model$^+$)?

$$ P(x \mid \text{model}^+) = \prod_{i=0}^{L} a_{x_i \cdot x_{i+1}}, $$

with $x_0 = b$ and $x_{L+1} = e$.

We use the following score:

$$ S(x) = \log \frac{P(x \mid \text{model}^+)}{P(x \mid \text{model}^-)} = \sum_{i=0}^{L} \log \frac{a_{x_{i-1}x_i}^+}{a_{x_{i-1}x_i}^-}. $$
The higher this score is, the higher the probability is, that \( x \) comes from a \( \text{CpG} \)-island.

### 13.10 Questions that a Markov chain can answer

**Example** weather in Tübingen, daily at midday: Possible states are rain, sun or clouds.

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R )</td>
<td>.5</td>
<td>.2</td>
<td>.3</td>
</tr>
<tr>
<td>( S )</td>
<td>.1</td>
<td>.6</td>
<td>.3</td>
</tr>
<tr>
<td>( C )</td>
<td>.4</td>
<td>.2</td>
<td>.4</td>
</tr>
</tbody>
</table>

Types of questions that the model can answer:
- If it is sunny today, what is the probability that the sun will shine for the next seven days?
- How large is the probability, that it will rain for a month?

### 13.11 Hidden Markov Models (HMM)

Motivation: Question 2, how to detect \( \text{CpG} \)-islands inside a long sequence?

E.g., window techniques: a window of width \( w \) is moved along the sequence and the score is plotted. Problem: it is hard to determine the boundaries of \( \text{CpG} \)-islands, which window size \( w \) should one choose?...

Approach: Merge the two Markov chains model\(^+\) and model\(^-\) to obtain a so-called *Hidden Markov Model*.

### 13.12 Hidden Markov Models

**Definition** A *HMM* is a system \( M = (\mathcal{S}, Q, A, e) \) consisting of

- an alphabet \( \mathcal{S} \),
- a set of states \( Q \),
- a matrix \( A = \{a_{kl}\} \) of transition probabilities \( a_{kl} \) for \( k, l \in Q \), and
- an emission probability \( e_k(b) \) for every \( k \in Q \) and \( b \in \mathcal{S} \).

### 13.13 Example

An HMM for \( \text{CpG} \)-islands:
(Additionally, we have all transitions between states in either of the two sets that carry over from the two Markov chains model$^+$ and model$^-$.)

13.14 **HMM for CpG-islands**

<table>
<thead>
<tr>
<th># Number of states:</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td># Names of states (begin/end, A+, C+, G+, T+, A-, C-, G- and T-):</td>
<td></td>
</tr>
<tr>
<td>0 A C G T a c g t</td>
<td></td>
</tr>
<tr>
<td># Number of symbols:</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td># Names of symbols:</td>
<td></td>
</tr>
<tr>
<td>a c g t</td>
<td></td>
</tr>
<tr>
<td># Transition matrix, probability to change from +island to -island (and vice versa) is 10E-4</td>
<td></td>
</tr>
</tbody>
</table>

| Transition matrix, probability to change from +island to -island (and vice versa) is 10E-4 |
|---|---|---|---|---|---|---|
| 0.0000000000 | 0.0725193101 | 0.1637630296 | 0.1789247200 | 0.0754545682 | 0.1226380452 | 0.1278901313 |
| 0.0100000000 | 0.1762237676 | 0.2669517483 | 0.4170624970 | 0.1322050994 | 0.1267006624 | 0.1226380452 |
| 0.0100000000 | 0.1672451303 | 0.3992019597 | 0.2679840319 | 0.4138722556 | 0.1322050994 | 0.0506906169 |
| 0.0100000000 | 0.1576223776 | 0.3188811119 | 0.3671588671 | 0.1223776224 | 0.0312167832 | 0.0074950855 |
| 0.0100000000 | 0.1574245793 | 0.3470514486 | 0.3795405597 | 0.1789247200 | 0.0105794216 | 0.0077625277 |
| 0.0100000000 | 0.0029979103 | 0.002047952 | 0.0028273713 | 0.002907952 | 0.0041900594 | 0.0249504906 |
| 0.0100000000 | 0.0021267903 | 0.002977023 | 0.0007822921 | 0.000316689 | 0.0213586434 | 0.2974504906 |
| 0.0100000000 | 0.0024579102 | 0.002877023 | 0.000297952 | 0.024755480 | 0.0245049064 | 0.0079443558 |
| 0.0010000000 | 0.0007863512 | 0.0002387612 | 0.0028970583 | 0.0002917083 | 0.0176643536 | 0.2885224775 |
| 0.0010000000 | 0.0002387612 | 0.0028970583 | 0.0002917083 | 0.1766435363 | 0.2885224775 | 0.2914155844 |
| 0.0010000000 | 0.1766435363 | 0.2885224775 | 0.2914155844 |

From now one we use 0 for the begin and end state.

13.15 **Example fair.loaded dice**

Casino uses two dice, fair and loaded:

Casino guest only observes the number rolled:
6 4 3 2 3 4 6 5 1 2 3 4 5 6 6 3 2 1 2 6 3 4 2 1 6 6...
Which dice was used remains hidden:
F F F F F F F F F F F F U U U U U F F F F F F F F F F...

13.16 Example urns model

Given \( p \) urns \( U_1, U_2, \ldots, U_p \). Each urn \( U_i \) contains \( r_i \) red, \( g_i \) green and \( b_i \) blue balls. An urn \( U_i \) is randomly selected and from it a random ball \( k \) is taken (with replacement). The color of the ball \( k \) is reported.

\[
\begin{array}{c}
\text{r1 red} \\
\text{g1 green} \\
\text{b1 blue} \\
\end{array}
\quad
\begin{array}{c}
\text{r2 red} \\
\text{g2 green} \\
\text{b2 blue} \\
\end{array}
\quad
\begin{array}{c}
\text{rp red} \\
\text{gp green} \\
\text{bp blue} \\
\end{array}
\]

\( r \ r \ g \ g \ b \ b \ g \ g \ g \ b \ b \ b \ r \ g \ g \ b \ b \ b \ g \ g \ b \ g \ g \ b \ldots \)
Again, (a part of) the actual state (namely which urn was chosen) is hidden.

13.17 HMM for the urns model

```
# Four urns
# Number of states:
5
# Names of states:
# (0 begin/end, and urns A-D)
0 A B C D
# Number of symbols:
3
# red, green, blue
r g b
# Transition matrix:
0 .25 .25 .25 .25
0.01 .69 .30 0 0
0.01 0 .69 .30 0
0.01 0 0 .69 .30
0.01 .30 0 0 .69
# Emission probabilities:
0 0 0
.8 .1 .1
.2 .5 .3
.1 .1 .8
.3 .3 .4
# EOF
```
13.18 Generation of simulated data

We can use HMMs to generate data:

Algorithm

Start in state 0.

While we have not reentered state 0:

Choose a new state using the transition probabilities

Choose a symbol using the emission probabilities and report it.

13.19 A sequence generated for the casino example

We use the fair/loaded HMM to generate a sequence of states and symbols:

Symbols: 2433564261134166666526562426612134635535566462666636664253
States : FFFFFFFFFFFFFFFUUUUUUUUUUUUUUUUUUFFFFFFFFUUUUUUUUUUUUUFFFF

Symbols: 352463632521655615445656366665111145445656621261532516435
States : FFFFFFFFFFFFFFFFFUUUUUUUUUUUUFFFFFFFFF

Symbols: 5146526666
States : FFUUUUUU

How probable is a given sequence of data?

If we can observe only the symbols, can we reconstruct the corresponding states?

13.20 Determining the probability, given the states and symbols

Definition A path $\pi = (\pi_1, \pi_2, \ldots, \pi_L)$ is a sequence of states in the model $M$.

Given a sequence of symbols $x = (x_1, \ldots, x_L)$ and a path $\pi = (\pi_1, \ldots, \pi_L)$ through $M$. The joint probability is:

$$P(x, \pi) = a_{0\pi_1} \prod_{i=1}^{L} e_{\pi_i}(x_i)a_{\pi_i\pi_{i+1}},$$

with $\pi_{L+1} = 0$.

Unfortunately, we usually do not know the path through the model.

13.21 “Decoding” a sequence of symbols

Problem: We have observed a sequence $x$ of symbols and would like to “decode” the sequence:
Example: The sequence of symbols \texttt{C G C G} has a number of explanations within the \texttt{CpG}-model, e.g.: 
\((C_+, G_+, C_+, G_+), (C_-, G_-, C_-, G_-)\) and \((C_-, G_+, C_-, G_+)\).

A path through the HMM determines which parts of the sequence \(x\) are classified as \texttt{CpG}-islands, such a classification of the observed symbols is called a \textit{decoding}.

### 13.22 The most probable path

To solve the decoding problem, we want to determine the path \(\pi^*\) that maximizes the probability of having generated the sequence \(x\) of symbols, that is:

\[
\pi^* = \arg \max_{\pi} P(x, \pi).
\]

This \textit{most probable path} \(\pi^*\) can be computed recursively.

\textbf{Definition:} Given a prefix \((x_1, x_2, \ldots, x_i)\), let \(v_k(i)\) denote the probability that the most probable path is in state \(k\) when it generates symbol \(x_i\) at position \(i\). Then:

\[
v_l(i + 1) = e_l(x_{i+1}) \max_{k \in Q} (v_k(i) a_{lk}),
\]

with \(v_0(0) = 1\), initially.

(Exercise: We have: \(\arg \max_{\pi} P(x, \pi) = \arg \max_{\pi} P(\pi \mid x)\))

### 13.23 Most probable path

\[
\begin{array}{cccccccccc}
  & x_0 & x_1 & x_2 & x_3 & \ldots & x_{i-2} & x_{i-1} & x_i & x_{i+1} \\
G_+ & G_+ & G_+ & G_+ & G_+ & G_+ & G_+ & G_+ & G_+ & G_+ \\
T_+ & T_+ & T_+ & T_+ & T_+ & T_+ & T_+ & T_+ & T_+ & T_+ \\
0 & A_- & A_- & A_- & A_- & A_- & A_- & A_- & A_- & A_- \\
C_- & C_- & C_- & C_- & C_- & C_- & C_- & C_- & C_- & C_- \\
G_- & G_- & G_- & G_- & G_- & G_- & G_- & G_- & G_- & G_- \\
T_- & T_- & T_- & T_- & T_- & T_- & T_- & T_- & T_- & T_- \\
\end{array}
\]
13.24 The Viterbi-algorithm

Input: HMM $M = (S, Q, A, e)$ and symbol sequence $x$
Output: Most probable path $\pi^*$.  

Initialization ($i = 0$): $v_0(0) = 1$, $v_k(0) = 0$ for $k \neq 0$.

For all $i = 1 \ldots L, l \in Q$: $v_l(i) = e_l(x_i) \max_{k \in Q} (v_k(i-1)a_{kl})$
$\text{ptr}_i(l) = \arg \max_{k \in Q} (v_k(i-1)a_{kl})$

Termination: $P(x, \pi^*) = \max_{k \in Q} (v_k(L)a_{k0})$
$\pi^*_L = \arg \max_{k \in Q} (v_k(L)a_{k0})$

Traceback: For all $i = L - 1 \ldots 1$: $\pi^*_{i-1} = \text{ptr}_i(\pi^*_i)$

Implementation hint: instead of multiplying many small values, add their logarithms!
(Exercise: Run-time complexity)

13.25 Example for Viterbi

Given the sequence C G C G and the HMM for CpG-islands. Here is a table of possible values for $v$:

<table>
<thead>
<tr>
<th>$v$</th>
<th>C</th>
<th>G</th>
<th>C</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A_+</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C_+</td>
<td>0</td>
<td>.13</td>
<td>0</td>
<td>.012</td>
</tr>
<tr>
<td>State</td>
<td>G_+</td>
<td>0</td>
<td>0</td>
<td>.034</td>
</tr>
<tr>
<td>T_+</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A_-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C_-</td>
<td>0</td>
<td>.13</td>
<td>0</td>
<td>.0026</td>
</tr>
<tr>
<td>G_-</td>
<td>0</td>
<td>0</td>
<td>.010</td>
<td>0</td>
</tr>
<tr>
<td>T_-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
13.26 Viterbi-decoding of the casino example

We used the *fair.loaded* HMM to first generate a sequence of symbols and then use the Viterbi-algorithm to decode the sequence, result:

Symbols: 243356426134666665624266121346355355566666666664253
States : FFFFFFFFFFFFFFFUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
Viterbi: FFFFFFFFFFFFFFFUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU

Symbols: 35246363252165515445666366666651114544656621261532516435
States : FFFFFFFFFFFFFFFUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
Viterbi: FFFFFFFFFFFFFFFUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU

Symbols: 5146526666
States : FFUUUUUUUU
Viterbi: FFFFFUUUUUU

13.27 Three “Hauptprobleme” for HMMs

Let $M$ be an HMM, $x$ a sequence of symbols.

(Q1) For $x$, determine the most probable sequence of states through $M$: *Viterbi-algorithm*

(Q2) Determine the probability that $M$ generated $x$: $P(x) = P(x \mid M)$: *forward-algorithm*

(Q3) Given $x$ and perhaps some additional sequences of symbols, how do we train the parameters of $M$? E.g., *Baum-Welch-algorithm*

13.28 Computing $P(x \mid M)$

Assume we are given an HMM $M$ and a sequence of symbols $x$. For the probability, that $x$ was generated by $M$ we have:

$$P(x \mid M) = \sum_{\pi} P(x, \pi \mid M),$$

summing over all possible state sequences $\pi$ through $M$!

(Exercise: how fast does the number of paths increase as a function of length?)

13.29 Forward-algorithm

This algorithm is obtained from the Viterbi-algorithm by replacing max by a sum. More precisely, we define the *forward-variable*:

$$f_k(i) = P(x_1 \ldots x_i, \pi_i = k),$$

that equals the probability, that model reports the prefix sequence $(x_1, \ldots, x_i)$ and reaches in state $\pi_i = k$. 
We obtain the recursion: \( f_l(i + 1) = e_l(x_{i+1}) \sum_{k \in Q} f_k(i) a_{kl} \).

\[
\begin{align*}
  f_p(i) \bullet a_{kl} \\
  f_q(i) \quad \bullet f_l(i+1) \\
  f_r(i) \\
  f_s(i) \\
\end{align*}
\]

Input: HMM \( M = (\mathcal{S}, Q, A, e) \)
and sequence of symbols \( x \)

Output: probability \( P(x \mid M) \)

Initialization \( (i = 0) \): \( f_0(0) = 1, f_k(0) = 0 \) for \( k \neq 0 \).

For all \( i = 1 \ldots L, l \in Q \): \( f_l(i) = e_l(x_i) \sum_{k \in Q} f_k(i - 1) a_{kl} \)

Result: \( P(x \mid M) = \sum_{k \in Q} (f_k(L) a_{k0}) \)

Implementation hint: Logarithms can not be employed here easily, but there are scaling methods...

This solves “Hauptproblem” Q2!

### 13.30 Backward-algorithm

The backward-variable contains the probability to start in state \( p_i = k \) and then to generate the suffix sequence \( (x_{i+1}, \ldots, x_L) \): \( b_k(i) = P(x_{i+1} \ldots x_L \mid \pi_i = k) \).

Input: HMM \( M = (\mathcal{S}, Q, A, e) \)
and sequence of symbols \( x \)

Output: probability \( P(x \mid M) \)

Initialization \( (i = L) \): \( b_k(L) = a_{k0} \) for all \( k \).

For all \( i = L - 1 \ldots 1, k \in Q \): \( b_k(i) = \sum_{l \in Q} a_{kl} e_l(x_{i+1}) b_l(i + 1) \)

Result: \( P(x \mid M) = \sum_{l \in Q} (a_{0l} e_l(x_1) b_l(1)) \)
13.31 Comparison of the three variables

Viterbi \( v_k(i) \) probability, with which the most probable state path generates the sequence of symbols \( (x_1, x_2, \ldots, x_i) \) and the system is in state \( k \) at time \( i \).

Forward \( f_k(i) \) probability, that the prefix sequence of symbols \( x_1, \ldots, x_i \) is generated, and the system is in state \( k \) at time \( i \).

Backward \( b_k(i) \) probability, that the system starts in state \( k \) at time \( i \) and then generates the sequence of symbols \( x_{i+1}, \ldots, x_L \).

13.32 Posterior probabilities

Assume an HMM \( M \) and a sequence of symbols \( x \) are given.

Let \( P(\pi_i = k \mid x) \) denote the probability that the HMM is in state \( \pi_i = k \), given that the symbol \( x_i \) is reported. We call this the posterior probability, as it is computed after observing the sequence \( x \).

We have:

\[
P(\pi_i = k \mid x) = \frac{P(\pi_i = k, x)}{P(x)} = \frac{f_k(i)b_k(i)}{P(x)},
\]

as \( P(g, h) = P(g \mid h)P(h) \) and by definition of the forward- and backward-variable.

13.33 Decoding with posterior probabilities

There are alternatives to the Viterbi-decoding that are useful e.g., when many other paths exist that have a similar probability to \( \pi^* \).

We define a sequence of states \( \hat{\pi} \) thus:

\[
\hat{\pi}_i = \arg \max_{k \in Q} P(\pi_i = k \mid x),
\]
in other words, at every position we choose the most probable state for that position.

This decoding may be useful, if we are interested in the state at a specific position \( i \) and not in the whole sequence of states.

Warning: if the transition matrix forbids some transitions (i.e., \( a_{kl} = 0 \)), then this decoding may produce a sequence that is not a valid path, because its probability is 0!

13.34 Training the parameters

How does one generate an HMM?

**First step:** Determine its “topology”, i.e. the number of states and how they are connected via transitions of non-zero probability.
**Second step:** Set the parameters, i.e. the transition probabilities $a_{kl}$ and the emission probabilities $e_k(b)$.

We consider the second step. Given a set of example sequences. Our goal is to “train” the parameters of the HMM using the example sequences, e.g. to set the parameters in such a way that the probability, with which the HMM generates the given example sequences, is maximized.

### 13.35 Supervised learning

*Supervised learning:* estimation of parameters when both input (symbols) and output (states) are provided.

Let $M = (S, Q, A, e)$ be an HMM.

Given a list of sequences of symbols $x^1, x^2, \ldots, x^n$ and a list of corresponding paths $\pi^1, \pi^2, \ldots, \pi^n$. (E.g., DNA sequences with annotated CpG-islands.)

We want to choose the parameters $(A, e)$ of the HMM $M$ *optimally*, such that:

$$P(x^1, \ldots, x^n, \pi^1, \ldots, \pi^n \mid M = (S, Q, A, e)) = \max_{(A', e')} P(x^1, \ldots, x^n, \pi^1, \ldots, \pi^n \mid M = (S, Q, A', e')).$$

In other words, we want to determine the *Maximum Likelihood Estimator* (*ML-estimator*) for $(A, e)$.

### 13.36 ML-Estimation for $(A, e)$

(Recall: If we consider $P(D \mid M)$ as a function of $D$, then we call this a *probability*, as a function of $M$, then we use the word *likelihood*.)

ML-estimation:

$$(A, e)^{\text{ML}} = \arg \max_{(A', e')} P(x^1, \ldots, x^n, \pi^1, \ldots, \pi^n \mid M = (S, Q, A', e')).$$

Computation:

- $A_{kl}$: Number of transitions from state $k$ to $l$
- $E_k(b)$: Number of emissions of $b$ in state $k$

We obtain the ML-estimation for $(A, e)$ by setting:

$$a_{kl} = \frac{A_{kl}}{\sum_{q \in Q} A_{kq}} \quad \text{and} \quad e_k(b) = \frac{E_k(b)}{\sum_{s \in S} E_k(s)}.$$

### 13.37 Training the *fair.loaded* HMM

Given example data $x$ and $\pi$: 
Symbols $x$: 1 2 5 3 4 6 1 2 6 6 3 2 1 5  
States $\pi$: F F F F F F U U U F F F  

State transitions:

<table>
<thead>
<tr>
<th>$A_{kl}$</th>
<th>0</th>
<th>F</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

$\Rightarrow$

<table>
<thead>
<tr>
<th>$a_{kl}$</th>
<th>0</th>
<th>F</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Emissions:

<table>
<thead>
<tr>
<th>$E_k(b)$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\Rightarrow$

<table>
<thead>
<tr>
<th>$e_k(b)$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Given example data $x$ and $\pi$:

Symbols $x$: 1 2 5 3 4 6 1 2 6 6 3 2 1 5  
States $\pi$: F F F F F F U U U F F F  

State transitions:

<table>
<thead>
<tr>
<th>$A_{kl}$</th>
<th>0</th>
<th>F</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

$\Rightarrow$

<table>
<thead>
<tr>
<th>$a_{kl}$</th>
<th>0</th>
<th>F</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Emissions:

<table>
<thead>
<tr>
<th>$E_k(b)$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\Rightarrow$

<table>
<thead>
<tr>
<th>$e_k(b)$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

13.38 Pseudocounts

One problem in training is overfitting. For example, if some possible transition $k \rightarrow l$ is never seen in the example data, then we will set $\bar{a}_{kl} = 0$ and the transition is then forbidden.

If a given state $k$ is never seen in the example data, then $\bar{a}_{kl}$ is undefined for all $l$.

To solve this problem, we introduce pseudocounts $r_{kl}$ and $r_k(b)$ and define:

$$A_{kl} = \text{number of transitions from } k \text{ to } l \text{ in the example data} + r_{kl}$$
$$E_k(b) = \text{number of emissions of } b \text{ in } k \text{ in the example data} + r_k(b).$$

Small pseudocounts reflect “little pre-knowledge”, large ones reflect “more pre-knowledge”.

13.39 Unsupervised learning

Unsupervised learning: In practice, one usually has access only to the input (symbols) and not to the output (states).
Given sequences of symbols $x^1, x^2, \ldots, x^n$, for which we do NOT know the corresponding state paths $\pi^1, \ldots, \pi^n$.

Unfortunately, the problem of choosing the parameters $(A, e)$ of HMM $M$ optimally so that
\[
P(x^1, \ldots, x^n \mid M = (S, Q, A, e)) = \max_{(A', e')} P(x^1, \ldots, x^n \mid M = (S, Q, A', e'))
\]
holds, is \textit{NP-hard}.

13.40 Log-likelihood

Given sequences of symbols $x^1, x^2, \ldots, x^n$.

Let $M = (S, Q, A, e)$ be an HMM. We define the score of the model $M$ as:
\[
l(x^1, \ldots, x^n \mid (A, e)) = \log P(x^1, \ldots, x^n \mid (A, e)) = \sum_{j=1}^n \log P(x^j \mid (A, e)).
\]
(Here we assume, that the sequences of symbols are independent and therefore $P(x^1, \ldots, x^n) = P(x^1) \cdot \ldots \cdot P(x^n)$ holds.)

The goal is to chooses parameters $(A, e)$ so that we maximize this score, called the log likelihood:
\[
(A, e) = \arg \max_{(A', e')} l(x^1, \ldots, x^n \mid (A', e')).
\]

13.41 The expected number of transitions and emissions

Suppose we are given an HMM $M$ and training data $x^1, \ldots, x^n$. The probability that transition $k \rightarrow l$ is used at position $i$ in sequence $x$ is:
\[
P(\pi_i = k, \pi_{i+1} = l \mid x, (A, e)) = \frac{f_{k}(i)a_{kl}e_l(x_{i+1})b_l(i+1)}{P(x)}.
\]
(This follows from:
\[
P(\pi_i = k, \pi_{i+1} = l \mid x, (A, e)) = \frac{P(\pi_i = k, \pi_{i+1} = l, x \mid (A, e))}{P(x)} = \frac{P(x_i = k, x_{i+1} = l, x_{i+1}, \ldots \mid (A, e))}{P(x)} = \frac{f_{k}(i)a_{kl}e_l(x_{i+1})b_l(i+1)}{P(x)}.
\]

An estimation for the expected number of times that transition $k \rightarrow l$ is used is given by summing over all positions and all training sequences:
\[
A_{kl} = \sum_j \frac{1}{P(x^j)} \sum_i f_{k}^j(i) a_{kl} e_l(x_{i+1}^j) b_l^j(i+1),
\]
where $f_{k}^j$ and $b_l^j$ are the forward and backward variable computed for sequence $x^j$, respectively.

The expected number of times that letter $b$ appears in state $k$ is given by:
\[
E_k(b) = \sum_j \frac{1}{P(x^j)} \sum_{\{i \mid x_i^j = b\}} f_{k}^j(i) b_l^j(i),
\]
where the inner sum is only over those positions $i$ for which the symbol emitted is $b$. 
13.42 The Baum-Welch-algorithm

Let $M = (S, Q, A, e)$ be an HMM and suppose that training sequences $x^1, x^2, \ldots, x^n$ are given. The parameters $(A, e)$ are to be iteratively improved as follows:

- Using the current value of $(A, e)$, estimate the expected number $\bar{A}_{kl}$ of transitions from state $k$ to state $l$ and the expected number $\bar{e}_k(b)$ of emissions of symbol $b$ in state $l$.

- Then, set
  
  $a_{kl} = \frac{\bar{A}_{kl}}{\sum_{q \in Q} \bar{A}_{kq}}$ and $e_k(b) = \frac{\bar{E}_k(b)}{\sum_{s \in S} \bar{E}_k(s)}$.

- This is repeated until some halting criterion is met.

This is a special case of the so-called EM-technique. (EM = expectation maximization).

**Baum-Welch Algorithm**

**Input:** HMM $M = (S, Q, A, e)$, training data $x^1, x^2, \ldots, x^n$

**Output:** HMM $M' = (S, Q, A', e')$ with an improved score.

**Initialize:** Set $A$ and $e$ to arbitrary model parameters

**Recursion:**
For every sequence $x^j$:

- Compute $f^j$, $b^j$ and $P(x^j)$
- Increase $\bar{A}_{kl}$ by $\frac{1}{P(x^j)} \sum_i f^j_k(i)a_{kl}e_i(x_{i+1}^j)\mathbb{1}(i + 1)$
- Increase $\bar{E}_k(b)$ by $\frac{1}{P(x^j)} \sum_{\{i | x^j_i = b\}} f^j_k(i)b_l^j(i)$

Add pseudocounts to $\bar{A}_{lk}$ and $\bar{E}_k(b)$, if desired

Set $a_{kl} = \frac{\bar{A}_{kl}}{\sum_{q \in Q} \bar{A}_{kq}}$ and $e_k(b) = \frac{\bar{E}_k(b)}{\sum_{s \in S} \bar{E}_k(s)}$. Determine the new Log-likelihood $l(x^1, \ldots, x^n | (A, e))$

**End:** Terminate, when the score does not improve or a maximum number of iterations is reached.

13.43 Convergence

**Remark** One can prove that the log-likelihood-score converges to a local maximum using the Baum-Welch-algorithm.

However, this doesn’t imply that the parameters converge!

Local maxima can be avoided by considering many different starting points.

Additionally, any standard optimization approach can also be applied to solve the optimization problem.
13.44 Viterbi training

Here is an alternative to the Baum-Welch training algorithm:

Let $M = (S, Q, A, e)$ be an HMM and suppose that training sequences $x^1, x^2, \ldots, x^n$ are given. The parameters $(A, e)$ are to be iteratively improved as follows:

- Use the Viterbi algorithm to compute the currently most probable paths $\pi_1, \ldots, \pi_n$.
- Based on these paths, let $A_{kl}$ be the number of transitions from state $k$ to $l$ and $E_k(b)$ be the number of emissions of $b$ in state $k$.
- Update the model parameters:
  
  $a_{kl} = \frac{A_{kl}}{\sum_{q \in Q} A_{kq}}$ and $e_k(b) = \frac{E_k(b)}{\sum_{s \in S} E_k(s)}$.

- This is repeated until the halting criterion is met.

13.45 Example of Baum-Welch training

Assume that a casino uses two dice, fair and loaded. Here are the original HMM and two models obtained using the Baum-Welch algorithm, one based on a sequence of 300 and other on 30,000 observations (see DEKM98, pg. 65-66):

![Diagram showing the original and Baum-Welch models with parameters]

13.46 Pairwise Alignment and FSA

Pairwise alignment with affine gap penalties can be described by a finite state automaton (FSA). For example, compare the recursion for global alignment with the corresponding FSA:

- $V^M(i, j) = s(x_i, y_j) + \max \left\{ V^M(i-1, j-1), V^X(i-1, j-1), V^Y(i-1, j-1) \right\}$
- $V^X(i, j) = \max \left\{ V^M(i-1, j) - d, V^X(i-1, j) - e \right\}$
- $V^Y(i, j) = \max \left\{ V^M(i, j-1) - d, V^Y(i, j-1) - e \right\}$

Can we turn the FSA into an HMM?
13.47 Pair HMMs

A **pair HMM** is an HMM that emits pairs of symbols. We obtain a pair HMM from the FSA by making two sets of changes. First, we must give probabilities to emissions and transitions:

There are two free parameters for transitions between the three main states, $\delta$ the probability of a transition from $M$ to $X$ or $Y$ and $\epsilon$, the probability of staying in an insert state ($X$ or $Y$).

Second, we must add a begin and end state to provide a probability distribution over all possible sequences:

We assume here that the probability of transitioning into the end state from any other state is $\tau$.

