Foundations of Sequence Analysis

Lecture notes for a course in the Winter Semester 2000/2001

Stefan Kurtz

July 18, 2002

Contents

CHAPTER 1

Overview

1.1 Application Areas

Sequences or equivalently texts, strings, or words are a natural way to represent information. We give a short list of areas where sequences to be analyzed come from:

- molecular biology

DNA	$\ldots a\ a\ c\ g\ a\ c\ g\ t \ldots$	4 nucleotides,	length: $\approx 10^3 - 10^9$
RNA	$\ldots a\ u\ c\ g\ g\ c\ u\ t \ldots$	4 nucleotides,	length: $\approx 10^2 - 10^3$
proteins	$\ldots L\ I\ S\ A\ I\ S\ T\ L\ I\ E\ B \ldots$	20 aminoacids,	length: $\approx 10^2 - 10^3$

L = Leucin
I = Isoleucin
S = Serin
A = Alanin etc.

- phonetic spelling:

english	40 phoneme
japanese	113 “morae” (syllables)

- spoken language, birdsong: discretized measurements, multidimensional (frequency, energy) on a dynamic time scale
- graphics: (r, g, b) vectors with $r, g, b \in [0, 255]$ for the intensity of the red, green, and blue color of a pixel.
- text processing: sequences in ASCII format
  (comparison of files, search in index, spelling correction)
- information transmission: bitsequences, blockcodes in noisy channels
  substitution/synchronisation errors — decoding and correction

Usually sequences encoding experimental or natural information are almost always inexact. Thus similar sequences have in a lot of cases the same or similar meaning or effect. Thus a main part of this lecture will be devoted to notions of similarity, and we will show how to handle these notions algorithmically.

1.2 Problems on Strings

Here is a short list of problems occurring on strings:

1. sequence comparison: compare two sequences and show the similarities and differences.
2. string matching: find all positions in a text where a pattern string occurs.
3. regular expression matching: find all positions in a text where a regular expression matches.
4. multiple string matching: find all positions in a text where one of the strings in a given set matches.
5. approximate string matching: find all positions in a text where a string matches, allowing for errors in the match.
6. dictionary matching: for a given word w find the word v in a given set of words with maximal similarity to w.
7. text compression: find long duplicated substrings in a given text.
8. text compression: sort all suffixes of a given string lexicographically.
9. structural pattern matching: find regularities in sequences, like repeats, tandems ww, palindromes, or unique subsequences.

1.3 Topics

Section 2 introduces the datatype "sequence". Section 3 considers different notions of similarity (edit distance, maximal matches distance, q-gram-distance) and methods to compute the similarity of two sequences according to these notions. Section 4 will introduce an index structure, called suffix tree, which stores all substrings of a given string very efficiently. We consider how to build a suffix tree and show several applications. Section 5 will be devoted to approximative string matching, i.e. finding occurrences of patterns, allowing for errors.

CHAPTER 2

Basic Notions and Definitions

Let S be a set. $|S|$ denotes the number of elements in S and $\mathcal{P}(S)$ refers to the set of subsets of S.

$\mathbb{N}$ denotes the set of positive integers including 0. $\mathbb{R}^+$ denotes the set of positive reals including 0. The symbols $h, i, j, k, l, m, n, q, r$ refer to integers if not stated otherwise. $|i|$ is the absolute value of i and $i \cdot j$ denotes the product of i and j.

Let $\mathcal{A}$ be a finite set, the *alphabet*. The elements of $\mathcal{A}$ are *characters*. Strings are written by juxtaposition of characters. In particular, ε denotes the *empty string*. The set $\mathcal{A}^*$ of *strings over* $\mathcal{A}$ is defined by

$$\mathcal{A}^* = \bigcup_{i \geq 0} \mathcal{A}^i$$

where $\mathcal{A}^0 = \{\varepsilon\}$ and $\mathcal{A}^{i+1} = \{aw \mid a \in \mathcal{A}, w \in \mathcal{A}^i\}$. $\mathcal{A}^+$ denotes $\mathcal{A}^* \setminus \{\varepsilon\}$. The symbols a, b, c, d refer to characters and $p, s, t, u, v, w, x, y, z$ to strings, unless stated otherwise.

Example 1

1. ASCII: 8-bit characters, encoding as defined by the ASCII standard
2. $\{A, \ldots, Z, a, \ldots, z, 0, \ldots, 9, \}$: alphanumeric subset of the ASCII-set
3. $\{A, \ldots, Z\} \setminus \{B, J, O, U, X, Z\}$: letter code for 20 aminoacids
4. $\{a, c, g, t\}$: DNA-alphabet (Adenin, Cytosin, Guanin, Thymin)
5. $\{R, Y\}$: purine/pyrimidine-alphabet
6. $\{I, O\}$: hydrophile/hydrophobe nucleotides/aminoacids
7. $\{+, -\}$: positive/negative electrical charge □

These examples show that the size of the alphabets can be quite different. The alphabet size is an important parameter when determining the efficiency of several algorithms.

The *length* of a string s, denoted by $|s|$, is the number of characters in s. We make no distinction between a character and a string of length one. If $s = uvw$ for some (possibly empty) strings u, v and w, then

- u is a *prefix* of s,
- v is a *substring* of s, and
- w is a *suffix* of s.

A prefix or suffix of s is *proper* if it is different from s. A suffix of s is *nested* if it occurs more than once in s. A set S of strings is *prefix-closed* if $u \in S$ whenever $ua \in S$. A set S of strings is *suffix-closed* if $u \in S$ whenever $au \in S$. A substring v of s is *right-branching* if there are different characters a and b such that va and vb are substrings of s. Let $q > 0$. A *q-gram* of s is a substring of s of length q. q-grams are sometimes called q-tuples.

s_i is the i-th character of s. That is, if $|s| = n$, then $s = s_1 \dots s_n$ where $s_i \in \mathcal{A}$. $s_n \dots s_1$, denoted by s^{-1}, is the *reverse* of $s = s_1 \dots s_n$. If $i \leq j$, then $s_i \dots s_j$ is the substring of s beginning with the i-th character and ending with the j-th character. If $i > j$, then $s_i \dots s_j$ is the empty string. A string w begins at position i and ends at position j in s if $s_i \dots s_j = w$.

CHAPTER 3

String Comparisons

The comparison of strings is an important operation applied in several fields, such as molecular biology, speech recognition, computer science, and coding theory. The most important model for string comparison is the model of edit distance. It measures the distance between strings in terms of edit operations, that is, deletions, insertions, and replacements of single characters. Two strings are compared by determining a sequence of edit operations that converts one string into the other and minimizes the sum of the operations' costs. Such a sequence can be computed in time proportional to the product of the lengths of the two strings, using the technique of dynamic programming.

The edit distance is a measure of local similarities in which matches between substrings are highly dependent on their relative positions in the strings. There are situations where this property is not desired. Suppose one wants to consider strings as similar which differ only by an exchange of large substrings. This occurs, for instance, if a text file has been created from another by a block move operation. In such a case, the edit distance model should not be used since it gives a large edit distance. There are two other string comparison models that are more appropriate for this case: The maximal matches model and the q-gram model.

The idea of the maximal matches model is to count the minimal number of occurrences of characters in one string such that if these characters are "crossed out", the remaining substrings are all substrings of the other string. Thus, strings with long common substrings have a small distance. The idea of the q-gram model is to count the number of occurrences of different q-grams in the two strings. Thus, strings with many common q-grams have a small distance. A very interesting aspect is that the maximal matches distance and the q-gram distance of two strings can be computed in time proportional to the sum of the lengths of the two strings.

When comparing biological sequences, the edit distance computation is often

to expensive, while the order of the sequence characters is still important. Therefore heuristics have been developed, which approximate the edit distance model. Two of these heuristics are described in Sections 3.7 and 3.8.

In the following, we first consider the issue of string comparison in general. Then we describe the three models of string comparison in details and give algorithm to compute the respective distances. For the rest of this section let u and v be strings of length m and n, respectively.

3.1 The Problem

The trivial method to compare two sequences is to compare them character by character: u and v are equal if and only if $|u| = |v|$ and $u_i = v_i$ for $i \in [1, n]$. However, this comparison model is too restrictive for several problems:

- searching for a name of which the spelling is not exactly known
- finding diffracted forms of a word
- accounting for typing errors
- tolerating error prone experimental measurements
- allowing for ambiguities in the genetic code, e.g. *gcu*, *gcc*, *gca*, and *gcg* all code for alanin.
- searching for a protein with unknown function, a "similar" protein sequence, whose biological function is known.

To be more general one has to define a function $f : \mathcal{A}^* \times \mathcal{A}^* \to \mathbb{R}$, which delivers a qualitative measure of distance/similarity. Note that there is a duality in the notions "distance" and "similarity": the smaller the distance, the larger the similarity.

Let M be a set and $f : M \times M \to \mathbb{R}^+$ be a function. f is a *metric* on M if for all $x, y, z \in M$ the following properties hold:

Zero Property	$f(x,y) = 0 \iff x = y$
Symmetry	$f(x,y) = f(y,x)$
Triangle Inequality	$f(x,y) \leq f(x,z) + f(z,y).$

If the symmetry and the triangle inequality hold, and also

$$x = y \Rightarrow f(x,y) = 0$$

then f is a *pseudo-metric* on M.

Example 2 Let $\mathcal{A}$ be a finite subset of $\mathbb{N}$. Suppose $n > 0$ and $M = \mathcal{A}^n$. Then we define the following distance notions:

$$\text{euclidian distance:} \qquad f(u,v) = \sqrt{\sum_{i=1}^{n} (u_i - v_i)^2}$$

$$\text{block distance:} \qquad f(u,v) = \sum_{i=1}^{n} |u_i - v_i|$$

$$\text{hamming distance:} \qquad f(u,v) = |\{i \mid 1 \leq i \leq n, u_i \neq v_i\}| \quad \square$$

These distance notions only make sense for sequences of the same length. The distance notions we consider now are also defined for sequences of different length.

3.2 The Edit Distance Model

The notion of edit operations is the key to the edit distance model.

Definition 1 An *edit operation* is a pair $(\alpha, \beta) \in (\mathcal{A}^1 \cup \{\varepsilon\}) \times (\mathcal{A}^1 \cup \{\varepsilon\}) \setminus \{(\varepsilon, \varepsilon)\}$. $\square$

α and β denote *strings* of length ≤ 1. However, if $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$, then the edit operation (α, β) is identified with a pair of characters.

An edit operation (α, β) is usually written as $\alpha \to \beta$. This reflects the operational view which considers edit operations as rewrite rules transforming a source string into a target string, step by step. In particular, there are three kinds of edit operations:

- $\alpha \to \varepsilon$ denotes the *deletion* of the character α,
- $\varepsilon \to \beta$ denotes the *insertion* of the character β,
- $\alpha \to \beta$ denotes the *replacement* of the character α by the character β.

Notice that $\varepsilon \to \varepsilon$ is not an edit operation. Insertions and deletions are sometimes referred to collectively as *indels*.

Sometimes string comparison just means to measure how different strings are. Often it is additionally of interest to analyze the total difference between two strings into a collection of individual elementary differences. The most important mode of such analyses is an alignment of the strings.

Definition 2 An *alignment* A of u and v is a sequence

$$(\alpha_1 \to \beta_1, \ldots, \alpha_h \to \beta_h)$$

of edit operations such that $u = \alpha_1 \ldots \alpha_h$ and $v = \beta_1 \ldots \beta_h$. $\square$

Note that the unique alignment of ε and ε is the empty alignment, that is, the empty sequence of edit operations. An alignment is usually written by placing the characters of the two aligned strings on different lines, with inserted dashes denoting ε. In such a representation, every column represents an edit operation.

Example 3 The alignment $A = (\varepsilon \to d, b \to b, c \to a, \varepsilon \to d, a \to a, c \to \varepsilon, d \to d)$ of the sequences *bcacd* and *dbadad* is written as follows:

$$\begin{pmatrix} - & b & c & - & a & c & d \\ d & b & a & d & a & - & d \end{pmatrix}$$

$\square$

Example 4 Five alignments of $u = gabh$ and $v = gcdhb$

$$L_1 = \begin{pmatrix} g & - & a & b & - & h & - \\ g & c & - & - & d & h & b \end{pmatrix} \quad L_2 = \begin{pmatrix} g & - & a & - & b & h & - \\ g & c & - & d & - & h & b \end{pmatrix} \quad L_3 = \begin{pmatrix} g & - & - & a & b & h & - \\ g & c & d & - & - & h & b \end{pmatrix}$$

$$L_4 = \begin{pmatrix} g & a & - & - & b & h \\ g & c & d & h & b & - \end{pmatrix} \qquad L_5 = \begin{pmatrix} g & a & b & h & - \\ g & c & d & h & b \end{pmatrix} \quad \square$$

Observation 1 Let $A = (\alpha_1 \to \beta_1, \ldots, \alpha_h \to \beta_h)$ be an alignment of u and v. Then $m + n \geq h \geq \max\{m, n\}$.

Proof:

1. The alignment
$$\begin{pmatrix} u_1 & u_2 & \ldots & u_m & - & - & \ldots & - \\ - & - & \ldots & - & v_1 & v_2 & \ldots & v_n \end{pmatrix}$$
of u and v is of maximal length. Its length is $m + n$. Hence $m + n \geq h$.

2. Let $m \geq n$. Then
$$\begin{pmatrix} u_1 & u_2 & \ldots & u_n & u_{n+1} & \ldots & u_m \\ v_1 & v_2 & \ldots & v_n & - & \ldots & - \end{pmatrix}$$
is an alignment of u and v of minimal length. Hence $h \geq m = \max\{m, n\}$.

3. The case $m < n$ is similar to case 2. $\square$

3.2.1 The Number of Alignments

For all $i, j \geq 0$ let $Aligns(i, j)$ be the number of alignments of two fixed sequences of length i and j. The following holds:

$$\begin{array}{lcl} Aligns(0,0) & = & 1 \\ Aligns(m+1,0) & = & 1 \\ Aligns(0,n+1) & = & 1 \\ Aligns(m+1,n+1) & = & Aligns(m,n+1) + Aligns(m+1,n) + Aligns(m,n) \end{array}$$

$Aligns(n, n)$ can be approximated by the Stanton-Cowan-Numbers:

$$Aligns(n,n) \approx \left(1+\sqrt{2}\right)^{2n+1} \cdot \sqrt{n}$$

For $n = 1000$ we have $Aligns(n, n) \approx \left(1 + \sqrt{2}\right)^{2001} \cdot \sqrt{1000} = 10^{767.4\ldots}$

The order of insertions and deletions immediately following each other is not important. For example, the two alignments

$$\begin{pmatrix} a & - \\ - & b \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} - & a \\ b & - \end{pmatrix} \tag{3.1}$$

should be considered equivalent. This results in the notion of subsequences:

Definition 3 A *subsequence* of u and v is a sequence of index pairs

$$(i_1, j_1), \ldots, (i_r, j_r)$$

such that

$$1 \leq i_1 < \ldots < i_r \leq m \text{ and}$$
$$1 \leq j_1 < \ldots < j_r \leq n. \ \square$$

The index pair (i_h, j_h) stands for the replacement $u_{i_h} \rightarrow v_{j_h}$. All characters in u and v not occurring in a subsequence are considered to be deleted in u or v. For example, the empty subsequence stands for the alignments in (3.1). In a graphical representation, the index pairs of the subsequence appear as lines connecting the characters in the subsequence.

Example 5 The following subsequences of $u = gabh$ and $v = gcdhb$ represent the alignments of Example 4. In particular, P_1 represents L_1, L_2, and L_3, while P_2 represents L_4, and P_3 represents L_5.

$$P_1 = \begin{pmatrix} g\ a\ b\ h \\ |\quad\quad| \\ g\ c\ d\ h\ b \end{pmatrix} \qquad P_2 = \begin{pmatrix} g\ a\ b\ h \\ |\ |\ \diagdown \\ g\ c\ d\ h\ b \end{pmatrix} \qquad P_3 = \begin{pmatrix} g\ a\ b\ h \\ |\ |\ |\ | \\ g\ c\ d\ h\ b \end{pmatrix} \ \square$$

Observation 2 Let $Subseqs(m, n)$ be the number of subsequences of two fixed sequences of length m and n. Then

$$Subseqs(m, n) = \sum_{r=0}^{\min(m,n)} \binom{m}{r} \cdot \binom{n}{r}$$

Proof: For each $r \in [0, \min\{m, n\}]$ we have: for the ordered selection of the indices $i_1, \ldots, i_r$ there are $\binom{m}{r}$ possibilities; for the ordered selection of the indices $j_1, \ldots, j_r$ there are $\binom{n}{r}$ possibilities. All these possibilities have to be combined. $\square$

$Subseqs(n, n)$ can be approximated by $2^{2n} \left(4 \cdot \sqrt{n\pi}\right)^{-1}$, e.g. $Subseqs(1000, 1000) \approx 10^{600}$.

3.2.2 The Edit Distance Problem

The notion of optimal alignment requires some scoring or optimization criterion. This is given by a cost function.

Definition 4 A *cost function* δ assigns to each edit operation $\alpha \rightarrow \beta$, $\alpha \neq \beta$ a positive real cost $\delta(\alpha \rightarrow \beta)$. The cost $\delta(\alpha \rightarrow \alpha)$ of an edit operation $\alpha \rightarrow \alpha$ is 0. If $\delta(\alpha \rightarrow \beta) = \delta(\beta \rightarrow \alpha)$ for all edit operations $\alpha \rightarrow \beta$ and $\beta \rightarrow \alpha$, then δ is *symmetric*. If $\delta(\alpha \rightarrow \beta) = 1$, for all edit operations $\alpha \rightarrow \beta$, $a \neq b$ then δ is the *unit cost function*. δ is extended to alignments in a straightforward way: The cost $\delta(A)$ of an alignment $A = (\alpha_1 \rightarrow \beta_1, \ldots, \alpha_h \rightarrow \beta_h)$ is the sum of the costs of the edit operations A consists of. More precisely,

$$\delta(A) = \sum_{i=1}^{h} \delta(\alpha_i \rightarrow \beta_i). \ \square$$

Example 6

1. Let
$$\delta(\alpha \to \beta) = \begin{cases} 0 & \text{if } \alpha, \beta \in \mathcal{A} \text{ and } \alpha = \beta \\ 1 & \text{otherwise} \end{cases}$$
Then δ is the *unit cost*.

2. Let
$$\delta(\alpha \to \beta) = \begin{cases} 0 & \text{if } \alpha, \beta \in \mathcal{A} \text{ and } \alpha = \beta \\ 1 & \text{else if } \alpha, \beta \in \mathcal{A} \text{ and } \alpha \neq \beta \\ \infty & \text{otherwise} \end{cases}$$
Then δ is the *hamming cost*.

3. Let
$$\delta(\alpha \to \beta) = \begin{cases} 0 & \text{if } \alpha, \beta \in \mathcal{A} \text{ and } \alpha = \beta \\ 2 & \text{else if } \alpha, \beta \in \mathcal{A} \text{ and } \alpha \neq \beta \\ 1 & \text{otherwise} \end{cases}$$
Then δ is the *LCS cost*. We will later see that this cost function is related to the LCS problem, hence the name.

4. Suppose δ is given by the following table:

δ	ε	A	C	G	T
ε		3	3	3	3
A	3	0	2	1	2
C	3	2	0	2	1
G	3	1	2	0	2
T	3	2	1	2	0

Then δ is the transversion/transition cost function. Bases A and G are called *purine*, and bases C and T are called *pyrimidine*. The transversion/transition cost function reflects the biological fact that a purine/purine and a pyrimidine/pyrimidine replacement is much more likely to occur than a purine/pyrimidine replacement. Moreover, it takes into account that a deletion or an insertion of a base occurs more seldom.

5. The following tables shows costs for replacements of amino acids, as suggested by Willy Taylor. For a cost function we would have to define the indel costs:

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	0	14	7	9	20	9	8	7	12	13	17	11	14	24	7	6	4	32	23	11
R	14	0	11	15	26	11	14	17	11	18	19	7	16	25	13	12	13	25	24	18
N	7	11	0	6	23	5	6	10	7	16	19	7	16	25	10	7	7	30	23	15
D	9	15	6	0	25	6	3	9	11	19	22	10	19	29	11	11	10	34	27	17
C	20	26	23	25	0	26	25	21	25	22	26	25	26	26	22	20	21	34	22	21
Q	9	11	5	6	26	0	5	12	7	17	20	8	16	27	10	11	10	32	26	16
E	8	14	6	3	25	5	0	9	10	17	21	10	17	28	11	10	9	34	26	16
G	7	17	10	9	21	12	9	0	15	17	21	13	18	27	10	9	9	33	26	15
H	12	11	7	11	25	7	10	15	0	17	19	10	17	24	13	11	11	30	21	16
I	13	18	16	19	22	17	17	17	17	0	9	17	8	17	16	14	12	31	18	4
L	17	19	19	22	26	20	21	21	19	9	0	19	7	14	20	19	17	27	18	10
K	11	7	7	10	25	8	10	13	10	17	19	0	15	26	11	10	10	29	25	16
M	14	16	16	19	26	16	17	18	17	8	7	15	0	18	17	16	13	29	20	8
F	24	25	25	29	26	27	28	27	24	17	14	26	18	0	27	24	23	24	8	19
P	7	13	10	11	22	10	11	10	13	16	20	11	17	27	0	9	8	32	26	14
S	6	12	7	11	20	11	10	9	11	14	19	10	16	24	9	0	5	29	22	13
T	4	13	7	10	21	10	9	9	11	12	17	10	13	23	8	5	0	31	22	10
W	32	25	30	34	34	32	34	33	30	31	27	29	29	24	32	29	31	0	25	32
Y	23	24	23	27	22	26	26	26	21	18	18	25	20	8	26	22	22	25	0	20
V	11	18	15	17	21	16	16	15	16	4	10	16	8	19	14	13	10	32	20	0

Definition 5 The *edit distance* of u and v, denoted by $edist_\delta(u,v)$, is the minimum possible cost of an alignment of u and v. That is,

$$edist_\delta(u,v) = \min\{\delta(A) \mid A \text{ is an alignment of } u \text{ and } v\}.$$

An alignment A of u and v is *optimal* if $\delta(A) = edist_\delta(u,v)$. If δ is the unit cost function, then $edist_\delta(u,v)$ is the *unit edit distance* between u and v. □

By definition, δ satisfies the zero property. If δ is symmetric and satisfies the triangle inequality, then $edist_\delta$ is a metric. Note that there can be more than one optimal alignment. The unit edit distance is sometimes called Levenshtein distance. The following observation states a simple property of the edit distance.

Observation 3 For any cost function δ and any two strings $u, v \in \mathcal{A}^*$ the following equation holds:

$$edist_\delta(u,v) = edist_\delta(u^{-1}, v^{-1})$$

Proof: Let $A = (\alpha_1 \to \beta_1, \ldots, \alpha_h \to \beta_h)$ be an optimal alignment of u and v. Obviously, $A^{-1} = (\alpha_h \to \beta_h, \ldots, \alpha_1 \to \beta_1)$ is an alignment of u^{-1} and v^{-1}. Now suppose there is an alignment X of u^{-1} and v^{-1} such that $\delta(X) < \delta(A^{-1})$. That is, A^{-1} is not the optimal alignment of u^{-1} and v^{-1}. Now X^{-1} is an alignment of u and v and we have $\delta(X^{-1}) = \delta(X) < \delta(A^{-1}) = \delta(A)$. Thus A is not an optimal alignment. This is a contradiction. Hence our assumption above was wrong, i.e. there is no alignment X of u^{-1} and v^{-1} with $\delta(X) < \delta(A^{-1})$. As a consequence

$$edist_\delta(u,v) = \delta(A) = \delta(A^{-1}) = edist_\delta(u^{-1}, v^{-1}) \quad \square$$

Definition 6 The *edit distance problem* is to compute the edit distance and all optimal alignments. □

By specifying the cost functions, we obtain special forms of edit distances:

Definition 7

- If δ is the unit cost, then $edist_\delta$ is the *unit edit distance* or *Levenshtein distance*.
- If δ is the hamming cost, then $edist_\delta$ is the *hamming distance*.
- If δ is the LCS cost, then $edist_\delta$ is the *simple Levenshtein distance*. □

3.2.3 A Dynamic Programming Algorithm

Suppose a cost function δ is given and $u, v \in \mathcal{A}^*$ are fixed but arbitrary. We will now develop some recursive equation for $edist_\delta(u,v)$ from which we derive a dynamic programming algorithm.

Consider an optimal alignment

$$A = \begin{pmatrix} \alpha_1 & \alpha_2 & \ldots & \alpha_h \\ \beta_1 & \beta_2 & \ldots & \beta_h \end{pmatrix}$$

of $u = u_1 \ldots u_m$ and $v = v_1 \ldots v_n$. Then $\delta(A) = edist_\delta(u,v)$.

- Case (1): $u = \varepsilon$. Since $u = \alpha_1\alpha_2\ldots\alpha_h$, we conclude $\alpha_i = \varepsilon$ for all $i \in [1,h]$. Hence $h = n$, and $\beta_j = v_j$ for $j \in [1,h]$. Thus the cost of A is $\delta(A) = \sum_{j=1}^{n} \delta(\varepsilon \to \beta_j)$.
- Case (2): $v = \varepsilon$. Since $v = \beta_1\beta_2\ldots\beta_h$, we conclude $\beta_j = \varepsilon$ for all $j \in [1,h]$. Hence $h = m$, and $\alpha_i = u_i$ for $i \in [1,h]$. Thus the cost of A is $\delta(A) = \sum_{i=1}^{m} \delta(\alpha_i \to \varepsilon)$.
- Case (3): $u \neq \varepsilon$ and $v \neq \varepsilon$. Then $u = u'a$ and $v = v'b$ for some $u', v' \in \mathcal{A}^*$ and some $a, b \in \mathcal{A}$. Now split A into an alignment A' (consisting of the first $h-1$ edit operations) and the hth edit operation:

$$A = \begin{pmatrix} A' & \alpha_h \\ & \beta_h \end{pmatrix}$$

  - Case (3a): $\alpha_h = a$ and $\beta_h = \varepsilon$. Then A' is an alignment of u' and v. Suppose that A' is not optimal. Then $\delta(A') > edist_\delta(u', v)$. Now

    $$edist_\delta(u,v) = \delta(A) = \delta(A') + \delta(\alpha_h \to \beta_h) > edist_\delta(u', v) + \delta(a \to \varepsilon) \geq edist_\delta(u,v)$$

    This is a contradiction. Hence A' is an optimal alignment of u' and v, and $edist_\delta(u,v) = \delta(A) = edist_\delta(u', v) + \delta(a \to \varepsilon)$. The following case (3b) handles insertions and case (3c) handles replacements in an analogous way.
  - Case (3b): $\alpha_h = \varepsilon$ and $\beta_h = b$. Then A' is an alignment of u and v'. Suppose that A' is not optimal. Then $\delta(A') > edist_\delta(u, v')$. Now

    $$edist_\delta(u,v) = \delta(A) = \delta(A') + \delta(\alpha_h \to \beta_h) > edist_\delta(u, v') + \delta(\varepsilon \to b) \geq edist_\delta(u,v)$$

    This is a contradiction. Hence the A' is an optimal alignment of u and v'. Hence $edist_\delta(u,v) = \delta(A) = edist_\delta(u, v') + \delta(\varepsilon \to b)$.
  - Case (3c): $\alpha_h = a$ and $\beta_h = b$. Then A' is an alignment of u' and v'. Suppose that A' is not optimal. Then $\delta(A') > edist_\delta(u', v')$. Now

    $$edist_\delta(u,v) = \delta(A) = \delta(A') + \delta(\alpha_h \to \beta_h) > edist_\delta(u', v') + \delta(a \to b) \geq edist_\delta(u,v)$$

    This is a contradiction. Hence A' is an optimal alignment of u' and v', and $edist_\delta(u,v) = \delta(A) = edist_\delta(u', v') + \delta(a \to b)$.

  Since all three cases (3a), (3b), and (3c) may occur, we have to compute the minimum over all three cases. Altogether, we get the following system of recursive equations:

$$\begin{aligned}
edist_\delta(\varepsilon, \varepsilon) &= 0 \\
edist_\delta(\varepsilon, v'b) &= edist_\delta(\varepsilon, v') + \delta(\varepsilon \to b) \\
edist_\delta(u'a, \varepsilon) &= edist_\delta(u', \varepsilon) + \delta(a \to \varepsilon) \\
edist_\delta(u'a, v'b) &= \min \left\{ \begin{array}{l} edist_\delta(u'a, v') + \delta(\varepsilon \to b) \\ edist_\delta(u', v'b) + \delta(a \to \varepsilon) \\ edist_\delta(u', v') + \delta(a \to b) \end{array} \right\}
\end{aligned}$$

Of course, the direct implementation of $edist_\delta$ as a recursive function would be inefficient, since the same function calls might appear in different contexts. However, note that $edist_\delta(u', v')$ is evaluated for all pairs of prefixes u' of u and v' of v. So the idea is to tabulate these intermediate results. That is, we compute an $(m+1) \times (n+1)$ matrix E_δ defined as follows:

$$E_\delta(i,j) = edist_\delta(u_1 \dots u_i, v_1 \dots v_j)$$

for all $i \in [0,m]$ and $j \in [0,n]$. Using the equations above, it is easy to prove that the following recurrences hold:

$$E_\delta(0,0) = 0 \tag{3.2}$$
$$E_\delta(i+1,0) = E_\delta(i,0) + \delta(u_{i+1} \to \varepsilon) \tag{3.3}$$
$$E_\delta(0,j+1) = E_\delta(0,j) + \delta(\varepsilon \to v_{j+1}) \tag{3.4}$$
$$E_\delta(i+1,j+1) = \min \left\{ \begin{array}{l} E_\delta(i,j+1) + \delta(u_{i+1} \to \varepsilon) \\ E_\delta(i+1,j) + \delta(\varepsilon \to v_{j+1}) \\ E_\delta(i,j) + \delta(u_{i+1} \to v_{j+1}) \end{array} \right\} \tag{3.5}$$

By definition, $E_\delta(m,n)$ gives the edit distance of u and v. The values in E_δ are computed in topological order, i.e. consistent with the data dependencies. The following algorithm, for example, employs a computation column by column.

Algorithm DP Algorithm for the Edit Distance
Input: sequences $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$
cost function δ
Output: $edist_\delta(u,v)$

$E_\delta(0,0) := 0$
for $i := 1$ **to** m **do**
  $E_\delta(i,0) := E_\delta(i-1,0) + \delta(u_i \to \varepsilon)$
for $j := 1$ **to** n **do**
  $E_\delta(0,j) := E_\delta(0,j-1) + \delta(\varepsilon \to v_j)$
  for $i := 1$ **to** m **do**

$$E_\delta(i,j) := \min \left\{ \begin{array}{l} E_\delta(i,j-1) + \delta(\varepsilon \to v_j) \\ E_\delta(i-1,j) + \delta(u_i \to \varepsilon) \\ E_\delta(i-1,j-1) + \delta(u_i \to v_j) \end{array} \right\}$$

print $E_\delta(m,n)$

Example 7 Let $u = bcacd$, $v = dbadad$, and assume that δ is the unit cost function.

Then E_δ is as follows:

			d	b	a	d	a	d
	$E_\delta(i,j)$	0	1	2	3	4	5	6
	0	0	1	2	3	4	5	6
b	1	1	1	1	2	3	4	5
c	2	2	2	2	2	3	4	5
a	3	3	3	3	2	3	3	4
c	4	4	4	4	3	3	4	4
d	5	5	4	5	4	3	4	4

Hence the edit distance of u and v is 4. □

Each entry in E_δ is computed in constant time. This leads to an $O(m \cdot n)$ time complexity. Note that the values in each column only depend on the values of the previous column. Hence, if we only want to compute the edit distance, then it suffices to store only two columns in each step of the algorithm. Hence, in this case, the space requirement is $O(\min\{m,n\})$. The corresponding algorithm is then also termed "distance-only algorithm" for computing the edit distance.

In molecular biology, the above algorithm is usually called "the dynamic programming algorithm". However, dynamic programming (DP, for short) is a general programming paradigm. A problem can be solved by DP, if the following holds:

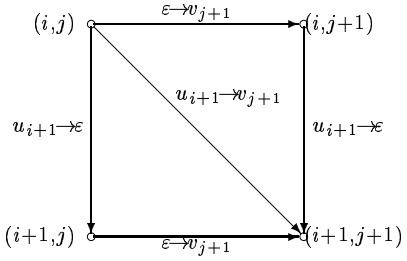
- optimal solutions to the problem can be derived from optimal solutions to subproblems.
- the optimal solutions can efficiently be determined, if a table of solutions for increasing subproblems are computed.

To completely solve the edit distance problem, we also have to compute the optimal alignments. An optimal alignment is recovered by tracing back from the entry $E_\delta(m,n)$ to an entry in its three-way minimum that yielded it, determining which entry gave rise to that entry, and so on back to the entry $E_\delta(0,0)$. This requires saving the entire table, giving an algorithm that takes $O(m \cdot n)$ space. This backtracking algorithm can best be explained by giving a graph theoretic formulation of the problem.

Definition 8 The *edit graph* $G(u,v)$ of u and v is an edge labeled graph. The nodes are the pairs $(i,j) \in [0,m] \times [0,n]$. The edges are given as follows:

- For $0 \le i \le m-1, 0 \le j \le n$ there is a deletion edge $(i,j) \xrightarrow{u_{i+1} \to \varepsilon} (i+1,j)$.
- For $0 \le i \le m, 0 \le j \le n-1$ there is an insertion edge $(i,j) \xrightarrow{\varepsilon \to v_{j+1}} (i,j+1)$.
- For $0 \le i \le m-1, 0 \le j \le n-1$ there is a replacement edge $(i,j) \xrightarrow{u_{i+1} \to v_{j+1}} (i+1, j+1)$.

Figure 3.1: A Part of the Edit Graph $G(u, v)$



□

This is illustrated in Figure 3.1.

The central feature of $G(u, v)$ is that each path from (i', j') to (i, j) is labeled by an alignment of $u_{i'+1} \dots u_i$ and $v_{j'+1} \dots v_j$, and a different path is labeled by a different alignment. An edge $(i', j') \xrightarrow{\alpha \to \beta} (i, j)$ is *minimizing* if $E_\delta(i, j)$ equals $E_\delta(i', j') + \delta(\alpha \to \beta)$. A *minimizing path* is any path from $(0, 0)$ to (m, n) that consists of minimizing edges only. In this framework, the edit distance problem means to enumerate the minimizing paths in $G(u, v)$. This is done by starting at node (m, n) and tracing the minimizing edges back to node $(0, 0)$. The back tracing procedure can be organized in such a way that each optimal alignment A of u and v is computed in $O(|A|)$ time. To facilitate the backtracking, we store with each entry $E_\delta(i, j)$ three bits. Each of these bits tells us whether an incoming edge into (i, j) is minimizing. Thus we conclude:

Theorem 1 The edit distance problem for two sequences u of length m and v of length n can be solved in $O(m \cdot n + z)$ time, where z is the total length of all alignments of u and v. □

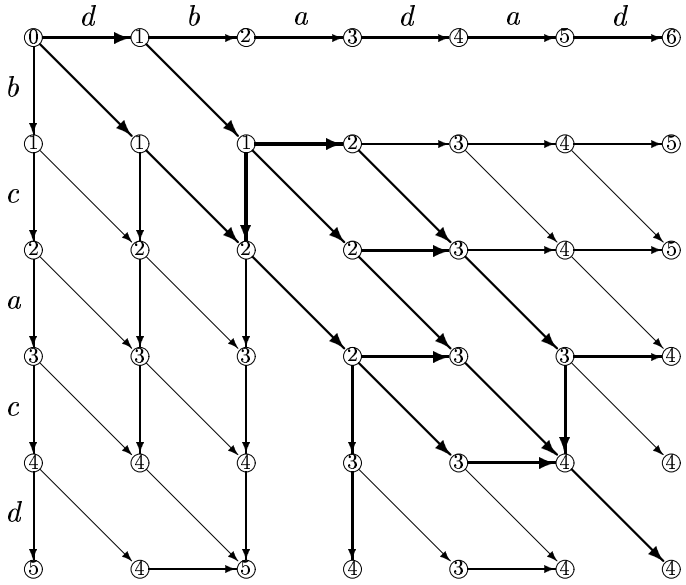
Example 8 Let $u = bcacd$ and $v = dbadad$. Suppose δ is the unit cost function. Then we have $edist_\delta(u, v) = 4$ and there are the following optimal alignments of u and v.