13.48 Viterbi algorithm for pair HMMs

**Input:** Sequences $x = x_1 \ldots x_n$ and $y = y_1 \ldots y_m$.

**Output:** Alignment based on most probable path

**Init.:** Set $v^M(0,0) = 1$, set all other $v(i,0), v(0,j)$ to 0.

For $i = 1, \ldots, n$, $j = 1, \ldots, m$ do:

**For $v^M$:**

\[
v^M(i, j) = p_{x_iy_j} \max \left\{ \begin{array}{l}
(1 - 2\delta - \tau)v^M(i-1, j-1), \\
(1 - \epsilon - \tau)v^X(i-1, j-1), \\
(1 - \epsilon - \tau)v^Y(i-1, j-1), 
\end{array} \right. 
\]

**For $v^X$:**

\[
v^X(i, j) = q_{x_i} \max \left\{ \begin{array}{l}
dv^M(i-1, j) - d, \\
ev^X(i-1, j) - e; 
\end{array} \right. 
\]

**For $v^Y$:**

\[
v^Y(i, j) = q_{y_j} \max \left\{ \begin{array}{l}
dv^M(i, j-1) - d, \\
ev^Y(i, j-1) - e. 
\end{array} \right. 
\]

**Termination:**

\[
v^E = \tau \max(v^M(n, m), v^X(n, m), v^Y(n, m)).
\]
One can show that the most probable path corresponds to an optimal global alignment (in the FSA sense).

### 13.49 Protein identification

Alignment of seven Globin sequences

### 13.50 Characterization?

How can one characterize a family of protein sequences?

- Exemplary sequence?
- Consensus sequence?
- Regular expression (Prosite):
- HMM?

### 13.51 Simple HMM
We first consider a simple HMM that is equivalent to a PSSM (Position Specific Score Matrix):

![Diagram of a simple HMM](image)

(The listed amino-acids have a higher emission-probability.)

13.52 **Insert-states**

We introduce so-called *insert*-states that emit symbols based on their background probabilities.

![Diagram of insert-states](image)

This allows us to model segments of sequence that lie outside of conserved domains.

13.53 **Delete-states**

We introduce so-called *delete*-states that are silent and do not emit any symbols.

![Diagram of delete-states](image)

This allows us to model the absence of individual domains.
13.54 Topology of a profile-HMM

![Diagram of a profile-HMM topology]

- Match-state, ◇ Insert-state, □ Delete-state

13.55 Design of a profile-HMM

Given a multiple alignment of a family of sequences.

First we must decide which positions are to be modeled as match- and which positions are to be modeled as insert-states. Rule-of-thumb: columns with more than 50% gaps should be modeled as insert-states.

We determine the transition and emission probabilities simply by counting the observed transitions \( A_{kl} \) and emissions \( E_k(B) \):

\[
a_{kl} = \frac{A_{kl}}{ \sum_l A_{kl}' } \quad \text{and} \quad e_k(b) = \frac{E_k(b)}{ \sum_b E_k(b') }.
\]

Obviously, it may happen that certain transitions or emissions do not appear in the training data and thus we use the Laplace-rule and add 1 to each count.
14 Gene Prediction

This exposition is based on the following sources, which are all recommended reading:


14.1 Introduction

In the 1960s, it was discovered that a gene and its protein product are colinear structures with a direct correlation between the triplets of nucleotides in the gene and the amino acids in the protein.

It soon became clear that genes can be difficult to determine, due to the existence of overlapping genes, and genes within genes etc.

Moreover, the paradox arose that the genome size of many eukaryotes does not correspond to “genetic complexity”, for example, the salamander genome is 10 times the size of that of human.

In 1977, the surprising discovery of “split” genes was made: genes that consist of multiple pieces of coding DNA called exons, separated by stretches of non-coding DNA called introns.

The existence of split genes and junk-DNA raises a computational gene prediction problem that is still unsolved:
Given a string of DNA. The gene prediction problem is to reliably predict all genes contained in the sequence.

## 14.2 Three approaches to gene finding

One can distinguish between three types of approaches:

- **Statistical or ab initio methods.** These methods attempt to predict genes based on statistical properties of the given DNA sequence. Programs are e.g. GENSCAN, GeneID, GENIE and FGENEH.

- **Homology methods.** The given DNA sequence is compared with known protein structures, e.g. using “spliced alignments”. Programs are e.g. Procrustes and GeneWise.

- **Comparative methods.** The given DNA string is compared with a similar DNA string from a different species at the appropriate evolutionary distance and genes are predicted in both sequences based on the assumption that exons will be well conserved, whereas introns will not. Programs are e.g. CEM (conserved exon method) and TWINSAN.

## 14.3 Simplest approach to gene prediction

The simplest way to detect potential coding regions is to look at *Open Reading Frames (ORFs)*. An ORF is a sequence of codons in DNA that starts with a Start codon (ATG), ends with a Stop codon (TAA, TAG or TGA) and has no other (in-frame) stop codons inside.

The average distance between stop codons in “random” DNA is $\frac{64}{3} \approx 21$, much smaller than the number of codons in an average protein ($\approx 300$).

Essentially, long ORFs indicate genes, whereas short ORF may or may not indicate genes or short exons.

Additionally, features such as *codon usage* or hexamer counts can be taken into account. The *codon usage* of a string of DNA is given by a 64-component vector that counts how many times each codon is present in the string. These values can differ significantly between coding and non-coding DNA.

## 14.4 Eukaryotic gene structure

For our purposes, a eukaryotic gene has the following structure:
Ab initio gene prediction methods use statistical properties of the different components of such a gene model to predict genes in unannotated DNA. For example, for the bases around the start site we may have the following observed frequencies (given by this position weight matrix):

<table>
<thead>
<tr>
<th>Pos.</th>
<th>-8</th>
<th>-7</th>
<th>-6</th>
<th>-5</th>
<th>-4</th>
<th>-3</th>
<th>-2</th>
<th>-1</th>
<th>+1</th>
<th>+2</th>
<th>+3</th>
<th>+4</th>
<th>+5</th>
<th>+6</th>
<th>+7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.16</td>
<td>.29</td>
<td>.20</td>
<td>.25</td>
<td>.22</td>
<td>.66</td>
<td>.27</td>
<td>.15</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>.28</td>
<td>.24</td>
<td>.11</td>
<td>.26</td>
</tr>
<tr>
<td>C</td>
<td>.48</td>
<td>.31</td>
<td>.21</td>
<td>.33</td>
<td>.36</td>
<td>.56</td>
<td>.05</td>
<td>.50</td>
<td>.58</td>
<td>0</td>
<td>0</td>
<td>.16</td>
<td>.29</td>
<td>.24</td>
<td>.40</td>
</tr>
<tr>
<td>G</td>
<td>.18</td>
<td>.16</td>
<td>.46</td>
<td>.21</td>
<td>.17</td>
<td>.27</td>
<td>.12</td>
<td>.22</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>.48</td>
<td>.20</td>
<td>.45</td>
<td>.21</td>
</tr>
<tr>
<td>T</td>
<td>.19</td>
<td>.24</td>
<td>.21</td>
<td>.21</td>
<td>.06</td>
<td>.02</td>
<td>.11</td>
<td>.05</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>.09</td>
<td>.26</td>
<td>.21</td>
<td>.21</td>
</tr>
</tbody>
</table>

14.5 GENSCAN’s model

We are going to discuss the popular program GENSCAN in detail, which is based on a semi-Markov model:

GENSCAN’s model can be formulated as an explicit state duration HMM. This is an HMM in which, additionally, a duration period is explicitly modeled for each state, using a probability distribution. The model is thought of generating a parse $\phi$, consisting of:

- a sequence of states $q = (q_1, q_2, \ldots, q_n)$, and
- an associated sequence of durations $d = (d_1, d_2, \ldots, d_n)$,

which, using probabilistic models for each of the state types, generates a DNA sequence $S$ of length $L = \sum_{i=1}^{n} d_i$.

The generation of a parse of a given sequence length $L$ proceeds as follows:

1. An initial state $q_1$ is chosen according to an initial distribution $\pi$ on the states, i.e. $\pi_i = P(q_1 = Q^{(i)})$, where $Q^{(j)}$ ($j = 1, \ldots, 27$) is an indexing of the states of the model.
2. A state duration or length $d_1$ is generated conditional on the value of $q_1 = Q^{(i)}$ from the duration distribution $f_{Q(i)}$.

3. A sequence segment $s_1$ of length $d_1$ is generated, conditional on $d_1$ and $q_1$, according to an appropriate sequence-generating model for state type $q_1$.

4. The subsequent state $q_2$ is generated, conditional on the value of $q_1$, from the (first-order Markov) state transition matrix $T$, i.e. $T_{i,j} = P(q_{k+1} = Q^{(j)} | q_k = Q^{(i)})$.

This process is repeated until the sum $\sum_{i=1}^{n} d_i$ of the state durations first equals or exceeds $L$, at which point the last state duration is appropriately truncated, the final stretch of sequence is generated and the process stops.

The resulting sequence is simply the concatenation of the sequence segments, $S = s_1s_2\ldots s_n$.

Note that the generated sequence is not restricted to correspond to a single gene, but could represent multiple genes, in both strands, or none.

In addition to its topology involving the 27 states and 46 transitions depicted above, the model has four main components:

- a vector of initial probabilities $\pi$,
- a matrix of state transition probabilities $T$,
- a set of length distributions $f$, and
- a set of sequence generating models $P$.

(Recall that an HMM has initial-, transition- and emission probabilities).

### 14.6 Maximum likelihood prediction

Given such a model $M$. For a fixed sequence length $L$, consider

$$\Omega = \Phi_L \times S,$$

where $\Phi_L$ is the set of all possible parses of $M$ of length $L$ and $S_L$ is the set of all possible sequences of length $L$.

The model $M$ assigns a probability density to each point (parse/sequence pair) in $\Omega$. Thus, for a given sequence $S \in S_L$, a conditional probability of a particular parse $\phi \in \Phi_L$ is given by:

$$P(\phi | S) = \frac{P(\phi, S)}{P(S)} = \frac{P(\phi, S)}{\sum_{\phi' \in \Phi_L} P(\phi', S)},$$

using $P(M, D) = P(M | D)P(D)$.

The essential idea is to specify a precise probabilistic model of what a gene looks like in advance and then to select the parse $\phi$ through the model $M$ that has highest likelihood, given the sequence $S$. 
14.7 Computational issues

Given a sequence $S$ of length $L$, the joint probability $P(\phi, S)$ of generating the parse $\phi$ and the sequence $S$ is given by:

$$P(\phi, S) = \pi_{q_1} f_{q_1}(d_1) P(s_1 | q_1, d_1)$$

$$\times \prod_{k=2}^{n} T_{q_{k-1}, q_k} f_{q_k}(d_k) P(s_k | q_k, d_k),$$

where the states of $\phi$ are $q_1, q_2, \ldots, q_n$ with associated state lengths $d_1, d_2, \ldots, d_n$, which break the sequence into segments $s_1, s_2, \ldots, s_n$.

Here, $P(s_k | q_k, d_k)$ is the probability of generating the segment $s_k$ under the appropriate sequence generating model for a type-$q_k$ state of length $d_k$.

A modification of the Viterbi algorithm may be used to calculate $\phi_{opt}$, the parse with maximal joint probability (under $M$), that gives the predicted gene or set of genes in the sequence.

We can compute $P(S)$ using the “forward algorithm” discussed under HMMs. With the help of the “backward algorithm”, certain additional quantities of interest can also be computed.

For example, consider the event $E^{(k)}_{[x,y]}$ that a particular sequence segment $[x,y]$ is an internal exon of phase $k \in \{0, 1, 2\}$. Under $M$, this event has probability

$$P(E^{(k)}_{[x,y]} | S) = \frac{\sum_{\phi: E^{(k)}_{[x,y]} \in \phi} P(\phi, S)}{P(S)},$$

where the sum is taken over all parses that contain the given exon $E^{(k)}_{[x,y]}$. This sum can be computed using the forward and backward algorithms.

14.8 Details of the model

So far, we have discussed the topology and the other main components of the GENSCAN model in general terms. The following details need to be discussed:

- the initial and transition probabilities,
- the state length distributions,
- transcriptional and translational signals,
- splice signals, and
- reverse-strand states.
14.9 Initial and transition probabilities

For gene prediction in randomly chosen blocks of contiguous human DNA, the initial probability of each state should be chosen proportionally to its estimated frequency in bulk human genomic DNA. This is a non-trivial problem, because gene density and certain aspects of gene structure vary significantly in regions of differing C+G content (so-called “isochores”) of the human genome, with a much higher gene density in C+G-rich regions.

Hence, in practice, initial and transitional probabilities are estimated for four different categories: (I) < 43% C+G, (II) 43 – 51% C+G, (III) 51 – 57% C+G, and (IV) > 57% C+G.

The following initial probabilities were obtained from a training set of 380 genes by comparing the number of bases corresponding to each of the different states:

<table>
<thead>
<tr>
<th>Group</th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>C+G-range</td>
<td>&lt; 43%</td>
<td>43 – 51%</td>
<td>51 – 57%</td>
<td>&gt; 57%</td>
</tr>
<tr>
<td>Initial probabilities:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intergenic (N)</td>
<td>0.892</td>
<td>0.867</td>
<td>0.540</td>
<td>0.418</td>
</tr>
<tr>
<td>Intron (I^+, I^-)</td>
<td>0.095</td>
<td>0.103</td>
<td>0.338</td>
<td>0.388</td>
</tr>
<tr>
<td>5’ UTR (F^+, F^-)</td>
<td>0.008</td>
<td>0.018</td>
<td>0.077</td>
<td>0.122</td>
</tr>
<tr>
<td>3’ UTR (T^+, T^-)</td>
<td>0.005</td>
<td>0.011</td>
<td>0.045</td>
<td>0.072</td>
</tr>
</tbody>
</table>

For simplicity, the initial probabilities for the exon, promoter and poly-A states were set to 0. Transition probabilities are obtained in a similar way.

14.10 State length distributions

In general, the states of the model correspond to sequence segments of highly variable length.

For certain states, most notably for internal exon states E_k, length is probably important for proper biological function, i.e. proper splicing and inclusion in the final processed mRNA.

For example, it has been shown in vivo that internal deletions of exons to sizes below about 50 bp may often lead to exon skipping, and there is evidence that steric interference between factors recognizing splice sites may make splicing of small exons more difficult. There is also evidence that spliceosomal assembly is inhibited if internal exons are expanded beyond 300 bp.

In summary, these arguments support the observation that internal exons are usually ≈ 120 – 150 bp long, with only a few of length less that 50 bp or more than 300 bp.

Constraints for initial and terminal exons are slightly different.

The duration in initial, internal and terminal exon states is modeled by a different empirical distribution for each of the types of states.

In contrast to exons, the length of introns does not seem critical, although a minimum length of 70 – 80 may be preferred.

The length distribution for introns appears to be approximately geometric (exponential). However, the average length of introns differs substantially between the different C+G groups: In group I, the average length is 2069 bp, whereas for group IV, the average length is only 518 bp.
Hence, the duration in intron states is modeled by a geometric distribution with parameter $q$ estimated for each $C+G$ group separately.

Empirical length distributions for introns and exons:

![Graphs showing length distributions](image)

Note that the exon lengths generated must be consistent with the phases of adjacent introns. To account for this, first the number of complete codons is generated from the appropriate length distribution, then the appropriate number (0, 1 or 2) of bp is added to each end to account for the phases of the preceding and subsequent states.

For example, if the number of complete codons generated for an internal exon is $C = 6$, and the phase of the previous and next intron is 1 and 2, respectively, then the total length of the exon is $l = 3C + 2 + 2 = 22$:

```
TA CGC GCT CGC TT ACTGTTTGT
```

For the 5' UTR and 3' UTR states, geometric distributions are used with mean values of 769 and 457 bp, respectively.

### 14.11 Simple signal models

There are a number of different models of biological signal sequences, such as donor and acceptor sites, promoters, etc.

One of the earliest and most influential approaches is the weight matrix method (WMM), in which the frequency $p_a^{(i)}$ of each nucleotide $a$ at position $i$ of a signal of length $n$ is derived from a collection of aligned signal sequences.

The product $P(A) = \prod_{i=1}^{n} p_{a_i}$ is used to estimate the probability of generating a particular sequence $A = a_1a_2 \ldots a_n$. 
The weight array matrix (WAM) is a generalization that takes dependencies between adjacent positions into account. In this model, the probability of generating a particular sequence is

\[ P(A) = p_{a_1} \prod_{i=2}^{n} p_{a_i-1,a_i} \]

where \( p_{a_i-1,a_i} \) is the conditional probability of generating a particular nucleotide \( w \) at position \( i \), given nucleotide \( v \) at position \( i - 1 \).

Here is a WMM for recognition of a start site:

<table>
<thead>
<tr>
<th>Pos.</th>
<th>-8</th>
<th>-7</th>
<th>-6</th>
<th>-5</th>
<th>-4</th>
<th>-3</th>
<th>-2</th>
<th>-1</th>
<th>+1</th>
<th>+2</th>
<th>+3</th>
<th>+4</th>
<th>+5</th>
<th>+6</th>
<th>+7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.16</td>
<td>.29</td>
<td>.20</td>
<td>.25</td>
<td>.22</td>
<td>.66</td>
<td>.27</td>
<td>.15</td>
<td>1</td>
<td>0</td>
<td>.28</td>
<td>.24</td>
<td>.11</td>
<td>.26</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>.48</td>
<td>.31</td>
<td>.21</td>
<td>.33</td>
<td>.56</td>
<td>.05</td>
<td>.50</td>
<td>.58</td>
<td>0</td>
<td>0</td>
<td>.16</td>
<td>.29</td>
<td>.24</td>
<td>.40</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>.18</td>
<td>.16</td>
<td>.46</td>
<td>.21</td>
<td>.17</td>
<td>.27</td>
<td>.12</td>
<td>.22</td>
<td>0</td>
<td>0</td>
<td>.48</td>
<td>.20</td>
<td>.45</td>
<td>.21</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>.19</td>
<td>.24</td>
<td>.14</td>
<td>.21</td>
<td>.06</td>
<td>.02</td>
<td>.11</td>
<td>.05</td>
<td>0</td>
<td>1</td>
<td>.09</td>
<td>.26</td>
<td>.21</td>
<td>.21</td>
<td></td>
</tr>
</tbody>
</table>

Under this model, the sequence \(...)CCGCCACC ATG GCGC(...) has the highest probability of containing a start site, namely:

\[ P = 0.48 \cdot 0.31 \cdot 0.46 \cdot 0.33 \cdot 0.56 \cdot 0.66 \cdot 0.5 \cdot 0.58 \cdot 1 \cdot 1 \cdot 1 \cdot 0.48 \cdot 0.29 \cdot 0.45 \cdot 0.4 = 0.006. \]

The sequence \(...)AGTTTTTT ATG TAAT(...) has the lowest non-zero probability of containing a start site at the indicated position, namely:

\[ P = 0.16 \cdot 0.16 \cdot 0.14 \cdot 0.21 \cdot 0.06 \cdot 0.02 \cdot 0.11 \cdot 0.05 \cdot 1 \cdot 1 \cdot 1 \cdot 0.09 \cdot 0.24 \cdot 0.11 \cdot 0.21 = 20.4 \cdot 10^{-11}. \]

14.12 Transcriptional and translational signals

Poly-A signals are modeled as a 6 bp WMM model with consensus sequence AATAAA.

A 12 bp WMM, beginning 6 bp prior to the start codon, is used for the translation initiation signal. In both cases, one can estimate the probabilities using the GenBank annotated “polyA signal” and “CDS” features of sequences.

Approximately 30% of eukaryotic promoters lack a TATA signal. Hence, a TATA-containing promoter is generated with 0.7 probability, and a TATA-less one with probability 0.3.

TATA-containing promoters are modeled as a 15 bp TATA WMM and an 8 bp cap site WMM. The length between the two WMMs is generated uniformly from the range 14 – 20 bp.

TATA-less ones are modeled as intergenic regions of 40 bp.

14.13 Splice signals

The donor and acceptor splice signals are probably the most important signals, as the majority of exons are internal ones. Previous approaches use WMMs or WAMs to model them, thus assuming independence of sites, or that dependencies only occur between adjacent sites.

The consensus region of the donor splice sites covers the last 3 bp of the exon (positions -3 to -1) and the first 6 bp of the succeeding intron (positions 1 to 6):
14.14 Donor site model

However, donor sites show significant dependencies between non-adjacent positions, which probably reflect details of donor splice site recognition by U1 snRNA and other factors.

Given a sequence \( S \). Let \( C_i \) denote the consensus indicator variable that is 1, if the given nucleotide at position \( i \) matches the consensus at position \( i \), and 0 otherwise. Let \( X_j \) denote the nucleotide at position \( j \).

For example, consider:

\[
\begin{array}{ccccccccccc}
\ldots & \text{exon} & \ldots & \text{intron} & \ldots \\
\text{Position} & -3 & -2 & -1 & +1 & +2 & +3 & +4 & +5 & +6 \\
\text{Consensus} & c/a & A & G & G & T & a/g & A & G & t \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
S & \ldots & T & A & A & C & G & T & A & A & G & C & C & \ldots \\
\end{array}
\]

Here, \( C_{-1} = 0 \) and \( C_{+6} = 0 \), and \( = 1 \), for all other positions. Similarly, \( X_{-3} = A, X_{-2} = A, X_{-1} = C \) etc.

For each pair of positions \( i \neq j \), consider the \( C_i \) versus \( X_j \) contingency table computed from the given learning set of gene structures:

\[
\begin{array}{c|cccc}
C_i & A & C & G & T \\
\hline
0 & f_0(A) & f_0(C) & f_0(G) & f_0(T) \\
1 & f_1(A) & f_1(C) & f_1(G) & f_1(T),
\end{array}
\]

where \( f_c(x) \) is the frequency at which the training set has the consensus indicator value \( c \) at position \( i \) and the base \( x \) at position \( j \). The (Pearson’s) \( \chi^2 \)-test assigns a score \( \chi^2(C_i, X_j) \) to each pair of variables \( C_i \) and \( X_j \):

\[
\chi^2(C_i, X_j) = \sum_{c \in \{0,1\}} \sum_{x \in \{A,C,G,T\}} \left( \frac{f_c(x) - f(x)}{f(x)} \right)^2,
\]

where \( f(x) \) denotes the frequency with which we observe \( X_j = x \) in the training set (corresponding to the null-hypothesis that \( X_j \) does not depend on \( C_i \)).

A significant score indicates that a dependency exists between \( C_i \) and \( X_j \).

In donor site prediction, the positions \( i \) are ordered by decreasing discriminatory power \( Z_i = \sum_{j \neq i} \chi^2(C_i, X_j) \) and separate WMMs for each of the different cases are derived, thus obtaining a so-called maximal dependence decomposition:
Here, $H = A|C|U$, $B = C|G|U$ and $V = A|C|G$. For example, $G_5$, or $H_5$, is the set of donor sites with, or without, a $G$ at position +5, respectively.

### 14.15 Acceptor site model

Intron/exon junctions are modeled by a (first-order) WAM for bases $-20$ to $+3$, capturing the pyrimidine (C,T) rich region and the acceptor splice site itself.

It is difficult to model the branch point in the preceding intron, and only 30% of the test data had an YYRAY sequence in the appropriate region $[-40, -21]$.

A modified variant of a second-order WAM is employed in which nucleotides are generated conditional on the previous two ones, in an attempt to model the weak but detectable tendency toward YYY triplets as well as certain branch point-related triplets such as TGA, TAA, GAC, and AAC in this region, without requiring the occurrence of any specific branch point consensus.

(A windowing and averaging process is used to obtain estimates from the limited training data.)

### 14.16 Exon models

Coding portions of exons are modeled using an inhomogeneous 3-periodic fifth order Markov model. Here, separate Markov transition matrices, $c_1$, $c_2$ and $c_3$, are determined for hexamers ending at each of the three codon positions, respectively.
This is based on the observation that frame-shifted hexamer counts are generally the most accurate compositional discriminator of coding versus non-coding regions.

However, A+T rich genes are often not well predicted using hexamer counts based on bulk DNA and so Genscan uses two different sets of transition matrices, one trained for sequences with < 43% C+G content and one for all others.

14.17 Example of Genscan summary output

```
GENSCAN 1.0 Date run: 28-Apr-104 Time: 02:56:56
Sequence HUMAN DNA : 36741 bp : 52.90% C+G : Isochore 3 (51 - 57 C+G%)
Parameter matrix: HumanIso.smat
Predicted genes/exons:
Gn.Ex Type S .Begin ...End .Len Fr Ph I/Ac Do/T CodRg P.... Tscr..
----- ---- - ------ ------ ---- -- -- ---- ---- ----- ----- ------
1.01 Intr + 3420 3538 119 1 2 104 66 32 0.106 3.29
1.02 Intr + 3950 4063 114 1 0 22 61 133 0.276 5.15
1.03 Term + 4310 4426 117 1 0 83 44 64 0.725 0.34
1.04 PlyA + 5029 5034 6

2.05 PlyA - 5519 5514 6
2.04 Term - 7619 7455 165 2 0 36 47 124 0.792 1.33
2.03 Intr - 9364 9309 56 2 2 90 84 48 0.603 3.79
2.02 Intr - 9557 9478 80 1 2 90 95 49 0.904 -0.71
2.01 Init - 9986 9910 77 2 2 90 90 58 0.701 6.79
2.00 Prom - 13449 13410 40

3.02 PlyA - 14233 14228 6
3.01 Sngl - 14748 14287 462 0 0 13 51 222 0.425 7.51
3.00 Prom - 17109 17070 40

4.00 Prom + 19924 19963 40

...```

14.18 Performance studies

The performance of a gene prediction program is evaluated by applying it to DNA sequences for which all contained genes are known and annotated with high confidence.

To calculate accuracy statistics, each nucleotide of a test sequence is classified as:

- a predicted positive (PP) if it is predicted to be contained in a coding region,
• a predicted negative (PN) if it is predicted to be contained in non-coding region,
• an actual positive (AP) if it is annotated to be contained in coding region, and
• an actual negative (AN) if it is annotated to be contained in non-coding region.

The performance is measured both on the level of nucleotides and on whole predicted exons, using a similar classification.

Based on this classification, we compute the number of:

• true positives, \( TP = PP \cap AP \),
• false positives, \( FP = PP \cap AN \),
• true negatives, \( TN = PN \cap AN \), and
• false negatives, \( FN = PN \cap AP \).

The sensitivity \( Sn \) and specificity \( Sp \) of a method are then defined as

\[
Sn = \frac{TP}{AP} \quad \text{and} \quad Sp = \frac{TP}{PP},
\]

respectively, measuring both the ability to predict true genes and to avoid predicting false ones.

### 14.19 Performance of GENSCAN

GENSCAN was run on a test set of 570 vertebrate sequences and the forward strand exons in the optimal GENSCAN parse of the sequence were compared to the annotated exons. The following table shows the results and compares them with results obtained using other programs:

<table>
<thead>
<tr>
<th>Program</th>
<th>Sequences</th>
<th>Accuracy per nucleotide</th>
<th>Accuracy per exon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sn</td>
<td>Sp</td>
</tr>
<tr>
<td>GENSILK</td>
<td>570 (8)</td>
<td>0.93</td>
<td>0.93</td>
</tr>
<tr>
<td>GENEH</td>
<td>569 (22)</td>
<td>0.77</td>
<td>0.88</td>
</tr>
<tr>
<td>Gene1D</td>
<td>570 (2)</td>
<td>0.63</td>
<td>0.81</td>
</tr>
<tr>
<td>Genie</td>
<td>570 (6)</td>
<td>0.76</td>
<td>0.77</td>
</tr>
<tr>
<td>GenLang</td>
<td>570 (60)</td>
<td>0.72</td>
<td>0.79</td>
</tr>
<tr>
<td>GeneParser2</td>
<td>562 (6)</td>
<td>0.66</td>
<td>0.79</td>
</tr>
<tr>
<td>GRAIL2</td>
<td>570 (23)</td>
<td>0.72</td>
<td>0.87</td>
</tr>
<tr>
<td>SORFIND</td>
<td>561 (6)</td>
<td>0.71</td>
<td>0.85</td>
</tr>
<tr>
<td>Xpoint</td>
<td>570 (28)</td>
<td>0.61</td>
<td>0.87</td>
</tr>
<tr>
<td>GeneID+</td>
<td>478 (1)</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>GeneParser3</td>
<td>478 (1)</td>
<td>0.86</td>
<td>0.91</td>
</tr>
</tbody>
</table>

(Source: Burge and Karlin 1997)

GENSCAN performs very well here and is currently the most popular gene finding method.
14.20 Comparative gene finding

GENSCAN’s model makes use of statistical features of the genome under consideration, obtained from an annotated training set.

More recently, a number of methods have been suggested that attempt to also make use of comparative data. They are based on the observation that

* the level of sequence conservation between two species depends on the function of the DNA, e.g. coding sequence is more conserved than intergenic sequence.

One such program is Rosetta, which first computes a global alignment of two homologous sequences and then attempts to predict genes in both sequences simultaneously. A second is the conserved exon method, that uses local conservation.

The TWINSCAN program is an extension of GENSCAN, that additionally models a conserved sequence.

14.21 TWINSCAN

The input to TWINSCAN consists of a *target sequence*, i.e. a genomic sequence in which genes are to be predicted, and an *informant sequence*, i.e. a genomic sequence from a related organism.

For example, the target may come from the mouse genome and the informant from the whole human genome.

Given a target and an informant, in a preprocessing step, one determines a set of *top homologs* (e.g. using BLAST) from the informant sequence, i.e. one or more sequences from the informant sequence that best match the target sequence.