- b c a c - d	b c a c - d	- b c - a c d	- b - c a c d	- b c a - c d	b c a - c d	- b c a c d
d b - a d a d	d b a d a d	d b a d a - d	d b a d a - d	d b - a d a d	d b a d a d	d b a d a d

Figure 3.2 shows $G(u, v)$ with all minimizing edges. The minimizing paths are given by the thick edges. Each node is marked by the corresponding edit distance. It is straightforward to read the optimal alignments of u and v from the edit graph. □

The space requirement for the above procedure is $O(m \cdot n)$. Using a distance-only algorithm as a sub-procedure, there are divide and conquer algorithms that can determine each optimal alignment in $O(m+n)$ space and $O(m \cdot n)$ time. These algorithms are very important, since space, not time, is often the limiting factor when computing optimal alignments between large strings. However, we will not consider them further.

Figure 3.2: The Minimizing Edges and Paths in the Edit Graph $G(bcacd, dbadad)$. Edge labels are not shown.



3.2.4 Fast Computation of the Simple Levenshtein Distance

Recall that the simple Levenshtein distance is the edit distance given the cost function δ defined by

$$\delta(\alpha \to \beta) = \begin{cases} 0 & \text{if } \alpha, \beta \in \mathcal{A} \text{ and } \alpha = \beta \\ 2 & \text{else if } \alpha, \beta \in \mathcal{A} \text{ and } \alpha \neq \beta \\ 1 & \text{otherwise} \end{cases}$$

Now consider the edit graph $G(u, v)$ when computing the simple Levenshtein distance. Consider a minimizing edge from node (i, j) to node $(i + 1, j + 1)$ labeled by the replacement operation $u_{i+1} \to v_{j+1}$. If $u_{i+1} \neq v_{j+1}$, then the minimizing edge has weight 2. So a detour from (i, j) to $(i + 1, j + 1)$ via $(i + 1, j)$ or $(i, j + 1)$ involving the deletion of u_{i+1} and the insertion of v_{j+1} has the same total weight 2. In other words, if we want to compute the simple Levenshtein distance, then we can restrict to minimizing paths which do not contain a diagonal edge labeled by a replacement of two distinct characters. So, in this subsection, when we talk about diagonal edges, we always refer to those with weight 0, i.e. those labeled by a replacement of two identical characters.

As a consequence, the computation of the simple Levenshtein distance means to minimize the number of horizontal/vertical edges in the edit graph. Equivalently, we can maximize the number of diagonal edges in the edit graph. Thus the simple Levenshtein distance is closely related to the LCS-problem which we define now.

Definition 9 A common subsequence of u and v is a subsequence

$$(i_1, j_1), \ldots, (i_r, j_r)$$

such that $u_{i_l} = v_{j_l}$ for $l \in [1, r]$. The *longest common subsequence problem* (LCS-problem, for short) is to find a common subsequence of u and v of maximal length. This length is denoted by $lcs(u, v)$. Each common subsequence denotes a string $u_{i_1} u_{i_2} \ldots u_{i_r} = v_{j_1} v_{j_2} \ldots v_{j_r}$. □

Example 9 Let $u = cbabac$ and $v = abcabba$. Then $(0, 2), (2, 3), (3, 4), (4, 6)$ is a longest common subsequence denoting the string $caba$. Hence $lcs(u, v) = 4$. □

Observation 4 Let δ be the LCS-cost function. Then the following property holds for all strings u and v:

$$2 \cdot lcs(u, v) + edist_\delta(u, v) = m + n \tag{3.6}$$

Proof: Consider an optimal alignment A of u and v, which does not contain any replacement $a \to b$ with $a \neq b$. As shown above, this must exist. Since $m = |u|$ and $n = |u|$, there are $m + n$ characters occurring in A. $edist_\delta(u, v)$ is the number of deletions and insertions in A, and this is identical to the number of characters occurring in a deletion or an insertion. The number of replacements $a \to b$ with $a = b$ is identical to $lcs(u, v)$. Each such replacement contains 2 characters. So the alignment contains $2 \cdot lcs(u, v) + edist_\delta(u, v)$ characters. Thus (3.6) holds. □

Due to (3.6) the LCS-problem and the problem to compute the simple Levenshtein distance are equivalent. A solution to one problem can in constant time be transformed into a solution to the other problem.

We now give an output-sensitive algorithm for computing the simple Levenshtein distance. That is, an algorithm, whose running time depends on the computed distance value. The smaller this value, the faster it runs.

Definition 10 Let $d \in \mathbb{N}_0$. A *d-path* is a path in $G(u, v)$ which begins at node $(0, 0)$ and which contains d non-diagonal edges. That is, a d-path has cost d. Let $h \in [-m, n]$. The forward diagonal h ($\searrow$) consists of all pairs (i, j) satisfying $j - i = h$. □

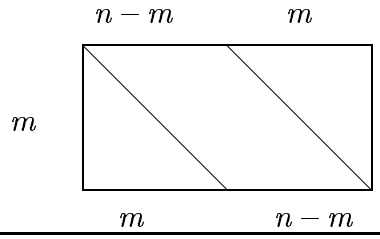
By definition, $(0, 0)$ is on diagonal 0, and (m, n) on diagonal $n - m$. Hence it is clear that any path from $(0, 0)$ to (m, n) must cross the diagonal band between diagonal 0 and diagonal $n - m$, as shown in Figure 3.3.

Observation 5 A d-path must end in a diagonal $h \in D_d := \{-d, -d+2, \ldots, d-2, d\}$.

Proof: We prove the claim by induction on d.

- Case $d = 0$: A 0-path begins at $(0, 0)$ (on diagonal 0) and it only has diagonal edges. Hence it ends on diagonal $0 \in D_d = \{0\}$.
- Consider a $(d+1)$-path. It contains at least one non-diagonal edge, and thus can be split into 3 parts:

Figure 3.3: The diagonal band from diagonal 0 (left) to diagonal $n-m$ (right)



part 1: maximal prefix which is a d-path. By assumption this path ends in diagonal $h \in D_d$.

part 2: either a horizontal edge from diagonal h to $h+1$ or a vertical edge from diagonal h to $h-1$.

part 3: a path on diagonal $h+1$ or $h-1$ depending on part 2.

Hence the $(d+1)$-path ends in diagonal

$$\begin{aligned} h' &\in \{-d+1, -d+2+1, \ldots, d-2+1, d+1\} \cup \{-d-1, -d+2-1, \ldots, d-2-1, d-1\} \\ &= \{-(d+1)+2, -(d+1)+4, \ldots, (d+1)-2, d+1\} \cup \\ &\quad \{-(d+1), -(d+1)+2, \ldots, (d+1)-2, d+1\} \\ &= \{-(d+1), -(d+1)+2, \ldots, (d+1)-2, d+1\} \\ &= D_{d+1} \quad \square \end{aligned}$$

Definition 11 A d-path of maximal length on diagonal h is a maximal d-path on h. $\square$

The idea of the algorithm is to compute how far we come in the edit graph using d vertical or horizontal edges. More precisely, compute for each $d = 0, 1, \ldots$ and all $h \in [-d, d]$ the endpoint of a maximal d-path on h. Now recall that (m, n) is on diagonal $n-m$. Hence, if d is minimal such that (m, n) is the endpoint of a maximal path on $n-m$, then we have $edist_\delta(u, v) = E_\delta(m, n) = d$. The endpoint is defined in terms of the *front* of a diagonal:

Definition 12 For any $d \in \mathbb{N}_0$ and any $h \in [-d, d]$ define

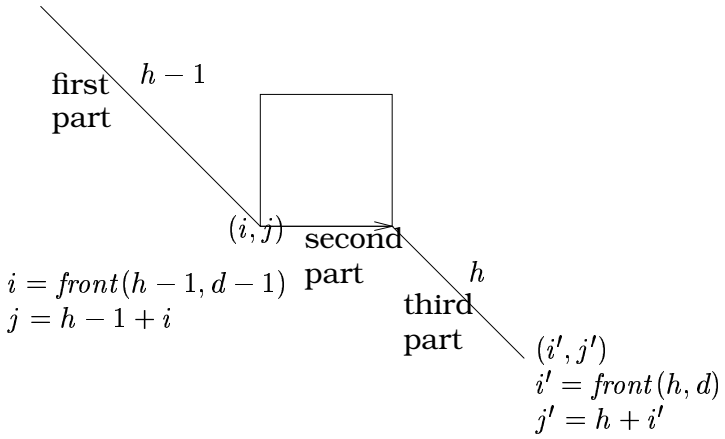
$$front(h, d) = \max\{i \in [0, m] \mid E_\delta(i, h+i) = d\}. \square$$

That is, the end point of a d-path on a particular diagonal h is given as the row number of the end point.

Observation 6 Let $d_{\min} := \min\{d \in \mathbb{N}_0 \mid front(n-m, d) = m\}$. Then $d_{\min}$ is the simple Levenshtein distance of u and v.

Proof: Let $d = edist_\delta(u, v)$. We have $d = E_\delta(m, n) = E_\delta(m, (n-m)+m)$ and hence $front(n-m, d) = m$. Thus $d \geq d_{\min}$. Now suppose $d > d_{\min}$. We have $front(n-m, d_{\min}) = m$ which implies $d_{\min} = E_\delta(m, (n-m)+m) = E_\delta(m, n) = d$. This is a contradiction, which implies that the assumption $d > d_{\min}$ was wrong. Hence $d = d_{\min}$. $\square$

Figure 3.4: Case 1.: Splitting of a d-path into 3 parts



We will now develop recurrences for computing $front$.

Consider the case $d = 0$. A maximal 0-path ends on (i, i) where $i = |lcp(u, v)|$ and $lcp(u, v)$ is the longest common prefix of u and v. Hence we derive

$$front(0, 0) = |lcp(u, v)| \tag{3.7}$$

Now let $d > 0$ and consider a maximal d-path ending on h. There are two ways to split this path into three parts.

- Suppose the d-path on h consists of the following three parts (as shown in Figure 3.4):

  1. a maximal $(d-1)$-path on $h-1$.
  2. a horizontal edge from diagonal $h-1$ to diagonal h.
  3. a maximal path on diagonal h.

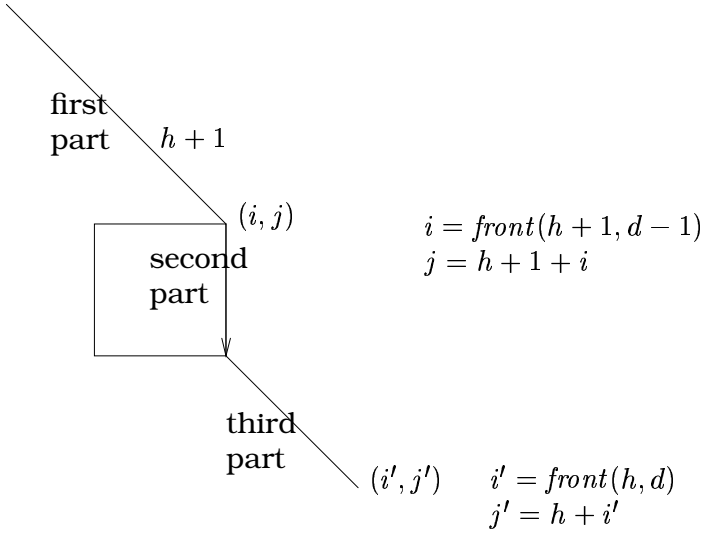
  Suppose that the maximal $(d-1)$-path on $h-1$ ends in (i, j), i.e. $i = front(h-1, d-1)$ and $j = h-1+i$. Then the maximal path on diagonal h ends in some point (i', j') where $i' = front(h, d)$ and $j' = h + i'$. The length of the maximal path on diagonal h is the length of $lcp(u_{i+1} \ldots u_m, v_{h+i+1} \ldots v_n)$. Hence we conclude

  $$front(h, d) = i + |lcp(u_{i+1} \ldots u_m, v_{h+i+1} \ldots v_n)|$$

- Suppose the d-path on h consists of the following three parts (as shown in Figure 3.5):

  1. a maximal $(d-1)$-path on $h+1$.
  2. a vertical edge from diagonal $h+1$ to diagonal h.
  3. a maximal path on diagonal h.

Figure 3.5: Case 2.: Splitting of a d-path into 3 parts



Suppose that the maximal $(d-1)$-path on $h+1$ ends in (i,j), i.e. $i = front(h+1, d-1)$ and $j = h+1+i$. Then the maximal path on diagonal h ends in some point (i',j') where $i' = front(h,d)$ and $j' = h+i'$. The length of the maximal path on diagonal h is the length of $lcp(u_{i+2} \ldots u_m, v_{h+i+2} \ldots v_n)$. Hence we conclude

$$front(h,d) = i + 1 + |lcp(u_{i+2} \ldots u_m, v_{h+i+2} \ldots v_n)|$$

Since both cases can occur we have to combine them to obtain the following recurrence:

$$\begin{aligned} front(h,d) &= l + |lcp(u_{l+1} \ldots u_m, v_{h+l+1} \ldots v_n)| \\ &\quad \text{where } l = \max\{front(h-1, d-1), front(h+1, d-1)+1\} \end{aligned} \tag{3.8}$$

We can now define the greedy algorithm for computing the simple Levenshtein distance.

Algorithm Greedy DP Algorithm for the simple Levenshtein distance
Input: sequences $u = u_1 \ldots u_m$ and $v = v_1 \ldots v_n$
δ is the LCS cost function
Output: $edist_\delta(u,v)$
for $d := 0$ **to** $n+m$ **do**
  for $h := -d$ **to** d **do**
    Compute $front(h,d)$ *according to* (3.7) *and* (3.8)
  if $front(n-m,d) = m$ **then** *return* d

Let $e := edist_\delta(u,v)$. For each $d \in [0,e]$, the algorithm computes a front of width $2 \cdot d + 1 \in O(m+n)$. Hence the running time is $O((m+n) \cdot e)$. Thus the algorithm

is output sensitive. The smaller the distance, the faster it runs. Each generation of *front*-values $front(-d,d), front(-d+1,d), \ldots, front(d-1,d), front(d,d)$ with $d > 0$ can be computed from the previous generation. Thus we only need to store two generations at any time. Hence the space requirement is $O(m+n)$. The expected running time of the algorithm is $O(m+n+e)$. We do not give a proof for this.

Note that newer versions of the UNIX command *diff* are based on this algorithm. This gave large speedups in comparison to a previous version of the algorithm.

3.2.5 Fast Computation of the Unit Edit Distance

The algorithm from the previous section can be generalized to also compute the unit edit distance. We just have to add a third case. But before we consider the details we show some properties of the unit edit distance, and the corresponding edit distance table. Assume for this subsection that δ is the unit cost function. From Section 3.2.3 we can derive the following equations for table E_δ:

$$
\begin{aligned}
&E_\delta(i,0) = i \\
&E_\delta(0,j) = j \\
&E_\delta(i+1,j+1) = \min \left\{ \begin{array}{l} E_\delta(i,j+1)+1 \\ E_\delta(i+1,j)+1 \\ E_\delta(i,j) + (\text{if } u_{i+1} = v_{j+1} \text{ then } 0 \text{ else } 1) \end{array} \right\}
\end{aligned}
$$

Consecutive entries in E_δ-columns, E_δ-rows, and E_δ-diagonals differ by at most one. Additionally the entries in E_δ-diagonals are non-decreasing. This is formally stated in the following observation. We do not give a proof.

Observation 7 Table E_δ has the following properties:

1. $E_\delta(i,j) - 1 \leq E_\delta(i+1,j) \leq E_\delta(i,j) + 1$, $i \in [0, m-1]$, $j \in [0,n]$.
2. $E_\delta(i,j) \leq E_\delta(i+1,j+1) \leq E_\delta(i,j) + 1$, $i \in [0, m-1]$, $j \in [0,n-1]$.
3. $E_\delta(i,j+1) - 1 \leq E_\delta(i,j) \leq E_\delta(i,j+1) + 1$, $i \in [0, m]$, $j \in [0,n-1]$. □

From the properties stated in Observation 7 we can conclude the following:

Observation 8 For all $(i,j) \in [0,m-1] \times [0,n-1]$ the following properties hold.

1. If $E_\delta(i,j) \leq E_\delta(i,j+1)$ and $E_\delta(i,j) \leq E_\delta(i+1,j)$, then $E_\delta(i,j) = E_\delta(i+1,j+1)$ if and only if $u_{i+1} = v_{j+1}$.
2. $$E_\delta(i+1,j+1) = \left\{ \begin{array}{ll} E_\delta(i,j) & \text{if } u_{i+1} = v_{j+1} \\ 1 + E_\delta(i,j+1) & \text{else if } E_\delta(i,j+1) < E_\delta(i,j) \\ 1 + \min\{E_\delta(i+1,j), E_\delta(i,j)\} & \text{otherwise} \end{array} \right.$$

Proof:

1. By assumption, we have

$$E_\delta(i+1,j+1) = \min \left\{ \begin{array}{l} E_\delta(i+1,j)+1, \\ E_\delta(i,j+1)+1, \\ E_\delta(i,j)+\delta(u_{i+1} \to v_{j+1}) \end{array} \right\} = E_\delta(i,j) + \delta(u_{i+1} \to v_{j+1})$$

   Hence, $E_\delta(i,j) = E_\delta(i+1,j+1) \iff \delta(u_{i+1} \to v_{j+1}) = 0 \iff u_{i+1} = v_{j+1}$.

2. By Case distinction. □

Due to the previous observation, we do not have to evaluate E_δ completely. Whenever pair (u_{i+1}, v_{j+1}) is identical, the corresponding edge in the edit distance graph is minimizing. Hence it suffices to evaluate an entry along this edge. If $u_{i+1} \neq v_{j+1}$, then we additionally have to test if $E_\delta(i,j+1) < E_\delta(i,j)$ holds. If so, then we can evaluate $E_\delta(i+1,j+1)$ without computing $E_\delta(i+1,j)$. Thus matrix E_δ can be evaluated in a lazy strategy. Requesting the evaluation $E_\delta(m,n)$ then triggers the computation of all necessary values in E_δ in a band around the main diagonal. The smaller $E_\delta(m,n)$, the smaller the band. However, we can also compute $edist_\delta(u,v)$ by extending the greedy algorithm for the simple Levenshtein distance. For $d > 0$, we have to consider an additional case, since we now have diagonal edges with weight 1:

Suppose the d-path on h consists of the following three parts (as shown in Figure 3.6):

1. a maximal $(d-1)$-path on h.
2. a diagonal edge with weight 1 on diagonal h.
3. a maximal path on diagonal h.

Suppose that the maximal $(d-1)$-path on h ends in (i,j), i.e. $i = \mathit{front}(h, d-1)$ and $j = h+i$. Then the maximal path on diagonal h ends in some point (i',j') where $i' = \mathit{front}(h,d)$ and $j' = h+i'$. The length of the maximal path on diagonal h is the length of $lcp(u_{i+2} \ldots u_m, v_{h+i+2} \ldots v_n)$. Hence we conclude

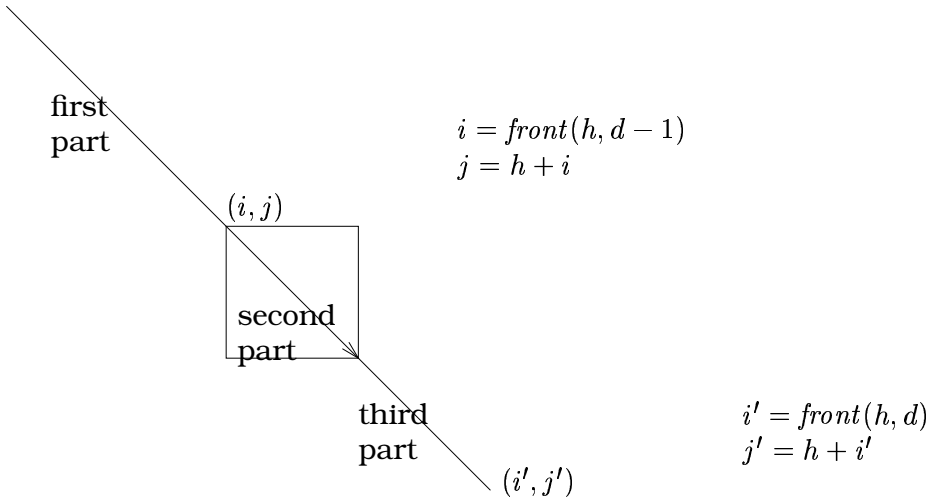
$$\mathit{front}(h,d) = i + 1 + |lcp(u_{i+2} \ldots u_m, v_{h+i+2} \ldots v_n)|$$

and we obtain the following recurrence for *front*:

$$\begin{array}{rcl} \mathit{front}(h,d) & = & l + |lcp(u_{l+1} \ldots u_m, v_{h+l+1} \ldots v_n)| \\ & & \text{where } l = \max \left\{ \begin{array}{l} \mathit{front}(h-1,d-1) \\ \mathit{front}(h+1,d-1)+1 \\ \mathit{front}(h,d-1)+1 \end{array} \right\} \end{array} \tag{3.9}$$

The greedy algorithm can be used verbatim, except that instead of (3.8) we use (3.9) to compute the front values. The worst case running time remains $O((m+n) \cdot e)$ where e is the unit edit distance. However, the expected running time becomes $O(m+n+e^2)$. See Figure 3.7 for an example of the values implicitly computed by this algorithm.

Figure 3.6: Case 3.: Splitting of a d-path into 3 parts



3.3 Local Similarity

Up to this point we have focussed on global comparison. That is, we have compared the complete sequence u with the complete sequence v. In biological sequences we often have long non-coding regions and small coding regions. Thus if two coding regions are similar, this does not imply that the sequences have a small edit distance. As a consequence, when comparing biological sequences it is sometimes important to perform local similarity comparisons: Find all pairs of substrings in u and v which are similar. To clarify the notion of similarity, we introduce score functions.

Definition 13 A *score function* σ assigns to each edit operation $\alpha \to \beta$ a *score* $\sigma(\alpha \to \beta) \in \mathbb{R}$. For each alignment $A = (\alpha_1 \to \beta_1, \ldots, \alpha_h \to \beta_h)$ we define the score $\sigma(A) = \sum_{i=1}^{h} \sigma(\alpha_i \to \beta_i)$. The similarity score of u and v is defined by

$$score_\sigma(u, v) = \max\{\sigma(A) \mid A \text{ is an alignment of } u \text{ and } v\}.\square$$

Note that, while distances are minimized, similarity scores are maximized. Table 3.8 shows the BLOSUM62 similarity matrix, which is currently widely used when comparing proteins. With some additional scores for insertions and deletions we would obtain a score function.

Definition 14 Let σ be a score function. We define

1. $loc_\sigma(u, v) = \max\{score_\sigma(u', v') \mid u' \text{ is substring of } u \text{ and } v' \text{ is substring of } v\}$
2. Let u' be a substring of u and v' be a substring of v such that $score_\sigma(u', v') = loc_\sigma(u, v)$. An alignment A of u' and v' satisfying $score_\sigma(u', v') = \sigma(A)$ is a *local optimal alignment* of u and v.

Figure 3.7: A complete distance matrix E_δ and the values implicitly computed for $d = 3$ and $k = 5$

		Z	E	I	T	G	E	I	S	T	
	0	1	2	3	4	5	6	7	8	9	
F	1	1	2	3	4	5	6	7	8	9	
R	2	2	2	3	4	5	6	7	8	9	
E	3	3	2	3	4	5	5	6	7	8	
I	4	4	3	2	3	4	5	5	6	7	
Z	5	4	4	3	3	4	5	6	6	7	
E	6	5	4	4	4	4	4	5	6	7	
I	7	6	5	4	5	5	5	4	5	6	
T	8	7	6	5	4	5	6	5	5	5	

complete distance matrix

		Z	E	I	T	G	E	I	S	T	
	0	1	2	3							
F	1	1	2	3							
R	2		2	3							
E	3										
I				2	3						
Z				3	3						
E											
I											
T											

values implicitly computed for $d = 3$
e.g. $front(0, 3) = 4, front(-1, 3) = 5, front(-2, 3) = 5$

		Z	E	I	T	G	E	I	S	T	
	0	1	2	3	4	5					
F	1	1	2	3	4	5					
R	2		2	3	4						
E	3										
I				2	3	4	5	5			
Z				3	3						
E				4	4	4					
I						5		4			
T				5	4	5		5	5	5	

values implicitly computed for $d = 5$. Since $m = 8$ and $n = 9$, we have $front(n - m, d) = front(1, 5) = 8$ and therefore $d = 5$.

Figure 3.8: The BLOSUM62 similarity score matrix specifying replacements score for each pair of amino acid

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4

3. The *local optimal alignment problem* is to compute $loc_\sigma(u, v)$ and a local optimal alignment of u and v. □

A brute force solution to the local optimal alignment problem would be as follows:

compute for each pair (u', v') of substrings u' of u and v' of v the value $score_\sigma(u', v')$.

Since there are $O(n^2m^2)$ pairs (u', v') of substrings and each computation of $score_\sigma(u', v')$ requires $O(mn)$, this method would require $O(n^3m^3)$ time. This is, of course, too expensive.

Now note that each substring u' of u is a suffix of a prefix of u and each substring v' of v is a suffix of a prefix of v. So the idea is to compute a matrix where each entry (i, j) contains the score for all pairs of suffixes of prefixes ending at position i in u and position j in v. More precisely, we compute an $(m+1)\times(n+1)$-Matrix L defined by

$$L(i, j) = \max\{score_\sigma(x, y) \mid x \text{ is suffix of } u_1 \dots u_i \text{ and } y \text{ is suffix of } v_1 \dots v_j\}$$

It is easy to see that $loc_\sigma(u, v) = \max\{L(i, j) \mid i \in [0, m], j \in [0, n]\}$. That is, $loc_\sigma(u, v)$ can be computed by maximizing over all entries in table L. Consider an edit graph representing all local alignments. Since we are interested in alignments of all pairs of substrings of u and v, we are interested in each path. The paths do not necessarily have to start at $(0, 0)$ or end at (m, n). Since a path can begin at any node, we have to allow the score 0 in any entry of the matrix. These considerations lead to the following result:

Theorem 2 Let σ be a score function satisfying

$$loc_\sigma(s,\varepsilon) = loc_\sigma(\varepsilon,s) = 0 \tag{3.10}$$

for any sequence $s \in \mathcal{A}^*$. Then the following holds:

- If $i = 0$ or $j = 0$, then $L(i,j) = 0$.
- Otherwise,

$$L(i,j) = \max \left\{ \begin{array}{l} 0 \\ L(i-1,j) + \sigma(u_i \to \varepsilon) \\ L(i,j-1) + \sigma(\varepsilon \to v_j) \\ L(i-1,j-1) + \sigma(u_i \to v_j) \end{array} \right\}$$

Condition (3.10) is very important for the Theorem: prefixes with negative score are suppressed, and similar substrings occur as positive islands in a matrix dominated by 0-entries. In general, it is not easy to verify condition (3.10). However, the following simple condition implies (3.10): $\sigma(\alpha \to \beta) < 0$ for all insertions and deletions $\alpha \to \beta$. This is because:

$$\begin{aligned} loc_\sigma(s,\varepsilon) &= \max\{score_\sigma(x,\varepsilon) \mid x \text{ is a substring of } s\} \\ &= \max\{score_\sigma(\varepsilon,\varepsilon)\} \cup \{score_\sigma(x,\varepsilon) \mid x \text{ is a substring of } s, x \neq \varepsilon\} \\ &= 0 \end{aligned}$$

One can similarly show $loc_\sigma(\varepsilon,s) = 0$.

Using these observations, we can derive a simple algorithm for the local similarity search problem:

Algorithm Smith-Waterman Algorithm
Input: sequences $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$
score function σ satisfying 3.10
Output: $loc_\sigma(u,v)$ and a local optimal alignment of u and v.

1. Compute Matrix L according to Theorem 2.
2. Compute a maximal entry, say $L(i,j)$, in L.
3. Compute local optimal alignments by backtracking on a maximizing path starting at (i,j) and ending in some entry $L(i',j') = 0$.

The Smith-Waterman Algorithm requires $O(mn)$ time and space.

Example 10 Consider the similarity score

$$\sigma(a \to b) = \left\{ \begin{array}{ll} -1 & \text{if } a = \varepsilon \vee b = \varepsilon \\ -2 & \text{if } a,b \in \mathcal{A}, a \neq b \\ 2 & \text{if } a,b \in \mathcal{A}, a = b \end{array} \right.$$

and the sequences $u = xyaxbacsll$ and $v = pqraxabcstvq$. Then matrix L is as follows:

		x	y	a	x	b	a	c	s	l	l
	0	0	0	0	0	0	0	0	0	0	0
p	0	0	0	0	0	0	0	0	0	0	0
q	0	0	0	0	0	0	0	0	0	0	0
r	0	0	**0**	0	0	0	0	0	0	0	0
a	0	0	0	**2**	1	0	2	1	0	0	0
x	0	2	1	1	**4**	**3**	2	1	0	0	0
a	0	1	0	3	3	2	**5**	4	3	2	1
b	0	0	0	2	2	5	**4**	3	2	1	0
c	0	0	0	1	1	4	3	**6**	5	4	3
s	0	0	0	0	0	3	2	5	**8**	7	6
t	0	0	0	0	0	2	1	4	7	6	5
v	0	0	0	0	0	1	0	3	6	5	4
q	0	0	0	0	0	0	0	2	5	4	3

The maximum value is 8. Tracing back the path along the bold face numbers gives a path representing the local optimal alignment (with score 8)

$$\begin{array}{cccccccc} a & x & b & a & - & c & s \\ a & x & - & a & b & c & s \end{array}$$

3.4 Advanced Problems

There is a multitude of more advanced problems concerning the edit distance model. We only mention a few important here:

- The Multiple Alignment Problem: Given sequences $S_1, S_2, \ldots, S_r$, compute an optimal Alignment of all these sequences. This can be done by generalizing the Algorithm for computing the edit distance. The modified algorithm computes an $(|S_1| + 1) \times (|S_2| + 1) \times \cdots \times (|S_r| + 1)$ matrix. Each entry (except for the boundary entries) has $2^r - 1$ predecessors over which the minimum/maximum has to be computed. Thus the algorithm runs in $O(2^r \cdot \prod_{i=1}^{r} |S_i|)$ time. There are heuristic algorithms which only compute a part of the matrix, but the worst case running time remains.

- Determining biologically important score functions. There are several methods to do this: One method is to take multiple alignments which have been thoroughly studied by biologists, and considered to be correct in the biological sense. From the alignment one determines a score function such that a dynamic programming algorithm would nearly compute the "correct alignment". This method involves several techniques from statistics. For example, the BLOSUM62 matrix has been determined by this method.

- In biology, the uniform scoring of gaps (i.e. a contiguous sequence of insertions and deletions) is not always correct. One would like a more general

cost/score for gaps. For example, a gap of length l could have the cost $g(l) := \alpha + \beta \cdot l$, where α and β are constants: α is the cost for starting a gap and β is the cost for extending the gap. The cost function is then called "affine gap cost." One usually chooses $\alpha > \beta$. There is a modification of the dynamic programming algorithm which can handle affine gap costs, while maintaining the running time of $O(mn)$.

- We have learned about global and about local comparison of sequences. There are problems in between these, e.g. the approximate string matching problem. We will learn about this problem in Section 5.

3.5 The Maximal Matches Model

The idea of this model is to measure the distance between strings in terms of common substrings. Strings are considered similar if they have long common substrings. The key to the model is the notion of partition. Recall that u and v are strings of length m and n, respectively.

Definition 15 A *partition* of v w.r.t. u is a sequence $(w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ of substrings $w_1, \ldots, w_r, w_{r+1}$ of u and characters $c_1, \ldots, c_r$ such that $v = w_1 c_1 \ldots w_r c_r w_{r+1}$. Let $\Psi = (w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ be a partition of v w.r.t. u. $w_1, \ldots, w_r, w_{r+1}$ are the *submatches* in Ψ. $c_1, \ldots, c_r$ are the *marked characters* in Ψ. The size of Ψ, denoted by $|\Psi|$, is r. $\mathit{mmdist}(v, u)$ is the size of any minimal partition of v w.r.t. u. We call $\mathit{mmdist}(v, u)$ *maximal matches distance* of v and u. □

Example 11 Let $v = cbaabdcb$ and $u = abcba$. $\Psi_1 = (cba, a, b, d, cb)$ is a partition of v w.r.t. u, since cba, b, and cb are substrings of u. $\Psi_2 = (cb, a, ab, d, cb)$ is a partition of v w.r.t. u, since cb and ab are substrings of u. It is clear that Ψ_1 and Ψ_2 are of minimal size. Hence, $\mathit{mmdist}(v, u) = 2$. □

There are two canonical partitions.

Definition 16 Let $\Psi = (w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ be a partition of v w.r.t. u. If for all $h \in [1, r]$, $w_h c_h$ is not a substring of u, then Ψ is the *left-to-right partition* of v w.r.t. u. If for all $h \in [1, r]$, $c_h w_{h+1}$ is not a substring of u, then Ψ is the *right-to-left partition* of v w.r.t. u. The left-to-right partition of v w.r.t. u is denoted by $\Psi_{lr}(v, u)$. The right-to-left partition of v w.r.t. u is denoted by $\Psi_{rl}(v, u)$. □

Example 12 For the strings $v = cbaabdcb$ and $u = abcba$ of Example 11 we have $\Psi_{lr}(v, u) = \Psi_1$ and $\Psi_{rl}(v, u) = \Psi_2$. □

One can show that $\Psi_{lr}(v, u)$ and $\Psi_{rl}(v, u)$ are of minimal size. Hence, we can conclude $|\Psi_{lr}(v, u)| = \mathit{mmdist}(v, u) = |\Psi_{rl}(v, u)|$. This property leads to a simple algorithm for calculating the maximal matches distance. The partition $\Psi_{lr}(v, u)$ can be computed by scanning the characters of v from left to right, until a prefix wc of v is found such that w is a substring of u, but wc is not. w is the first submatch and c is the first marked character in $\Psi_{lr}(v, u)$. The remaining submatches and marked characters are obtained by repeating the process on the remaining

suffix of v, until all of the characters of v have been scanned. Using the suffix tree of u (see Section 4), the longest prefix w of v that is a substring of u, can be computed in $O(|\mathcal{A}| \cdot |w|)$ time. This gives an algorithm to calculate *mmdist*(v,u) in $O(|\mathcal{A}| \cdot (m+n))$ time and $O(m)$ space.

$\Psi_{rl}(v,u)$ can be computed in a similar way by scanning v from right to left. However, one has to be careful since the reversed scanning direction means to compute the longest prefix of v^{-1} that occurs as substring of u^{-1}. This can, of course, be accomplished by using $ST(u^{-1})$ instead of $ST(u)$.

It is easily verified that *mmdist*$(u,v) = 1$ and *mmdist*$(v,u) = 2$ if v and u are as in Example 11. Hence, *mmdist* is not a metric on $\mathcal{A}^*$. However, one can obtain a metric as follows:

Theorem 3 Let $mmm(u,v) = \log_2((\textit{mmdist}(u,v)+1) \cdot (\textit{mmdist}(v,u)+1))$. mmm is a metric on $\mathcal{A}^*$. □

From the above it is clear that $mmm(u,v)$ can be computed in $O(|\mathcal{A}| \cdot (m+n))$ steps and $O(\max\{m,n\})$ space. Next we study the relation of the maximal matches distance and the unit edit distance. We first show an important relation of alignments and partitions.

Observation 9 Let A be an alignment of v and u. Then there is an $r \in [0, \delta(A)]$, and a partition $(w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ of v w.r.t. u such that w_1 is a prefix and w_{r+1} is a suffix of u.