```
mouse --------------------------------
     \-----------
     | conserved human (top homologs)
```

The top homologs represent the regions of conserved informant sequence, which we will simply call “the informant sequence” in the following.

14.22 Conservation sequence

Similarity is represented by a *conservation sequence*, which pairs one of three symbols with each nucleotide of the target:

```
. unaligned | matched : mismatched
```

Gaps in the informant sequence become mismatch symbols, gaps in the target sequence are ignored.

Consider:

```
123456789 position
GAATTCGCT target sequence
```
and suppose that BLAST yields the following HSP:

<table>
<thead>
<tr>
<th>345 6789 target position</th>
<th>123456789 position</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATT−CCGT target alignment</td>
<td>GAATTCCGT target sequence</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ATCACC−T Informant alignment</td>
<td></td>
</tr>
</tbody>
</table>

The conservation sequence derived from this HSP is:

```
ATT-CCGT
```

The following algorithm takes a list of HSPs and computes the conservation sequence $C$:

**Algorithm**

Input: target sequence, list of HSPs  
Output: conservation sequence $C$

Init.: $C[1..n] := \text{unaligned}$

Sort HSPs by alignment score

for each position $i$ in the target sequence:

for each HSP $H$ from best to worst:

if $H$ covers position $i$:

if $C[i] = \text{unaligned}$:

$C[i] := \text{'}\text{'}$, in case of a match, and $C[i] := \text{'}\text{'}$ otherwise

end

Note that the conservation symbol assigned to the target nucleotide at position $i$ is determined by the best HSP that covers $i$, regardless of which homologous sequence it comes from. Position $i$ is classified as unaligned only if none of the HSPs overlap it.

**14.23 Probability of sequence and conservation sequence**

Recall that GENSCAN assigns each nucleotide of an input sequence to one of seven categories: promoter, 5' UTR, exon, intron, 3' UTR, poly-A signal and intergenic.

GENSCAN chooses the most likely assignment of categories to nucleotides according to the GENSCAN model, using an optimization algorithm (that is, a modification of the Viterbi algorithm).

Given a sequence, the GENSCAN model assigns a probability to each parse of the sequence (that is, a path of states and durations through the model that generates the sequence.)

The TWINSSCAN model assigns a probability to any parsed DNA sequence together with a parallel conservation sequence. Under this model, the probability of a DNA sequence and the probability of the parallel conservation sequence are independent, given the parse.

Consider the following example:

```
10  20  30
123456789|123456789|123456789|123456789
ATTAGCCCTACTGAATGGAGCCGTTCCAAGATGGTATCC target sequence T
```
What is the probability of observing the target sequence \( T_{7,33} \) and conservation sequence \( C_{7,33} \) extending from position 7 to 33, given the hypothesis \( E_{7,33} \) that an internal exon extends from position 7 to 33?

This is simply the probability of the target sequence \( T_{7,33} \) under the GENSCAN model times the probability of the conservation sequence \( C_{7,33} \) under the conservation model, assuming the parse \( E_{7,33} \):

\[
P(T_{7,33}, C_{7,33} | E_{7,33}) = P(T_{7,33} | E_{7,33})P(C_{7,33} | E_{7,33}).
\]

### 14.24 TWINSCAN’s model

TWINSCAN consists of a new, joint probability model on DNA sequences and conservation sequences, together with the same optimization algorithm used by GENSCAN.

TWINSCAN augments the state-specific sequence models of GENSCAN with models for the probability of generating any given conservation sequence from any given state.

Coding, UTR, and intron/intergenic states all assign probabilities to stretches of conservation sequence using homogeneous 5th-order Markov chains:

\[
\begin{align*}
\text{ccc} & \quad c1 \quad c2 \quad c3 \quad c4 \quad c5 \quad c6 \quad \text{ccc} \\
\end{align*}
\]

One set of parameters is estimated for each of these types of regions.

Again, consider:

\[
\begin{align*}
\text{ATTTAGCCTACTGAAATGGACCGCTTCAGCATGGTATCC} & \quad \text{target sequence T} \\
\|:||:|.........|:|:|||||||||:||:|||::|| & \quad \text{conservation sequence C}
\end{align*}
\]

The probability of observing \( C_{7,33} \), given \( E_{7,33} \), is:

\[
P_C(C_{7,33} | E_{7,33}) = P_E(C_{7,7} | C_{2,6}) \cdot \ldots \cdot P_E(C_{33,33} | C_{28,32}),
\]

where \( P_E(C_{33,33} | C_{28,32}) \), for example, is the estimated probability of a \( \| \) (match) following the give context symbols “\( \|:||: \)” in the conservation sequence of an exon.

Models of conservation at splice donor and acceptor sites are modeled using 2nd-order WAMs of length 9 bp and 43 bp, respectively (lengths as in GENSCAN).

### 14.25 TWINSCAN’s performance

TWINSCAN was tested on two data sets. The first set consists of 86 mouse sequences totaling 7.6 Mb and used top homologs from human:
The second set is a subset containing 8 pairs of finished orthologs:

<table>
<thead>
<tr>
<th>Program</th>
<th>Exons</th>
<th>Exon Sn</th>
<th>Exon Sp</th>
<th>Genes</th>
<th>Genes Sn</th>
<th>Genes Sp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>610</td>
<td>0.798</td>
<td>0.666</td>
<td>51</td>
<td>0.167</td>
<td>0.157</td>
</tr>
<tr>
<td>GENSCAN</td>
<td>731</td>
<td>0.631</td>
<td>0.581</td>
<td>395</td>
<td>0.153</td>
<td>0.106</td>
</tr>
<tr>
<td>TWINSNAN</td>
<td>684</td>
<td>0.683</td>
<td>0.660</td>
<td>464</td>
<td>0.244</td>
<td>0.144</td>
</tr>
</tbody>
</table>

### 14.26 The conserved exon method (CEM)

Based on a model of sequence conservation, TWINSNAN uses an informant sequence to obtain better gene predictions for a given target sequence.

Input to the conserved exon method (CEM) are two related sequences and the method predicts gene structures in both sequences simultaneously. The underlying assumption is that exons are well preserved, whereas introns and intergenic DNA have very little similarity.

For this assumption to hold, the two input sequences must be at an appropriate evolutionary distance. Coding regions are generally well conserved in species as far back as 450 Myrs. At evolutionary distances of 50–100 Myrs (human and mouse), the conservation also extends to other functional regions important for gene expression and maintaining genome structure.

The main idea of CEM is to look for conserved protein sequences by comparing pairs of DNA sequences, to identify putative exons based on sequence and splice site conservation, and then to chain such pairs of conserved exons together to obtain gene structure predictions in both sequences.

**Identifying conserved coding sequence** The first part of the CEM is not new. For example, the tBLASTx program performs precisely this task. Additionally, a number of tools exist for comparing two genomic sequences, finding conserved exons and regulatory regions etc.

**Building gene models** The second part of the CEM is more interesting, in which gene structures are generated from the identified matches and complete gene structures are predicted in both input sequences.

### 14.27 Application of tBLASTx

Throughout the following, we are given two similar DNA sequences $S$ and $T$.

The program tBLASTx produces a list of high-scoring pairs (HSPs) of locally aligned substrings of $S$ and $T$, where the two substrings are interpreted as amino-acid coding strings and the score of the alignment is computed using a BLOSSUM or PAM protein scoring matrix.

This is how an HSP is reported by tBLASTx:

```
Score = 214 (98.4 bits), Expect = 0.0, Sum P(24) = 0.0
Identities = 44/46 (95%), Positives = 46/46 (100%), Frame = +1 / +1
```
In this example, the positions 5284–5421 of sequence $S$ and positions 3871–4008 of sequence $T$ are aligned together and interpreted as amino-acids as shown. The “frame” indicates the directions and the offsets of the two substrings.

**tBLASTx** matches between two similar pieces of human and mouse DNA:

### 14.28 Key assumption for conserved exons

Note that programs such as tBLASTx predict putative coding regions, but not actual splice boundaries. Also, many HSPs are due to other conserved features, not exons.

In the CEM, the local alignments produced by tBLASTx are used as seeds for dynamic programming alignments that are computed to detect complete exons.

**Key assumption** Any pair of conserved exons $E_1$ (in $S$) and $E_2$ (in $T$) possesses a **witness**, that is, an HSP $h$ whose middle codon is a portion of the correct local alignment of $E_1$ and $E_2$, in the correct frame.

![Diagram showing exons and HSP alignment](image-url)
**14.29 Conserved exon pairs**

A putative conserved exon pair (CEP) consists of a pair of substrings $E_1$ (in $S$) and $E_2$ (in $T$) that are both flanked by appropriate splice junctions and have a high scoring local amino-acid alignment. We now discuss how to obtain putative CEPs.

Given an HSP $h$. Let $m_S(h)$ and $m_T(h)$ denote the position of the middle codon of $h$ in $S$ and in $T$, respectively.

Let $b_S(h)$ and $e_S(h)$ denote the position of the left-most possible intron-exon splice site and right-most possible exon-intron splice site for any putative exon in $S$ that is witnessed by $h$. Define $b_T(h)$ and $e_T(h)$ in the same way.

In a simple approach, we use empirical bounds on the lengths of exons to find the values of $b_S$, $e_S$, $b_T$ and $e_T$. A more sophisticated approach takes the amount of coverage by HSPs etc. into account.

Start, stop and splice sites are detected by WMMs or more advanced techniques.

If the values of $b_S$, $e_S$, $b_T$ and $e_T$ were chosen large enough, then the key assumption implies that the two exons $E_1$ (in $S$) and $E_2$ (in $T$) of the true CEP (witnessed by $h$) will start in $[b_S(h), m_S(h)]$ and $[b_T(h), m_T(h)]$, and will end in $[m_S(h), e_S(h)]$ and $[m_T(h), e_T(h)]$, respectively.

We evaluate all possible pairs of exons in this region by running two dynamic programs: one starts at $(m_S(h), m_T(h))$ and ends at $(e_S(h), e_T(h))$, the other runs in reverse direction from $(m_S(h), m_T(h))$ to $(b_S(h), b_T(h))$:

![Diagram of exon alignment](image)

**14.30 Exon alignment**

The actual algorithms used for the local alignment computations are variants of the standard algorithm.

Note that the alignments are forced to start in the frame defined by the HSP. Frame-shifts are allowed subsequently (with an appropriate indel penalty).

Each splice-junction pair is a cell in the dynamic programming matrix, and its score is maintained in a separate list.

Let $(i, j)$ be the coordinates of a cell corresponding to a splice-pair $(z_S(h), z_T(h))$. The score assigned to $(z_S(h), z_T(h))$ is not $\text{Score}[i, j]$, but

$$\text{Score}(z_S(h), z_T(h)) = \max_{0 \leq k_S(h), k_T(h) \leq 2} \{\text{Score}[i - k_S(h)][j - k_T(h)]\}$$
This is to allow for the possibility of an intron splitting a codon. In this way, the alignment (which only scores codons) allows terminal nucleotide gaps without incurring a frame-shift penalty.

The amount of overhang $$(o_S(h), o_T(h)) = \arg \max_{0 \leq k_S(h), k_T(h) \leq 2} \{\text{Score}[i - k_S(h)][j - k_T(h)]\}$$ is also stored along with the score.

As the alignment is done at the protein level, there is a direction associated with it. The dynamic programming computation from the mid-point to the acceptor splice junctions is done by reversing each codon before scoring.

### 14.3.1 The CEP gadget

For each HSP $h$ we construct a *CEP gadget*. Each node $u$ in the CEP gadget corresponds to a coordinate pair $$(i, j),$$ which is the starting point, mid-point or terminating point of a candidate exon pair $$(E_1, E_2).$$ More precisely, $u$ is one of the following:

- a **center** node, if $$(i, j) = (m_S(h), m_T(h))$$ is the position of the middle codon of $h$,
- a **donor** node if $i \in [m_S(h), e_S(h)]$ & $j \in [m_T(h), e_T(h)]$ are sites of donor splice signals in $S$ and $T$,
- an **acceptor** node if $i \in [b_S(h), m_S(h)]$ and $j \in [b_T(h), m_T(h)]$ are sites of acceptor signals,
- a **start** node if $i \in [b_S(h), m_S(h)]$ and $j \in [b_T(h), m_T(h)]$ are sites of translation initiation signals, or
- a **terminal** node if $i \in [m_S(h), e_S(h)]$ and $j \in [m_T(h), e_T(h)]$ are sites for a stop codon.

Each node $u$ has some additional information associated with it. The coordinates of the cell are maintained as $$(u_S, u_T).$$ For each acceptor or donor node $u,$ we maintain information on the nucleotide overhang at the boundary as $\text{overhang}(u) = (o_S(u), o_T(u)).$

A directed edge is constructed from each acceptor or start node to the center, and from the center to each donor or terminal node. The weight of the edge is the score of the corresponding local alignment.
14.32 The CEM graph

As discussed above, each HSP gives rise to a CEP gadget. (In practice, however, different HSPs often lead to the same CEP gadget and such redundancies should be removed.)

Each CEP gadget is a concise representation of alignments of pairs of exons. At most one pair can actually be a conserved-exon-pair in the true gene structures. The Conserved-Exon-Method takes all CEP gadgets of HSPs and chains them together, thus obtaining the full “CEM graph”. It builds gene models from this graph based on the assumption that the transcripts derived from correct orthologous gene structures will have the highest alignment score.

Let $S$ and $T$ be the two genomic sequences.

For each HSP $h$, compute the CEP gadget. We build a candidate exon graph $G = (V, E)$ (which we call the CEM graph), as follows: $V$ is the union of all the nodes in the CEP gadgets, and $E$ contains all the edges in each CEP gadget. Further, add an edge from donor or terminal node $u$ to an acceptor or start node $v$ if both:

- $v_S \geq u_S + M$, and $v_T \geq u_T + M$, where $M$ is a suitably chosen minimum intron length, and:
- Let $(o_S(u), o_T(u)) = \text{overhang}(u)$, and $(o_S(v), o_T(v)) = \text{overhang}(v)$. Then, $(o_S(u) + o_S(v)) = 0(\text{mod } 3)$, and $(o_T(u) + o_T(v)) = 0(\text{mod } 3)$,

The weight of the edge $(u, v)$ is the score of aligning the amino-acids obtained by concatenating the overhangs on either side added to the penalty for an intron gap.

Example of linking two CEPs, nodes are labeled by their offsets $(o_S, o_T)$:

Example of a complete graph:
14.33 Obtaining a gene prediction

By construction, a path in the CEM graph corresponds to a prediction of orthologous gene structures in the two genomes. Based on the assumption that the correct gene models will have the highest alignment score, we can extract the correct gene structures simply by choosing the highest-scoring path. As this is a directed acyclic graph, the highest-scoring path can be computed via a topological sort:

\[
\text{getGeneModelScores}(\text{CEMGraph } G(V, E))
\begin{align*}
&\text{OrderedNodeList } L = \text{TopologicalSort}(G) \\
&\text{for each } v \text{ in } L \\
&\quad \text{Initialize}(\text{Score}(v)) \\
&\quad \text{for each incoming edge } e = (x, v) \\
&\quad\quad \text{if } (\text{Score}(v) < \text{Score}(x) + w(e)) \\
&\quad\quad\quad \text{Score}(v) = \text{Score}(x) + w(e) \\
&\quad\quad\quad \text{predecessor}(v) = x
\end{align*}
\]

(An ordering \( \phi \) of the nodes of an acyclic graph is a topological sorting if for any edge \((v, w)\) we have \( \phi(v) < \phi(w) \).

For an arbitrary node \( u \), \( \text{score}(u) \) is the best score of an alignment of two sequence prefixes \( S[1..u_i] \), and \( T[1..u_j] \), allowing for frame-shifts, amino-acid indels and intron penalties.

Once the scores on the nodes are computed, the gene models are built by starting at the node with the highest score, and following the predecessors. The coordinates of start-, terminal-, donor- and
acceptor nodes reveal the gene structure in the two genomic sequences. As the boundaries of the path are not limited to start and terminal nodes, partial gene structures can be predicted.

### 14.34 Multiple genes

Additionally, we add an edge from every stop node \( v \) with coordinates \((v_S, v_T)\) to every downstream start node \( w \) (with coordinates \( w_S > v_S \) and \( w_T > v_T \)). Such edges are given weight 0. The role of such edges is to allow prediction of multiple genes.

HSPs with negative frames in one or both sequences are possible witnesses for exons in the reverse strand of one or both sequences. The CEP gadgets derived from such HSPs are simply added to the CEM graph.

To enable a prediction of genes in both strands simultaneously, appropriate additional edges must be inserted between the start and stop nodes of the CEPs. For example, a stop node obtained from an HSP with \(+/+/\) frame is connected to all downstream stop nodes that have frame \(-/-\).

### 14.35 Summary of CEM algorithm

1. Compute HSPs using tBLASTx.
2. For each HSP \( h \), determine the range of possible exons and their possible splice sites.
3. For each HSP \( h \), build the corresponding CEP gadget.
4. Sort CEP gadgets lexicographically according to their ranges.
5. Compute the CEM graph by joining the CEP gadgets.
6. Compute the gene model scores.
7. Determine the highest-scoring path through the CEM graph.
8. Extract the corresponding gene model.

### 14.36 Performance of CEM

Here is a comparison of the performance of CEM and GENSCAN on a test data set of 60 pairs of gene from human and mouse:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CEM</td>
<td>120</td>
<td>0.76</td>
<td>0.80</td>
<td>0.94</td>
<td>0.95</td>
</tr>
<tr>
<td>GENSCAN</td>
<td>120</td>
<td>0.74</td>
<td>0.78</td>
<td>0.92</td>
<td>0.94</td>
</tr>
</tbody>
</table>

The gain in performance obtained is not spectacular. However, it provides a proof of concept and additional work may well lead to a useful tool for comparative gene finding, especially for genomes for which little is known of the statistical properties of the contained genes.
14.37 Homology method: Procrustes

Any newly sequence gene has a good chance of having an already known relative and progress in large-scale sequencing projects is rapidly increasing the number of known genes and protein sequences.

Hence, homology-based gene prediction methods are becoming more and more useful. In particular, such a method may be able to detect exons that are missed by statistical methods because they are small, or statistically unusual.

Procrustes is a popular program that uses homology to predict genes and is based on the following idea:

Given a genomic sequence $G$ and a target protein $P$. Determine a chain $\Gamma$ of blocks in $G$ that has the highest spliced-alignment score with target $T$. These blocks are interpreted as exons and the chain $\Gamma$ is the predicted gene structure.

14.38 Example

Given the genomic sequence $G = \text{baa baa black sheep have you any wool}$ and the target protein $T = \text{barbara sleeps on wool}$, find the best spliced alignment of $T$ to $G$ and thus obtain a gene prediction in $G$:

Genome sequence:

\begin{align*}
\text{baa baa black sheep have you any wool}
\end{align*}

Assume that these are the possible blocks:

- baa baa black sheep have you any wool
- baa baa black sheep have you any wool
- baa baa black sheep have you any wool

Best spliced alignment:

\begin{align*}
\text{barbara sleeps on wool}
\end{align*}

Resulting gene structure prediction:

\begin{align*}
\text{baa baa sheep any wool}
\end{align*}

There are many possible chainings of blocks in the given example:

\begin{align*}
\text{baa baa black sheep have you any wool}
\end{align*}

However, we choose the one that yields the best alignment to the given target sequence. In general, a number of possible target sequence will be given and then we choose the one that gives rise to the
best alignment.

14.39 Preprocessing: determining the blocks

Given a genomic sequence $G$. The first computational step is to determine the set $B$ of all candidate blocks for $G$, which should contain all true exons. Naively, this is done by selecting all blocks between potential acceptor and donor sites, which are detected using e.g. a WMM:

```
acacacG aggtAgttagggtctcagttcactgcacgtcagcagtG TatcactttacgacacGtacac;
block 1
block 2
block 3
block 4
```

Clearly, this set of blocks will contain many false exons. Statistical methods may be used in an attempt to remove blocks that are obviously not true exons.

Any chain of blocks corresponds to a gene prediction and the number of such chains can be huge. Dynamic programming is used to obtain an algorithm that runs in polynomial time.

14.40 The spliced alignment problem

Let $G = g_1 \ldots g_n$ be a string of letters, and $B = g_i \ldots g_j$ and $B' = g_{i'} \ldots g_{j'}$ be substrings of $G$. We write $B \prec B'$, if $B$ ends before $B'$ starts, i.e. $j < i'$. A sequence $\Gamma = (B_1, \ldots, B_b)$ of substrings of $G$ is a chain, if $B_1 \prec B_2 \prec \ldots \prec B_b$. We denote the concatenation of the strings in $\Gamma$ by $\Gamma^* = B_1^* B_2^* \ldots B_b$.

For two strings $A$ and $B$, we set $s(A, B)$ to the score of an optimal alignment between $A$ and $B$.

**Spliced Alignment Problem (SAP)** Let $G = g_1 \ldots g_n$ be a genomic sequence, $T = t_1 \ldots t_m$ a target sequence and $B = \{B_1, \ldots, B_b\}$ a set of blocks in $G$. Given $G$, $T$ and $B$, the Spliced Alignment Problem is to find a chain $\Gamma$ of strings from $B$ such that the score $s(\Gamma^*, T)$ is maximum among all chains of blocks from $B$.

14.41 Solving the spliced alignment problem

The SAP can be reduced to the search of a path in some (unweighted) graph. Vertices of this graph correspond to the blocks, arcs correspond to potential transitions between blocks, and the path weight is defined as the weight of the optimal alignment between the concatenated blocks of this path and the target sequence.

For simplicity, we will consider sequence alignment with linear gap penalties and define the $\Delta_{\text{match}}$, $\Delta_{\text{mismatch}}$ and $\Delta_{\text{indel}}$ scores as usual.

We set $\Delta(x, y) = \begin{cases} 
\Delta_{\text{match}} & \text{if } x = y, \\
\Delta_{\text{mismatch}} & \text{else.}
\end{cases}$
14.42 The score of a prefix alignment

For a block $B_k = g_m \ldots g_i$ in $G$, define $\text{first}(k) = m$, $\text{last}(k) = l$ and $\text{size}(k) = l - m + 1$. Let $B_k(i)$ denote the $i$-prefix $g_m \ldots g_i$ of $B_k$, if $m \leq i \leq l$.

Given a position $i$ and let $\Gamma = (B_1, \ldots, B_k, \ldots, B_t)$ be a chain such that some block $B_k$ contains $i$. We define

$$\Gamma^*(i) = B_1 * B_2 * \ldots * B_k(i)$$

as the concatenation of $B_1 \ldots B_{k-1}$ and the $i$-prefix of $B_k$.

Then

$$S(i, j, k) = \max_{\text{all chains } \Gamma \text{ containing block } B_k} s(\Gamma^*(i), T(j)),$$

is the optimal score for aligning a chain of blocks up to position $i$ in $G$ to the $j$-prefix of $T$. As we will see, the values of this matrix is computed using dynamic programming.

14.43 The dynamic program

Let $B(i) = \{ k \mid \text{last}(k) < i \}$ be the set of all blocks that end (strictly) before position $i$ in $G$. The following recurrence computes $S(i, j, k)$ for $1 \leq i \leq n$, $1 \leq j \leq m$ and $1 \leq k \leq b$:

$$S(i, j, k) = \max \begin{cases} S(i - 1, j - 1, k) + \Delta(g_i, t_j), & \text{if } i \neq \text{first}(k) \\ S(i - 1, j, k) + \Delta_{\text{indel}}, & \text{if } i \neq \text{first}(k) \\ \max_{l \in B(i)} S(\text{last}(l), j - 1, l) + \Delta(g_l, t_j), & \text{if } i = \text{first}(k) \\ \max_{l \in B(i)} S(\text{last}(l), j, l) + \Delta_{\text{indel}}, & \text{if } i = \text{first}(k) \\ S(i, j - 1, k) + \Delta_{\text{indel}}. & \end{cases}$$

The score of the optimal spliced alignment can be found as:

$$\max_k S(\text{last}(k), m, k).$$

Note that $S(i, j, k)$ is only defined if $i \in B_k$ and therefore only a portion of entries in the three-dimensional $n \times m \times b$ matrix $S$ needs to be computed, where $n = |G|$, $m = |T|$ and $b$ is the number of blocks.

The total number of such entries is:

$$m \sum_{k=1}^{b} \text{size}(k) = nmc,$$
where \( c = \frac{1}{n} \sum_{k=1}^{b} \text{size}(k) \) is the coverage of the genomic sequence by blocks.

Hence, a naive implementation of the recurrence runs in \( O(mncb) \) time.

### 14.44 Example

Consider the following string \( G \) with the possible blocks indicated by boxes:

| baa baa | black sheep have you any wool |

The recurrence corresponds to the following graph:

The target sequence is:

**barbara sleeps on wool**

Here are three spliced alignments of \( G \) and \( T \) obtainable from the above graph:

- **barbara sleeps on wool**
  - baa baa black sheep have you any wool
  - baa baa black any wool
  - baa baa sheep have you any wool
  - black sheep any wool

### 14.45 Speed up

The time and space requirements of the algorithm can be reduced significantly. Here we only discuss one such improvement.

Define \( P(i, j) = \max_{l \in B(i)} S(\text{last}(l), j, l) \). The recurrence can be rewritten as follows:

\[
S(i, j, k) = \max \begin{cases} 
S(i - 1, j - 1, k) + \Delta(g_i, t_j), & \text{if } i \neq \text{first}(k) \\
S(i - 1, j, k) + \Delta_{\text{indel}}, & \text{if } i \neq \text{first}(k) \\
P(i, j - 1) + \Delta(g_i, t_j), & \text{if } i = \text{first}(k) \\
P(i, j) + \Delta_{\text{indel}}, & \text{if } i = \text{first}(k) \\
S(i, j - 1, k) + \Delta_{\text{indel}}, & \end{cases}
\]
where

\[ P(i, j) = \max \left\{ P(i-1, j) \max_{l \in B(i) : \text{last}(l) = i-1} S(i-1, j, l) \right\}. \]

With this modification, we maintain and update the maximal score for all preceding blocks explicitly and thus do not reconsider all preceding blocks in each evaluation of the recurrence. This reduces the run time of the algorithm to \( O(mnc + mnb) \).

The corresponding network that indicates which computations are performed looks like this:

---

### 14.4.6 Evaluation of the method

The authors of Procrustes evaluated the performance of the program on a test sample of human genes with known mammalian relatives. In their study, the average correlation between the predicted and actual proteins was 99%. The algorithm correctly reconstructed 87% of the genes.

They also reported that the algorithm predicts human genes reasonably well when the homologous protein is non-vertebrate or even prokaryotic.

Additionally, predictions were made using simulated targets that gradually diverged from the analyzed gene. For targets up to 100 PAM distance, the predictions were almost 100% correct. (This distance roughly corresponds to 40% similarity).

This indicates that for an average protein family the method is likely to correctly predict a human gene given a mammalian relative.
15 Suffix trees

This lecture is based on the following sources, which are all recommended reading:


History:
Weiner 1973: linear-time algorithm
McCreight 1976: reduced space
Ukkonen 1995: new algorithm, easier to describe

15.1 Importance of sequence comparison

The first fact of biological sequence analysis: In biomolecular sequences (DNA, RNA, or amino acid sequences), high sequence similarity usually implies significant functional or structural similarity.

“Duplication with modification”: The vast majority of extant proteins are the result of a continuous series of genetic duplications and subsequent modifications. As a result, redundancy is a built-in characteristic of protein sequences, and we should not be surprised that so many new sequences resemble already known sequences. ... all of biology is based on enormous redundancy...

We didn’t know it at the time, but we found everything in life is so similar, that the same genes that work in flies are the ones that work in humans. (Eric Wieschaus, co-winner of the 1995 Nobel prize in medicine for work on the genetics of Drosophila development.)

Dan Gusfield, 1997, 212 ff

15.2 Searching for short queries in a long text

Problem Given a long text $t$ and many short queries $q_1, \ldots, q_k$. For each query sequence $q_i$, find all its occurrences in $t$.

We would like to have a data-structure that allows us to solve this problem efficiently.

Example: The text $t$ is a genomic sequence and the queries are short signals such as transcription factor binding sites, splice sites etc.

Important applications are in the comparison of genomes (in programs such as MUMmer that computes maximum unique matches) and in the analysis of repeats.
15.3 Basic definitions

Let $\Sigma$ denote an alphabet and $\Sigma^*$ the set of strings over $\Sigma$. Let $\epsilon$ denote the empty string and $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$. Let $t = t_1 t_2 \ldots t_n$ be the text and $\$ \in \Sigma^* \setminus t$. For $i \in \{1, 2, \ldots, n + 1\}$, let $s_i = t_i \ldots t_n \$ denote the $i$-th suffix of $t\$. 

In bioinformatics, the alphabet $\Sigma$ is usually of size 4 or 20. In other applications, the alphabet can be much larger, for example, to detect common web-surfing patterns, $\Sigma$ may be very large, e.g. consisting of the set of all links contained in a collection of web sites.

15.4 The role of suffixes

Consider the text $\text{abab\$}$. It has the following suffixes:

$\text{abab\$, bab\$, ab\$, b\$, and \$.}$

To determine whether a given query $q$ is contained in the text, we check whether $q$ is the prefix of one of the suffixes.

E.g., the query $\text{ab}$ is the prefix of both $\text{abab\$}$ and $\text{ab\$}$.

To speed up the search for all suffixes that have the query as a prefix, we use a tree structure to share common prefixes between the suffixes.

15.5 Sharing prefixes

(a) The suffixes $\text{abab\$}$ and $\text{ab\$}$ both share the prefix $\text{ab}$.
(b) The suffixes $\text{bab\$}$ and $\text{b\$}$ both share the prefix $\text{b}$.
(c) The suffix $\$ $ doesn’t share a prefix.
15.6 First example

Suffix tree for \textit{abab$\$} is obtained by sharing prefixes wherever possible. The leaves are annotated by the positions of the corresponding suffixes in the text.

15.7 $\Sigma^+$-tree

A $\Sigma^+$-tree $T$ is a finite, directed tree with root $\text{root}$. Its edges are labeled with strings in $\Sigma^+$, such that: For every letter $a \in \Sigma$ and node $u$ there exists at most one $a$-edge $u \xrightarrow{a} w$ (for some string $s$ and some node $w$).

A leaf is a node with no children and an edge leading to a leaf is called a leaf edge. A node with at least two children is called a branching node.

15.8 Naming nodes by strings

Let $u$ be a node of $T$. We use the name $\bar{s}$ for $u$, if $s$ is the concatenation of all labels of the edges along the path from the root of the tree to $u$.

For example $u = \overline{pqr}$:

The root is called $\tau$.

Definition: A string $g$ is said to occur in $T$, if there exists a string $h$ such that $\overline{gh}$ is a node in $T$. 

15.9 Suffix tree

**Definition:** A suffix tree $ST(t)$ for $t$ is a $\Sigma^+$-tree with the following properties:

1. Every node is either a leaf or a branching node, and
2. A string $s$ occurs in $ST(t)$ $\iff$ $s$ is a substring of $t$.

There exists a one-to-one correspondence between the non-empty suffixes of $t$ and the leaves of $ST(t)$.

For every leaf $s_j$ we define $l(s_j) = \{ j \}$. Recursively, for every branching node $\pi$ we define:

$$l(\pi) = \{ j \mid \pi \xrightarrow{u} \pi v \text{ is an edge of } ST(t), j \in l(\pi v) \}.$$

In other words,

$$l(\pi) = \bigcup_{\pi \text{ is child of } \pi} l(\pi).$$

We call $l(\pi)$ the **leaf set** of $\pi$.

15.10 Example

Text: $xabxaxc$

15.11 Ukkonen’s online construction

We will now discuss an algorithm that constructs $ST(t)$ in linear time. It operates *online* and generates $ST(\epsilon), ST(t_1), ST(t_1t_2), \ldots, ST(t_1t_2\ldots t_n)$ for all prefixes of $t$, without knowledge of the remaining part of the input string.

**Induction:** First, note that $ST(\epsilon)$ consists of a root node only. To completely define the algorithm we must describe the *induction step* $i \rightarrow i + 1$ or $(i+1)$-th phase:

$$ST(t_1\ldots t_i) \to ST(t_1\ldots t_i t_{i+1}), \quad \text{for any } i < n.$$

More precisely, we will generate a series of so-called *implicit suffix trees* that can at any time be made “explicit” by processing a terminator character $\$$.
15.12 Implicit Suffix Tree

Every tree $T$ computed by the induction step of Ukkonen’s algorithm is an implicit suffix tree in the following sense:

**Definition.** An implicit suffix tree for string $t$ is a tree obtained from the suffix tree for $t\$ by removing every copy of the terminal symbol $\$ from the edge labels of the tree, then removing any edge that has no label, and then removing any node that does not have at least two children.

Note that we will be able to obtain the latter from the former straightforwardly in linear time, simply by processing an additional terminator symbol $\$.

15.13 Updating the tree

Assume that $T$ is an implicit suffix tree for $t_1 \ldots t_i$. This means that any word $w$ is contained in $T$ if and only if it is contained in $t_1 \ldots t_i$.

Given the implicit suffix tree $T = ST(t_1 \ldots t_i)$, which modifications must we make to it to obtain a new tree $T' = ST(t_1 \ldots t_it_{i+1})$ that represents all substrings of $t_1 \ldots t_{i+1}$?

In the following we will use $a := t_{i+1}$ to indicate the new letter currently being added to the text.

Let us consider each suffix $wt_{i+1}$ of $t_1 \ldots t_{i+1}$ in turn. Note:

1. If $wt_{i+1}$ was already a substring of $t_1 \ldots t_i$, then no modification to $T$ is necessary to accommodate this string.

2. If $w$ is a leaf in $T$, then $w$ occurs precisely once in $t_1 \ldots t_i$ and $wt_{i+1}$ also occurs precisely once in the extended text $t_1 \ldots t_it_{i+1}$. In this case, we can simply extend the label of the edge leading to the leaf $w$ by the letter $t_{i+1}$.

3. Otherwise, $w$ occurs at least twice in $t_1 \ldots t_i$ and $wt_{i+1}$ does not occur in $t_1 \ldots t_i$. In this case, we call $wt_{i+1}$ a relevant suffix and the insertion of $wt_{i+1}$ is a more complex operation, which we will describe in detail below.

15.14 Processing the suffixes

In the $(i + 1)$-th phase, the algorithm considers all extensions of suffixes $w$ in order of decreasing length: $t_1 \ldots t_i, t_2 \ldots t_i$ etc.
Note that the list of extensions can be partitioned as follows:

\[
\begin{align*}
(A) & \quad w^1a = t_1 \ldots t_{i+1} \\
& \ldots \\
& \quad w^p a = t_p \ldots t_{i+1} \\
\end{align*}
\]

\(w^j\) occurs precisely once in \(t_1 \ldots t_i\),

\[
\begin{align*}
(B) & \quad w^{p+1}a = t_{p+1} \ldots t_{i+1} \\
& \ldots \\
& \quad w^q a = t_{q} \ldots t_{i+1} \\
\end{align*}
\]

\(w^j\) is repeated in \(t_1 \ldots t_i\), but

\[
\begin{align*}
(C) & \quad w^{q+1}a = t_{q+1} \ldots t_{i+1} \\
& \ldots \\
& \quad \epsilon a = t_{i+1} \\
\end{align*}
\]

\(w^j a\) is not contained in \(t_1 \ldots t_i\),

\(w^j a\) is contained in \(t_1 \ldots t_i\).

Each of these three sub-lists of prefixes is treated differently:

(A) To process suffixes \(w^1a, \ldots, w^p a\), simply extend all leaf edge labels by the new character \(a\).

(B) To process all relevant suffixes \(w^{p+1}a, \ldots, w^{q} a\), insert each suffix into the tree \(T\) using the procedure \textit{insert} defined below.

(C) No processing of the suffixes \(w^{q+1} \ldots \epsilon a\) is necessary as they are already contained in the tree.

### 15.15 Inserting a relevant suffix

To insert a relevant suffix \(wa\) into \(T\), we must navigate into the tree matching as many letters of \(wa\) to letters in edge-labels as possible. There are then two cases:

- If we finish matching letters inside an edge labeled \(uv\), then we insert a new leaf edges as follows:
  \[ \begin{array}{c}
  \text{If we finish matching letters inside an edge labeled } uv, \text{ then we insert a new leaf edges as follows:}
  \end{array} \]

- If we finish matching letters at a node \(uv\), then we insert a new leaf edge under \(uv\) as follows:
  \[ \begin{array}{c}
  \text{If we finish matching letters at a node } uv, \text{ then we insert a new leaf edge under } uv \text{ as follows:}
  \end{array} \]

### 15.16 Linear time construction

A \(O(n)\) runtime can be achieved by showing:
1. how to extend all suffixes $w_1a, \ldots, w_pa$ in one step per phase,
2. that only $\leq n$ relevant suffixes occur throughout the whole run,
3. how to insert a relevant suffix in constant time, and
4. how to determine which suffixes are relevant in constant time.

Lemma: In each phase we can extend all suffixes $w_1a, \ldots, w_pa$ in one step.
This is because the label of any leaf edge ends at the end of the current text. Hence, we simply maintain a global variable that contains the current text length and let this specify the end of all leaf labels.

Also, note that the topology of the tree is only changed when we process a relevant suffix and this step only modifies edges or internal nodes, but not leaves, hence: once a leaf, always a leaf.

Lemma: At most $n$ relevant suffixes occur.
Proof: Let $\alpha(t_1 \ldots t_i)$ denote the longest nested suffix in $t_1 \ldots t_i$, i.e. the longest suffix that occurs more than once as a substring of $t_1 \ldots t_i$. Then the total number of relevant suffixes that can occur is

$$R = \sum_{i=1}^{n-1} (q_i - p_i) = \sum_{i=1}^{n-1} (|\alpha(t_1 \ldots t_i)| - |\alpha(t_1 \ldots t_{i+1})|)$$

$$= \sum_{i=1}^{n-1} (|\alpha(t_1 \ldots t_i)| + 1 - |\alpha(t_1 \ldots t_{i+1})|)$$

$$= n - 1 + |\alpha(t_1)| - |\alpha(t_1 \ldots t_n)| \leq n,$$

where $p_i$ and $q_i$ denote the values of $p$ and $q$ in the $i$-th phase.\qed

15.17 Tracking the location of a suffix

To be able to perform steps (3) and (4) in constant time, we define the concept of the location of a suffix in the tree $T$ and keep track of our current location in the tree.

For a string $s$ contained in $T$, define $\text{loc}(s)$ as follows:

- if $s$ is a branching node, then set $\text{loc}(s) := s$,
- if $s$ is a leaf, then there is a leaf edge $\overrightarrow{uv} \rightarrow s$ in $T$ and we set $\text{loc}(s) := (u,v,\epsilon,s)$,
- otherwise, there is an edge $\overrightarrow{uv} \rightarrow uvw$ such that $s = uv, v \neq \epsilon, w \neq \epsilon$ and we set $\text{loc}(s) := (u,v,w,uvw)$.

Additionally, we will introduce so-called suffix links that will help us to navigate through the tree as we move down the list of suffixes for a given phase.
15.18 Operations on locations

We define the following three operations on locations:

1. Given the location \( \text{loc}(w) \) of a suffix \( w \) of \( t_1 \ldots t_i \) in \( T \) and a new letter \( a \), determine whether \( wa \) occurs in \( T \), that is, define \( \text{occurs}(\text{loc}(w), a) := \text{true} \Leftrightarrow wa \) occurs in \( T \). This operation can be implemented in constant time.
   This tells us when we have seen the last relevant suffix in the current phase.

2. Given the location \( \text{loc}(s) \) of a string \( s \) and a string \( w \) such that \( sw \) occurs in \( T \). The operation \( \text{getloc}(\text{loc}(s), w) := \text{loc}(sw) \) returns the location of \( sw \) in \( T \) in \( O(|w|) \) time, simply by following characters of \( w \) in \( T \).
   This will be used to update so-called “suffix links”, as described defined below.

3. Given a suffix \( s \) of \( t_1 \ldots t_i \) and \( \text{loc}(s) \), if \( sa \) is relevant, then we insert it into the tree as follows:
   - if \( \text{loc}(s) = \pi \), then \( \pi \) is a branching node and we add a leaf edge \( \pi \rightarrow sa \), and
   - otherwise, \( \text{loc}(s) = (\overline{u}, v, w, uvw) \) and we update the tree by splitting the edge \( \overline{u} \rightarrow uvw \) into \( \overline{u} \rightarrow v \rightarrow w \rightarrow uvw \) and adding a new leaf edge \( \pi \rightarrow sa \).
   This is the most important operation that inserts a relevant suffix \( sa \) into the tree \( T \).

15.19 Suffix links

To be able to process the list of relevant suffixes in the \( i \)-th phase quickly enough, we need to be able to move from \( \text{loc}(bw) \) to \( \text{loc}(w) \) in constant time.
To facilitate this, we maintain for each branching node, say \( \overline{bw} \), an auxiliary edge called a suffix link that points to the branching node \( \overline{w} \), if it exists, or to the root, else.

In the operation \( \text{insert}(\text{loc}(s), a) \) described above, we update the suffix links as follows:

- If \( \text{loc}(s) = \pi \), then no new internal node is generated and we do nothing.
- Otherwise, assume that \( \text{loc}(s) = (\overline{u}, v, w, uvw) \) and \( \pi \) is obtained as described above. Then the target of the suffix link from \( \pi \) is given by the following location:

\[
\text{linkloc}(\pi) := \begin{cases} 
\text{loc}(v_2 \ldots v_k) & \text{if } \overline{u} = \text{root}, \\
\text{getloc}(\text{linkloc}(\overline{u}), v) & \text{else},
\end{cases}
\]

where \( v = v_1 \ldots v_k \).

15.20 Example

A suffix tree with a suffix links (dotted arcs):
15.21 Ukkonen’s algorithm

Algorithm
Input: text \( t_1 \ldots t_n \)
Output: chain of implicit suffix trees \( ST(t_1), \ldots, ST(t_1 \ldots t_n) \)
Initialization: Generate a single node \( \varepsilon \) and set \( L = \varepsilon \).

for \( i = 1 \) to \( n \) do // phase \( i \) generates \( ST(t_1 \ldots t_i) \)
    while occurs\( (L, t_i) = \text{false} \) do // for all relevant suffixes
        if \( L \) is not a leaf then // i.e., not of the form \( (\overline{u}, v, \varepsilon, uv) \)
            insert\( (L, t_i) \) // insert the suffix
        Set \( L := \text{linkloc}(L) \) // move to next suffix
    Set \( L := \text{getloc}(L, t_i) \) // prepare for next phase
end

This generates an implicit suffix tree \( ST(t_1 \ldots t_i) \) for each \( i \). (Recall that all leaf labels are implicitly extended as we assume that they extend to the last processed position in the text.) To obtain a full suffix tree for the text \( t_1 \ldots t_i \), simply run the main loop one more time using a character $ that does not occur in the text.

Theorem This algorithm builds the suffix tree \( ST(t_1 \ldots t_n) \) by inserting \( \leq n \) relevant suffixes, each in constant time, thus achieving a linear run time.

15.22 Example

Consider the text \textit{pucupcupu}. Each column contains all suffixes of a prefix of the string:
15.23 The WOTD Algorithm

Ukkonen’s algorithm can build the suffix tree in linear time. Although this is a nice algorithm, it has two drawbacks:

First, it does not have good memory locality, which means that there is little correlation between how close two pieces of data are in the tree, and how close they are in physical memory. Hence, an average query will produce many cache misses and can lead to poor performance in practice.

Second, if we only have a few queries to pose, then it may be wasteful to compute the whole suffix tree.

The “write only, top down” algorithm due to R. Giegerich, S. Kurtz and J. Stoye (1999) has good memory locality and can also be used for a lazy construction of the suffix tree, building only as much of the tree as is necessary to satisfy a given query.

Although it requires $O(n \log n)$ average run-time, in practice it is often competitive.

15.24 Idea: Compute tree recursively

Note that the subtree below a branching node $\pi$ is determined by the set of all suffixes of $t\$ that start with the prefix $u$:
So, if we know the set of remaining suffixes
\[ R(\omega) := \{ s \mid us \text{ is a suffix of } t\$\}, \]
then we can evaluate the node \( \omega \), i.e. construct the subtree below \( \omega \).

15.25 The main evaluation step

An unevaluated node is evaluated as follows: We partition the set \( R(\omega) \) into groups by the first letter of the strings, i.e. for every letter \( c \in \Sigma \), we define the \( c \)-group as:
\[ R_c(\omega) := \{ cw \in \Sigma^* \mid cw \in R(\omega) \}. \]
Consider \( R_c(\omega) \) for \( c \in \Sigma \). If \( R_c(\omega) \neq \emptyset \), then there are two possible cases:

1. If \( R_c(\omega) \) contains precisely one string \( w \), then we construct a new leaf edge starting at \( \omega \) and label it with \( w \).

2. Otherwise, the set \( R_c(\omega) \) contains at least two different strings and let \( p \) denote their longest common prefix (lcp). We create a new \( c \)-edge with label \( p \) whose source node is \( \omega \). The new unevaluated node \( \omega p \) and set \( R(\omega p) = \{ w \mid pw \in R(\omega) \} \) will be (recursively) processed later.

15.26 Evaluating the root

This \textit{wotd}-algorithm (write-only, top-down) starts by evaluating the root node, with \( R(\text{root}) \) equal to the set of all suffixes of \( t\$ \). All nodes of \( ST(t) \) are then recursively constructed using the appropriate sets of remaining suffixes.

15.27 Example

Text: \textbf{abab}$

The algorithm proceeds as follows: We first evaluate the root node \text{root} using \( R(\text{root}) = \{abab\$, bab\$, ab\$, b\$\} \). There are three groups of suffixes:
\[ R_a(\text{root}) = \{abab\$, ab\$\}, R_b(\text{root}) = \{bab\$, b\$\} \text{ and } R_\$ = \{\$\}. \]
The letter \$ gives rise to a leaf edge with label \$. The letter \( a \) gives rise to an internal edge with label \( ab \), because \( ab = \text{lcp}(R_a(\text{root})) \). Similarly, for \( b \) we obtain an internal edge with label \( b \).

For the node \( ab \) we have \( R(ab) = \{ab\$, \$\} \) and thus \( R_a(ab) = \{ab\$\} \text{ and } R_\$(ab) = \{\$\}. \) Because both latter sets have cardinality one, we obtain two new leaf edges with labels \( ab\$ \text{ and } \$, \) respectively.

Similarly, we obtain two new leaf edges with labels \( ab\$ \text{ and } \$ \) for the node \( b \).

Text: \textbf{abab}$
15.28 Properties of the wotd-algorithm

Complexity: Space requirement? Worst case time complexity? (Exercises!)

The expected running time is $O(n \log_k n)$ and experimental studies indicate that the algorithm often performs in linear time for moderate sized strings.

Good memory locality.

Algorithm can be parallelized.

15.29 Suffix tree data-structure

An implementation of a suffix tree must represent its nodes, edges and edge labels. To be able to describe the implementation, we define a total ordering on the set of children of a branching node:

Let $\pi$ and $\tau$ be two different children of the same node in $ST(t)$. We write

$$\pi < \tau \quad \text{iff} \quad \min l(\pi) < \min l(\tau),$$

in other words, iff the first occurrence $\min l(\pi)$ of $\pi$ in $t\$ comes before the first occurrence $\min l(\tau)$ of $\tau$ in $t\$.

$$\min l(\overline{u}) \quad \min l(\overline{v})$$

in $t$. 

$$t \quad u \quad v \quad u \quad v$$
15.30 Representing edge labels efficiently

Remark 1 Because an edge label $s$ is a substring of the text $t$, we can represent it by a pair of pointers $(i,j)$ into $t' = t$ such that $s = t'_i, t'_{i+1}, \ldots, t'_j$.

However, note that we have $j = n + 1$ for any leaf edge and so in this case the right pointer is redundant. Hence:

Remark 2 A leaf edge requires only one pointer.

The following is not so easy to see:

Remark 3 An internal edge requires only one pointer.

This is made possible by defining a left pointer on the set of nodes (not edges) in such a way that these can be used to reconstruct the original left and right pointers of each edge, as follows:

Consider an edge $uv \rightarrow uv$. Define the left pointer $lp(uv)$ as the position $p$ of the first occurrence of $uv$ in $t$ plus the length of $u$:

$$lp(uv) = \min l(uv) + |u|.$$ 

This gives the start position $i$ of a copy of $v$ in $t$.

To get the end position of $v$, consider the $≺$-smallest child $uvw$ of $uv$. We have $\min l(uv) = \min l(uvw)$, i.e. the corresponding suffix starts at the same position $p$. By definition, we have $lp(uvw) = \min l(uvw) + |uv|$ and the end position of $v$ equals $lp(uvw) - 1$.

15.31 The main data table

For each node $\overline{u}$, we store a reference $firstchild(\overline{u})$ to its smallest child.

We store the values of $lp$ and $firstchild$ together in a single (integer) table $T$. We store the values of all children of a given node $\overline{u}$ consecutively, ordered w.r.t. $≺$. (We will indicate the last child of $\overline{u}$ by setting its lastchild-bit.)

So, only the edge from a given node $\overline{u}$ to its first child is represented explicitly. Edges from $\overline{u}$ to its other children are given implicitly and are found be scanning consecutive positions in $T$ that follow the position of the smallest child.

We reference the node $\overline{u}$ using the index of the position in $T$ that contains the value $lp(\overline{u})$.

15.32 Example

The table $T$ for $ST(abab)$. All indices start at 1. The first value in $T$ for a branching node $\overline{u}$ is $lp(\overline{u})$, the second value is $firstchild(\overline{u})$: 

\[
\begin{array}{cccc}
\text{t} & \text{u} & \text{v} & \text{w} \\
lp(\overline{u}) & lp(uvw) & \min l(\overline{uv}) = \min l(\overline{uvw}) \uparrow \\
\end{array}
\]
To be able to decode this representation of the suffix tree, we need two extra bits: A **leaf-bit** (*) indicates that the given position in $T$ corresponds to a leaf node and a **lastchild-bit** (†) indicates that the node at this position does not have a larger brother w.r.t. $\prec$.

### 15.33 Storing an unevaluated node

We consider the wotd-algorithm as a process that evaluates the nodes of a suffix tree. It starts at the root and then evaluates all nodes recursively.

First we discuss how to store an unevaluated node $\overline{v}$.

To be able to evaluate $\overline{v}$, we (only) need to know the set of remaining suffixes $R(\overline{v})$. To make these available, we define a global array called $\text{suffixes}$ that contains pointers to suffixes in $t\$ and use it as follows: For every unevaluated node $\overline{v}$, the $\text{suffixes}$ array contains an interval of pointers to start positions in $t\$ that correspond precisely to the suffixes contained in $R(\overline{v})$, in increasing order.

We can now represent $R(\overline{v})$ in $T$ using two numbers, $\text{left}(\overline{v})$ and $\text{right}(\overline{v})$, that define an interval of entries in the $\text{suffixes}$ array.

As a branching node, $\overline{v}$ will occupy two positions in $T$, one for $\text{lp}(\overline{v})$ and followed by $\text{firstchild}(\overline{v})$. Until $\overline{v}$ is actually evaluated, we will use these two positions to store $\text{left}(\overline{v})$ and $\text{right}(\overline{v})$. We use a third bit called the **unevaluated**-bit to distinguish between unevaluated and evaluated nodes.

### 15.34 Evaluating a node $\overline{v}$

First note that the left pointer $\text{lp}(\overline{v})$ of the node $\overline{v}$ is given by the left-most entry of $\text{suffixes}$ over the interval $[\text{left}(\overline{v}), \text{right}(\overline{v})]$.

Determine the length of the longest common prefix $\text{lcp}$ of entries in $\text{suffixes}$ over $[\text{left}(\overline{v}), \text{right}(\overline{v})]$ and add it to all these entries.

The $\text{lcp}$ is computed by stepping through a simple loop $j = 1, 2, \ldots$ and checking the equality of all letters $t_{\text{suffixes}[i]+j}$ for all start positions $i$ in $[\text{left}(\overline{v}), \text{right}(\overline{v})]$. As soon as a difference is detected, the loop is aborted and $j$ is the length of the $\text{lcp}$.

Sort and count all entries of $\text{suffixes}$ in the interval $[\text{left}(\overline{v}), \text{right}(\overline{v})]$, by the first letter $c$ of the suffixes as the sort key. (Do this stably, i.e. don’t change the order of suffixes that start with the same letter.)

Each letter $c$ that has count $> 0$ will give rise to a new node $\overline{v}$ below $\overline{v}$ and the suffixes in the $c$-group $R_c(\overline{v})$ determine the tree below $\overline{v}$.

For each non-empty $c$-group of $\overline{v}$, we store one child in the table $T$, as follows:

**Leaf case:** A $c$-group containing only one string gives rise to a leaf node $\overline{v}$ and we write the number $\text{lp}(\overline{v})$ in the first available position of $T$. This number $\text{lp}(\overline{v})$ is obtained as the single entry of $\text{suffixes}$.
that corresponds to the c-group.

**Internal node case:** A c-group containing more than one string gives rise to branching node \( \overline{v} \) and we store \( \text{left}(\overline{v}) \) and \( \text{right}(\overline{v}) \) in the first two available positions of \( T \). The values of \( \text{left} \) and \( \text{right} \) are computed during the sort and count step.

### 15.35 Lazy vs. complete evaluation

To build the complete suffix tree, we proceed breath-first, from left to right.

In a lazy approach, we only evaluate those nodes that are necessary to answer a query (and have not yet been evaluated).

### 15.36 Example

**Input:** Text: \( a \ b \ a \ b \ $ \)

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

**Initial:** suffixes: 1 2 3 4 5

Evaluate(\text{root}): 
Sort and count: 
\[
\begin{align*}
R_a(\text{root}) &= \{1, 3\}, \ lcp = ab \\
R_b(\text{root}) &= \{2, 4\}, \ lcp = b \\
R_S(\text{root}) &= \{5\}
\end{align*}
\]

The suffixes are ordered, \( \text{left} \) and \( \text{right} \) are entered in the table and the three bits \( (u, *, \dagger): \) unevaluated, leaf, lastchild) are set:

\[
\begin{array}{ccc}
\text{suffixes:} & 1 & 2 & 3 & 4 & 5 \\
T: & u & u & * & \dagger
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & \\
\end{array}
\]

Evaluate(1):
Note \( lp(1) = \text{suffixes}[T[1]] = \text{suffixes}[1] = 1 \) and \( \text{firstchild}(1) = 6 \), thus:

\[
\begin{array}{cccccc}
1 & 6 & 3 & 4 & 5 & \\
u & u & * & \dagger
\end{array}
\]

Determine \( lcp \) of entries \( \text{suffixes}[1 \ldots 2] \) and add it to them:

\[
\begin{array}{ccc}
\text{suffixes:} & 3 & 5 & 2 & 4 & 5 \\
\end{array}
\]

Determine c-groups and then add appropriate nodes:
\[
\begin{align*}
R_a(1) &= \{3\} \\
R_s(1) &= \{5\}
\end{align*}
\]

\[
\begin{array}{cccccc}
1 & 6 & 3 & 4 & 5 & 3 & 5 \\
(u) & u & * & \dagger & * & \dagger
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & \\
\end{array}
\]

(Text: a b a b $)
Evaluate(3):
Note \( l_p(3) = \text{suffixes}[T[3]] = \text{suffixes}[3] = 2 \) and \( \text{firstchild}(3) = 8 \), thus:

\[
T: \begin{array}{cccccccc}
1 & 6 & 2 & 8 & 5 & 3 & 5 & 1 \\
\end{array}
\]

Determine \( lcp \) of entries \( \text{suffixes}[3...4] \) and add it to them:

\[
\begin{array}{c}
\text{suffixes: } 3 & 5 & 3 & 5 & 5 \\
\end{array}
\]

Determine \( c \)-groups and then add appropriate nodes:
\[
R_a(3) = \{3\} \\
R_s(3) = \{5\}
\]

\[
T: \begin{array}{cccccccc}
1 & 6 & 2 & 8 & 5 & 3 & 5 & 3 & 5 \\
\end{array}
\]

\[
\begin{array}{c}
(u) & * & * & * & * & * & * \\
\end{array}
\]

Done!

15.37 Finding occurrences in a WOTD tree

Suppose we are given a text \( text \) and have computed the WOTD tree \( T \). How do we determine all occurrences of a given query string \( q \)?

This is done by navigating into the tree and matching the letters of the query with the letters of the edge labels until we have used-up all the letters of the query. Once we have established that the query is contained in the tree, we visit all nodes below the location at which the query was fulfilled and report one occurrence for each leaf.

Recall that the position of an occurrence is not stored in the tree, however, it can be obtained by keeping track of the \( \text{depth} \) of nodes which is the the sum of lengths of all edge labels along the path from the root to the node.

The following algorithm is initially called with \( t = 1 \), the first cell in \( T \):

**Algorithm** Find-WOTD

**Input:** A query string

**Output:** All occurrences of query

**repeat**

\[ label := \text{getEdgeLabel}(t) \]
\[ \text{newDepth} := \text{depth} + |\text{edgeLabel}| \]

**compare** query and label

if query is prefix of label then // query found

if \( t \) is leaf then

return the singleton list \( \{|text| - \text{newDepth}\} \)

else return \text{collectAllHits}(T[t + 1], \text{newDepth})

else if label is proper prefix of query then // move to child

\[ t := T[t + 1], \text{depth} := \text{newDepth}, \]
\[ \text{query} := \text{query}[|\text{label}| + 1, |\text{query}|] \]

else if query[0] = label[0] then return empty-list // not in tree

else // try next child, if available

if \( t \) is last child then return empty-list
if $t$ is leaf then $t := t + 1$ else $t := t + 2$ // move to sibling
end

Sub-Routine collectAllHits $(t, \text{depth})$
Input: A parent node $t$ and its depth below the root
Returns: List $hits$ of all positions corresponding to leaves below $t$
repeat
newDepth = depth + $|\text{getEdgeLabel}(t)|$
if $t$ is a leaf then
Append $(|text| - newDepth)$ to $hits$
if $t$ is last child, return
$t := t + 1$
else
call collectAllHits$(T[t + 1], newDepth)$
if $t$ is last child, return
$t := t + 2$
end

Sub-Routine getEdgeLabel $(t)$
Input: A node $t$
Returns: The corresponding edge label
begin
Set $p_1 = T[t]$
if $t$ is a leaf node then
Set $p_2 = |text|$
else
Set $p_2 = T[T[t + 1]] - 1$
return $text[p_1, p_2]$
end

15.38 Applications of suffix trees

1. Searching for exact patterns
2. Minimal unique substrings
3. Maximal unique matches
4. Maximal repeats
5. Approximate repeats

Additional literature:
15.39 Searching for exact patterns

To determine whether a string $q$ occurs in a string $t$, follow the path from the root of suffix tree $ST(t)$ as directed by the characters of $q$. If at some point you cannot proceed, then $q$ does not occur in $t$, otherwise it does.

Text $abab\$.

The query $abb$ is not contained in $abab$: Following $ab$ we arrive at the node $\overline{ab}$, however there is no $b$-edge leaving from there. The query $baa$ is not contained in $abab$: Follow the $b$-edge to $\overline{b}$ and then continue along the leaf edge whose label starts with $a$. The next letter of the label is $b$ and doesn’t match the next letter of the query string.

Clearly, the algorithm that matches a query $q$ against the text $t$ runs in $O(|q|)$ time.

15.40 Finding all occurrences

To find all positions where the query $q$ is contained in $t$, annotate each leaf $\overline{s_i}$ of the suffix tree with the position $i$ at which the suffix $i$ starts in $t$.

Then, after matching $q$ to a path in the tree, visit all nodes below the path and return the annotated values.

This works because any occurrence of $q$ in $t$ is the prefix of one of these suffixes.

The number of nodes below the path is at most twice the number of hits and thus finding and collecting all hits takes time $O(|q| + k)$, where $k$ is the number of occurrences.

(Nota that in the discussed lazy suffix tree implementation we do not use this leaf annotation but rather compute the positions from the the $lp$ values, to save space...)

15.41 Application: Maximal Unique Matches

Standard dynamic programming is too slow for aligning two large genomes. If the genomes are similar, then one can expect to see long identical substrings which occur in both genomes. These maximum unique matches (MUMs) are almost surely part of a good alignment of the two sequences and so the alignment problem can be reduced to aligning the sequence in the gaps between the MUMs.

Given two sequences $s$ and $t$, and a number $l > 0$. The maximal unique matches problem (MUM-problem) is to find all sequences $u$ with:
• $|u| \geq l$,

• $u$ occurs exactly once in $s$ and once in $t$, and

• for any character $a$ neither $ua$ nor $au$ occurs both in $s$ and $t$.

In other words, a MUM is a sequence $m$ that occurs precisely once in $s$ and once in $t$, and is both right maximal and left maximal with this property.

This problem can be solved in $O(|s| + |t|)$ time: To find all MUMs, generate the suffix tree $T$ for $s \% t$, where $\%$ is a “separator” with $\% \notin s$ and $\% \notin t$. Any path in $T$ from the root to some node $u$ that has precisely two children, one in $s$ and one in $t$, corresponds to a right maximal unique match.

To determine whether $u$ is left maximal, too, simply check whether both preceding letters in $s$ and $t$ differ.
15.42 Example

**Nucleic Acids Research, 1999, Vol. 27, No. 11**

**Figure 8.** Alignment of a 222 930 bp subsequence of human chromosome 12p13, accession no. U47924, to a 227 538 bp subsequence of mouse chromosome 6, accession no. AC002397. Each point in the plot corresponds to an MUM of 21 bp.

15.43 Example
15.44 Repeats in human

| Table 1: Number of copies and fraction of genome for classes of inter-spersed repeat |
|---------------------------------|-----------------|-----------------|-----------------|
|                                | Number of copies (x 1,000) | Total number of bases in the draft genome sequence (Mb) | Fraction of the draft genome sequence (%) | Number of families [subfamilies] |
| SINEs                          | 1,058                        | 359.6                      | 13.14                        |                     |
| Alu                            | 1,000                        | 250.1                      | 10.00                        | 1 (~20)             |
| MIR                            | 363                          | 60.1                       | 2.20                         | 1 (1)               |
| MIR3                           | 75                           | 9.3                        | 0.34                         | 1 (1)               |
| LINEs                          | 868                          | 558.8                      | 20.42                        |                     |
| LINE1                          | 516                          | 462.1                      | 16.89                        | 1 (~55)             |
| LINES                          | 315                          | 88.2                       | 3.22                         | 1 (2)               |
| LINE3                          | 37                           | 4.4                        | 0.31                         | 1 (2)               |
| LTR elements                   | 443                          | 227.0                      | 8.29                         |                     |
| ERV-class I                    | 112                          | 79.2                       | 2.89                         | 72 (132)            |
| ERV-class II                   | 8                            | 8.5                        | 0.31                         | 10 (20)             |
| ERV (L)-class III              | 63                           | 39.5                       | 1.44                         | 21 (42)             |
| MLTR                           | 240                          | 99.8                       | 3.65                         | 1 (31)              |
| DNA elements                   | 294                          | 77.6                       | 2.64                         |                     |
| IAP group                      | 152                          | 38.1                       | 1.39                         | 25 (50)             |
| ZAPJAC                        | 13                           | 4.3                        | 0.16                         | 4 (10)              |
| Tc-1 group                     | 152                          | 38.1                       | 1.39                         | 25 (50)             |
| MER1-Charlie                   | 57                           | 29.0                       | 1.02                         | 12 (26)             |
| Tc2                            | 4                            | 0.9                        | 0.03                         | 1 (3)               |
| MER2-Tigger                    | 47                           | 9.5                        | 0.32                         | 10 (20)             |
| MER3 Hornor                    | 14                           | 2.6                        | 0.10                         | 4 (5)               |
| Peggy-Bac-like                 | 2                            | 0.5                        | 0.02                         |                    |
| Unclassified                   | 22                           | 3.2                        | 0.12                         | 7 (7)               |
| Unclassified                   | 3                            | 0.6                        | 0.14                         | 2 (4)               |
| Total interspersed repeats     | 1,226.8                      | 44.83                      |                     |


15.45 Definition of a maximal repeat

Given a sequence \( t = t_1t_2 \ldots t_n \).

A substring \( t[i, j] := t_i \ldots t_j \) is represented by the pair \((i, j)\). A pair \( R = (l, r) \) of different substrings \( l = (i, j) \) and \( r = (i', j') \) of \( t \) is called a repeat, if \( i < i' \) and \( t_i \ldots t_j = t_{i'} \ldots t_{j'} \). We call \( l \) and \( r \) the right and left instance of the repeat \( R \), respectively.

A repeat \( R = ((i, j), (i', j')) \) is called left maximal, if \( i = 1 \) or \( t_{i-1} \neq t_{i'-1} \), and right maximal, if \( j' = n \) or \( t_{j+1} \neq t_{j'+1} \), and maximal, if both.

15.46 Example

\[ a c \]

The string \( g \ a \ g \ c \ t \ c \ g \ a \ g \ c \) contains the following repeats of length \( \geq 2 \):
15.47 An algorithm for computing all maximal repeats

We will discuss how to compute all maximal repeats. Let $t$ be a string of length $n$ and assume that the first and last letter of $t$ both occur exactly once on $t$, e.g.:

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
t = & x & g & g & c & g & c & y & g & c & g & c & c & z \\
\end{array}
\]

Let $T$ be the suffix tree for $t$.

We can ignore all leaf edges from the root.

The algorithm proceeds in two phases:

In the first phase, every leaf node $v$ of $T$ is annotated by $(a, i)$, where $v = t_i \ldots t_n$ is the suffix associated with $v$ and $a = t_{i-1}$ is the letter that occurs immediately before the suffix.

15.48 Example

Partial suffix tree for $t = x g g c g c y g c g c c z$:

With leaf annotations:
15.49 Second phase of the algorithm

For every leaf node $v$ set:

$$A(v, c) = \begin{cases} \{i\}, & \text{if } c = t_{i-1}, \text{ and} \\ \emptyset, & \text{else,} \end{cases}$$

where $i$ is the start position of the corresponding suffix $v$.

In the second phase of the algorithm, we extend this annotation to all branching nodes bottom-up:

Let $w$ be a branching node with children $v_1, \ldots, v_h$ and assume we have computed $A(v_j, c)$ for all $j \in \{1, \ldots, h\}$ and all $c \in \Sigma$.

For each letter $c \in \Sigma$ set:

$$A(w, c) := \bigcup_{j=1}^{h} A(v_j, c).$$

Note that this is a disjoint union and $A(w, c)$ is the set of all start positions of $w$ in $t$ for which $t_{i-1} = c$.

15.50 Example
Annotation of branching nodes and output repeats of length $\geq 2$

15.51 Reporting all maximal repeats

In a bottom-up traversal, for each branching node $w$ we first determine $A(w, c)$ for all $c \in \Sigma$ and then report all maximal repeats of the word $w$:

Let $q$ be the current depth, i.e., number of characters from the root node, i.e., the length of $w$.

If $\text{depth} \geq l$ do:

\begin{verbatim}
for each pair of children $v_f$ and $v_g$ of $w$ with $v_f \prec v_g$:
    for each letter $c \in \Sigma$ with $A(v_f, c) \neq \emptyset$:
        for each letter $d \in \Sigma$ with $d \neq c$ and $A(v_g, d) \neq \emptyset$:
            for each $i \in A(v_f, c)$:
                for each $j \in A(v_g, d)$:
                    Print $((i, i + q - 1), (j, j + q - 1))$
\end{verbatim}

15.52 Maximality of output

**Lemma** The algorithm prints precisely the set of all maximal repeats in $t$ of length $\geq l$.

**Proof**

1. Each printed pair $R$ is a repeat, as the word $w$ is the common prefix of two or more different suffixes.

2. Each repeat $R$ is left-maximal, as $c \neq d$.

3. Each repeat $R$ is right-maximal, as $v_f \neq v_g$, and by definition of a suffix tree, the labels of the edges leading to these two children begin with distinct letters.

4. No maximal repeat is reported twice, as $v_f \prec v_g$ and all unions are disjoint. \hfill $\square$

15.53 Performance analysis

**Lemma** Computation of all maximal repeats of length $\geq l$ can be done in $O(n + z)$ time and $O(n)$ space, where $z$ is the number of maximal repeats.

**Proof** The suffix tree can be built in $O(n)$ time and space. We can annotate the tree in $O(n)$ time and space, if we use the fact that we only need to keep the annotation of a node until its father has been fully processed. (Also, we maintain the sets as linked links and then each disjoint-union operation can be done in constant time.)

In the nested loop we enumerate in total all $z$ maximal repeats in $O(z)$ steps. \hfill $\square$

Hence, the algorithm is both time and space optimal.
15.54 Significance of repeats

How significant is a detected maximal repeat? In a long random text we will expect to find many short repeats purely by chance.

The $E$-value associated with a maximum repeat $R$ in $t$ is the expected number of repeats of the same length or longer that are found in a random sequence of length $|t|$.

To compute this in the case of DNA, consider a simple Bernoulli model where each base $\alpha \in \{A,C,G,T\}$ has the same fixed probability of occurrence: $p_\alpha = p = \frac{1}{4}$.

Note that the number of maximal exact repeats of length $\geq l$ equals the number of (only) left-maximal repeats of length exactly $l$.

Ignoring boundary effects:

\[
\mathbb{E}[\# \text{ of maximal exact repeats of length } \geq l] = \mathbb{E}[\# \text{ of left-maximal exact repeats of length } l] = \sum_{1 \leq i_1 < i_2 \leq n} Pr(t[i_1,i_1 + l - 1] = t[i_2,i_2 + l - 1], t[i_1 - 1] \neq t[i_2 - 1]) = \sum_{1 \leq i_1 < i_2 \leq n} p^l(1 - p) = \frac{1}{2}n(n - 1)p^l(1 - p)
\]

With boundary effects: $= \frac{1}{2}(n - l + 1)(n - l)p^l(1 - p) + (n - l)p^{l+1}$.

(Note that this is only an estimation, as the effect of overlapping of instances of the repeat is ignored.)

15.55 Palindromic repeats

Let $t$ be a DNA sequence. We call $((i,j), (i',j'))$ a palindromic repeat, if $t_i \ldots t_j = \overline{t_{i'}} \ldots \overline{t_{j'}}$, where $\overline{w}$ denotes the reverse complement of a DNA string $w$.

All maximal palindromic repeats can be found using a modification of the described algorithm for maximal repeats, based on the suffix tree for $xty\overline{t}z$, where $x,y,z$ are three characters that do not appear in $t$ or $\overline{t}$.

15.56 Degenerate repeats

Let $u$ and $w$ be two strings of the same length. The Hamming distance $d_H(u, w)$ between $u$ and $w$ is the number of positions $i$ such that $u_i \neq w_i$.

In the following, we assume that we are given a sequence $t = t_1 \ldots t_n$, an error threshold $k \geq 0$ and a minimum length $l > 0$. 
**Definition** A pair of equal-length substrings $R = ((i_1, j_1), (i_2, j_2))$ is called a $k$-mismatch repeat in $t$, iff $(i_1, j_1) \neq (i_2, j_2)$ and $d_H(t[i_1, j_1], t[i_2, j_2]) = k$. The length of $R$ is $j_1 - i_1 + 1 = j_2 - i_2 + 1$. A $k$-mismatch repeat is maximal if it is not contained in any other $k$-mismatch repeat.

As with exact repeats, a $k$-mismatch repeat $R = ((i_1, j_1), (i_2, j_2))$ is maximal iff $(i_1 = 1$ or $i_2 = 1$ or $t_{i_1-1} \neq t_{i_2-1})$ and $(j_1 = n$ or $j_2 = n$ or $t_{j_1+1} \neq t_{j_2+1})$

15.57 The Mismatches Repeat Problem

The Mismatches Repeat Problem (MMR) is to enumerate all maximal $k$-mismatch repeats of length $\geq l$ contained in $t$.

15.58 Example

Maximal $k$-mismatch repeats ($k = 0,\ldots, 4$) for $l = 5$ in *mississippi*:

<table>
<thead>
<tr>
<th>$k$</th>
<th>Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td>s i s s i</td>
</tr>
<tr>
<td>2</td>
<td>s i p p i</td>
</tr>
<tr>
<td>3</td>
<td>s i p p i</td>
</tr>
<tr>
<td>4</td>
<td>???</td>
</tr>
</tbody>
</table>

text : m i s s i s i p p i

15.59 The Seed Lemma

The following result is a key observation and is the basis of many seed-and-extend approaches to sequence comparison:

**Lemma** Every maximal $k$-mismatch repeat $R$ of length $l$ contains a maximal exact repeat of length $\geq \left\lceil \frac{l-k}{k+1} \right\rceil$, called a seed.

**Proof** Let $R = ((i_1, j_1), (i_2, j_2))$ be a $k$-mismatch repeat of length $\geq l$. The $k$ mismatches divide $t[i_1, j_1]$ and $t[i_2, j_2]$ into exact repeats $w_0, w_1, \ldots, w_k$. Now $\max_{i \in [0, k]} |w_i|$ is minimal if the mismatching character pairs are equally distributed over $R$. Obviously, for such an equal distribution the length of the longest $w_i$ is $\geq \left\lceil \frac{l-k}{k+1} \right\rceil = \left\lfloor \frac{l}{k+1} \right\rfloor$.

(A “•” indicates a mismatch and we have: $l = 11$, $k = 3$, $\left\lceil \frac{l-k}{k+1} \right\rceil = \left\lceil \frac{11-3}{3+1} \right\rceil = \left\lceil \frac{8}{4} \right\rceil = 2 = \left\lfloor \frac{l}{k+1} \right\rfloor = \left\lfloor \frac{11}{4} \right\rfloor$)
15.60 An algorithm for the MMR problem

Suppose we are given a text \( t \) of length \( n \). To find all maximal \( k \)-mismatch repeats in \( t \) of length \( \geq l \), do the following:

1. Build the suffix tree \( T \) for \( t \) and use it to detect all seeds, i.e., all exact maximal repeats of length \( \geq \lfloor \frac{l}{k+1} \rfloor \).
2. For each seed \( s = ((i_1, j_1), (i_2, j_2)) \) do the following:
   (a) For \( q = 0, 1, \ldots, k \), compute:
       \[
       \text{left}(q) := \max\{p \mid d_H(t[i_1-p, i_1], t[i_2-p, i_2]) = q\},
       \]
       i.e. the length of the maximal extension of the seed to the left with precisely \( q \) mismatches.
   (b) For \( q = 0, 1, \ldots, k \), compute:
       \[
       \text{right}(q) := \max\{p \mid d_H(t[j_1+j_1+p], t[j_2+j_2+p]) = q\},
       \]
       i.e. the length of the maximal extension of the seed to the right with precisely \( q \) mismatches.
   (c) For \( q = 0, 1, \ldots, k \):
       if \( (j_1 - i_1 + 1 + \text{left}(q) + \text{right}(k - q)) \geq l \), then print \((i_1 - \text{left}(q), j_1 + \text{right}(k - q)), (i_2 - \text{left}(q), j_2 + \text{right}(k - q))\).

15.61 Correctness of the algorithm

The correctness of the algorithm follows from the Seed Lemma: every maximal \( k \)-mismatch repeat of length \( \geq l \) contains an exact repeat of length \( \geq \lfloor \frac{l}{k+1} \rfloor \) and can be obtained from the seed by extending the seed match with \( q \) mismatches to the left and with \( k - q \) mismatches to the right, for some \( q \leq k \).

Note that the same maximal \( k \)-mismatch repeat can be obtained via more than one seed. To avoid this, when computing \( \text{left} \) for a given seed, we stop the computation of the table \( \text{left} \) if we observe a second seed to the left of the original seed. This ensures that we only output those maximal \( k \)-mismatch repeats for which the given seed is leftmost.

15.62 Efficient solution of MMR problem

**Lemma** The mismatch repeats problem MMR can be solved in \( O(n + kq) \) time, where \( n \) is the length of the text, \( k \) is the number of mismatches permitted and \( q \) is the number of different seeds.

We can find all seeds in \( O(n + q) \) time. Thus the result follows, if we can compute the \( k \)-mismatch extension of any seed in \( k \) steps.

The latter can indeed be achieved, if we can determine the maximal common extension of two matches in constant time. This is indeed possible, due to the following amazing result on rooted trees:

15.63 The lowest common ancestor problem

Let \( T \) be a rooted tree. A node \( u \) is called an ancestor of a node \( v \) if \( u \) lies on the unique path from root to \( v \). The lowest common ancestor (lca) of two nodes \( i \) and \( j \) is the last node that is both on the
path from root to $i$ and on the path from root to $j$. 

**Theorem** Let $T$ be a rooted tree. After a linear amount of preprocessing, we can determine the lowest common ancestor of any two nodes $x$ and $y$ in constant time. 

(This is due to Harel and Tarjan (1984) and later simplified by Schieber and Vishkin (1988), see Chapter 8 of Dan Gusfield’s book for details.)

15.64 Determining whether one node is an ancestor of another

In the the lowest common ancestor problem, there is a particularly simple case, namely when one node, $i$ say, is an ancestor of the other, $j$ say.

Consider two nodes $i$ and $j$ in a rooted tree $T$. How to determine whether $i$ is an ancestor of $j$ in constant time, after a linear amount of preprocessing?

Preprocess the tree as follows: number all nodes in a depth-first traverse and let $l(i)$ denote the depth-first number of $i$. Let $s(i)$ denote the number of nodes (including $i$) contained in the subtree rooted at $i$, obtained as the count of depth-first numbers used while processing $i$.

**Lemma** Node $i$ is an ancestor of node $j$ if and only if $l(i) \leq l(j)$ and $l(j) \leq l(i) + s(i)$.

15.65 Complete binary trees

How to compute the least common ancestor $lca(i, j)$ of two nodes $i$ and $j$ in the case that $lca(i, j) \notin \{i, j\}$? We will discuss this for the simple case of a complete binary tree. (See Gusfield 1997 for the general case.)

In the following, suppose that $T$ is a rooted complete binary tree with $p$ leaves, and $n = 2^p - 1$ nodes in total, such that every internal node has precisely two children and the number of edges on the path from the node to any leaf is $d = \log_2 p$.

15.66 Computational Model

Any run-time complexity analysis is based on a computational model. Given an input of size $n$, we usually assume the unit cost RAM model, in which:

- any number of size $O(\log n)$ bits can be written, read or used as an address in constant time, and
- any two $O(\log n)$-bit numbers can be added, subtracted, multiplied or divided in constant time.

In the following we also explicitly assume that bit operations such as exclusive-or (XOR), left- or right-shift and detecting the first non-zero bit can be performed in constant time on numbers of size $O(\log n)$. 
15.67 In-order node numbering

Let us assume that the nodes of $T$ have been numbered using an in-order traverse. We will view these in-order numbers as bit-strings of length $d + 1$, e.g.:

Alternatively, we may consider assigning to each node $v$ a $d + 1$-bit number called the path number of $v$, defined as follows.

Counting from the left-most bit, the $k^{th}$ bit in the path number of $v$ corresponds to the $k^{th}$ edge in the path from the root to $v$: a 0 indicates that the path goes to a left child and a 1 indicates that the path goes to a right child. The right-most 1 indicates the end-of-path (and not a choice of child).

For example, $0110$ is the node obtained by choosing the left child of the root and then the right child of that node.

We have:

**Lemma** The in-order number of a node equals its path number.

15.68 Solving lca for a complete binary tree

Let $T$ be a complete binary tree with an in-order numbering of nodes. How to compute $lca(i, j)$ in constant time, after a linear amount of preprocessing, when $lca(i, j) \in \{i, j\}$?

Let $x_{ij} = \text{XOR}(i, j)$ be the number obtained by computing the bitwise-exclusive-or of the two $d + 1$-bit numbers $i$ and $j$.

Let $k$ denote the position of the left-most 1 of $x_{ij}$. By definition of XOR, the first $k - 1$ bits of $i$ and $j$ are the same and thus the paths to $i$ and $j$ are the same for the first $k - 1$ edges, and then diverge (as we are assuming $lca(i, j) \notin \{i, j\}$).

Hence, the path-number for $lca(i, j)$ consists of the first $k - 1$ bits of $i$, followed by a 1 (to indicate end-of-path), followed by $d + 1 - k$ zeros.

**Example:**

\[
\begin{align*}
   i &= 0 \ 0 \ 1 \ 0 \\
   j &= 0 \ 1 \ 1 \ 1 \\
\end{align*}
\]
\[
\text{XOR}(i, j) = 0 \ 1 \ 0 \ 1
\]

So, we can compute $lca(i, j)$ in constant time using the following bit-operations:

- compute $x_{ij} = \text{XOR}(i, j)$
• Determine the position \( k \) of the left-most 1 in \( x_{ij} \)
• Shift \( i \) to the right by \( d + 1 - k \) places
• Set the right-most bit of \( i \) to 1
• Shift \( i \) to the left by \( d + 1 - k \) places.
16 Suffix arrays

This exposition is based on the following sources, which are all recommended reading:


Suppose we are given a text \( t = t_1t_2\ldots t_n \) of length \( n \). A suffix tree for \( t \) requires either \( O(n|\Sigma|) \) space or the minimum of \( O(n \log n) \) and \( O(n \log |\Sigma|) \) time, with \( |\Sigma| \) the size of the alphabet.

If \( |\Sigma| \) is large, or if space is limited, then it may be desirable to have a data structure that uses less space than a suffix tree, while retaining most of the advantages that a suffix tree has.

Udi Manber and Gene Myers proposed a new data structure, the suffix array, that addresses this issue.

A suffix array \( SA(t) \) for a text \( t = t_1t_2\ldots t_n \) is simply an array \( pos \) of length \( n + 1 \) containing the start positions of all suffixes of \( t\$, sorted in lexical order, with the understanding that \$ comes after all other symbols.

16.1 Example

Consider the text:

\[
1 2 3 4 5 6 7 8 \\
bar bar a \$
\]

It has the suffix array:

\[
5 \quad a r a \$
2 \quad a r b a r a \$
7 \quad a \$
4 \quad b a r a \$
1 \quad b a r b a r a \$
6 \quad r a \$
3 \quad r b a r a \$
8 \quad \$
\]

16.2 Suffix tree to suffix array in linear time

Suppose we are given a text \( t = t_1\ldots t_n \) and the corresponding suffix tree \( ST(t) \). We can compute a suffix array \( SA(t) \) from \( ST(t) \) by performing a lexical depth-first traverse of \( ST(t) \). Once \( SA(t) \) has been built, \( ST(t) \) can be discarded.

We say that an edge \( e = (v, u) \) in \( ST(t) \) is lexically less than an edge \( f = (v, w) \), if and only if the first character of the label of \( e \) is lexically less than the first character of the label of \( f \). (We assume that \( c \leq \$ \) for all \( c \in \Sigma \).)
As no two edges out of a node \( v \) start with the same character, this defines a strict lexical ordering of all the edges out of \( v \). In such a depth-first traverse, the order in which we visit the leaves gives us the order of the entries in the suffix array.

In the following example, the lexical depth-first visits the nodes in the order 2, 5, 3, 6, 1, 4, 7:

\[
\begin{array}{c}
\text{2} & \text{3} & \text{4} & \text{5} & \text{6} & \text{7} \\
\text{a} & \text{b} & \text{c} & \text{x} & \text{a} & \text{c} \\
\text{b} & \text{x} & \text{a} & \text{c} & \text{x} & \text{a} \\
\text{x} & \text{a} & \text{b} & \text{c} & \text{a} & \text{c} \\
\text{7} & \text{6} & \text{5} & \text{3} & \text{1} & \text{2} \\
\end{array}
\]

16.3 Computing a suffix array

Suppose we are given a text \( t = t_1 \ldots t_n \). In the following we will discuss how to directly compute the suffix array \( SA(t) \) in \( O(n \log n) \) time.

Let \( s_i \) denote the suffix starting at position \( i \). For a string \( s \), let \( s^h \) denote the prefix consisting of the first \( h \) symbols of \( s \). For two suffixes \( s_i \) and \( s_j \), define \( s_i \leq_h s_j \), if \( s_i^h \) is lexically less then \( s_j^h \), and define \( =_h \) and \( \geq_h \) similarly.