Proof: By structural induction on A. If A is the empty alignment, then $\delta(A) = 0$, $v = u = \varepsilon$, and the statement holds with $r = 0$ and $w_1 = \varepsilon$. If A is not the empty alignment, then A is of the form $(A', \beta \to \alpha)$ where A' is an alignment of some strings v' and u' and $\beta \to \alpha$ is an edit operation. Obviously, $v = v'\beta$ and $u = u'\alpha$. Assume the statement holds for A'. That is, there is an $r' \in [0, \delta(A')]$ and a partition $(w_1, c_1, \ldots w_{r'}, c_{r'}, w_{r'+1})$ of v' w.r.t. u' such that w_1 is a prefix and $w_{r'+1}$ is a suffix of u'. First note that w_1 is a prefix of u since it is prefix of u'. There are three cases to consider:

- If $\beta = \varepsilon$, then $\alpha \neq \varepsilon$ and $\delta(A) = \delta(A')+1$. Hence, $v = v'\beta = w_1c_1 \ldots w_{r'}c_{r'}w_{r'+1}$. If $w_{r'+1}$ is the empty string, then it is a suffix of $u = u'\alpha$. If $w_{r'+1} = wc$ for some string w and some character c, then $v'\beta = w_1c_1 \ldots w_{r'}c_{r'}wcw'$ where $w' = \varepsilon$ is a suffix of $u = u'\alpha$. Thus, the statement holds with $r = r' + 1 \leq \delta(A)$.

- If $\beta \neq \varepsilon$ and $\alpha \neq \beta$, then $\delta(A) = \delta(A')+1$. Hence, $v = v'\beta = w_1c_1 \ldots w_{r'}c_{r'}w_{r'+1}\beta w$ where $w = \varepsilon$ is a suffix of $u = u'\alpha$. Thus, the statement holds with $r = r'+1 \leq \delta(A)$.

- If $\beta \neq \varepsilon$ and $\alpha = \beta$, then $\delta(A) = \delta(A')$. Let $w = w_{r'+1}\beta$. Then $v = v'\beta = w_1c_1 \ldots w_{r'}c_{r'}w$, and w is a suffix of $u = u'\alpha$ since $w_{r'+1}$ is a suffix of u'. Thus, the statement holds with $r = r' \leq \delta(A)$. □

The following theorem shows that *mmdist*(v,u) is a lower bound for the unit edit distance of v and u.

Theorem 4 Suppose δ is the unit cost function. Then $mmdist(v,u) \leq edist_\delta(v,u)$.

Proof: Let A be an optimal alignment of v and u. Then by Observation 9 there is a partition Ψ of v w.r.t. u such that $|\Psi| \leq \delta(A)$. Hence, $mmdist(v,u) \leq |\Psi| \leq \delta(A) = edist_\delta(v,u)$. □

The relation between *mmdist* and $edist_\delta$ suggests to use *mmdist* as a filter in contexts where the unit edit distance is of interest only below some threshold k. In fact, there are algorithms for the approximate string searching problem (see Section 5) using filtering techniques based on maximal matches.

3.6 The q-Gram Model

Like the maximal matches model, the q-gram model considers common substrings of the strings to be compared. However, while the former model considers substrings of possibly different length, the latter restricts to substrings of a fixed length q. In this section let q be a positive integer. Recall that u and v are sequences of length m and n, respectively.

Definition 17 The *q-gram profile* of u is the function $G_q(u) : \mathcal{A}^q \to \mathbb{N}$, such that $G_q(u)(w)$ is the number of different positions in u where the sequence $w \in \mathcal{A}^q$ ends. □

The parameters q and $|\mathcal{A}|$ are very important for the q-gram distance. For example, if $q = 3$ and $|\mathcal{A}| = 4$, then $|\mathcal{A}|^q = 64$. That is, we can assume that in a short string, all q-gram occur. If $q = 4$ and $|\mathcal{A}| = 20$, then $|\mathcal{A}|^q = 160000$ and the string has to be very long to contain all q-grams. In general, one chooses $q \ll n$, e.g. $q \in [3,6]$ for DNA sequences.

Definition 18 The *q-gram distance* $qgdist(u,v)$ of u and v is defined by

$$qgdist(u,v) = \sum_{w \in \mathcal{A}^q} |G_q(u)(w) - G_q(v)(w)|. \quad \square$$

One can show that the symmetry and the triangle inequality hold for $qgdist$. The zero property does not hold as shown by the following example. Hence, $qgdist$ is not a metric.

Example 13 Let $q = 2$, $u = aaba$ and $v = abaa$. Then u and v have the same q-gram profile $\{aa \mapsto 1, ab \mapsto 1, ba \mapsto 1, bb \mapsto 0\}$. Hence, the q-gram distance of u and v is 0. □

The simplest method to compute the q-gram distance is to encode each q-gram into a number, and to use these numbers as indices into tables holding the counts for the corresponding q-gram.

Definition 19 Let $\mathcal{A} = \{a_1, \ldots, a_r\}$. Then

$$\overline{a_l} = l - 1$$

is the code of a_l and

$$\overline{w} = \sum_{i=1}^{q} \overline{w_i} \cdot r^{q-i}$$

is the code of $w \in \mathcal{A}^q$.

An important property is that the code of each q-gram can be computed incrementally in constant time, due to the fact that $\overline{xc} = (\overline{ax} - \overline{a} \cdot r^{q-1}) \cdot r + \overline{c}$ for any $x \in \mathcal{A}^*$ and any $a, b \in \mathcal{A}$.

The algorithm to compute the q-gram distance follows the following strategy:

1. Accumulate the q-gram profiles of u and v in two arrays τ_u and τ_v such that $\tau_u[\overline{w}] = G_q(u)(w)$ $\tau_v[\overline{w}] = G_q(v)(w)$ for all $w \in \mathcal{A}^q$.
2. Compute the list $C = \{\overline{w} \mid w \text{ is } q\text{-gram of } u \text{ or } v\}$.
3. Compute $qgdist(u,v) := \sum_{c \in C} |\tau_u[c] - \tau_v[c]|$.

Algorithm Computing the q-gram distance
Input: sequences $u = u_1 \ldots u_m$ and $v = v_1 \ldots v_n$
$q > 0$
Output: $qgdist(u,v)$

$r := |\mathcal{A}|$
for $c := 0$ **to** $r^q - 1$ **do**
  $\tau_u[c] := 0$
  $\tau_v[c] := 0$
$c := \sum_{i=1}^{q} \overline{u_i} \cdot r^{q-i}$
$\tau_u[c] := 1$
$C := \{c\}$
for $i := 1$ **to** $m - q$ **do**
  $c := (c - \overline{u_i} \cdot r^{q-1}) \cdot r + \overline{u_{i+q}}$
  if $\tau_u[c] = 0$ **then** $C := C \cup \{c\}$
  $\tau_u[c] := \tau_u[c] + 1$
$c := \sum_{i=1}^{q} \overline{v_i} \cdot r^{q-i}$
$\tau_v[c] := 1$
if $\tau_u[c] = 0$ **then** $C := C \cup \{c\}$
for $i := 1$ **to** $n - q$ **do**
  $c := (c - \overline{v_i} \cdot r^{q-1}) \cdot r + \overline{v_{i+q}}$
  if $\tau_u[c] = 0$ **and** $\tau_v[c] = 0$ **then** $C := C \cup \{c\}$
  $\tau_v[c] := \tau_v[c] + 1$
return $\sum_{c \in C} |\tau_u[c] - \tau_v[c]|$

Let us consider the efficiency of the algorithm. The space for the arrays τ_u and τ_v is $O(r^q)$. The space for the set C is $O(m-q+1+n-q+1) = O(m+n)$. Hence the

total space requirement is $O(m+n+r^q)$. We need $O(r^q)$ time to initialize the arrays τ_u and τ_v. The computation of the codes requires $O(m+n)$ time. Each array lookup and update requires $O(1)$. Hence the total time requirement is $O(m+n+r^q)$. If $r^q \in O(n+m)$, then this method is optimal. There are other techniques to compute the q-gram distance. These are based on suffix trees, see Section 4.

Like the maximal matches distance, the q-gram distance provides a lower bound for the unit edit distance.

Theorem 5 Let δ be the unit cost function. Then $qgdist(u,v)/(2 \cdot q) \leq edist_\delta(u,v)$.

3.7 The Fasta Similarity Model

This model is based on the Fasta-program, which is a very popular tool for comparing biological sequences. First consider the problem the Fasta-program was designed for: Let w be a *query sequence* (e.g. a novel DNA-sequence or an unknown protein). Let S be a set of sequences (the database) and $k \geq 0$ be a threshold value. The problem is to find all sequences in S, whose similarity to w is at least k.

We now need to define the similarity notion used by Fasta. Consider for each $u \in S$, the corresponding matrix E_δ defined by $E_\delta(i,j) = edist_\delta(u_1 \ldots u_i, w_1 \ldots w_j)$ where δ is the unit cost function. The idea is to count for each diagonal the number of minimizing subpaths of length q on this diagonal. Each such minimizing subpath stands for a common q-gram in u and w. This number gives a score, according to the following definition:

Definition 20 Let $m = \min\{|u|,|w|\}$ and $n = \max\{|u|,|w|\}$. For $d \in [-m,n]$ let

$$count(d) = |\{(i,j) \mid j - i = d \text{ and } u_i \ldots u_{i+q-1} = w_i \ldots w_{i+q-1}\}|$$

The Fasta score is now defined by $score_{\text{fasta}}(u,w) = \max\{count(d) \mid d \in [-m,n]\}$. □

Example 14 Let $w =$ `freizeit`, $u =$ `zeitgeist`, and $q = 2$. Then $count(-4) = 3$, $count(-1) = 1$, $count(0) = 1$, $count(3) = 1$ and $count(d) = 0$ for $d \notin \{-4,-1,0,3\}$. This can be easily verified in Figure 3.9. □

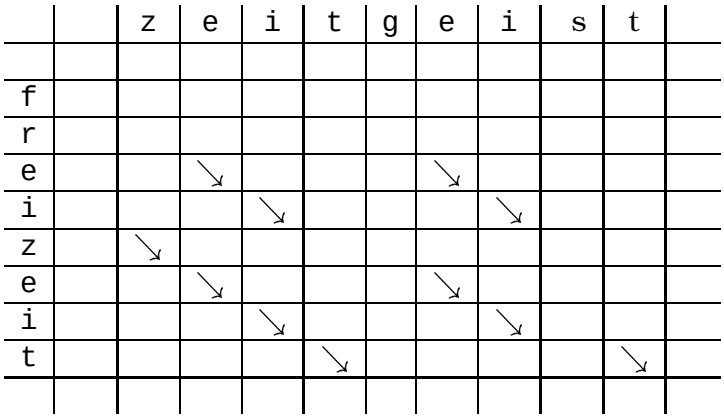
Note that only the subpaths on the same diagonal are counted. In other words, the matching q-grams have to be at the same distance in both u and v. This is the main difference to the q-gram distance model, where the order of the q-grams is not important.

We now sketch an algorithm to compute $score_{\text{fasta}}(u,w)$.

1. Encode each q-gram as an integer $c \in [0, r^q - 1]$, where $r = |\mathcal{A}|$. The details of this encoding are described in Section 3.6.

2. The query sequence w is preprocessed into a function $h : [0, r^q - 1] \to \mathcal{P}(\mathbb{N})$ defined by
$$h(c) := \{i \in [1, |w| - q + 1] \mid c = \overline{w_i \ldots w_{i+q-1}}\}$$
That is, each "bucket" $h[c]$ holds the positions in w where the q-gram with code c occurs.

Figure 3.9: The matching diagonals for `freizeit` and `zeitgeist`



3. In the final phase, the data base is processed.

foreach $u \in S$
$n := |u|$
for $d := -m$ **to** n **do** $count(d) = 0$
for $j := 1$ **to** $n - q + 1$ **do**
$c := \overline{u_j \dots u_{j+q-1}}$
foreach $i \in h(c)$
$count(j - i) := count(j - i) + 1$
$score_{\text{fasta}}(u, w) := \max\{count(d) \mid d \in [-m, n]\}$
if $score_{\text{fasta}}(u, w) \geq k$ **then** $print$ "u and w are similar"

The running time of this algorithm is clearly $O(r^q + m + n + \sum_{d=-m}^{n} count(d))$ for one database sequence u. That is, the more similar w and u the more time the algorithm requires.

It is often not sufficient to just know that the two sequences under consideration are similar. One also would like to know where the similarities are. Therefore, in an earlier version of the Fasta-program, an alignment is constructed which contains a maximal number of matching q-grams. New versions of the Fasta-program simply apply the Smith-Waterman algorithm to the sequences u and v, whenever $score_{\text{fasta}}(u, w) \geq k$. Thus the Fasta-score serves as a heuristic filter of the database search.

3.8 The BlastP Similarity Model

This similarity model is based on the Blast-program which is perhaps the most popular program to perform sequence database searches. Here we will describe the program for the case where the input sequences are proteins (hence the name BlastP). We will restrict ourselves to an older version of Blast which was used until about 1998 (Blast 1.4). The newer version of Blast (Blast 2.0) is more complicated.

Suppose we want to search a protein sequence database, given a score function σ, satisfying $\sigma(\alpha \to \beta) = -\infty$ for any deletion or insertion operation $\alpha \to \beta$. That is, the model does not allow for insertions and deletions.

Definition 21 Let $q \in \mathbb{N}$ and $k \geq 0$ be a threshold. Two sequences u and w are *similar in the Blast model* if there is a pair (i, j) (called hit) such that $score_\sigma(u_i \dots u_{i+q-1}, w_j \dots w_{j+q-1}) \geq k$. □

In practice, w is the query sequence and u is a sequence from a database, e.g. from SWISSPROT (version 39.11 of 8. December 2000 contains 91131 sequences of total length 33,206,837).

We now sketch an algorithm to find the hits between the query sequence w of length m and a database sequence u of length n. This algorithm is iterated over all u in the database.

1. In the first step, we construct the following set:

$$M = \{(s, i) \mid s \in \mathcal{A}^q, i \in [1, |w| - q + 1], score_\sigma(s, w_i \dots w_{i+q-1}) \geq k\}$$

2. Then we construct a deterministic finite automaton (DFA) which accepts exactly the set $S = \{s \mid \exists i \in \mathbb{N}, (s, i) \in M\}$. Identify the accepting state corresponding to s by s. For each $s \in S$, the DFA stores the set $I_s := \{i \mid (s, i) \in M\}$ with state s.

3. The string v is then processed using the DFA. Whenever, the DFA reaches an accepting state, say s, after processing j characters, then $(i, j - q + 1)$ is a hit for all $i \in I_s$.

The first two steps only depend on the query sequence w. Hence they only have to be performed once for the database.

The size of the DFA grows exponentially with q and $1/k$. So these parameters should be selected carefully. For protein sequences, $q = 4$ and $k = 17$ (if σ is the PAM250-scorefunction) seem to be a reasonable choice.

Once the DFA has been constructed v is processed in $O(|v|)$ time.

Finding the hits is only one step in the BlastP-program. Each hit (i, j) is separately extended to the left and to the right. The user specifies a "drop-off" parameter X_d according to which the extension is stopped. For the extension to the left the sequences $u_1 \dots u_{i-1}$ and $w_1 \dots w_{j-1}$ are compared from right to left. For the extension to the right the sequences $u_{i+q} \dots u_m$ and $w_1 \dots w_n$ are compared from left to right. Each pair of characters (u_{i-l}, w_{j-l}), $l \in [1, \min\{i - 1, j - 1\}]$, and (u_{i+q+r}, w_{j+q+r}), $r \in [0, \min\{m - q - i, n - q - j\}]$, delivers a score according to the score function σ. For both extensions the scores are accumulated and the maximum value X reached during the extension is kept track of. As soon as a score smaller than $X - X_d$ is reached, the extension is stopped. The pair of sequences around the hit delivered by this extension is called maximum segment pair (MSP). For any such MSP, a significance score is computed. If this is better than some predefined significance threshold, then the MSP is reported.

CHAPTER 4

Suffix Trees

4.1 Motivation

The amount of sequence information in today's data bases is growing very fast. This is especially true for the domain of genomics: The current and future projects to sequence large genomes (e.g. human, mouse, rice) produce gigabytes and will soon produce terabytes of sequence data, mainly DNA sequences and Protein sequences derived from the former. To make use of these sequences, larger and larger instances of string processing problems have to be solved. While the sequence databases are growing rapidly, the sequence data they already contain does not change much over time. As a consequence, this domain is very suitable for applying indexing methods. These methods preprocess the sequences in order to answer queries much faster than methods that work sequentially. Apart from the size of the string processing problems in genomics, their diversity is also remarkable. For example, to assemble the Genome of Drosophila melanogaster, a multitude of string processing problems involving exact and approximate pattern matching tasks had to be solved. These problems are often complicated by additional constraints about uniqueness, containment, or repetitiveness of sequences. *Suffix trees* are perfectly well suited to solve such problems. In this chapter, we will introduce the concept of suffix trees, show their most important properties, and take a look at some applications of biological relevance.

4.2 The Concept of Suffix Trees

Let us consider a *sequence* S over some alphabet $\mathcal{A}$. Think of S as being a database of sequences concatenated (e.g. Swissprot) or a genome, etc. We will mostly use the synonym *string* for sequence.

Problems on strings are often formulated like this:

... enumerate all substrings of S satisfying the property ...

In order to answer such queries it would be helpful to have an index of all substrings of S. To pursue this idea further, we first should consider how many substrings exist. For example, if $S = caacacacca$, then there are 40 different substrings. Since the alphabet is so small, many of them occur more than once. Now assume that S is of length n. One can show that a string of length n has at most n^2 different substrings. There are examples where this number is exact, e.g. if $S =$ *abcdef* (no character appears more than once). There are examples where there are only n substrings, e.g. if $S = aaaaaaa$. But n^2 is a good estimation in practice. As a consequence of these considerations we see that there are too many substrings of S to represent them all *explicitly*. We need an *implicit* representation.

4.3 An Informal Introduction to Suffix Trees

It is easy to see that each substring of S is a prefix of a suffix of S. And S has only n suffixes. So consider an index consisting of all suffixes of S. But let us append a unique character \$ to the end of S, so that no suffix is a prefix of another suffix. For example, if $S = abab$, then we get an index consisting of all suffixes of $abab$\$:

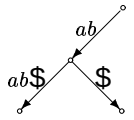
$$\$, b\$, ab\$, bab\$, abab\$$$

Unfortunately, the index consisting just of the suffixes of S\$ does not support queries like

"does u occur only once as a substring of S?"