The algorithm proceeds inductively in \( \lceil \log_2(n+1) \rceil \) stages.

In stage 1, all suffixes are put into buckets according to their first letter. This is done by recording their positions in an integer array \( pos \), setting a newbucket bit for each index \( i \) that contains a suffix that starts with a different letter than the previous one.

Inductively, in each stage we will further partition the buckets by sorting according to twice the number of letters. We will compute stage \( 2h \) from state \( h \), that is, successively consider \( h = 2, 4, 8, 16, \ldots \).

We assume that after stage \( h \) the suffixes are partitioned into a set of \( h \)-buckets, as indicated by the newbucket bits, such that for any two suffixes \( s_i \) and \( s_j \), we have \( s_i =_h s_j \), iff the two suffixes are contained in the same \( h \)-bucket, and \( s_i \leq_h s_j \), iff the \( h \)-bucket containing \( s_i \) comes before the one containing \( s_j \).

Our goal is to reach stage \( 2h \) by sorting all suffixes by the relationship \( \leq_{2h} \), in \( O(n) \) time.

Let \( s_i \) and \( s_j \) be two suffixes belonging to the same \( h \)-bucket after the \( h^{th} \)-stage. We need to compare their next \( h \) symbols.

Note that the next \( h \) symbols of \( s_i \) or \( s_j \) are the first \( h \) symbols of \( s_{i+h} \) and \( s_{j+h} \), respectively. By induction, we already know the relative \( \leq_h \)-order of \( s_{i+h} \) and \( s_{j+h} \). How can we use this information to efficiently partition each \( h \)-bucket into smaller \( 2h \)-buckets?

Here is the main idea of the algorithm: Let \( s_i \) be the first suffix in the first \( h \)-bucket. Since \( s_i \) starts with the \( \leq_h \)-smallest string, \( s_{i-h} \) should be the first in its \( 2h \)-bucket (if it exists). Hence we can move \( s_{i-h} \) to the beginning of its \( 2h \)-bucket and record this fact.

One by one, the algorithm considers each suffix \( s_i \) as it appears in \( \leq_h \) order and moves \( s_{i-h} \) to the
next available place in its $h$-bucket. (If $i - h < 1$, then ignore the suffix $s_i$ and move on to the next one.)
16.4 Example

How to build the suffix array for $b a r b a r a$:

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 a r b a r a</td>
<td>2 a r b a r a</td>
<td>a r a</td>
</tr>
<tr>
<td>2 a r a</td>
<td>1 a r a</td>
<td>a r b a r a</td>
</tr>
<tr>
<td>3 a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>1 b a r a b a r a</td>
<td>2 b a r a b a r a</td>
<td>b a r a</td>
</tr>
<tr>
<td>2 b a r a</td>
<td>1 b a r a</td>
<td>b a r a b a r a</td>
</tr>
<tr>
<td>2 r a b a r a</td>
<td>r a</td>
<td>r a</td>
</tr>
<tr>
<td>1 r a</td>
<td>r b a r a</td>
<td>r b a r a</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
</tbody>
</table>

Here, horizontal lines indicate bucket boundaries. The numbers in front of the suffixes indicate the new position of a suffix $s_{i-h}$ within its bucket obtained by considering suffix $s_i$.

16.5 Implementation details

We maintain three integer arrays, $pos$, $inv$, $count$, and two boolean arrays, $bh$ and $b2h$, all of length $n + 1$. After stage $h$:

- $pos[i]$ contains the start position of the $i^{th}$ smallest suffix according to $\leq h$,
- $inv$ is the inverse of $pos$, with $pos[inv[i]] = i$ for all $i$, and
- $bh[i] = 1$ iff $pos[i]$ contains the left-most suffix of an $h$-bucket (i.e., $s_{pos[i]} \neq h s_{pos[i-1]}$).

The arrays $count$ and $b2h$ are temporary arrays.

We can easily fill areas $pos$, $inv$ and $bh$ for stage 1 in $O(n)$ time.

Now, assume that $pos$, $inv$ and $bh$ have the correct values after stage $h$ and consider stage $2h$.

First, in linear time:

- Reset $inv[i]$ to point to the left-most cell of the $h$-bucket containing the $i^{th}$ suffix rather than to its precise place in the bucket.
- Set $count[i] = 0$ for all $i$.

We then scan the $pos$ array in increasing order, one $h$-bucket at a time.

Let $l$ and $r$ (with $l \leq r$) mark the left and right boundary of the current $h$-bucket.

For every $i$ from $l$ to $r$ do:

- define $f_i = pos[i] - h$ (if $f_i \leq 0$, go to next $i$),
- increment $count[inv[f_i]]$ (to keep track of how many suffixes have been moved in the $h$-bucket that contains the suffix that starts at position $f_i$),
set \( \text{inv}[f_i] = \text{inv}[f_i] + \text{count}[\text{inv}[f_i]] - 1 \) (to make this the next suffix in the bucket, but without changing \( \text{pos} \)),

- set \( \text{b2h}[\text{inv}[f_i]] = 1 \) (to mark the suffix as moved.)

Before moving to the next \( h \)-bucket, do the following:

- Find all the moved suffixes and reset their \( \text{b2h} \) fields such that only the left-most one of each \( 2h \)-bucket is set to 1. This way, the \( \text{b2h} \) fields correctly mark the beginning of the \( 2h \)-buckets.

After processing all buckets, finally update \( \text{pos} \) as inverse to \( \text{inv} \) and set \( \text{b2h} \) to \( \text{b2h} \).

All these steps can be done in linear time and since there are at most \( \lceil \log_2(n + 1) \rceil \) stages, the sorting requires \( O(n \log n) \) time in the worst case.

### 16.6 Binary search for occurrences

Suppose we are given a text \( t \), suffix array \( \text{SA}(t) \) and a query \( q = q_1 \ldots q_h \).

If \( q \) occurs in \( t \), then all occurrences of \( q \) will be listed consecutively in the array \( \text{pos} \), by definition of a lexical ordering of strings.

Define

\[
L(q) = \min\{k \mid q \le_h s_{\text{pos}[k]} \text{ or } k = n\}
\]

and

\[
R(q) = \max\{k \mid s_{\text{pos}[k]} \le_h q \text{ or } k = 0\}.
\]

If \( L(q) \le R(q) \), then these two values denote the left-most and right-most entries of \( \text{pos} \) that contain the positions of occurrences of \( q \) in \( t \).

We can determine \( L(q) \) using a binary search. This takes \( O(\log n) \) comparisons and each comparison looks at \( O(h) \) positions. We can determine \( R(q) \) in a second such search. Hence:

**Lemma** By using binary search on the array \( \text{pos} \), all occurrences of \( q \) in \( t \) can be found in \( O(h \log n) \) time.

This run-time analysis is quite pessimistic: it implicitly assumes that many prefixes of \( q \) occur in the text. If this is not the case, then the comparisons will contribute a lot less to the run time. Indeed, for a “random text”, the method will run in \( O(h + \log n) \) time.

**Algorithm** (Binary search)

Input: suffix array \( \text{pos} \) and query \( q = q_1 \ldots q_h \)

Output: Left-most entry \( L(q) \) for occurrence of \( q \)

if \( q \le_h s_{\text{pos}[1]} \) then

\( L(q) = 1 \)

else if \( q >_h s_{\text{pos}[n]} \) then

\( L(q) = n \)

else

\( (L, R) = (1, N) \)
while $R - L > 1$ do
  \[
  M = \left\lceil \frac{L + R}{2} \right\rceil
  \]
  if $q \leq h \cdot s_{\text{pos}[M]}$ then
    $R = M$
  else
    $L = M$
  end
\[
L(q) = R
\]
\end

### 16.7 First speed-up of binary search

In an iteration of the binary search algorithm, let $L$ and $R$ denote the left and right boundaries of the “current search interval”. Now,

- let $l$ denote the length of the lcp (longest common prefix) of $q$ and $s_{\text{pos}[L]}$;
- let $r$ denote the length of the lcp of $q$ and $s_{\text{pos}[R]}$;
- and let $mlr = \min(l, r)$.

As the entries in $\text{pos}$ are lexically ordered, the first $mlr$ characters of $s_{\text{pos}[i]}$ must be the same for all $i \in [L, R]$.

Hence, when comparing $q$ and $s_{\text{pos}[M]}$ in the algorithm, we may skip the first $mlr$ characters.

The two variables $l$ and $r$ are easily maintained during execution of the algorithm and save many redundant comparisons.

At the start of the algorithm with $L = 1$ and $R = n$, explicitly compare $q$ with $s_{\text{pos}[1]}$ and $s_{\text{pos}[n]}$ to obtain the initial values for $l$, $r$ and $mlr$. Then, in each iteration, update the values appropriately after comparing $q$ and $s_{\text{pos}[M]}$.

This modification speeds up the algorithm and Myers and Manber report that in the practice the algorithm runs as fast as $O(h + \log n)$.

However, the worst-case complexity still is $O(h \log n)$.

Exercise: construct a query and an infinite family of texts that require the worst-case runtime.

### 16.8 Second speed-up of binary search

An examination of a character in $q$ is called redundant, if that character has been examined before.

A second speed-up modification of the binary search can be made to reduce the number of such redundant examinations to at most one per iteration of the binary search, thus to $O(\log n)$. This then gives a total time bound of $O(h + \log n)$.

In the first speed-up idea, we avoided some redundant examinations by starting a comparison of $q$ and $s_{\text{pos}[M]}$ at $mlr$. However, when $l \neq r$, all positions from $mlr = \min(l, r)$ to $\max(l, r)$ have already been examined and any further examinations of them will be redundant. This is avoided by using additional
information on the longest common prefix between certain suffixes that allows one to always start the comparison at position $\max(l, r)$. See the original paper or the book by Gusfield for details.
17 Sequence Assembly

This exposition is based on the following sources, which are all recommended reading:


17.1 Genome Sequencing

Using a method that was basically invented in 1980 by Sanger, current sequencing technology can only determine $500 - 1000$ consecutive base pairs of DNA in any one read. To sequence a larger piece of DNA, shotgun sequencing is used.

Originally, shotgun sequencing was applied to small viral genomes and to $30 - 40kb$ segments of larger genomes.

In 1994, the $1.8Mb$ genome of the bacteria $H.influenzae$ was assembled from shotgun data.

At the beginning of 2000, an assembly of the $130Mb$ Drosophila genome was published.

At the beginning of 2001, two initial assemblies of the human genome were published.
17.2 Shotgun Sequencing

Source sequence...

is copied many times...

and randomly broken into fragments, e.g. using sonication or nebulation, ...

that are then size selected, size e.g. 2kb, 10kb, 50kb or 150kb, ...

and inserted into cloning vectors.

In double barrel shotgun sequencing, each clone is sequenced from both ends, to obtain a mate-pair of reads, each read of average length 550.

17.3 Shotgun sequencing data

Given an unknown DNA sequence \( a = a_1a_2 \ldots a_L \).

Shotgun sequencing of \( a \) produces a set of reads

\[ \mathcal{F} = \{ f_1, f_2, \ldots, f_R \}, \]
of average length 550 (at present).

Essential characteristics of the data:

- Incomplete coverage of the source sequences.
- Sequencing introduces errors at a rate of about 1% for the first 500 bases, if carefully performed.
- The reads are sampled from both strands of the source sequence and thus the orientation of any given read is unknown.

17.4 The fragment assembly problem

The input is a collection of reads (or fragments) \( F = \{f_1, f_2, \ldots, f_R\} \), that are sequences over the alphabet \( \Sigma = \{A, C, G, T\} \).

An \( \epsilon \)-layout of \( F \) is a string \( S \) over \( \Sigma \) and a collection of \( R \) pairs of integers \( (s_j, e_j)_{j \in \{1, 2, \ldots, R\}} \), such that

- if \( s_j < e_j \) then \( f_j \) can be aligned to the substring \( S[s_j, e_j] \) with less than \( \epsilon \cdot |f_j| \) differences, and
- if \( s_j > e_j \) then \( f_j \) can be aligned to the substring \( S[e_j, s_j] \) with less than \( \epsilon \cdot |f_j| \) differences, then

\[ \cup_{j=1}^{R} [\min(s_j, e_j), \max(s_j, e_j)] = [1, |S|]. \]

The string \( S \) is the reconstructed source string. The integer pairs indicate where the reads are placed and the order of \( s_i \) and \( e_i \) indicate the orientation of the read \( f_i \), i.e. whether \( f_i \) was sampled from \( S \) or its complement \( \overline{S} \).

The set of all \( \epsilon \)-layouts models the set of all possible solutions. There are many such solutions and so we want a solution that is in some sense best. Traditionally, this has been phrased as the Shortest Common Superstring Problem (SCS) of the reads within error rate \( \epsilon \).

Unfortunately, the SCS Problem often produces “overcompressed” results.

Consider the following source sequence that contains two instances \( R, R' \) of a high identity repeat, and three stretches of unique sequence \( A, B \) and \( C \):

![Reads and reconstruction](image)

The shortest answer isn’t always the best and the interior part \( R_c \approx R'_c \) of the repeat region is overcompressed:
17.5 Sequence assembly in three stages

The sequence assembly problem is usually divided into three phases:

1. In the **overlap** phase, every read is compared with every other read, and the overlap graph is computed.

2. In the **layout** phase, the pairs \((s_j, e_j)\) are determined that position every read in the assembly.

3. In the **consensus** phase, a multialignment of all the placed reads is produced to obtain the final sequence.

17.6 The overlap phase

For a read \(f_i\), we must calculate how it overlaps any other read \(f_j\) (or its reverse complement, \(\overline{f_j}\)). Holding \(f_i\) fixed in orientation, \(f_i\) and \(f_j\) can overlap in the following ways:

\[
\begin{array}{c}
\text{\(f_i\)} \quad \text{\(f_j\)} \\
\text{\(f_i\)} \quad \text{\(f_j\)} \\
\text{\(f_i\)} \quad \text{\(f_j\)} \\
(\text{\(f_i\)} \quad \text{\(f_j\)})
\end{array}
\]

The number of possible relationships doubles, when we also consider \(\overline{f_j}\).

The overlap phase is the computational bottleneck in large assembly projects. For example, assembling all 27 million human reads produced at Celera requires

\[
2 \cdot \binom{27000000}{2} \approx 1458000000000000
\]

comparisons.

For any two reads \(a\) and \(b\) (and either orientation of the latter), one searches for the overlap alignment with the highest alignment score, based on a similarity score \(s(a, b)\) on \(\Sigma\) and an indel penalty \(g(k) = k\delta\).

Let \(S(a, b)\) be the maximum score over all alignments of two reads \(a = a_1a_2\ldots a_m\) and \(b = b_1b_2\ldots b_n\), we want to compute:

\[
A(a, b) = \max \left\{ S(a_k, a_{k+1}\ldots a_i, b_l b_{l+1}\ldots b_j) \mid \begin{cases} 1 \leq k \leq i \leq m, \\
1 \leq l \leq j \leq n, \\
\text{and } i = m \text{ or } j = n \text{ holds} \end{cases} \right\}.
\]
17.7 Overlap alignment

This is a standard pairwise alignment problem (similar to local alignment, except we don’t have a 0 in the recursion) and we can use dynamic programming to compute:

\[ A(i, j) = \max\{S(a_k, a_{k+1} \ldots a_i, b_l b_{l+1} \ldots b_j) \mid 1 \leq k \leq i \text{ and } 1 \leq l \leq j\}. \]

**Algorithm (Overlap alignment)**

Input: \(a = a_1 a_2 \ldots a_n\) and \(b = b_1 b_2 \ldots b_m\), \(s(\cdot, \cdot)\) and \(\delta\)

Output: \(A(i, j)\)

begin

\(A(0, j) = A(i, 0) \leftarrow 0\) for \(i = 1, \ldots, n, j = 1, \ldots, m\)

for \(i = 1, \ldots, n;\)

for \(j = 1, \ldots, m;\)

\(A(i, j) \leftarrow \max\left\{ A(i - 1, j) - \delta, A(i, j - 1) - \delta, A(i - 1, j - 1) + s(a_i, b_j) \right\} \)

end

Runtime is \(O(nm)\).

Given two reads \(a = a_1 a_2 \ldots a_m\) and \(b = b_1 b_2 \ldots b_n\). For the matrix \(A(i, j)\) computed as above, set

\((p, q) := \arg \max\{A(i, j) \mid i = m \text{ or } j = n\}\).

There are two cases:

- \(p = m\)
- \(q = n\)

The trace-back paths look like this:

The alignments look like this:

---

17.8 Faster overlap detection

Dynamic programming is too slow for large sequencing projects. Indeed, it is wasteful, as in assembly, only high scoring overlaps with more than 96% identity, say, play a role.

One can use a *seed and extend* approach (as used in BLAST):
1. Produce the concatenation of all input reads $H = f_1 f_2 \ldots f_L$.

2. For each read $f_i \in \mathcal{F}$: Find all seeds, i.e. exact matches between $k$-mers of $f_i$ and the concatenated sequence $H$. (Merge neighboring seeds.)

3. Compute extensions: Attempt to extend each (merged) seed to a high scoring overlap alignment between $f_i$ and the corresponding read $f_j$.

(A $k$-mer is a string of length $k$. In this context, $k = 18 \ldots 22$)

Computation of seeds:

![Diagram showing the computation of seeds]

Extension of seeds using banded dynamic programming (running time is linear in the read length):

![Diagram showing the extension of seeds]

17.9 True and repeat-induced overlaps

Assume that we have found a high quality overlap $o$ between $f_i$ and $f_j$. There are three possible cases:

- The overlap $o$ corresponds to an overlap of $f_i$ and $f_j$ in the source sequence. In this case we call $o$ a true overlap.
- The reads $f_i$ and $f_j$ come from different parts of the source sequence and their overlapping portions are contained in different instances of the same repeat, this is called a repeat-induced overlap.
- The overlap exists by chance. To avoid short random overlaps, one requires that an overlap is at least 40bp long, say.

![Diagram showing true and repeat-induced overlaps]

True overlap between $f_i$ and $f_j$, repeat induced overlap between $f_k$ and $f_l$. 
17.10 Avoiding repeat-induced overlaps

To avoid the computation of repeat-induced overlaps, one strategy is to only consider seeds in the seed-and-extend computation whose \( k \)-mers are not contained inside a repeat. In this way we can ensure that any computed overlap has a significant unique part.

There are two strategies for this:

- **Screening known repeats**: Each read is aligned against a database of known repeats, i.e. using Repeatmasker. Portions of reads that match a known repeat are labeled repetitive.

- **De novo screening**: Choose \( k = 20 \), say. Let \( H \) denote the concatenation of all reads. For each \( k \)-mer contained in \( H \), we determine how many times it occurs in \( H \) and then label those \( k \)-mers (and their occurrences in the set of reads) as repetitive, whose number of occurrences is unexpectedly high.

17.11 Celera’s overlapper

The assembler developed at Celera Genomics employs an overlapper than compares up to 32 million pairs of reads per second.

Overlapping all pairs of 27 million reads of human DNA using this program takes about 10 days, running on about 10-20 four processor machines (Compaq ES40), each with 4GB of main memory.

The input data file is about 50GB. To parallelize the overlap compute, each job grabs as many reads as will fit into 4GB of memory (minus the memory necessary for doing the computation) and then streams all 27 million reads against the ones in memory.

17.12 The overlap graph

The overlap phase produces an overlap graph \( OG \), defined as follows: Each read \( f_p \in \mathcal{F} \) is represented by a directed edge \((s_p, e_p)\) from node \( s_p \) to \( e_p \), representing the start and end of \( f_p \), respectively. The length of such a read edge is simply the length of the corresponding read.

An overlap between \( f_p = f_{p_1}f_{p_2} \cdots f_{p_m} \) and \( f_q = f_{q_1}f_{q_2} \cdots f_{q_n} \) gives rise to an undirected overlap edge \( e \) between \( s_p \), or \( e_p \), and \( s_q \), or \( e_q \), depending on the orientation of the overlap, e.g.:

\[
\begin{align*}
\text{The label (or \"length\") of the overlap edge } e \text{ is defined to be } -1 \text{ times the overlap length, e.g. } -(\frac{m-i+j-1}{2} + 1) \text{ in the figure.}
\end{align*}
\]
17.13 Example

Assume we are given 6 reads \( F = \{ f_1, f_2, \ldots, f_6 \} \), each of length 500, together with the following overlaps:

Here, for example, the last 320 bases of read \( f_1 \) align to the first 320 bases of the reverse complement \( \overline{f_2} \) of \( f_2 \), whereas \( f_1 \) and \( f_5 \) overlap in the first 50 bases of each.

We obtain the following overlap graph \( OG \):

Each read \( f_p \) is represented by a read edge \((s_p, e_p)\) of length \( |f_p| \). Overlaps off the start \( s_p \) or end \( e_p \) of \( f_p \) are represented by overlap edges starting at the node \( s_p \) or \( e_p \), respectively. Each overlap edge is labeled by \(-1\) times the overlap length.

17.14 The layout phase

The goal of the layout phase is to arrange all reads into an approximate multi-alignment. This involves assigning coordinates to all nodes of the overlap graph \( OG \), and thus, determining the value of \( s_i \) and \( e_i \) for each read \( f_i \).

A simple heuristic is to select a spanning forest of the overlap graph \( OG \) that contains all read edges. (A spanning forest is a set \( F \) of edges such that any two nodes in the same connected component of \( OG \) are connected by a unique, simple, unoriented path of edges in \( F \).)
component of the graph:

Such a putative alignment of reads is called a contig.

The spanning tree is usually constructed using a greedy heuristic in which the overlap edges are chosen in decreasing overlap length (i.e., increasing edge “length”).

17.15 Repeats and the layout phase

Consider the following situation:

This gives rise to the following overlap graph:
Consider this spanning tree:

A layout produced using the edge $e$ or $f$ does not reflect the true ordering of the reads and the obtained contig is called misassembled:

However, avoiding the repeat-induced edges $e$ and $f$, one obtains a correct layout:

Note that neither of the two layouts is “consistent” with all overlap edges in the graph.

### 17.16 Unitigging

The main difficulty in the layout phase is that we can’t distinguish between true overlaps and repeat-induced overlaps. The latter produce “inconsistent” layouts in which the coordinate assignment induces overlaps that are not reflected in the overlap graph (e.g., reads $f_4$ and $f_7$ in the example above).

Thus, the layout phase proceeds in two stages:

1. **Unitigging:** First, all uniquely assemblable contigs are produced, as just described. These are called unitigs.

2. **Repeat resolution:** Then, at a later stage, one attempts to reconstruct the repetitive sequence that lies between such unitigs.

Reads are sampled from a source sequence that contains repeats:

Reads that form consistent chains in the overlap graph are assembled into unitigs and the remaining “repetitive” reads are processed later:
17.17 Unique unitigs

As defined above, a “unitig” is obtained as a chain of consistently overlapping reads. However, a unitig does not necessarily represent a segment of unique source sequence. For example, its reads may come from the interior of different instances of a long (many copy) repeat:

Non-unique unitigs can be detected by virtue of the fact that they contain significantly more reads than expected.

17.18 Identifying unique unitigs

Let $R$ be the number of reads and $G$ the estimated length of the source sequence. For a unique unitig of approximate length $\rho$, the probability of the unitig containing $k$ reads, that is, of seeing $k - 1$ start positions in the interval of length $\rho$, is

$$e^{-ck} \frac{k!}{k!}, \quad \text{with } c := \frac{\rho R}{G},$$

if the unitig is not oversampled, and

$$e^{-2c(2c)^k} \frac{k!}{k!},$$

if the unitig is the result of collapsing two repeats.

(see Mike Waterman’s book, page 148, for details)

The arrival statistic used to identify unique unitigs is the (natural) log of the ratio of these two probabilities,

$$c - k \log 2.$$

A unitig is called unique, if it’s arrival statistic has a positive value of 10 or more, say.
17.19 Mate-pairs

Fragment assembly of reads produces contigs, whose relative placement and orientation with respect to each other is unknown.

Recall that modern shotgun sequencing protocols employ a so-called double barreled shotgun. That is, longer clones of a given fixed length are sequenced from both ends and one obtains a pair of reads, a mate-pair, whose relative orientation and mean $\mu$ (and standard deviation $\sigma$ of) length are known:

![Diagram of a mate-pair](image)

Typical clone lengths are $\mu = 2$kb, 10kb, 50kb or 150kb. For clean data, $\sigma \approx 10\%$ of $\mu$. Mate-pair mismatching is a problem and can effect $10 - 30\%$ of all pairs.

17.20 Scaffolding

Consider two reconstructed contigs. If they correspond to neighboring regions in the source sequence, then we can expect to see mate-pairs that span the gap between them:

![Diagram of scaffolding](image)

Such mate-pairs determine the relative orientation of both contigs, and we can compute a mean and standard deviation for the gap between them. In this case, the contigs are said to be scaffolded:

![Diagram of scaffolded contigs](image)

17.21 Determining the distance between two contigs

Given two contigs $c_1$ and $c_2$ connected by mate-pairs $m_1, m_2, \ldots, m_k$. Each mate-pair gives an estimation of the distance between the two contigs.

These estimations can viewed as independent measurements $(l_1, \sigma_1), (l_2, \sigma_2), \ldots (l_k, \sigma_k)$ of the distance $D$ between the two contigs $c_1$ and $c_2$. Following standard statistical practice, they can be combined as follows:

Define $p := \sum \frac{l_i}{\sigma_i^2}$ and $q = \sum \frac{1}{\sigma_i^2}$. We set the distance between $c_1$ and $c_2$ to

$$D := \frac{p}{q},$$

with standard deviation $\sigma := \frac{1}{\sqrt{q}}$.

Here is an example:
17.22 The significance of mate-pairs

Given two contigs $c_1$ and $c_2$. If there is only one mate-pair between the two contigs, then due to the high error rates associated with mate-pairs, this is not significant.

If, however, $c_1$ and $c_2$ are unique unitigs, and if there exist two different mate-pairs between the two that give rise to the same relative orientation and similar estimations of the gap size between $c_1$ and $c_2$, then this the scaffolding of $c_1$ and $c_2$ is highly reliable.

This is because that probability that two false mate-pairs occur that confirm each other, is extremely small.

17.23 Example

Let the sequence length be $G = 120Mb$, for example (Drosophila). For simplicity, assume we have 5-fold coverage of mate-pairs, with a mean length of $\mu = 10kb$ and standard deviation of $\sigma = 1kb$.

Consider a false mate-pair $m_1 = (f_1, f_2)$ with reads $f_1$ and $f_2$. Let $N_1$ and $N_2$ denote the two intervals (in the source sequence) of length $3\sigma$ centered at the starts of $f_1$ and $f_2$, respectively. Both have length $6kb$.

Consider a second false mate $m_2 = (g_1, g_2)$ with $g_1$ inside $N_1$. The probability that $g_2$ lies in $N_2$ is roughly

$$\frac{6kb}{120Mb} = \frac{1}{20000}.$$

Assume that the reads have length 600. Assume that 10% of all mate-pairs are false. At 5-fold coverage, the interval $N_1$ is covered by about $5 \cdot \frac{6000}{600} = 50$ reads, of which $\approx 5$ have false mates.
Hence, the probability that $m_1$ is confirmed by some second false mate-pair $m_2$ is

$$\approx 5 \cdot \frac{1}{20000} = \frac{1}{4000} = 0.00025.$$  

17.24 The overlap-mate graph

Given a set of reads $\mathcal{F} = \{f_1, f_2, \ldots, f_R\}$ and let $G$ denote the overlap graph associated with $\mathcal{F}$.

Given one set (or more) $M_{\mu, \sigma} = \{m_1, \ldots, m_k\}$ of mate-pairs $m_k = (f_i, f_j)$, with mean $\mu$ and standard deviation $\sigma$.

Let $f_i$ and $f_j$ be two mated reads represented by the edges $(s_i, e_i)$ and $(s_j, e_j)$ in $G$. We add an undirected mate edge between $e_i$ and $e_j$, labeled $(\mu, \sigma)$, to indicate that $f_i$ and $f_j$ are mates and thus obtain the overlap-mate graph:

17.25 The contig-mate graph

Given a set of $\mathcal{F}$ of fragments and a set of assembled contigs $\mathcal{C} = \{c_1, c_2, \ldots, c_t\}$. A more useful graph is obtained as follows:

Represent each assembled contig $c_i$ by a contig edge with nodes $s_i$ and $e_i$. Then, add mate edges between such nodes to indicate that the corresponding contigs contain fragments that are mates:
17.26 Edge bundling

Consider two contigs $c_1$ and $c_2$, joined by mate-pair edges $m_1, \ldots, m_k$ between node $e_1$ and $s_2$, say. Every maximal subset of mutually confirming mate edges is replaced by a single *bundled mate edge* $e$, whose mean length $\mu$ and standard deviation $\sigma$ are computed as discussed above. Any such bundled edge is labeled $(\mu, \sigma)$.

(A heuristic used to compute these subsets is to repeatedly bundle the median-length simple mate edge with all mate edges within three standard deviations of it, until all simple mate edges have been bundled.)

Additionally, we set the weight $w(e)$ of any mate edge to 1, if it is a simple mate edge, and to $\sum_{i=1}^{k} w(e_i)$, if it was obtained by bundling edges $e_1, \ldots, e_k$.

Consider the following graph:

Assuming that mate edges drawn together have similar lengths and large enough standard deviation, edge bundling will produce the following graph:

17.27 Transitive edge reduction

Consider the previous graph with some specific edge lengths:

The mate edge $e$ gives rise to estimation of the distance from the right node of contig $c_1$ to the left node of $c_3$ that is similar to the one obtained by following the path $P = (g, c_2, h)$. Based on this *transitivity* property we can *reduce* the edge $e$ on to the path $p$: 
to obtain:

Consider two nodes $v$ and $w$ that are connected by an alternating path $P = (m_1, b_1, m_2, \ldots, m_k)$ of mate-edges $(m_1, m_2, \ldots)$ and contig edges $(c_1, c_2, \ldots)$ from $v$ to $w$, beginning and ending with a mate-edge. We obtain a mean length and standard deviation for $P$ by setting $l(P) := \sum m_i \mu(m_i) + \sum c_i l(c_i)$ and $\sigma(P) := \sqrt{\sum m_i \sigma(m_i)^2}$.

We say that a mate-edge $e$ from $v$ to $w$ can be transitive reduced on to the path $P$, if $e$ and $P$ approximately have the same length, i.e., if $|\mu(e) - l(P)| \leq C \cdot \max\{\sigma(e), \sigma(P)\}$ for some constant $C$, typically 3. If this is the case, then we can reduce $e$ by removing $e$ from the graph and incrementing the weight of every mate-edge $m_i$ in $P$ by $w(e)$.

In the following, we will assume that any contig-mate graph considered has been edge-bundled and perhaps also transitive reduced to some degree.

### 17.28 Happy mate-pairs

Consider a mate-pair $m$ of two reads $f_i$ and $f_j$, obtained from a clone of mean length $\mu$ and standard deviation $\sigma$:

Assume that $f_i$ and $f_j$ are contained in the same contig or scaffold of an assembly. We call $m$ happy, if $f_i$ and $f_j$ have the correct relative orientation (i.e., are facing each other) and are at approximately the right distance, i.e., $|\mu - |s_i - s_j|| \leq 3\sigma$, say. Otherwise, $m$ is unhappy. Two unhappy mates are highlighted here:

### 17.29 Ordering and orientation of the contig-mate graph

Given a collection of contigs $C = \{c_1, c_2, \ldots, c_k\}$ constructed from a set of reads $F = \{f_1, f_2, \ldots, f_R\}$, together with the corresponding mate-pair information $M$. Let $G = (V, E)$ denote the associated contig-mate graph.

An ordering (and orientation) of $G$ (or $C$) is a map $\phi : V \to \mathbb{N}$ such that $|\phi(b_i) - \phi(c_i)| = l(c_i)$ for all contigs $c_i \in C$, in other words, an assignment of coordinates to all nodes that preserves contig lengths.
Additionally, we require \( \{\phi(b_i), \phi(e_i)\} \neq \{\phi(b_j), \phi(e_j)\} \) for any two distinct contigs \( c_i \) and \( c_j \).

### 17.30 Example

Given the following contig-mate graph:

![Contig-Mate Graph](image)

An ordering \( \phi \) assigns coordinates \( \phi(v) \) to all nodes \( v \) and thus determines a layout of the contigs:

![Contig Layout](image)

### 17.31 Happiness of mate edges

Let \( G = (V, E) \) be a contig-mate graph and \( \phi \) an ordering of \( G \).

Consider a mate-edge \( e \) with nodes \( v \) and \( w \). Let \( c_i \) denote the contig edge incident to \( v \) and let \( c_j \) denote the contig edge incident to \( w \). Let \( v' \) and \( w' \) denote the other two nodes of \( c_i \) and \( c_j \), respectively. We call \( e \) happy (with respect to \( \phi \)), if \( c_i \) and \( c_j \) have the correct relative orientation, and if the distance between \( v \) and \( w \) is approximately correct, in other words, we require that either

1. \( \phi(v') \leq \phi(v) \) & \( |\phi(w) - \phi(v) - \mu(e)| \leq 3\sigma(e) \) & \( \phi(w) \leq \phi(w') \), or
2. \( \phi(w') \leq \phi(w) \) & \( |\phi(v) - \phi(w) - \mu(e)| \leq 3\sigma(e) \) & \( \phi(v) \leq \phi(v') \).

Otherwise, \( e \) is unhappy.

### 17.32 The Contig-Matepair Ordering Problem

Given a collection of contigs \( C = \{c_1, c_2, \ldots, c_k\} \) constructed from a set of reads \( F = \{f_1, f_2, \ldots, f_R\} \), together with the corresponding mate-pair information \( M \). Let \( G = (V, E) \) denote the associated contig-mate graph.
**Problem** The *Contig-Matepair Ordering Problem* is to find an ordering of $G$ that maximizes the sum of weights of happy mate edges.

**Theorem** The corresponding decision problem is NP-complete.

(The decision problem is: Given a contig-mate graph $G$, does there exist an ordering of $G$ such that the total weight of all happy edges $\geq K$?)

### 17.33 Proof of NP-completeness

Recall: to prove that a problem $X$ is NP-complete one must reduce a known NP-complete problem $N$ to $X$. In other words, one must show that any instance $I$ of $N$ can be translated into an instance $J$ of $X$ in polynomial time such that $I$ has the answer *true* iff $J$ does.

We will use the following NP-complete problem:

**BANDWIDTH**: For a given graph $G = (V, E)$ with node set $V = \{v_1, v_2, \ldots, v_n\}$ and number $K$, does there exist a *permutation* $\phi$ of $\{1, 2, \ldots, n\}$ such that for all edges $\{v_i, v_j\} \in E$ we have $|\phi(i) - \phi(j)| \leq K$? (See Garey and Johnson 1979 for details.)

![A graph with bandwidth 4:](image)

Problem is in NP: For a given ordering $\phi$, we can determine whether the number of happy mate-edges exceeds the given threshold $K$ in polynomial time by simple inspection of all mate edges.

Reduction of BANDWIDTH: Given an instance $G = (V, E)$ of this problem, we construct a contig graph $G' = (V', E')$ in polynomial time as follows:

First, set $V' := V$ and $E' := E$, and let these edges be the mate-edges, setting $\mu(e) := 1 + \frac{K-1}{2}$ and $\sigma(e) := \frac{K-1}{6}$ so as to obtain a happy range of $[1, K]$, and $w(e) := 1$, for every mate-edge $e$.

Then, for each initial node $v \in V$, add a new auxiliary node $v'$ to $V'$ and join $v$ and $v'$ by a contig edge of length 0.

The answer to the BANDWIDTH question is *true*, iff the graph $G'$ has an ordering $\phi$ such that all mate edges in $G'$ are happy:

A graph $G$ has BANDWIDTH $\leq K$ $\iff$ $\exists$ permutation $\phi$ such that $\{v_i, v_j\} \in E$ implies $|\phi(i) - \phi(j)| \leq K$ $\iff$ $\exists$ ordering $\phi$ such that $\{v_i, v_j\} \in E$ implies $1 \leq |\phi(i) - \phi(j)| \leq K$ $\iff$ $\exists$ ordering $\phi$ such that $e = \{v_i, v_j\} \in E$ implies $\mu(e) - 3\sigma(e) \leq |\phi(i) - \phi(j)| \leq \mu(e) + 3\sigma(e)$ $\iff$ all mate-edges of $G'$ are happy. □
17.34 Spanning tree heuristic for the Contig Ordering Problem

An ordering $\phi$ that maximizes the number of happy mate edges is a useful scaffolding of the given contigs.

The simplest heuristic for obtaining an ordering is to compute a maximum weight spanning tree for the contig-mate graph and use it to order all contigs, similar to the read layout problem.

Unfortunately, this method does not work well in practice, as false mate edges lead to incorrect interleaving of contigs from completely different regions of the source sequence:

17.35 Representing an ordering in the mate-contig graph

By the definition given above, an ordering is an assignment of coordinates to all nodes of the contig-mate graph that corresponds to a scaffolding of the contigs. When we are not interested in the exact coordinates, then the relative order and orientation of the contigs can be represented as follows:

Given a contig-mate graph $G = (V, E)$. A set $S \subseteq E$ of selected edges is called a scaffolding of $G$, if it has the following two properties:

- every contig edge is selected, and
- every node is incident to at most two selected edges.

Thus, a scaffolding of $G$ is a set of non-intersecting selected paths, each representing a scaffolding of its contained contigs.

The following example contains two chains of selected edges representing scaffolds $s_1 = (c_1, c_2, c_3, c_4)$ and $s_2 = (c_5, c_6, c_7)$:

However, to be able to represent the interleaved scaffolding discussed earlier, we need to add some inferred edges (shown here as dotted lines) to the graph:
17.36 Greedy path-merging

Given a contig-mate graph $G = (V, E)$. The greedy path merging algorithm is a heuristic for solving the Contig Ordering Problem. It proceeds “bottom up” as follows, maintaining a valid scaffolding $S \subseteq E$:

Initially, all contig edges $c_1, c_2, \ldots, c_k$ are selected, and none others. At this stage, the graph consists of $k$ selected paths $P_1 = (c_1), \ldots, P_k = (c_k)$.

Then, in ordering of decreasing weight we consider each mate edge $e = \{v, w\}$: If $v$ and $w$ lie in the same selected path $P_i$, then $e$ is a chord of $P_i$ and no action is necessary.

If $v$ and $w$ are contained in two different paths $P_i$ and $P_j$, then we attempt to merge the two paths to obtain a new path $P_k$ and accept such a merge, only if the increase of $H(G)$, the number of happy mate edges, is larger than the increase of $U(G)$, the number of unhappy ones.

17.37 The greedy path-merging algorithm

**Algorithm** Given a contig-mate graph $G$. The output of this algorithm is a node-disjoint collection of selected paths in $G$, each one defining an ordering of the contigs whose edges it covers.

begin
Select all contig edges.
for each mate-edge $e$ in descending order of weight:
  if $e$ is not selected:
    Let $v, w$ denote the two nodes connected by $e$
    Let $P_1$ be the selected path incident to $v$
    Let $P_2$ be the selected path incident to $w$
    if $P_1 \neq P_2$ and we can merge $P_1$ and $P_2$ (guided by $e$)
      to obtain $P$:
        if $H(P) - (H(P_1) + H(P_2)) \geq U(P) - (U(P_1) + U(P_2))$:
          Replace $P_1$ and $P_2$ by $P$
end

17.38 Merging two paths

Given two selected paths $P_1$ and $P_2$ and a guiding unselected mate-edge $e_0$ with nodes $v_0$ (incident to $P_1$) and $w_0$ (incident to $P_2$). Merging of $P_1$ and $P_2$ is attempted as follows:
This algorithm returns \textit{true}, if it successfully produced a new selected path $P$ containing all contigs edges in $P_1$ and $P_2$, and \textit{false}, if it fails.

Merging proceeds by “zipping” the two paths $P_1$ and $P_2$ together, first starting with $e_0$ and “zipping” to the right. Then, with the edge labeled $h$ now playing the role of $e_0$, zipper to the left. Merging is said to \textit{fail}, if the positioning of the “active” contig $c_1^i$ implies that it must overlap with some contig in $P_2$ by a significant amount, but no such alignment (of sufficiently high quality) exists.

### 17.39 Example

Here are we are given 5 contigs $c_1, \ldots, c_5$, each of length $l(c_i) = 10000$

The final scaffolding is $(c_1, c_2, c_3, c_5, c_4)$. 


17.40 Repeat resolution

Consider two unique unitigs $u_1$ and $u_2$ that are placed next to each other in a scaffolding, due to a heavy mate edge between them:

We consider all non-unique unitigs and singleton reads that potentially can be placed between $u_1$ and $u_2$ by mate edges:

Different heuristics are used to explore the corresponding local region of the overlap graph in an attempt to find a chain of overlapping fragments that spans the gap and is compatible with the given mate-pair information:

17.41 Summary

Given a collection $F = \{ f_1, f_2, \ldots, f_R \}$ of reads and mate-pair information, sampled from an unknown source DNA sequence. Assembly proceeds in the following steps:

1. compute the overlap graph, e.g. using a seed-and-extend approach,
2. construct all unitigs, e.g. using the minimal spanning tree approach,
3. scaffold the unitigs, e.g. using the greedy-path merging algorithm,
4. attempt to resolve repeats between unitigs, and
5. compute a multiple alignment of all reads in a given contig to obtain a consensus sequence for it.

Note that the algorithms for steps (2) and (3) that are used in actual assembly projects are much more sophisticated than ones described in these notes.

17.42 A WGS assembly of human (Celera)

Input: 27 million fragments of av. length 550bp, 70% paired:
5m pairs of length 2kb
4m pairs of length 10kb
0.9m pairs of length 50kb
0.35m pairs of length 150kb

Celera’s assembler uses approximately the following resources:

<table>
<thead>
<tr>
<th>Program</th>
<th>CPU hours</th>
<th>Max. memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Screener</td>
<td>4800</td>
<td>2GB</td>
</tr>
<tr>
<td>Overlapper</td>
<td>12000</td>
<td>4GB</td>
</tr>
<tr>
<td>Unitigger</td>
<td>120</td>
<td>32GB</td>
</tr>
<tr>
<td>Scaffolder</td>
<td>120</td>
<td>32GB</td>
</tr>
<tr>
<td>RepeatRez</td>
<td>50</td>
<td>32GB</td>
</tr>
<tr>
<td>Consensus</td>
<td>160</td>
<td>2GB</td>
</tr>
</tbody>
</table>

Total: ≈ 18000 CPU hours.

The size of the human genome is ≈ 3Gb. An unpublished 2001 assembly of the 27m fragments has the following statistics:

- The assembly consists of 6500 scaffolds that span 2.776Gb of sequence.
- The spanned sequence contains 150,000 gaps, making up 148Mb in total.
- Of the spanned sequence, 99.0% is contained in scaffolds of size 30kb or more.
- Of the spanned sequence, 98.7% is contained in scaffolds of size 100kb or more.
18 Sequencing From Compomers

This exposition is based on the following sources, which are recommended reading:


De-novo sequencing is performed today using electrophoresis-based approaches, based on ideas introduced by Sanger in 1977. We will discuss an alternative proposal that would use mass spectrometry to gather sequence data and produce reads of length 200+, if it can be made to work in practice.

18.1 Mass Spectrometry

Mass spectrometers are an analytical tool used for measuring the molecular weight (MW) of a sample. For large samples such as biomolecules, molecular weights can be measured to within an accuracy of 0.01% of the total molecular weight of the sample i.e. within a 4 Daltons (Da, $1 \text{ Da} = \frac{1}{12}$ mass of a carbon-12 nucleus) error for a sample of 40,000 Da. This is sufficient to allow minor mass changes to be detected, e.g. the substitution of one amino acid for another, or a post-translational modification.

For small organic molecules the molecular weight can be measured to within an accuracy of 5 ppm (parts-per-million), which is often sufficient to confirm the molecular formula of a compound, and is also a standard requirement for publication in a chemical journal.

Structural information can be generated using certain types of mass spectrometers, usually tandem mass spectrometers, and this is achieved by fragmenting the sample and analyzing the products generated.

This procedure is useful for the structural elucidation of organic compounds, for peptide or oligonucleotide sequencing, and for monitoring the existence of previously characterized compounds in complex mixtures with a high specificity by defining both the molecular weight and a diagnostic fragment of the molecule simultaneously e.g. for the detection of specific drug metabolites in biological matrices.

18.2 MALDI TOF MS

MALDI TOF MS stands for matrix assisted laser desorption ionization time-of-flight mass spectrometry.
MALDI is based on the bombardment of sample molecules with a laser light to bring about sample ionization. The sample is pre-mixed with a highly absorbing matrix compound for the most consistent and reliable results, and a low concentration of sample to matrix works best. The matrix transforms the laser energy into excitation energy for the sample, which leads to sputtering of analyte and matrix ions from the surface of the mixture.

The time-of-flight analyzer separates ions according to their mass(m)-to-charge(z) (m/z) ratios by measuring the time it takes for ions to travel through a field free region known as the flight, or drift, tube. The heavier ions are slower than the lighter ones.

The m/z scale of the mass spectrometer is calibrated with a known sample that can either be analyzed independently (external calibration) or pre-mixed with the sample and matrix (internal calibration). MALDI is a “soft” ionization method and so results predominantly in the generation of singly charged molecular-related ions regardless of the molecular weight, hence the spectra are relatively easy to interpret. Fragmentation of the sample ions does not usually occur.

Example Positive ionization MALDI m/z spectrum of the peptide ACTH (clip 18-39; MW 2465.7 Da) using cyano-4-hydroxycinnamic acid as matrix:

18.3 The Partial Cleavage Experiment

Suppose we are given many copies of a target (or sample) DNA molecule of length 100 – 500, and assume (for simplicity) that they are single stranded.
In four separate reactions, one for each base, we cleave the copies of the target using a base-specific (bio-)chemical cleavage reaction (e.g., RNAse A, RNAse T1, uracil-DNA-glycosylase, pn-bond cleavage and others.)

More precisely, we only perform partial cleavage reactions so that only a certain percentage of the possible “cut bases” get cut, thus ideally producing a mixture of all fragments obtainable by cutting the target sequence at any two cut bases:

![Diagram of partial cleavage reactions](Source: Böcker, 2003)

MALDI TOF mass spectrometry is then applied to the products of each of the four partial cleavage experiments in turn, producing four sample spectra that correlate mass and signal intensity of sample particles. Each is analyzed to extract a list of signal peaks with masses and intensities:

![Sample spectra graph](Source: Böcker, 2003)

The four nucleotides have different weights and so each peak in such a spectrum corresponds to a fragment of a specific composition of bases.

Indeed, given the measured weight $W$ of a fragment $f$, and the different molecular weights $\omega(A)$, $\omega(C)$, $\omega(G)$ and $\omega(T)$ of the four types of nucleotides, we can easily compute the base-composition of the fragment in $O(|f|^3)$ steps. (Exercise)

The key idea is to determine the precise sequence of the target molecule as the (hopefully unique) sequence whose calculated spectrum is the same as the measured one.

This is a difficult problem, for two reasons:

- given only the base-compositions of the fragments, the problem of determining a possible target sequence is NP-hard, and
- MS does not work well for sequences longer than $\approx 25$ nt.

It has been used in situations such as in pathogen identification in which information on the target sequence is already known.

### 18.4 Some Practical Observations

1. Current MS can only detect modules of weight $\leq 8000$ Dalton,

2. MS can show an inaccuracy of $1-2$ Da in high-throughput,
3. MS spectra are noisy and so low-intensity signals are hard to identify,
4. for a fixed cleavage reaction, different compositions can have nearly identical masses,
5. partial cleavage suffers from exponential decay, so long fragments will be difficult to detect, and
6. a small number of the terminal bases of the target are often known in advance.

### 18.5 The Composer Spectrum

Let \( s = s_1 \ldots s_n \) be a string over the alphabet \( \Sigma \). We denote the number of \((\text{non-overlapping})\) occurrences of a string \( x \) in \( s \) by:

\[
\text{ord}_x(s) = \max\{k \mid \text{there exist } t_0, \ldots, t_k \in \Sigma^* \text{ with } s = t_0xt_1 \ldots xt_k\}.
\]

So, \( x \) is a substring of \( s \) iff \( \text{ord}_x(s) > 0 \).

For two strings \( s, x \in \Sigma^* \) we define the string spectrum \( S(s, x) \) of \( s \) by:

\[
S(s, x) = \{y \in \Sigma^* \mid \text{there exist } a, b \in \Sigma^* \text{ with } s \in \{yxb, axyb, axy\}\} \cup \{s\},
\]

the set of all substrings of \( s \) that are bounded by \( x \) or by the ends of \( s \).

In the following, we will call \( s \) a sample string, \( x \) a cut-string and the elements of \( S(s, x) \) the fragments of \( s \) (under \( x \)).

**Example** Consider \( \Sigma = \{0, A, C, G, T, 1\} \), where \( 0 \) and \( 1 \) are only used to denote the start and end of the sample string. Let

\[
s = 0ACATGTG1 \quad \text{and} \quad x = T.
\]

Then we have:

\[
S(s, x) = \{0ACA, G, G1, 0ACATG, GTG1, 0ACTGTG1\}.
\]

We introduce the characters \( 0 \) and \( 1 \) to reduce symmetry, see observation (6) above.

A **compomer** is a map \( c : \Sigma \rightarrow \mathbb{N}^\geq \) that records the composition of a fragment of DNA, which we will often write as

\[
c = (A_i C_j G_k T_l),
\]

with \( i = c(A) \), \( j = c(C) \), \( k = c(G) \) and \( l = c(T) \), sometimes omitting the letters \( 0 \) and \( 1 \) and symbols with zero values.

The function \( \text{comp} \) maps every sequence \( s \in \Sigma^* \) to its compomer by counting the number of characters of each type in \( s \):

\[
\text{comp}(s) : \Sigma \rightarrow \mathbb{N}, \quad \sigma \mapsto c(\sigma) = |\{1 \leq i \leq |s| : s_i = \sigma\}|.
\]

Given a sample sequence \( s \) and cut string \( x \), the **compomer spectrum** \( C(s, x) \) consists of all compomers of all fragments in the string spectrum \( S(s, x) \):

\[
C(s, x) = \text{comp}(S(s, x)) = \{\text{comp}(y) : y \in S(s, x)\}.
\]

For the example \( s = 0ACATGTG1 \) we have:

\[
C(s, T) = \{0A_2C_1, G_1, G_11, 0A_2C_1G_1T_1, G_2T_11, 0A_2C_1G_2T_21\}.
\]
18.6 Reconstruction from Complete Compomer Spectra

We can now formulate the Reconstruction from Complete Compomer Spectra (RCCS) problem:

For an unknown string $s$ and a set of known cut strings $X$, can we uniquely reconstruct $s$ from its compomer spectra $C(S, x)$, $x \in X$?

Note that the problem is trivial, if there exist characters 0 and 1 that uniquely denote the start and end of the sample string. Then, for suitable $X$, e.g. $X = \Sigma^1 \setminus \{0, 1\}$, the subsets $\{c \in C(s, x) \mid c(0) = 1\}$, $x \in X$, are sufficient to reconstruct $s$. (We use $\Sigma^i \subset \Sigma^*$ to denote all substrings of length $i$.)

Unfortunately, this approach will fail when applied to real MS data, due to some of the observations listed above.

18.7 Reconstruction from k-Compomer Spectra

A consequence of observation (5) is that a fragment $f$ can only be detected by mass spec if it is sufficiently short, that is, contains only a small number of cut sites, say at most 4. This leads to the following two definitions:

For a sample string $s$, cut string $x$ and number $k$, we define the $k$-string spectrum of $s$ (of order $k$) as:

$$S_k(s, x) = \{ y \in S(s, x) : \text{ord}_x(y) \leq k \}.$$

The $k$-compomer spectrum of $s$ is defined by:

$$C_k(s, x) = \text{comp}(S_k(s, x)) = \{ \text{comp}(y) : y \in S(s, x), \text{ord}_x(y) \leq k \}.$$

For the example $s = 0\text{ACATGTG}1$ we calculate:

$$C_0(s, T) = \{0A_2C_1, G_1, G_11\},$$
$$C_1(s, T) = C_0(s, T) \cup \{0A_2C_1G_1T_1, G_2T_11\},$$
$$C_2(s, T) = C_1(s, T) \cup \{0A_2C_1G_2T_21\} = C(s, T).$$

Under what conditions can we uniquely reconstruct a sample string $s$ from its compomer spectra $C_k(s, x)$, $x \in X$?

**Example** Let $\Sigma = \{0, A, B, 1\}$. We cannot uniquely reconstruct the sample string

$$s = 0\text{BABAAB1}$$

from its complete cleavage compomer spectra $C_0(s, A)$ and $C_0(s, B)$ because the string

$$s = 0\text{BABAAB1}$$

leads to the same spectra.

Similarly,

$$s = 0\text{BABAABABAB1}$$

cannot be reconstructed from its compomer spectra $C_1(s, x)$ with $x \in \{A, B\}$, and one can create such examples for every order $k$. 
18.8 False Positive Compomers

The Reconstruction from k-Compomer Spectra problem suffers from the difficulty that the resulting answer is not necessarily unique.

An additional problem is that due to Observations (2) and (4), transforming a mass spectrum into a set of compomers will introduce many false positive compomers, since there is usually only one compomer that actually corresponds to an observed peak, but many that have a very similar weight. Also, Observation (3) can add to the problem.

To address the problem of false positives, a proposed solution should not be required to explain all observed compomers.

18.9 Sequencing From Compomers (SFC) Problem

We now formulate the Sequencing From Compomers (SFC) Problem:

For a fixed order $k$, let $X \subseteq \Sigma^*$ be a set of cut strings and, for each $x \in X$, let $C_x$ be a compomer set. Let $S \subseteq \Sigma^*$ be a set of sample string candidates. Find all strings $s \in S$ that satisfy $C_k(s, x) \subseteq C_x$ for all $x \in X$.

Note that the SFC problem has some trivial solutions if we allow $S = \Sigma^*$, for example, $s = \epsilon$ always satisfies the inclusion conditions. So, $S$ should be chosen to exclude such trivial solutions.

Note that even a small number of compomers can lead to an exponential number of strings, for a given fixed string length:

For example, let

- $\Sigma = \{A, B\}$,
- $k = 0$,
- $S = \Sigma^n$ for some $n \in \mathbb{N}$,
- $X = \Sigma^1$, and
- $C_A = \{B_1\}$ and $C_B = \{A_1, A_2\}$.

Note that every string $s \in S$ that is an arbitrary concatenation $s_0s_1s_2\ldots s_k$ with $s_0 \in \{A, AA\}$ and $s_j \in \{BA, BAA\}$ for $j = 1, \ldots, k$ satisfies the conditions

$$C_0(s, A) \subseteq C_A \text{ and } C_0(s, B) \subseteq C_B.$$
18.10 The undirected sequencing graph

To address the SFC problem of order \( k = 1 \), we introduce the **undirected sequencing graph**.

Let \( \mathcal{C} \) be an arbitrary set of compomers over \( \Sigma \). For each single-nucleotide cut string \( x \in \Sigma^1 \), we define the undirected sequencing graph \( G_u(\mathcal{C}, x) = (V, E) \) as follows:

- the vertex set \( V \) consists of all compomers \( c \in \mathcal{C} \) such that \( c(x) = 0 \) holds, and
- the edge set \( E \) consists of those \( \{v, w\} \subseteq V \) that satisfy:
  \[ v + \text{comp}(x) + w \in \mathcal{C}, \]

where \( v \) and \( w \) are not required to be distinct.

A sequence \( p = (p_0, p_1, \ldots, p_r) \) of nodes \( p_i \in V \) is called a **walk** in \( G = (V, E) \), if \( \{p_{i-1}, p_i\} \in E \) for all \( i = 1, \ldots, r \).

**Example** For \( \Sigma = \{0, A, C, G, T, 1\} \), \( s = 0CTAATCATAGTGCTG1 \), and \( x = T \), the compomer spectrum of order 1 is:

\[
\mathcal{C} := \mathcal{C}_1(s, T) = \{0C_1, 0A_2C_1T_1, A_2, A_3C_1T_1, A_1C_1, A_2C_1G_1T_1, A_1G_1, A_1C_1G_2T_1, C_1G_1, C_1G_2T_1, G_11\}.
\]

The resulting graph \( G_u(\mathcal{C}, T) \) is:

We say that a string \( s \in \Sigma^* \) is 1-**compatible** with a compomer set \( \mathcal{C} \) under \( x \), if \( \mathcal{C}_1(s, x) \subseteq \mathcal{C} \) holds. In this case \( s \) is a solution to the SFC problem of order 1 with respect to \( x \).

We say that \( s \) is **compatible** with a walk \( p = (p_0, \ldots, p_r) \) in \( G_u(\mathcal{C}, x) \), if there exist strings \( s_0, \ldots, s_r \in \Sigma^* \) such that

\[
s = s_0xs_1xs_2x \ldots xs_r,
\]

with \( \text{comp}(s_j) = p_j \) for \( j = 0, \ldots, r \). Such a set of strings \( s_0, \ldots, s_r \) is called an \( x \)-**partitioning** of \( s \), if additionally, \( \text{ord}_x(s_j) = 0 \) for all \( j = 0, \ldots, r \).

For \( x \in \Sigma^1 \), there exists exactly one \( x \)-partitioning of \( s \).

In the above example, the walk \( (0C_1, A_2, A_1C_1, A_1G_1, C_1G_1, G_11) \) is compatible with the input sequence \( s = 0CTAATCATAGTGCTG1 \). But also sequences like \( OCTATCGTG1 \) or \( 0CTAATCGTGATGCTG1 \) are compatible with walks in \( G_u(\mathcal{C}, T) \).

We have the following result:
Lemma Let $s \in \Sigma^*$ be a string and $C$ a set of compomers over $\Sigma$. Then, $s$ is 1-compatible with $C$ under $x \in \Sigma^1$ iff there exists a walk $p$ in $G_u(C, x)$ such that $s$ is compatible with $p$. Moreover, $p$ is unique.

Thus, an algorithm to solve the SFC problem can be based on the following:

For every walk $p$ of a sequencing graph $G_u(C, x)$ there exists one or more sequences $s \in \Sigma^*$ that are compatible with $p$, and therefore are 1-compatible with $C$. 

19 DNA arrays

This exposition is based on the following sources, which are recommended reading:

3. Ron Shamir, Analysis of Gene Expression Data, lectures 1 and 4, 2002.

- Also known as: biochips, DNA chips, oligo arrays, DNA microarrays or gene arrays.
- An array is an orderly arrangement of (spots of) samples.
- Samples are either DNA or DNA products.
- Each spot in the array contains many copies of the sample.
- Array provides a medium for matching known and unknown DNA samples based on base-pairing (hybridization) rules and for automating the process of identifying the unknowns.
- Sample spot size in microarray less than 200 microns and an array contains thousands of spots.
- Microarrays require specialized robotics and imaging equipment.
- High-throughput biology: a single DNA chip can provide information on thousands of genes simultaneously.

19.1 Two possible formats

We are given an unknown target nucleic acid sample and the goal is to detect the identity and/or abundance of its constituents using known probe sequences. Single stranded DNA probes are called oligo-nucleotides or oligos.

There are two different formats of DNA chips:

- Format I: The target (500-5000 bp) is attached to a solid surface and exposed to a set of probes, either separately or in a mixture. The earliest chips where of this kind, used for oligo-fingerprinting.
- Format II: An array of probes is produced either in situ or by attachment. The array is then exposed to sample DNA. Examples are oligo-arrays and cDNA microarrays.

In both cases, the free sequence is fluorescently or radioactively labeled and hybridization is used to determine the identity/abundance of complementary sequences.
19.2 Oligo arrays $C(l)$

The simplest oligo array $C(l)$ consists of all possible oligos of length $l$ and is used e.g. in sequencing by hybridization (SBH).

\[
\begin{array}{cccccccccccccccc}
AA & & & & & & & & & & & & & & \\
AT & & & & & & & & & & & & & & \\
AG & & & & & & & & & & & & & & \\
AC & & & & & & & & & & & & & & \\
TA & & & & & & & & & & & & & & \\
TT & & & & & & & & & & & & & & \\
TG & & & & & & & & & & & & & & \\
TC & & & & & & & & & & & & & & \\
GA & & & & & & & & & & & & & & \\
GT & & & & & & & & & & & & & & \\
GG & & & & & & & & & & & & & & \\
GC & & & & & & & & & & & & & & \\
CA & & & & & & & & & & & & & & \\
CT & & & & & & & & & & & & & & \\
CG & & & & & & & & & & & & & & \\
CC & & & & & & & & & & & & & & \\
\end{array}
\]

Example: oligo at $\Box$: TCGA

19.3 cDNA microarrays

The aim of this technology is to analyze the expression of thousands of genes in a single experiment and provides measurements of the differential expression of these genes.

Here, each spot contains, instead of short oligos, identical cDNA clones, which represents a gene. (Such complementary DNA is obtained by reverse transcription from some known mRNA.) The target is the unknown mRNA extracted from a specific cell. As most of the mRNA in a cell is translated into a protein, the total mRNA in a cell represents the genes expressed in the cell.

Since cDNA clones are much longer than the short oligos otherwise used, a successful hybridization with a clone is an almost certain match. However, because an unknown amount of cDNA is printed at each spot, one cannot directly associate the hybridization level with a transcription level and so cDNA chips are limited to to comparisons of a reference extract and a target extract.

19.4 Affymetrix chips

Affymetrix produces oligo arrays with the goal of capturing each coding region as specifically as possible. The length of the oligos is usually less than 25 bases. The density of oligos on a chip can be very high and a 1cm $\times$ 1cm chip can easily contain 100 000 types of oligos.
The chip contains both “coding” oligos and “control” oligos, the former corresponding to perfect
matches to known targets and the controls corresponding to matches with one perturbed base.
When reading the chip, hybridization levels at controls are subtracted from the level of match probes
to reduce the number of false positives. Actual chip designs use 10 match- and 10 mismatch probes
for each target gene.
Today, Affymetrix offers chips for the entire (known) human or yeast genomes.

19.5 Oligo fingerprinting

Format I chips were the first type used, namely for oligo fingerprinting which is, in a sense, the opposite
to what Affymetrix chips do. Such a chip consists of a matrix of target DNA and is exposed to a
solution containing many identical oligos.
After the positions in the matrix have been recorded at which hybridization of the tagged oligos has
occurred, the chip can be heated to separate the oligos from the target DNA and the experiment can
be repeated with a different type of oligo.
Finally, we obtain a data matrix $M$, with each row representing a specific target DNA from the matrix
and each column representing an oligo probe.
Example: cDNA’s extracted from a tissue. Cluster cDNA’s according to their fingerprints and then
sequence representatives from each cluster to obtain a sequence that identifies the gene.

19.6 Manufacturing oligo arrays

1. Start with a matrix created over a glass substrate.
2. Each cell contains a growing “chain” of nucleotides that ends with a terminator that prevents
   chain extension.
3. Cover the substrate with a mask and then illuminate the uncovered cells, breaking the bonds
   between the chains and their terminators.
4. Expose the substrate to a solution of many copies a specific nucleotide base so that each of the
   unterminated chains is extended by one copy of the nucleotide base and a new terminator.
5. Repeat using different masks.
Exposure to light replaces the terminators by hydrogen bonds (1–2), and (3) bonds forms with nucleotide bases provided in a solution, and then the process is repeated with a different base (4–6).

19.7 Experiment with a DNA chip

Labeled RNA molecules are applied to the probes on the chip, creating a fluorescent spot where hybridization has occurred.

19.8 Functional genomics

With the sequencing of more and more genomes, the question arises of how to make use of this data. One area that is now opening up is functional genomics, the understanding of the functionality of specific genes, their relations to diseases, their associated proteins and their participation in biological processes.
The functional annotation of genes is still at an early stage: e.g., for the plant Arabidopsis (whose sequence was recently completed), the functions of 40% of the genes are currently unknown.

Functional genomics is being addressed using high-throughput methods: global gene expression profiling (“transcriptome analysis”) and wide-scale protein profiling (“proteome analysis”).

19.9 Gene expression

The existing methods for measuring gene expression are based on two biological assumptions:

1. *The transcription level of genes indicates their regulation:* Since a protein is generated from a gene in a number of stages (transcription, splicing, synthesis of protein from mRNA), regulation of gene expression can occur at many points. However, we assume that most regulation is done only during the transcription phase.

2. *Only genes which contribute to organism fitness are expressed,* in other words, genes that are irrelevant to the given cell under the given circumstances etc. are not expressed.

Genes affect the cell by being *expressed,* i.e. transcribed into mRNA and translated into proteins that react with other molecules.

From the pattern of expression we may be able to deduce the function of an unknown gene. This is especially true, if the pattern of expression of the unknown gene is very similar to the pattern of expression of a gene with known function.

Also, the level of expression of a gene in different tissues and at different stages is of significant interest. Hence, it is highly interesting to analyze the expression profile of genes, i.e. in which tissues and at what stages of development they are expressed.

19.10 cDNA Clustering

It is not easy to determine which genes are expressed in each tissue, and at what level:

An average tissue contains more than 10 000 expressed genes, and their expression levels can vary by a factor of 10 000. Hence, we need to extract more than $10^5$ transcripts per tissue. There are about 100 different types of tissue in the body and we are interested in comparing different growth stages, disease stages etc., and so we should analyze more than $10^{10}$ transcripts.

⇒ Sequencing all cDNA’s is infeasible and we need cheap, efficient and large scale methods.

19.11 Representation of gene expression data

Gene expression data is represented by a *raw data matrix* $R$, where each row corresponds to one gene and each column represents one tissue or condition. Thus, $R_{ij}$ is the expression level for gene $i$ in condition $j$. The values can be ratios, absolute values or distributions.
Before it is analyzed, the raw data matrix is preprocessed to compute a similarity or distance matrix.

### 19.12 Clustering

The first step in analyzing gene expression data is clustering.

Clustering methods are used in many fields. The goal in a clustering problem is to group elements (in our case genes) into clusters satisfying:

1. **Homogeneity**: Elements inside a cluster are highly similar to each other.
2. **Separation**: Elements from different clusters have low similarity to each other.

There are two types of clustering methods:

- **Agglomerative** methods build clusters by looking at small groups of elements and performing calculations on them in order to construct larger groups.
- **Divisive** methods analyze large groups of elements in order to divide the data into smaller groups and eventually reach the desired clusters.

Why would we want to cluster gene expression data? We assume that:

- Distinct measurements of same genes cluster together.
- Genes of similar function cluster together.
- Many cluster-function specific insights are gained.

### 19.13 Hierarchical clustering

This approach attempts to place the input elements in a tree hierarchy structure in which distance within the tree reflects element similarity.

To be precise, the hierarchy is represented by a tree and the actual data is represented by the leaves of the tree. The tree can be rooted or not, depending on the method used.
Distance matrix → Dendrogram

<table>
<thead>
<tr>
<th>gene</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

19.14 Average linkage

Average linkage is similar to Neighbor-Joining, except that when computing the new distances of created clusters, the sizes of clusters that are merged are taken into consideration. This algorithm was developed by Lance and Williams (1967) and Sokal and Michener (1958).

1. Input: The distance matrix $D_{ij}$, initial cluster sizes $n_r$.
2. Iteration $k$: The same as in NJ, except that the distance from a new element $t$ is defined by:

   $$D_{it} := D_{ti} := \frac{n_r}{n_r + n_s} D_{ir} + \frac{n_s}{n_r + n_s} D_{is}$$

19.15 Non-Hierarchical clustering

Given a set of input vectors. For a given clustering $P$ of them into $k$ clusters, let $E^P := \sum_c \sum_{v \in c} D(v, z_c)$ denote the solution cost function, where $z_c$ is the centroid (average vector) of the cluster $c$ and $D(v, z_c)$ is the distance from $v$ to $z_c$.

The k-means clustering due to Macqueen (1965) operates as follows:

1. Initialize an arbitrary partition $P$ into $k$ clusters.
2. For each cluster $c$ and element $e$:
   - Let $E^P(c, e)$ be the cost of the solution if $e$ is moved to $c$.
3. Pick $c, e$ so that $E^P(c, e)$ is minimum.
4. Move $e$ to $c$, if it improves $E^P$.
5. Repeat until no further improvement is achieved.

19.16 Application to fibroblast cells

Eisen et al. (1998) performed a series of experiments on real gene expression data. One goal was to check the growth response of starved human fibroblast cells, which were then given serum. The expression level of about $n = 8600$ genes were monitored over $N_{\text{cond}} = 13$ time points.
The original data of test to reference ratios was first log transformed, and then normalized to have mean 0 and variance 1. Let $N_{ij}$ denote these normalized levels. A similarity matrix was constructed from $N_{ij}$ as follows:

$$S_{kl} := \frac{\sum_{j=1}^{n} N_{kj} N_{jl}}{N_{\text{cond}}}$$

where $N_{\text{cond}}$ is the number of conditions checked.

The average linkage method was then used to generate the following tree:

The Dendrogram resulting from the starved human fibroblast cells experiment. Five major clusters can be seen, and many non clustered genes. The cells in the five groups server similar functions: (A) cholesterol bio-synthesis, (B) the cell cycle, (C) the immediate-early response, (D) signaling and angiogenesis, and (E) wound healing and tissue remodeling.

(Color scale red-to-green corresponds to higher-to-lower expression level than in the control state.)

### 19.17 Testing the significance of the clusters

A standard method for testing the significance of clusters is to randomly permute the input data in different ways.

Original expression data (1), clustered data (2), and the results of clustering after random permutations within
19.18 Sequencing by Hybridization (SBH)

Originally, the hope was that one can use DNA chips to sequence large unknown DNA fragments using a large array of short probes:

1. Produce a chip \(C(l)\) spotted with all possible probes of length \(l\) (\(l = 8\) in the first SBH papers),
2. Apply a solution containing many copies of a fluorescently labeled DNA target fragment to the array.
3. The DNA fragments hybridize to those probes that are complementary to substrings of length \(l\) of the fragment
4. Detect probes that hybridize with the DNA fragment and obtain the \(l\)-tuple composition of the DNA fragment
5. Apply a combinatorial algorithm to reconstruct the sequence of the DNA target from the \(l\)-tuple composition

19.19 The Shortest Superstring Problem

SBH provides information of the \(l\)-tuples present in a target DNA sequence, but not their positions. Suppose we are given the spectrum \(S\) of all \(l\)-tuples of a target DNA sequence, how do we construct the sequence?

This is a special case of the Shortest Common Superstring Problem (SCS): A superstring for a given set of strings \(s_1, s_2, \ldots, s_m\) is a string that contains each \(s_i\) as a substring. Given a set of strings, finding the shortest superstring is NP-complete.

Define \(\text{overlap}(s_i, s_j)\) as the length of a maximal prefix of \(s_j\) that matches a suffix of \(s_i\). The SCS problem can be cast as a Traveling Salesman Problem in a complete directed graph \(G\) with \(m\) vertices \(s_1, s_2, \ldots, s_m\) and edges \((s_i, s_j)\) of length \(-\text{overlap}(s_i, s_j)\).

19.20 The SBH graph

SBH corresponds to the special case in which all substrings have the same length \(l\). We say that two SBH probes \(p\) and \(q\) overlap, if the last \(l - 1\) letters of \(p\) coincide with the first \(l - 1\) of \(q\).

Given the spectrum \(S\) of a DNA fragment, construct the directed graph \(H\) with vertex set \(S\) and edge set

\[E = \{ (p, q) \mid p \text{ and } q \text{ overlap} \}\]

There exists a one-to-one correspondence between paths that visit each vertex of \(H\) at least once and the DNA fragments with the spectrum \(S\).
19.21 Example of the SBH graph

Vertices: $l$ tuples of the spectrum $S$, edges: overlapping $l$-tuples:

$S = \{ \text{ATG, AGG, TGC, TCC, GTC, GGT, GCA, CAG} \}$

The path visiting all vertices corresponds to the sequence reconstruction $\text{ATGCAGGTCC}$.

A path that visits all nodes of a graph exactly once is called a Hamiltonian path. Unfortunately, the Hamiltonian Path Problem is NP-complete, so for larger graphs we cannot hope to find such paths.

19.22 Second example of the SBH graph

$S = \{ \text{AT, TGG, TGC, GTG, GGC, GCA, GCG, CGT} \}$

This example has two different Hamiltonian paths and thus two different reconstructed sequences:

$\text{ATGCCTGGCA, ATGGCGTGCA}$

19.23 Euler Path

Leonard Euler wanted to know whether there exists a path that uses all seven bridges in Königsberg exactly once:
Let $S$ be the spectrum of a DNA fragment. We define a graph $G$ whose set of nodes consists of all possible $(l - 1)$-tuples.

We connect one $l - 1$-tuple $v = v_1 \ldots v_{l-1}$ to another $w = w_1 \ldots w_{l-1}$ by a directed edge $(v, w)$, if the spectrum $S$ contains an $l$-tuple $u$ with prefix $v$ and suffix $w$, i.e. such that $u = v_1 \ldots v_{l-1}w_1 = v_{l-1}w_1 \ldots w_{l-1}$.

Hence, in this graph the probes correspond to edges and the problem is to find a path that visits all edges exactly once, i.e., an Eulerian path.

Finding all Eulerian paths is simple to solve.

(To be precise, the Chinese Postman Problem (visit all edges at least once in a minimal tour) can be efficiently solved for directed or undirected graphs, but not in a graph that contains both directed and undirected edges.)

$$S = \{ \text{ATG, TGG, TGC, GTG, GGC, GCA, GCG, CGT} \}$$

Vertices represent $(l - 1)$-tuples, edges correspond to $l$-tuples of the spectrum. There are two different solutions:
19.25 Eulerian graphs

A directed graph $G$ is called Eulerian, if it contains a cycle that traverses every edge of $G$ exactly once.

A vertex $v$ is called balanced, if the number of edges entering $v$ equals the number of edges leaving $v$, i.e. $\text{indegree}(v) = \text{outdegree}(v)$. We call $v$ semi-balanced, if $|\text{indegree}(v) - \text{outdegree}(v)| = 1$.

**Theorem** A directed graph is Eulerian, if and only if it is connected and each of its vertices is balanced.

**Lemma** A connected directed graph is Eulerian, if and only if it contains at most two semi-balanced nodes.

19.26 When does a unique solution exist?

**Problem:** Given a spectrum $S$. Does it possess a unique sequence reconstruction?

Consider the corresponding graph $G$. If the graph $G$ is Eulerian, then we can decompose it into simple cycles $C_1, \ldots, C_t$, that is, cycles without self-intersections. Each edge of $G$ is used in exactly one cycle, although nodes can be used in many cycles. Define the intersection graph $G_I$ on $t$ vertices $C_1, \ldots, C_t$, where $C_i$ and $C_j$ are connected by $k$ edges, if and only if they have precisely $k$ original vertices in common.

**Lemma** Assume $G$ is Eulerian. Then $G$ has only one Eulerian cycle if and only if the intersection graph $G_I$ is a tree.

19.27 Probability of unique sequence reconstruction

What is the probability that a randomly generated DNA fragment of $n$ can be uniquely reconstructed using a DNA array $C(l)$? In other words, how large must $l$ be so that a random sequence of length $n$ can be uniquely reconstructed from its $l$-tuples?

We assume that the bases at each position are chosen independently, each with probability $p = \frac{1}{4}$.

Note that a repeat of length $\geq l$ will always lead to a non-unique reconstruction. We expect about $\binom{n}{l}p^l$ repeats of length $\geq l$. Note that $\binom{n}{l}p^l = 1$ implies $l = \log_4 \binom{n}{l}$.

Thus for a given $l$ one should choose $n \approx \sqrt{2 \cdot 4^l}$, but not larger. (However, this is a very loose bound and a much tighter bound is known.)
19.28 SBH currently infeasible

The Eulerian path approach to SBH is currently infeasible due to two problems:

- Errors in the data
  - False positives arise, when the target DNA hybridizes to a probe even though an exact match is not present
  - False negatives arise, when an exact match goes undetected

- Repeats make the reconstruction impossible, as soon as the length of the repeated sequence is longer than the word length $l$

Nevertheless, ideas developed here are employed in a new approach to sequence assembly that uses sequenced reads and a Eulerian path representation of the data (Pavel Pevzner, Recomb'2001).

19.29 Masks for VLSIPS

DNA arrays can be manufactured using VLSIPS, very large scale immobilized polymer synthesis. In VLSIPS, probes are grown one layer of nucleotides at a time through a photolithographic process. In each step, a different mask is used and only the unmasked probes are extended by the current nucleotide. All probes are grown to length $l$ in $4l$ steps.

Problem: Due to diffraction, internal reflection and scattering, masked spots near an edge of the mask can be unintentionally illuminated.

Idea: To minimize the problem, design masks that have minimal border length!

For example, consider the $8 \times 8$ array for $l = 3$. Both of the following two masks add a $T$ to $\frac{1}{4}$ of the spots, with borders of very different length:

<table>
<thead>
<tr>
<th>Border length 58</th>
<th>Border length 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Diagram of border length 58 mask]</td>
<td>![Diagram of border length 16 mask]</td>
</tr>
</tbody>
</table>
19.30 The \( l \)-bit Gray code

In the above example, we can mask \( \frac{1}{4} \) of all spots using a mask that has a boundary of length \( 4 \cdot l \). Can we arrange the spots so that this minimal value is attained for every mask?

An \( l \)-bit Gray code is a permutation of the binary numbers \( 0,\ldots,2^{l}-1 \) such that any two neighboring numbers differ in exactly one bit.

The 4-bit “reflected” Gray code is:

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
\end{array}
\]

This is generated recursively from the 1-bit code \( G_1 = \{0, 1\} \) using:

\[
G_l = \{g_1, g_2, \ldots, g_{2^{l-1}}, g_{2^l}\} \rightarrow \quad G_{l+1} = \{0g_1, 0g_2, \ldots, 0g_{2^{l-1}}, 0g_{2^l}, 1g_2, 1g_{2^{l-1}}, \ldots, 1g_2, 1g_1\}.
\]

19.31 The two-dimensional Gray code

We want to construct a two-dimensional Gray code for strings of length \( l \) over the alphabet \( \{A, C, G, T\} \), in which every \( l \)-tuple is present and differs from each of its four neighbors in precisely one position.

Start: \( G_1 := \begin{bmatrix} A \\ C \\ G \end{bmatrix} \).

The induction step \( G_l \rightarrow G_{l+1} \):

Let \( G_l = \begin{bmatrix} g_{1,1} & \cdots & g_{1,2^l} \\ \cdots & \cdots & \cdots \\ g_{2^l,1} & \cdots & g_{2^l,2^l} \end{bmatrix} \) and set

\[
G_{l+1} := \begin{bmatrix}
A_{g_{1,1}} & \cdots & A_{g_{1,2^l}} & T_{g_{1,2^l}} & \cdots & T_{g_{1,1}} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
A_{g_{2^l,1}} & \cdots & A_{g_{2^l,2^l}} & T_{g_{2^l,2^l}} & \cdots & T_{g_{2^l,1}} \\
G_{g_{2^l,1}} & \cdots & G_{g_{2^l,2^l}} & C_{g_{2^l,2^l}} & \cdots & C_{g_{2^l,1}} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
G_{g_{1,1}} & \cdots & G_{g_{1,2^l}} & C_{g_{1,2^l}} & \cdots & C_{g_{1,1}}
\end{bmatrix}.
\]

19.32 Additional ideas

**SBH with universal bases** Use universal bases such as inosine that stack correctly, but don’t bind, and thus play the role of “don’t care” symbols in the probes. Arrays based on this idea can
be achieve the information-theoretic lower bound of the number of probes required for unambiguous reconstruction of an arbitrary DNA string of length $n$. (The full $C(l)$ array has redundancies that can be eliminated using such universal bases.) (Preparata et al. 1999)

**Adaptive SBH** If a sequencing by hybridization is not successful, analyze the critical problems and then build a new array to overcome them. Skiena and Sundaram (1993) gives theoretical bounds for the number of rounds needed for sequence reconstruction (in the error free case).

**SBH-style shotgun sequencing** The idea is to collect sequence reads from the target DNA sequence using traditional sequencing methods and then to treat each such read of length $k$ as a set of $k - l + 1$ individual $l$-tuples, with $l = 30$, say. Then, the Eulerian path method is used. Idury and Waterman suggested this in 1995 and it leads to an efficient assembly algorithm in the error-free case. More recent work by Pevzner and others (2001) has led to promising software.

**Fidelity probes for DNA arrays** As a quality control measure when manufacturing a DNA chip, one can produce fidelity probes that have the same sequence as probes on the chip, but are produced in a different order of steps. A known target is hybridized to these probes and the result reflects the quality of the array. Hubbel and Pevzner (1999) describe a combinatorial method for designing a small set of fidelity probes that can detect variations and can also indicate which manufacturing steps caused the errors.
20 Peptide Sequencing using MS and SCOPE (by Knut Reinert)

This is based on the following sources, which are all recommended reading:


Recall that a peptide chain consists of different amino acids joined by peptide bonds. Amino-acids are distinguished from each other by the secondary structure of the side chain R.

20 Tandem MS

In tandem mass spectrometry (MS/MS), MS is first used to select peptides of a certain $m/z$ value. The ionized peptides are then broken into fragments by collision-induced dissociation (CID). Those fragments that retain an ionizing charge after CID are subjected to a second MS step that gives an estimate of their new mass-to-charge ratio $m/z$.

Fragmentation by CID typically breaks a peptide-bond in the peptide chain and so the resulting spectrum implicitly contains information about the constituent amino-acids of the peptide.

20.1 Ion types

The fragmentation of the peptide in CID is a stochastic process governed by the physical-chemical properties of the peptide and the energy of collision.
The charged fragment can be characterized by the position of the broken bond and the side retaining the charge. Here we show the N-terminal $a_1, b_1, c_1$ fragments, and the C-terminal $x_{n-1}, y_{n-1}, z_{n-1}$ fragments:

While $b$ and $y$ correspond to the most frequently occurring fragments, a high energy collision often results in other fragments, including internal fragments formed by breakage at two points, and fragments formed by breaks in side-chains.

The above pictures show some $a, b, c, x, y, z$ ions as well as an internal ion.

Immonium ions appear in the very low m/z range of the MS-MS spectrum. Each amino acid residue leads to a diagnostic immonium ion, with the exception of the two pairs leucine (L) and iso-leucine (I), and lysine (K) and glutamine (Q), which produce immonium ions with the same m/z ratio, i.e. m/z 86 for I and L, m/z 101 for K and Q.

The immonium ions are useful for detecting and confirming many of the amino acid residues in a peptide, although no information regarding the position of these amino acid residues in the peptide sequence can be ascertained from the immonium ions.
20.2 MS/MS spectrum

If we measure the *MS/MS spectrum* of a set of peptide fragments and if we can identify the correct ions, then we can determine the respective amino acids.

A cartoon MS/MS spectrum for the peptide SGFLEEDK is shown below. The N-terminal \( b_1 \) ion has the mass of serine (87) plus one Dalton for the terminating H. The \( y_1 \) ion the mass of lysine (128) plus 19 for OHHH.

![MS/MS spectrum diagram]

Other possible fragments, the presence of multiply charged ions, the absence of some ions in a series, and finally noise and measurement error pose a real challenge for the identification algorithms.

Most algorithms for analyzing MS/MS data address the following three problems.

20.3 Problems addressed by MS/MS algorithms

**Interpretation:** The *input* is a *MS/MS spectrum*, the *output* is *interpreted-MS/MS-data*. Interpreted-MS/MS-data may include parent peptide mass, partial or complete sequence tags, and combinations of sequence tags and molecular masses.

**Filtering:** The *input* is *interpreted-MS/MS-data* and a peptide sequence database. The *output* is a list of *candidate-peptides* that might have generated the MS/MS spectrum.

**Scoring:** The *input* is a list of *candidate-peptides* and the *MS/MS spectrum*. The *output* is a ranking of the *candidate-peptides* along with a score and possibly a *p*-value (probability that the score was achieved by random chance).

We focus on the problem of scoring of peptides against theoretical spectra. That means we want to answer the question:

Given a peptide \( p \), how likely is it that it generated the observed spectrum?

To solve this, the SCOPE program, developed by Bafna and Edwards, addresses the following points:

- it explicitly models fragmentation depending on the peptide and experimental setting,
• it explicitly models measurement error, and
• it models noise peaks.

The SCOPE algorithm models the process of MS/MS spectrum generation by a two-step stochastic process.

1. The first step involves generation of fragments from a peptide, according to a probability distribution estimated from many training samples.
2. The second step involves the generation of a spectrum from the fragments according to the distribution of the instrument measurement error.

20.4 Definitions

We introduce some terminology.

• An MS/MS spectrum \( S \in \mathbb{R}^k_+ \) is a list of \( k \) positive real numbers specifying the \( k \) observed mass/charge ratios of the spectral peaks.

• A peptide \( p \in \Sigma^n \) is a sequence of \( n \) amino-acid residues from the alphabet of amino-acid symbols, \( \Sigma = \{A,C,\ldots,Y\} \).

• A fragment space is an enumeration \( \mathcal{F}(p) \) of all fragment mass/charge ratios that a peptide \( p \) might produce. Each element of \( \mathcal{F}(p) \), then, is a fragment-charge pair. Thus,

\[
\mathcal{F}(p) = \{(a_1, i), (b_1, i), (c_1, i), \ldots, (x_i, i), (y_i, i), (z_i, i), i = 1, 2, 3, \ldots\}
\]

• Let \((m/z)(f)\) denote the mass/charge ratio of a fragment \( f \in \mathcal{F}(p)\).

• The fragmentation space \( \phi(p) \) of a peptide \( p \) is the set of all fragmentation patterns of \( p \). That is,

\[
\phi(p) = \{F : F \subseteq \mathcal{F}(p)\}
\]

• A noise peak is any peak in the spectrum \( S \) for which \( \mathcal{F}(p) \) provides no explanation.

20.5 Fragmentation and measurement

**Fragmentation:** Each of the many copies of a peptide \( p \) that pass into the collision chamber is randomly broken into fragments.

The experimental conditions and the physical and chemical properties of the peptide determine the probability of observing a given fragment \( f \). In addition to the fragments modeled in \( \mathcal{F}(p) \), unexpected fragments or contaminant fragments might be observed.
We model this by randomly choosing a fragmentation pattern $F$ from $\phi(p)$. Noise peaks will be modeled later.

**Measurement:** Each fragment with a particular mass/charge ratio generates a mass/charge ratio observation close to, but not precisely at its true mass/charge ratio.

The observation of many fragments with the same mass/charge ratio leads to the formation of a distinctive peak close to the true mass/charge ratio of these fragments.

The observed peak can then be represented by a single real number, an estimate of the true mass/charge ratio of the fragments that generated it.

The deviation of a measured mass/charge ratio from its true value is modeled according to a probability distribution, typically the normal distribution.

### 20.6 Determining the most likely peptide

Let $\psi(S \mid p)$ denote the probability density function for the random vector $S$ representing the MS/MS spectrum, given peptide $p$. To identify the sample peptide, we search a database for the peptide $p^*$ that satisfies

$$p^* = \arg \max_p \psi(S \mid p)$$

A formal description of the two-step model of fragmentation followed by measurement is given by:

$$\psi(S \mid p) = \sum_{F \in \phi(p)} \psi(S \mid F, p) \Pr(F \mid p)$$

The quantity $\Pr(F \mid p)$ represents the probability of a particular fragmentation pattern of a peptide. It is in the computation of $\Pr(F \mid p)$ that the complex process of fragmentation can be modeled.

### 20.7 Fragmentation probability estimation

The SCOPE algorithm does not explicitly implement an automatic algorithm to estimate the probabilities $\Pr(F \mid p)$ but relies on the judgment of the user.

For example, experienced operators know that the presence of acidic amino-acids in a peptide makes the “neutral-water-loss-ion-type” cleavages much more likely.

In principle, these probabilities could also be learned from sample spectra of known peptides given a specific experimental setup.

### 20.8 Computing $\psi(S \mid F, p)$

The function $\psi(S \mid F, p)$ describes the probability of observing a collection of spectral peaks, given a particular fragmentation pattern of a peptide $p$.

Unfortunately, it is not obvious which fragment(s) are responsible for which peak(s), and which peaks should be considered noise.
In order to compute $\psi(S \mid F, p)$, we need to either sum over all the possible explanations of each peak, which is not feasible, or use our understanding of the mass spectrometer to limit the number of terms.

We make the following simplifying assumptions:

1. Each unique mass/charge ratio in the fragment space generates at most one spectral peak.
2. Each spectral peak is the observed mass/charge ratio of at most one of the (unique) mass/charge ratios in the fragment space.
3. The assignment of spectral peaks to fragments must be non-crossing: For all fragments $f_1, f_2$ and spectral peaks $S_1, S_2$, if

$$\frac{m}{z}(f_1) < \frac{m}{z}(f_2) \text{ and } S_1 < S_2,$$

then peak $S_1$ must have been generated by fragment $f_1$ and peak $S_2$ must have been generated by fragment $f_2$.

In addition, we augment the fragment space $\mathcal{F}(p)$ with noise fragments, one for each spectral peak. Each noise fragment has the same mass/charge ratio as its spectral peak. We denote this augmented fragment space by $\mathcal{F}'(p)$ and the corresponding fragmentation space by $\phi'(p)$.

Due to the addition of noise fragments all spectral peaks must either be assigned to a unique fragment from our original fragment space $\mathcal{F}(p)$ or to a noise fragment. Therefore we can make the following Observation: Only fragmentation patterns $F \subseteq \mathcal{F}'(p)$ with $|F| = k$ have non-zero probability mass. However, we can say something even stronger.

Let $S_i \overset{M}{=} f$ denote the event that peak $S_i$ is generated by fragment $f$, and $S = (S_1, S_2, \ldots, S_k)$ be a tandem MS spectrum ordered by mass/charge ratio. Further, let $F \subseteq \mathcal{F}'(p), |F| = k$ be an arbitrary fragmentation pattern, whose observed fragments $f_1, f_2, \ldots, f_k \in F$ are ordered by mass/charge ratio. Then only one assignment of spectral peaks to fragments has non-zero probability mass. All of the probability mass for $\psi(S \mid F, p)$ is captured by this unique non-crossing assignment. We write:

$$\psi(S \mid F, p) = \psi(S \mid \bigcap_{i=1}^{k} [S_i \overset{M}{=} f_i], F, p)$$

In isolation, the distribution of one measured mass/charge ratio about its true value is independent of any other measured mass/charge ratio about its true value.

We model the distribution of the measured mass/charge ratios as normal distributions centered at the fragment mass/charge ratio and the distribution of the measured mass/charge ratio of noise fragments by an impulse function at the mass/charge ratio of its spectral peak.
We expand the expression \( \psi(S \mid \bigcap_{i=1}^{k}[S_i = f_i], F, p) \) into its components in order to compute it:

\[
\psi(S \mid \bigcap_{i=1}^{k}[S_i = f_i], F, p) = \psi(S_1 \mid \bigcap_{i=1}^{k}[S_i = f_i], F, p) \times \\
\prod_{j=2}^{k} \psi(S_j \mid S_1, \ldots, S_{j-1}, \bigcap_{i=1}^{k}[S_i = f_i], F, p)
\]

In the last term, \( S_j \) is dependent on the previous \( S_i \) for \( i < j \). To simplify this we truncate the left-hand tail of the measurement distribution and rescale its total probability density to one.

Let \( \rho_f \) be the distribution of the observed peak about the true mass/charge ratio of fragment \( f \). Then

\[
\psi(S_1 \mid [S_1 = f_1], F, p) = \rho_{f_1}(S_1) \]

\[
\psi(S_j \mid S_{j-1}, [S_j = f_j], F, p) = \begin{cases} \\
\frac{\rho_{f_j}(S_j)}{\int_{S > S_{j-1}} \rho_{f_j}(S)}, & S_j > S_{j-1}; \\
0, & \text{otherwise}.
\end{cases}
\]

20.9 Computing \( \psi(S \mid p) \)

We now show how this choice of \( \psi \) allows an efficient algorithm for computing \( \psi(S \mid p) \).

We want to avoid the computation of an exponential number of terms in the expression \( \psi(S \mid p) = \sum_{F \subseteq F(p)} \psi(S \mid F, p) \Pr(F \mid p) \). To do this we need another assumption, namely that the probability of observing \( f \) must be independent of the observation of other fragments. This is not always true but allows us to compute \( \psi(S \mid p) \) efficiently with dynamic programming.

Given the spectrum \( S = (S_1, \ldots, S_k) \) and the fragments \( F'(p) = \{ f_1, \ldots, f_m \} \) ordered by mass/charge ratio, we define \( F'_j(p) = \{ f_1, \ldots, f_j \} \) to be the first \( j \) fragments of \( F'(p) \).

Then the dynamic programming recurrence function \( \Phi(i, j) \) represents the probability mass associated with the event that the first \( i \) peaks were generated by \( i \) fragments from the first \( j \) fragments of \( F'(p) \).
Clearly, $\Phi(k, m) = \psi(S \mid p)$ is the value we are interested in. The following recurrence holds:

$$\Phi(i, j) = \begin{cases} 
1, & \text{if } i = 0, \\
0, & \text{if } i > j, \\
\Phi(i - 1, j - 1) \\
\times \psi(S_i \mid S_{i-1}, S_i = f_j) \\
\times \Pr(f_j \mid p) \\
+ \Phi(i, j - 1) \Pr(f_j \mid p), 
\end{cases}$$

The above recursion corresponds to a special sequence alignment problem. We align the spectrum with all possible fragments in the augmented fragment space. The first term in the sum is for the case that the $j$-th fragment is assigned to a spectral peak. The probability of that is the probability of assigning the first $i - 1$ spectral peaks to $i - 1$ fragments among the first $j - 1$ fragments times the probability that $S_i$ is assigned $f_j$ times the probability of $f_j$ given $p$.

The second term in the sum is the probability that $f_j$ is not assigned to any peak in $S$. This might be large if we do not expect a fragment to occur!

The most likely assignment $F^*$ is given by

$$F^* = \arg \max_{F \subseteq F(p)} \psi(S \mid F, p) \Pr(F \mid p).$$

### 20.10 Summary

Using simplifying assumptions, one can compute $\psi(S \mid p)$ by dynamic programming and thus can score how well any given peptide $p$ explains a measured MS/MS spectrum $S$, thus allowing the identification of a most likely peptide sequence.