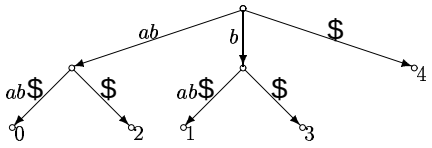
To solve the above minimal unique substring problem, we need to augment the set of suffixes with more structure. The idea is to share common prefixes between the suffixes. Let us consider this idea systematically for $S = abab$:

- suffixes $abab$\$ and ab\$ have a common prefix ab, and this is the longest common prefix. The remaining suffixes after dropping ab are ab\$ and \$. Let us draw a tree T_1 to illustrate this:



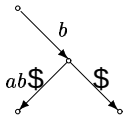
  T_1 consists of just four nodes and three edges; the $root$ node at the top, a branching node reached by an edge from the $root$, and two leaves reached by edges from the latter. The edges are marked by non-empty substrings of S\$. T_1 represents all prefixes of the suffixes $abab$\$ and ab\$, i.e. the set $Sub_1 = \{a, ab, aba, abab, abab\$, ab\$\}$ of substrings of S\$. Each string in this set can be found by following a path in T_1, reading the labels and stopping at a node or anywhere between two nodes. For example, the string a can be found as follows: follow the path from the $root$ and stop reading the label after the first character. The string ab can be found by continuing to read the label on the edge outgoing from the $root$ completely.

Figure 4.1: The suffix tree for $S = abab$ with leaf annotations. These are easily verified given the starting positions of the suffixes of $abab\$$ as depicted on the right.



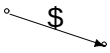
a	b	a	b	\$
0	1	2	3	4

- suffixes $bab\$$ and $b\$$ have a common prefix b, and this is the longest common prefix. The remaining suffixes after dropping b are $ab\$$ and \$. Let us draw a similar tree T_2 as above, to illustrate this:



  It should be clear now that T_2 represents all prefixes of the suffixes $bab\$$ and $b\$$, i.e. the set $Sub_2 = \{b, ba, bab, bab\$, b\$\}$ of substrings of $S\$$.

- There is only one suffix of $S\$$ which we have not considered yet, namely suffix \$. This has no common prefix with any other suffix, so we can represent it by a tree T_3 consisting of a $root$ node, a leaf and an edge labeled by \$:



  So T_3 represents the set $Sub_3 = \{\$\}$ of substrings of $S\$$.

Notice that the sets Sub_1, Sub_2, Sub_3 have no common element. The union of these sets is clearly the set of all substrings of $S\$$. So if we join the trees T_1, T_2, T_3 by giving them a common $root$, we obtain a tree representing all substrings of $S\$$. This is what we call the *suffix tree of* $S\$$. It is shown in Figure 4.1. (Ignore the numbers on the leaves for the moment. We will need them later.)

4.4 A Formal Introduction to Suffix Trees

We do not want to introduce the properties of suffix trees all at once. So we first introduce the raw material for suffix trees: An *$\mathcal{A}^+$-tree* T is a finite rooted tree with the following properties

(1) the edges are labeled by non-empty strings over alphabet $\mathcal{A}$.

(2) for every node α in T and each $a \in \mathcal{A}$, there is only one a-edge $\alpha \xrightarrow{av} \beta$ for some string v and some node β.

An $\mathcal{A}^+$-tree is more general than a suffix tree, but several notions important for suffix trees can already be introduced on $\mathcal{A}^+$-trees. Let T be an $\mathcal{A}^+$-tree. It consists of different types of nodes and edges: A node in T is *branching* if it has at least two outgoing edges. A *leaf* in T is a node in T with no outgoing edges. An *internal node* in T is either the $root$ or a node with at least one outgoing edge. An edge leading to an internal node is an *internal edge*. An edge leading to a leaf is a *leaf edge*.

$path(\alpha)$ denotes the concatenation of the edge labels on the path from the $root$ of T to the node α. Due to the requirement of unique a-edges at each node of T (see property (2) above), paths are also unique. Therefore, we denote v by $\overline{w}$ if and only if $path(v) = w$. The node $\overline{\varepsilon}$ is the $root$. (Remember that ε denotes the empty string). For any node $\overline{w}$ in T, $|w|$ is the *depth* of $\overline{w}$.

A string w *occurs* in T if T contains a node $\overline{wu}$, for some (possibly empty) string u. In other words, if we can extend a string w by some (possibly empty) string u and $\overline{wu}$ is a node in the suffix tree, then w occurs in the tree. This is equivalent, but more formal than stating "there is a path for w" as we did before.

If w occurs in T, then we also say that T represents w. $occ(T)$ denotes the set of strings occurring in T. Let $w \in occ(T)$. An $\mathcal{A}^+$-tree is *compact* if every node is either the $root$, a leaf, or a branching node. In other words, in a compact suffix $\mathcal{A}^+$, we do not allow nodes which only have one successor (except for the $root$, but this exception is not important).

From now on we assume that $S \in \mathcal{A}^+$ is a string of length $n \geq 1$. The *suffix tree* for S, denoted by $ST(S)$, is the compact $\mathcal{A}^+$-tree T s.t. $occ(T) = \{w \in \mathcal{A}^* \mid w \text{ is a substring of } S\}$. Thus the suffix tree for S represents exactly the substrings of S.

4.5 The Role of the Sentinel Character

Suppose T is the suffix tree for S. Then the following property holds: If $\overline{w}$ is a leaf in T, then w is a suffix of S. The reverse of this property does not hold: Consider, for example, the suffix tree for $acbcabcac$. Here the suffixes ac and c do not correspond to a leaf. This is because the suffixes are nested. If we restrict to suffixes which are not nested, then we can characterize the strings corresponding to leaves in the suffix tree: $\overline{w}$ is a leaf in T if and only if w is a suffix of S and w is not nested. Now what is the corresponding characterization for the branching nodes? To clarify it, we introduce the following notion: A substring w of S is *right-branching* if and only if there are different characters a and b such that wa and wb are substrings of S. Now the following holds: $\overline{w}$ is a branching node in T if and only if w is a right-branching substring of S.

Suffix trees are often only defined for strings which end with a character not occurring elsewhere in the strings. Such a character, usually the symbol \$, is called *sentinel*. With a sentinel at the end, there are no nested suffixes, and the one-to-one correspondence of leaves and suffixes is easier to express.

4.6 The Size of Suffix Trees

Due to the above considerations it is clear that the number of leaves in the suffix tree is bounded by n where n is the length of $|S|$. For example, the suffix tree for $abab\$$ in Figure 4.1 has 5 leaves. By definition, each internal node is branching or it is the $root$. Thus there cannot be more than $n-1$ internal nodes. For each node, except for the $root$ there is exactly one edge leading to this node. Vice versa, for each edge there is exactly one node it points to. As a consequence, the number of edges is one less than the number of nodes. So if there are at most $2n-1$ nodes in the suffix tree, there are at most $2n-2$ edges. Since the nodes are not labeled, we can surely represent each node in constant space. Each edge is labeled by a substring $S_i \dots S_j$ of S. Thus we do not need to store a copy of that substring. A pair (i, j) of integers suffices, to refer to $S_i \dots S_j$. As a consequence, each edge can be stored in constant space.

Altogether we need constant space for each of at most $2n-1$ nodes and at most $2n-2$ edges. Thus the space requirement for the suffix tree is $O(n)$. In Section 4.8, we will in more detail consider how to represent suffix trees in more detail.

4.7 Suffix Tree Constructions

We start with a suffix tree construction method that is fast in practice and easy to explain. In Sections 4.7.2 and 4.7.3 we explain two linear time suffix tree construction methods.

4.7.1 The Write Only Top Down Suffix Tree Construction

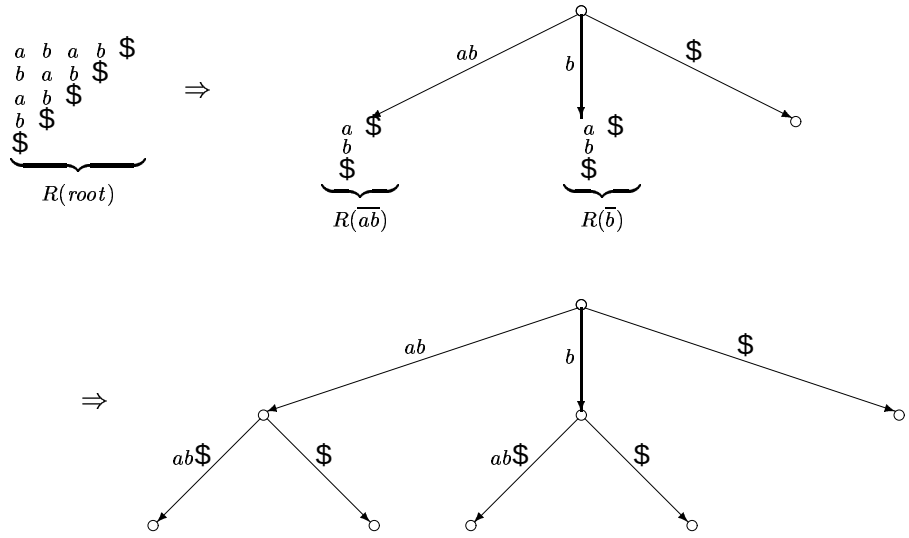
In this subsection, we assume that the input sequence ends with a sentinel. This is not really necessary, but it simplifies the explanation.

The *wotd*-algorithm adheres to the recursive structure of a suffix tree. The idea is that for each branching node $\overline{u}$ the subtree below $\overline{u}$ is determined by the set of all suffixes of $S\$$ that have u as a prefix. In other words, if we have the set $R(\overline{u}) := \{s \mid us \text{ is a suffix of } S\$\}$ of *remaining suffixes* available, we can evaluate the node $\overline{u}$. This works as follows: at first $R(\overline{u})$ is divided into groups according to the first character of each suffix. For any character $c \in \mathcal{A}$, let $group(\overline{u}, c) := \{w \in \mathcal{A}^* \mid cw \in R(\overline{u})\}$ be the *c-group of* $R(\overline{u})$. If for some $c \in \mathcal{A}$, $group(\overline{u}, c)$ contains only one string w, then there is a leaf edge labeled cw outgoing from $\overline{u}$. If $group(\overline{u}, c)$ contains at least two strings, then there is an edge labeled cv leading to a branching node $\overline{ucv}$, where v is the longest common prefix (*lcp*, for short) of all strings in $group(\overline{u}, c)$. The child $\overline{ucv}$ can then be evaluated from the set $R(\overline{ucv}) = \{w \mid vw \in group(\overline{u}, c)\}$ of remaining suffixes.

The *wotd*-algorithm starts by evaluating the $root$ from the set $R(root)$ of all suffixes of $S\$$. All nodes of the suffix tree can be evaluated recursively from the corresponding set of remaining suffixes in a top-down strategy.

Example 15 Consider the input string $S = abab$. The *wotd*-algorithm for $S\$$ works as follows: At first, the $root$ is evaluated from the set $R(root)$ of all non-empty suffixes of the string $S\$$, see the first five columns in Figure 4.2. The

Figure 4.2: The write-only top-down construction of the suffix tree for $abab$



algorithm recognizes three groups of suffixes. The a-group, the b-group, and the \$-group. The a-group and the b-group each contain two suffixes, hence we obtain two unevaluated branching nodes, which are reached by an a-edge and by a b-edge. The \$-group is singleton, so we obtain a leaf reached by an edge labeled \$. To evaluate the unevaluated branching node corresponding to the a-group, one first computes the longest common prefix of the remaining suffixes of that group. This is b in our case. So the a-edge from the *root* is labeled by ab, and the remaining suffixes ab\$ and \$ are divided into groups according to their first character. Since this is different, we obtain two singleton groups of suffixes, and thus two leaf edges outgoing from $\overline{ab}$. These leaf edges are labeled by ab\$ and \$. The unevaluated branching node corresponding to the b-group is evaluated in a similar way, see Figure 4.2. □

The worst case running time of the *wotd*-algorithm is $O(n^2)$. Consider, for example, the string $S = a^n$. The suffix tree for S\$ is a binary tree with exactly one branching node of depth i for any $i \in [0, n-1]$. To construct the branching node of depth i, exactly $n - i$ suffixes are considered. That is, the number of steps is:

$$\sum_{i=0}^{n-1} n - i = n^2 - \sum_{i=0}^{n-1} i = n^2 - \frac{n(n-1)}{2} = \frac{2n^2 - n^2 + n}{2} = \frac{n^2 + n}{2} \in O(n^2)$$

In the expected case, the maximal depth of the branching nodes is much smaller than $n-1$, namely $O(\log_{|A|} n)$. In other words, the length of the path to the deepest branching node in the suffix tree is $O(\log_{|A|} n)$. The suffixes along the leaf edges are not read any more. Hence the expected running time of the *wotd*-algorithm is $O(n \log_{|A|} n)$. Note that the *wotd*-algorithm has some nice properties, which make it interesting in practice:

- The subtrees of the suffix trees are constructed independently from each other. Hence the algorithm can easily be parallelized. Moreover, the locality behavior is very good: Due to the write-only-property, the construction of the subtrees only depends on the set of remaining suffixes. Thus the data required to construct the subtrees is very small. As a consequence, it often fits into the cache. This makes the algorithm fast in practice since a cache access is much faster than the access to the main memory. In a lot of cases the *wotd*-algorithm is faster than the linear time suffix tree construction we will explain in Sections 4.7.2 and 4.7.3.
- The paths in the suffix tree are constructed in the order they are searched, namely top-down. Thus one could construct a subtree only when it is traversed for the first time. This would result in a "lazy construction" which could also be implemented in an eager imperative language, like C. Experiments show that such a lazy construction is very fast.

4.7.2 The Linear Time Online Construction of Ukkonen

This algorithm constructs $ST(S)$ *online*, i.e. it generates a sequence of suffix trees

$$ST(\varepsilon), ST(S_1), ST(S_1S_2), \ldots, ST(S_1S_2 \ldots S_n)$$

for all prefixes of S. Here $ST(\varepsilon)$ is the *empty suffix tree* which consists only of root. The method is called *online* since in each step the suffix trees are constructed without knowing the remaining part of the input string. In other words, the algorithm may read the input string character by character from left to right.

Since we know the first suffix tree, we only have to consider the step from

$$ST(S_1 \ldots S_i) \text{ to } ST(S_1 \ldots S_iS_{i+1}) \tag{4.1}$$

for some $i \in [0, n-1]$. Now let i be arbitrary but fixed, and define $x := S_1 \ldots S_i$, $a := S_{i+1}$, and $y = S_{i+2} \ldots S_n$. When the algorithm is in step i, it has read x and a, but y is not known. Therefore, let us call xa the *visible part*, and y the *hidden part* of S. In terms of functional programming, y can be thought of as a lazy list, that is, an unevaluated list expression. As the online algorithm proceeds, more and more characters of S become visible, that is, more and more of y is evaluated.

By definition, $ST(xa)$ represents all substrings of xa, and $ST(x)$ represents all substrings of x. Thus in step (4.1) we have to add all substrings of xa which are not substrings of x. Let us call this set of strings *INSERT*. The following simple observations about *INSERT* help in developing the algorithm of Ukkonen:

Observation 10 For all $w \in$ *INSERT* there is a suffix s of x such that $w = sa$.

Proof: We know that w is not empty, since the empty string already occurs in $ST(x)$. Hence w is a suffix of xa since otherwise it would be a substring of x and thus occur in $ST(x)$. Thus the claim follows. □

Observation 11 For all $sa \in$ *INSERT* we have: $\overline{sa}$ is a leaf in $ST(xa)$.

Proof: To prove this property, assume that $\overline{sa}$ is not a leaf in $ST(xa)$. Then sa is a nested suffix of xa. This implies that sa occurs at least twice as a substring of xa.

Hence sa is a substring of x, and hence it occurs in $ST(x)$. This is a contradiction. Thus the assumption was wrong. □

Let us split *INSERT* into two disjoint subsets *INSERTleaf* and *INSERTrelevant* where

- *INSERTleaf* $= \{sa \in$ *INSERT* $\mid \overline{s}$ is a leaf in $ST(x)\}$
- *INSERTrelevant* $= \{sa \in$ *INSERT* $\mid \overline{s}$ is not a leaf in $ST(x)\}$

Now we show how to insert the substrings in the two different sets. Let us start with *INSERTleaf*. Suppose $sa \in$ *INSERTleaf*. Then $\overline{s}$ is a leaf in $ST(x)$. We insert sa without removing anything by extending the corresponding leaf edge. That is, by extending the label of the leaf edge $\overline{u} \xrightarrow{v} \overline{s}$ by a we obtain the leaf edge $\overline{u} \xrightarrow{va} \overline{sa}$. In other words, to insert all elements in *INSERTleaf* we have to extend all leaf edges in $ST(x)$ by the new character a. One of the basic ideas of Ukkonen's algorithm is to represent the leaf edges such that they automatically grow whenever a new character is read from the input.

Definition 22 An *open edge* in $ST(xa)$ is a leaf edge with a label vay representing the suffix va of xa. (Recall that y is the "unevaluated" remaining part of the input string.) For Ukkonen's algorithm all leaf edges are open edges. □

When implementing Ukkonen's algorithm in a lazy language, y could be a lazy (unevaluated) list. In an imperative language one would implement the label of a leaf edge by a pair of an integer, say i, and a pointer to the memory cell where the length of the input already read is stored. In this way, an increment of the input length would automatically mean that all leaf edges implicitly grow by one character.

Due to open edges, the labels of leaf edges grow while more and more characters become visible. Thus, to insert a new suffix sa into $ST(x)$, nothing must be done when $\overline{s}$ is a leaf. Hence, we only consider the complementary case (i.e. *INSERTrelevant*), when $\overline{s}$ is not a leaf in $ST(x)$, or equivalently s is a nested suffix of x.

Definition 23 A suffix sa of xa is *relevant* if s is a nested suffix of x and sa is not a substring of x. □

The step from $ST(x)$ to $ST(xa)$ now means the following: Insert all relevant suffixes sa of xa into $ST(x)$. To make this description more precise, we study some properties of relevant suffixes. In particular, we show that the relevant suffixes of xa form a contiguous segment of the list of all suffixes of xa, whose bounds are marked by "active suffixes":

Definition 24 The *active suffix* of x, denoted by $\alpha(x)$, is the longest nested suffix of x. □

Example 16 Consider the string $adcdacdad$ and a list of columns, where each column contains the list of all suffixes of a prefix of this string. The relevant suffixes in each column are marked by the symbol $\downarrow$ and the active suffix is printed in bold face.

ε	$\downarrow a$	ad	adc	$adcd$	$adcda$	$adcdac$	$adcdacd$	$adcdacda$	$adcdacdad$
	ε	$\downarrow d$	dc	dcd	$dcda$	$dcdac$	$dcdacd$	$dcdacda$	$dcdacdad$
		ε	$\downarrow c$	cd	cda	$cdac$	$cdacd$	$cdacda$	$cdacdad$
			ε	$\boldsymbol{d}$	$\downarrow da$	dac	$dacd$	$dacda$	$dacdad$
				ε	$\boldsymbol{a}$	$\downarrow ac$	acd	$acda$	$acdad$
					ε	$\boldsymbol{c}$	$\boldsymbol{cd}$	$\boldsymbol{cda}$	$\downarrow cdad$
						ε	d	da	$\downarrow dad$
							ε	a	$\boldsymbol{ad}$
								ε	d
									ε

Observation 12

1. For all suffixes s of x: s is nested $\iff |\alpha(x)| \geq |s|$.
2. For all suffixes s of x: sa is a relevant suffix of $xa \iff |\alpha(x)a| \geq |sa| > |\alpha(xa)|$.
3. $\alpha(xa)$ is a suffix of $\alpha(x)a$.
4. If $sa = \alpha(xa)$ and $\alpha(x)a \neq sa$, then s is a right-branching substring of x.

Proof:

1. Routine.
2. sa is a relevant suffix of $xa \iff s$ is a nested suffix of x and sa is not a substring of $x \iff |\alpha(x)| \geq |s|$ and sa is not a nested suffix of $xa \iff |\alpha(x)a| \geq |sa|$ and $|sa| > |\alpha(xa)| \iff |\alpha(x)a| \geq |sa| > |\alpha(xa)|$.
3. Since both $\alpha(xa)$ and $\alpha(x)a$ are suffixes of xa, it suffices to show $|\alpha(x)a| \geq |\alpha(xa)|$. If $\alpha(xa) = \varepsilon$, then this is obviously true. Let $\alpha(xa) = wa$. Since wa is a nested suffix of xa, we have $uwav = x$ for some strings u and v. Hence, w is a nested suffix of x. Since $\alpha(x)$ is the longest nested suffix of x, we have $|\alpha(x)| \geq |w|$ and hence $|\alpha(x)a| \geq |wa| = |\alpha(xa)|$.
4. Suppose $sa = \alpha(xa)$ and $\alpha(x)a \neq sa$. Then there is a suffix csa of xa such that $|\alpha(x)a| \geq |csa| > |\alpha(xa)|$. From Statement 2 we know that csa is a relevant suffix of xa. That is, cs is a nested suffix of x, and csa is not a substring of x. Hence, there is a character $b \neq a$ such that csb is a substring of x. Since sa is a substring of x, too, s is a right-branching substring of x. □

By Statement 2 the relevant suffixes of xa are "between" $\alpha(x)a$ and $\alpha(xa)$. Hence, by Statement 3, $\alpha(xa)$ is the longest suffix of $\alpha(x)a$ that is a substring of x. Based on this fact, the step from $ST(x)$ to $ST(xa)$ can be described as follows:

> Take the suffixes of $\alpha(x)a$ one after the other by decreasing length and insert them into $ST(x)$, until a suffix is found which occurs in the tree and therefore equals $\alpha(xa)$.

More formally this could be stated by the following pseudo-code:
$v := \alpha(x)a$
while v *does not occur in* $ST(x)$ **do**
    insert v *in* $ST(x)$
    $v :=$ *drop* 1 v
$\alpha(xa) := v$

Note that $\sum_{i=1}^{n-1} |\alpha(S_1 \dots S_i)S_{i+1}| - |\alpha(S_1 \dots S_iS_{i+1})| \leq n$. That is, the total number of all relevant suffixes is bounded by n. For the previous refinement we have to implement the following operations:

(1) decide if v occurs in $ST(x)$

(2) insert v in $ST(x)$

(3) drop the first character from v

These operations are executed $O(n)$ times. Thus we would obtain a linear time algorithm, if we could implement each operation in constant time. Note that either $v = \varepsilon$ or $v = sa$ for some string s occurring in $ST(x)$. The idea now is to represent v by the appropriate edges and nodes of $ST(x)$. In this way, operations (1) and (2) can be implemented in constant time. The second idea is to construct for each branching node, say $\overline{aw}$, a *suffix link* which points to branching node $\overline{w}$ (if it exists). See Figure 4.3, for an example, where also the leaves have suffix links. Using suffix links, (3) can be realized in constant time.

Definition 25 Let T be a compact $\mathcal{A}^+$-tree and $s \in occ(T)$. The *location* of s in T, denoted by $loc_T(s)$ is defined as follows:

- If $\overline{s}$ is a branching node, then $loc_T(s) = \overline{s}$.
- If $\overline{s}$ is a leaf, then there is a leaf edge $\overline{u} \xrightarrow{v} \overline{s}$ in T and $loc_T(s) = (\overline{u}, v, \varepsilon, \overline{s})$.
- If there is no node $\overline{s}$ in T, then there is an edge $\overline{u} \xrightarrow{vw} \overline{uvw}$ in T such that $s = uv$, $v \neq \varepsilon$, $w \neq \varepsilon$ and $loc_T(s) = (\overline{u}, v, w, \overline{uvw})$. If a location is a node, we call it *node location*, otherwise *edge location*. Sometimes we identify a node location with the corresponding node. □

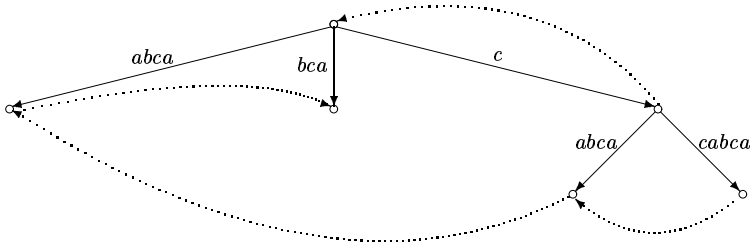
It is easy to see that a location can be represented in constant space.

Example 17 Let T be the compact $\mathcal{A}^+$-tree shown in Figure 4.3. Then, for instance,

$$\begin{array}{lcl} loc_T(\varepsilon) & = & root \\ loc_T(a) & = & (root, a, bca, \overline{abca}) \\ loc_T(abca) & = & (root, abca, \varepsilon, \overline{abca}) \\ loc_T(c) & = & \overline{c} \\ loc_T(cab) & = & (\overline{c}, ab, ca, \overline{cabca}) \ \square \end{array}$$

Definition 26 For any $\mathcal{A}^+$-tree T we define the following operations on locations:

Figure 4.3: An $\mathcal{A}^+$-tree with Suffix Links



1. $occurs(loc_T(s), a) \iff sa$ occurs in T. This operation can be implemented in constant time.

2. $getloc(loc_T(s), w) = loc_T(sw)$ for all $sw \in occ(T)$. This operation can be implemented in $O(|w|)$ time.

3. Insertion of say: $T[loc_T(s) \leftarrow ay]$ delivers the pair $(T', \overline{z})$ which is specified as follows:

   - If $loc_T(s) = \overline{s}$, then T' is obtained from T by adding a leaf edge $\overline{s} \xrightarrow{ay} \overline{say}$. Moreover, $\overline{z} = \perp$ which should be read as "$\overline{z}$ is undefined".
   - If $loc_T(s) = (\overline{u}, v, w, \overline{uvw})$, then T' is obtained from T by splitting the edge $\overline{u} \xrightarrow{vw} \overline{uvw}$ into $\overline{u} \xrightarrow{v} \overline{s} \xrightarrow{w} \overline{uvw}$, and adding a new leaf edge $\overline{s} \xrightarrow{ay} \overline{say}$. Moreover, $\overline{z} = \overline{s}$, that is, $\overline{z}$ is the new inner node created by the splitting.

4. Linking locations via suffix links:

$$linkloc(\overline{s}) = \overline{z}$$
where $\overline{s} \longrightarrow \overline{z}$ is the suffix link for $\overline{s}$

$$linkloc(\overline{u}, av, w, \overline{uavw}) = \begin{cases} loc_T(v) & \text{if } \overline{u} = root \\ getloc(\overline{z}, av) & \text{otherwise} \end{cases}$$
where $\overline{u} \longrightarrow \overline{z}$ is the suffix link for $\overline{u}$. □

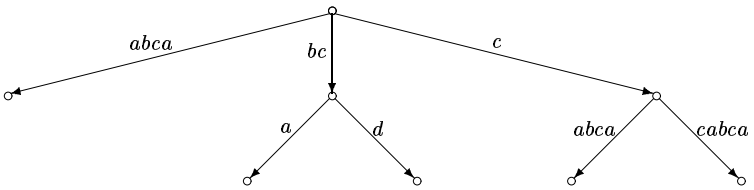
Example 18 Let T be the compact $\mathcal{A}^+$-tree of Figure 4.3 and $(T', \overline{bc}) = T[loc_T(bc) \leftarrow d]$. The compact $\mathcal{A}^+$-tree T' is shown in Figure 4.4. □

Observation 13 Let T be a compact $\mathcal{A}^+$-tree such that the suffix links for all branching nodes in T are defined. Suppose that cy and y occur in T. Then $linkloc(loc_T(cy)) = loc_T(y)$. □

We are now ready to define the algorithm of Ukkonen. The function $ukkstep$ implements the construction of $ST(xa)$ from $ST(x)$.

Definition 27 The function $ukkstep$ is defined as follows:

Figure 4.4: The Result of an Insert-Operation



$$ukkstep(T, L, ay, \overline{z}, loc) = \begin{cases} (T, L', getloc(loc, a)) & \text{if } occurs(loc, a) \\ (T', L', loc) & \text{else if } loc = root \\ ukkstep(T', L', ay, \overline{r}, linkloc(loc)) & \text{otherwise} \end{cases}$$

where $(T', \overline{r}) = T[loc \leftarrow ay]$

$$L' = \begin{cases} L & \text{if } \overline{z} = \bot \\ L \cup \{\overline{z} \longrightarrow loc\} & \text{else if } occurs(loc,a) \text{ or } \overline{r} = \bot \\ L \cup \{\overline{z} \longrightarrow \overline{r}\} & \text{otherwise} \end{cases}$$

Here the parameters of $ukkstep$ satisfy the following properties:

- T is the current $\mathcal{A}^+$-tree.
- L is the set of suffix links.
- a is the current input character.
- y is the remaining input string.
- $\overline{z}$ is the node for which the suffix link has to be set, or $\overline{z} = \bot$.
- loc is the location of s in T, where sa is a suffix of xa and $|\alpha(x)a| \geq |sa| \geq |\alpha(xa)|$.

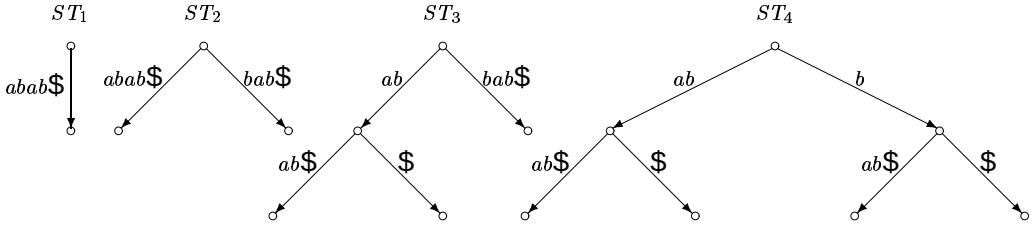
Note that for a new inner node $\overline{z}$ the suffix link $\overline{z} \longrightarrow \overline{r}$ cannot be set instantly, since $\overline{r}$ may not exist yet. However, $\overline{r}$ will be created in the next call to $ukkstep$. Therefore, $\overline{z}$ is taken as an argument of $ukkstep$ and the setting of the suffix link is delayed until $\overline{r}$ is constructed.

To complete the algorithm we iterate $ukkstep$, as defined by the function ukk:

$$\begin{aligned} ukk(T, L, \varepsilon, loc) &= (T, L) \\ ukk(T, L, ay, loc) &= ukk(T', L', y, loc') \\ &\quad \text{where } (T', L', loc') = ukkstep(T, L, ay, \bot, loc) \end{aligned}$$

Theorem 6 Let $S \in \mathcal{A}^n$. Then $ukk(ST(\varepsilon), \emptyset, S, root)$ returns $(ST(S), L)$ in $O(n)$ time and space, where L is the set of suffix links for all branching nodes of $ST(S)$. □

Figure 4.5: The sequence of $\mathcal{A}^+$-Trees constructed by *mcc* for $abab\$$. The last tree, ST_5, is omitted, since it is identical to the suffix tree shown in Figure 4.1.



4.7.3 The Linear Time Construction of McCreight

The suffix tree construction of McCreight is the classical method. It is also linear but not online and slightly faster than Ukkonen's algorithm.

McCreight's algorithm requires that the input string S ends with a unique sentinel character \$. So let us consider the suffix tree construction for the string $S\$$. For $i \in [1, n+1]$, let x_i be the suffix of $S\$$ starting at position i in $S\$$. Let ST_0 be the empty $\mathcal{A}^+$-tree. For any $i \in [1, n+1]$, let ST_i be the compact $\mathcal{A}^+$-tree such that

$$occ(ST_i) = \{w \in \mathcal{A}^* \mid w \text{ is a prefix of } x_j \text{ for some } j \in [1, i]\}$$

Notice that $ST_{n+1} = ST$. Let $head_1 = \varepsilon$ and for $i \in [2, n+1]$ let $head_i$ be the longest prefix of x_i which is also a prefix of x_j for some $j \in [1, i-1]$. $tail_i$ denotes the remaining suffix of x_i, i.e. $tail_i$ satisfies $head_i tail_i = x_i$. We obviously have $tail_{n+1} = \$$ and $tail_i \neq \varepsilon$ for any $i \in [1, n+1]$. One can show that there is a branching node $\overline{v} \neq root$ in ST_i if and only if $v = head_j \neq \varepsilon$ for some $j \in [2, i]$.

The *head*'s are represented by locations and the *tail*'s by pointers into the input string $S\$$. We therefore define $headloc_i = loc_{ST_{i-1}}(head_i)$ and $tailptr_i \in [1, n+1]$ such that $tail_i = S_{tailptr_i} \dots S_n\$$ for any $i \in [1, n+1]$.

The general structure of McCreight's algorithm is to construct ST by successively inserting the suffixes of $S\$$ into an initially empty tree, from longest to shortest. More precisely, the algorithm constructs the following sequence of compact $\mathcal{A}^+$-trees:

$$ST_1, ST_2, \ \dots\ , ST_n, ST_{n+1} \tag{4.2}$$

Figure 4.5 shows this sequence for the input string $abab\$$.

Additionally, with each ST_i, $i \in [1, n]$, the algorithm constructs the suffix links for all nodes $\overline{head_j} \neq root$, $j \in [2, i-1]$. In other words, the only branching node in ST_i whose suffix link is possibly not constructed, is node $\overline{head_i}$. Note furthermore that there is no suffix link for the $root$.

ST_1 is the compact $\mathcal{A}^+$-tree with only one edge $root \xrightarrow{S\$} \overline{x_1}$. This can easily be constructed in constant time. In order to compute ST_i for $i \in [2, n+1]$, it is crucial to compute $headloc_i$ and $tailptr_i$. Once this has been done, ST_i can be constructed by a single tree insertion operation. More precisely, we have $(ST_i, \overline{r}) = ST_{i-1}[headloc_i \leftarrow tail_i]$, where $\overline{r} = \perp$ or $\overline{r}$ is the node for which the suffix link has not been constructed yet.

The following function *mccstep* specifies how the construction of ST_i from ST_{i-1} works:

$$
\begin{array}{l}
mccstep(T, L, loc, cy, \overline{z}) \\
\quad = (T', L', loc'', y', \overline{r}) \\
\qquad \text{where } loc' = linkloc(loc) \\
\qquad\qquad (T', \overline{r}) = T[loc'' \leftarrow y'] \\
\qquad\qquad ((loc'', y'), L') = \left\{ \begin{array}{ll}
(scanprefix(loc, y), L) & \text{if } loc = root \\
(scanprefix(loc', cy), L) & \text{else if } loc \text{ is a node} \\
(scanprefix(loc', cy), L \cup \{\overline{z} \longrightarrow loc'\}) & \text{else if } loc' \text{ is a node} \\
((loc', cy), L \cup \{\overline{z} \longrightarrow \overline{r}\}) & \text{otherwise}
\end{array} \right.
\end{array}
$$

The parameters of *mccstep* satisfy the following properties:

1. T is ST_{i-1} and $T' = ST_i$.

2. L is the set of suffix links for all branching nodes in ST_{i-2}, and L' is the set of suffix links for all branching nodes in ST_{i-1}.

3. $loc = loc_{ST_{i-1}}(head_{i-1})$ and $loc' = loc_{ST_i}(head_i)$.

4. $cy = tail_{i-1}$.

5. $\overline{z}$ is the branching node created during the previous construction step, or $\overline{z} = \bot$.

It remains to define the function *scanprefix*:

Definition 28 Let T be an $\mathcal{A}^+$-tree. For each $s \in occ(T)$ and each string w the function *scanprefix* is specified as follows: $scanprefix(loc_T(s), w) = (loc_T(su), v)$, where $uv = w$ and u is the longest prefix of w such that $su \in occ(T)$. □

Example 19 Let T be the $\mathcal{A}^+$-tree as shown in Figure 4.3. Then, for instance,

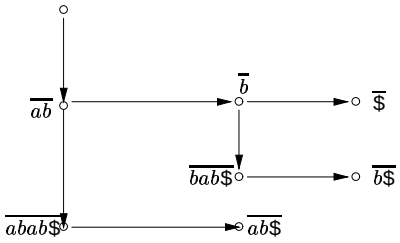
$$
\begin{array}{rcl}
scanprefix(loc_T(cab), cd) & = & (loc_T(cabc), d) \\
scanprefix(loc_T(bca), d) & = & (loc_T(bca), d) \quad \square
\end{array}
$$

McCreight's algorithm is specified by the function *mcc*.

$$
mcc(T, L, loc, cy, \overline{z}) = \left\{ \begin{array}{ll}
(T, L) & \text{if } y = \varepsilon \text{ and } loc = root \\
mcc(mccstep(T, L, loc, cy, \overline{z})) & \text{otherwise}
\end{array} \right.
$$

Theorem 7 Let $S \in \mathcal{A}^n$. Then $mcc(ST_1, \emptyset, root, S\$, \bot)$ returns $(ST(S\$), L)$ in $O(n)$ time and space, where L is the set of suffix links for all branching nodes of $ST(S\$)$. □

Figure 4.6: The references of the suffix tree for $x = abab$ (see Figure 4.1). Vertical arcs stand for *firstchild* references, and horizontal arcs for *branchbrother* and T_{leaf} references.



4.8 Representing Suffix Trees

In this section, we describe an implementation technique for suffix trees which is space efficient and allows a linear time construction using the algorithms of Ukkonen or of McCreight.

Again we assume that the suffix tree ends with a sentinel. $ST(S\$)$ is represented by two tables T_{leaf} and T_{branch} which store the following values: For each *leaf number* $j \in [1, n+1]$, $T_{leaf}[j]$ stores a reference to the right brother of leaf $\overline{x_j}$. If there is no such brother, then $T_{leaf}[j]$ is a nil reference. For each branching node $\overline{w}$, $T_{branch}[\overline{w}]$ stores a *branch record* consisting of five components *firstchild*, *branchbrother*, *depth*, *suffixpos*, and *suffixlink* whose values are specified as follows:

1. *firstchild* refers to the first child of $\overline{w}$
2. *branchbrother* refers to the right brother of $\overline{w}$. If there is no such brother, then *branchbrother* is a nil reference.
3. *depth* is the depth of $\overline{w}$
4. *suffixpos* is some $j \in [1, n+1]$ such that w is a prefix of x_j.
5. *suffixlink* refers to the branching node $\overline{v}$, if w is of the form av for some $a \in \mathcal{A}$ and some $v \in \mathcal{A}^*$

The successors of a branching node are therefore found in a list whose elements are linked via the *firstchild*, *branchbrother*, and T_{leaf} references. To speed up the access to the successors, each such list is ordered according to the first character of the edge labels. Figure 4.6 shows the child and brother references of the nodes of the suffix tree of Figure 4.1. We use the following notation to denote a record component: For any component c and any branching node $\overline{w}$, $\overline{w}.c$ denotes the component c stored in the branch record $T_{branch}[\overline{w}]$. Note that the suffix position j of some branching node $\overline{wu}$ tells us that the leaf $\overline{x_j}$ occurs in the subtree below node $\overline{wu}$. Hence wu is the prefix of x_j of length $\overline{wu}.depth$, i.e. the equality $wu = S_j \ldots S_{j+\overline{wu}.depth-1}$ holds. As a consequence, the label of the incoming edge

Figure 4.7: The tables T_{leaf} and T_{branch} representing the suffix tree for $x = abab$ (see Figure 4.1). A bold face number refers to table T_{leaf}.

T_{leaf}					
leaf	$\overline{abab\$}$	$\overline{bab\$}$	$\overline{ab\$}$	$\overline{b\$}$	$\overline{\$}$
leaf number j	1	2	3	4	5
$T_{leaf}[j]$	**3**	**4**	nil	nil	nil

T_{branch}			
branching node	*root*	$\overline{ab}$	$\overline{b}$
node number	1	2	3
firstchild	2	**1**	**2**
branchbrother	nil	3	**5**
depth	0	2	1
suffixpos	1	3	4
suffixlink		3	1

to node $\overline{wu}$ can be obtained by dropping the first $\overline{w}.depth$ characters of wu, where $\overline{w}$ is the predecessor of $\overline{wu}$:

Observation 14 If $\overline{w} \xrightarrow{u} \overline{wu}$ is an edge in $ST(S\$)$ and $\overline{wu}$ is a branching node, then we have $u = S_i \ldots S_{i+l-1}$ where $i = \overline{wu}.suffixpos + \overline{w}.depth$ and $l = \overline{wu}.depth - \overline{w}.depth$. □

Similarly, the label of the incoming edge to a leaf is determined from the leaf number and the depth of the predecessor:

Observation 15 If $\overline{w} \xrightarrow{u} \overline{wu}$ is an edge in $ST(S\$)$ and $\overline{wu} = \overline{x_j}$ for some $j \in [1, n+1]$, then $u = S_i \ldots S_n\$$ where $i = j + \overline{w}.depth$. □

Note that storing the depth of a branching node has some practical advantages over storing the length of the incoming edge to a node. At first, during the suffix tree constructions of Ukkonen and of McCreight, the depth of a node never changes. So it is not necessary to update the depth of a node (the same is true for the suffix position). Second, the depth of the nodes along a chain of suffix links is decremented by one, a property which can be exploited to store a suffix tree more space efficiently. The third advantage of storing the depth is that several applications of suffix trees assume that the depth of a node is available.

The references *firstchild*, *branchbrother*, and $T_{leaf}[j]$ can be implemented as integers in the range $[0, n+1]$. An extra bit with each such integer tells whether the reference is to a leaf or to a branching node. Each leaf $\overline{x_j}$ is referred to by leaf number j. Suppose there are q branching nodes in $ST(S\$)$. Let $b_1, b_2, \ldots, b_q$ be the sequence of branching nodes as generated during the Ukkonen's or McCreight's suffix tree construction. Each branching node b_i is referred to by it's node number i. Obviously, b_1 is the *root*. Figure 4.7 depicts T_{leaf} and T_{branch} for the suffix tree of Figure 4.1.

Like the references, the other components of the branch records can be implemented by an integer in the range $[0, n+1]$. Thus table T_{leaf} requires n integers and table T_{branch} requires $5q$ integers. The total space requirement of this suffix tree implementation is $n+5q$ integers. In the worst case we have $q = n$, so that the

implementation technique requires $6n$ integers. However, q is usually considerably smaller than n ($q = 0.62n$ is the theoretical average value for random strings). So in practice, we usually achieve a requirement of less than $4n$ integers.

4.9 Suffix Tree Applications

4.9.1 Searching for Exact Patterns

Since the suffix tree for $S\$$ contains all substrings of $S\$$, it is easy to verify whether some pattern string $P \in \mathcal{A}^+$ is a substring of S: just follow the path from the root directed by the characters of P. If at some point you cannot proceed with the next character in P, then P does not occur in the suffix tree and hence it is not a substring of S. Otherwise, if P occurs in the suffix tree, then it also is a substring of S. We, of course, assume that \$ does not occur in P.

Example 20 Let $S = abab$. The corresponding suffix tree is shown in Figure 4.1. Suppose $P = abb$ is the pattern. Then we read the first two characters of P and follow the left edge from the root. This leads to the branching node $\overline{ab}$. However, there is no b-edge outgoing from $\overline{ab}$, and hence we cannot proceed matching P against the suffix tree. In other words, P does not occur in the suffix tree and hence it is not a substring of S. Now suppose $P = baa$. Then we follow the second edge from the root to node $\overline{b}$. This has an a-edge leading to a leaf. The next character to read in P is the last character which is a. It does not match the next character on the leaf edge. Hence P does not occur in the suffix tree, i.e. it is not a substring of S. □

The algorithm to match a pattern string P against the suffix tree runs in $O(|P|)$ time, since for each character we can check in constant time if we can proceed in the suffix tree. Thus the running time does not depend on the size of the input string S. Of course one has to invest some time and space to construct the suffix tree. But this should pay off if we are looking for many patterns in a fixed input sequence, like e.g. a genome.

If we have found that a pattern occurs in the suffix tree, we can also find the positions in S where P occurs. We only have to annotate the leaves of the suffix tree. Recall that a leaf $\overline{u\$}$ corresponds to the suffix $u\$$ of $S\$$. $\overline{u\$}$ has a leaf number, that is, the position where the corresponding suffix $u\$$ starts. If the comparison of P against $ST(S\$)$ ends in an edge, then we go to the node, the edge leads to. If the comparison ends on a node, then we stay at that node. Suppose $\overline{w}$ is the node we reached in that way. We now enumerate all leaf numbers of the leaves in the subtree below $\overline{w}$. These leaf numbers are exactly those positions in S where P starts.

Example 21 Suppose $P = ab$ and assume the suffix tree of Figure 4.1. We follow the first edge from the root which brings us to the branching node $\overline{ab}$. Thus P is a substring of S. The leaf numbers in the subtree below $\overline{ab}$ are 0 and 2, and indeed ab starts in $S = abab$ at positions 0 and 2. □

There are many variations of this pattern search algorithm. One variation is to search for the longest prefix of P, say P', that is a substring of S. This can clearly be done in $O(|P'|)$ time. It is useful for computing the maximal matches distance, see Section 3.5.

4.9.2 Minimal Unique Substrings

This problem has applications in primer design. It consists of enumerating all substrings u of some string S satisfying the following properties:

1. u occurs exactly once in S (uniqueness),
2. all proper prefixes of u occur at least twice in S (minimality),
3. u is of length at least l for some given fixed parameter l.

Example 22 Let $S = abab$ and $l = 2$. Then the minimal unique substrings are aba and ba. Note that bab is *not* a minimal unique substring, since the proper prefix ba of bab is already unique, i.e. the minimality condition does not hold. □

To solve the minimal unique substring problem, we exploit two properties of the suffix tree of $S\$$:

- if a string w occurs at least twice in S, there are at least two suffixes in $S\$$ of which w is a proper prefix. Hence in the suffix tree $ST(S\$)$, w corresponds to a path ending with an edge to a branching node.
- if a string w occurs only once in S, there is only one suffix in $S\$$ of which w is a prefix. Hence in the suffix tree $ST(S\$)$, w corresponds to a path ending with an edge to a leaf.

According to the second property, we can find the unique strings by looking at the paths ending on the edges to a leaf. So if we have reached a branching node, say w, then we only have to enumerate the leaf edges outgoing from w. Suppose $w \xrightarrow{av} y$ is an edge from w leading to a leaf y, and av is the edge label, such that a is the first character on that edge. Then wa occurs only once in S, i.e. it is unique. Moreover, w corresponds to a path leading to a branching node, and by the first property, w occurs at least twice. Finally we only have to check if the length of wa is at least l. Thus the suffix tree based algorithm to solve the minimal unique substring problem is very simple.

Example 23 Let us apply the algorithm to the suffix tree of Figure 4.1, and assume $l = 2$. We can skip the root, since it would result in strings which are too short. Let us consider the branching node reached by the edge from the root labeled ab. Then $w = ab$ and with the first character a of the label of the first edge we obtain the minimal unique substring aba. The other solution ba can be found by looking at the other branching node reached by the label b from the root together with its first edge. □

The running time of this simple algorithm is linear in the number of nodes and edges in the suffix tree, since we have to visit each of these only once and for each we do a constant amount of work. The algorithm thus runs in linear time since the suffix tree can be constructed in linear time, and there are $O(n)$ nodes and edges in the suffix tree. This is optimal, since the running time is linear in the size of its input.

4.9.3 Maximal Unique Matches

The standard dynamic programming algorithm to compute the optimal alignment between two sequences of length m and n requires $O(mn)$ steps. This is too slow if the sequences are on the order of 100000 or millions of characters.

There are other methods which allow to align two genomes under the assumption that these are fairly similar. The basic idea is that the similarity often results in long identical substrings which occur in both genomes. These identities, called MUMs (for maximal unique matches) are almost surely part of any good alignment of the two genomes. So the first step is to find the MUMs. These are then taken as the fixed part of an alignment and the remaining parts of the genomes (those parts not included in a MUM) are aligned with traditional dynamic programming methods. In this section, we will show how to compute the MUMs in linear time. This is very important for the applicability of the method. We do not consider how to compute the final alignment. We first have to define the notion MUM precisely.

Suppose we are given sequences $S, S' \in \mathcal{A}^*$ (the genomes) and a positive integer l. The *maximal unique matches problem* (MUM-problem) is to find all sequences u with the following properties:

- $|u| \geq l$.
- u occurs exactly once in S and it occurs exactly once in S' (uniqueness).
- For any character a neither ua nor au occurs both in S and in S' (maximality).

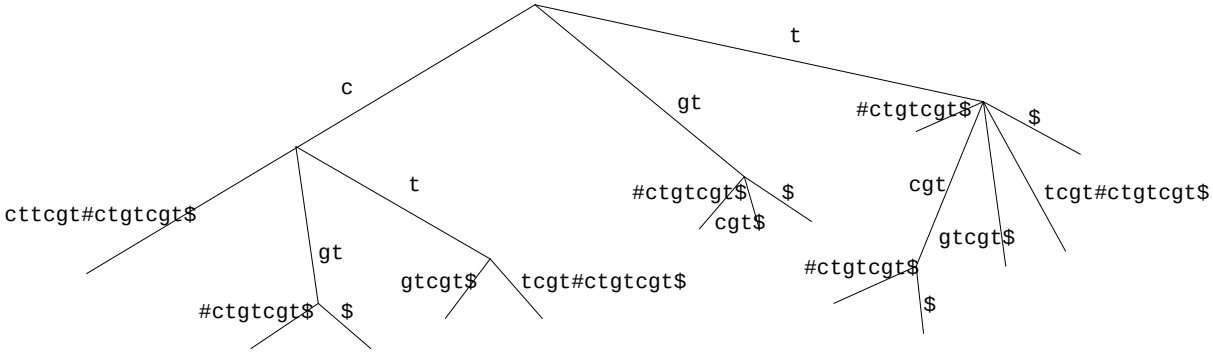
Example 24 Let $S = ccttcgt$, $S' = ctgtcgt$, and $l = 2$. Then there are two maximal unique matches ct and $tcgt$. Now consider an optimal alignment of these two sequences (assuming the same costs for insertions, deletions, and replacements):

```
cct-tcgt
-ctgtcgt
```

Clearly the two MUMs ct and $tcgt$ are part of this alignment. □

To compute the MUMs, we first have to construct the suffix tree for the concatenation of the two sequences S and S'. To prevent from considering any match that occurs on the borderline between S and S', we put a unique symbol # between S and S', i.e. we construct the suffix tree for $X = S\#S'\$$. Now observe that a MUM, say u, must occur exactly twice in X, once in S and once in S'. Hence u corresponds to a path in the suffix tree $ST(X)$ ending with an edge to a branching node. Since u is right-maximal by definition (i.e. for any symbol a, ua does not

Figure 4.8: The suffix tree for *ccttcgt#ctgtcgt*$ without the leaf edges from the root



occur both in S and in S'), u must even correspond to a branching node. In other words, for each MUM u there exists a branching node $\overline{u}$ in the suffix tree for X.

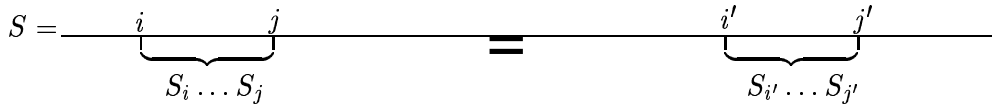
Since u occurs twice in X, there are exactly two leaves in the subtree below $\overline{u}$. The subtree can thus contain no branching node, and hence there are two leaf edges outgoing from $\overline{u}$. One edge must lead to a leaf, say $\overline{v}$, that corresponds to a suffix starting in S and the other edge must lead to a leaf, say $\overline{w}$, that corresponds to a suffix starting in S'. Given a branching node $\overline{u}$, the existence of exactly two such leaf edges can easily be checked. What remains is to verify left-maximality, i.e. to check if there is a character a such that au occurs both in S and in S'. But this is easy: Suppose that leaf $\overline{v}$ has leaf number i and leaf $\overline{w}$ has leaf number j. Then u is left maximal, if and only if $i = 0$ or $X_{i-1} \neq X_{j-1}$. In other words, we only have to look at the two positions immediately to the left of the two positions in X where u starts.

Example 25 Let $S = ccttcgt$, $S' = ctgtcgt$, and $l = 2$. Consider the suffix tree for $S\#S'\$$. This is shown in Figure 4.8. Obviously, the string $tcgt$ occurs once in S and S', since there are two corresponding edges from branching node $\overline{tcgt}$. Comparing the characters g and t immediately to the left of the occurrences of $tcgt$ in S and S' verifies left-maximality. The string ct also occurs once in S and once in S', as verified by the two leaf edges from $\overline{ct}$. The left-maximality is obvious, since the characters c and # to the left of the occurrences are different. □

4.9.4 Maximal Repeats

In this section we show how to compute all maximal repeats of a given input sequence. We generally assume that the input sequence S is indexed from 0 to $n-1$, i.e. $S = S_0 \dots S_{n-1}$. A substring $S_i \dots S_j$ of S is represented by the pair (i, j). A pair (l, r) of different substrings $l = (i, j)$ and $r = (i', j')$ in S is a *repeat* if $i < i'$ and $S_i \dots S_j = S_{i'} \dots S_{j'}$. l is the *left instance* of the repeat and r is the *right instance*.

See the following figure for an illustration.

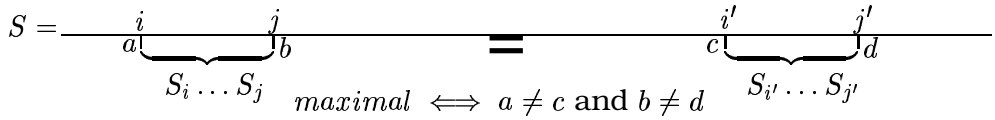


Note that the left instance and the right instance of a repeat may overlap.

Example 26 $S = gagctcgagc$ contains the following repeats of length ≥ 2:

$((0,3),(6,9))$	$gagc$
$((0,2),(6,8))$	gag
$((0,1),(6,7))$	ga
$((1,3),(7,9))$	agc
$((2,3),(8,9))$	gc

Example 26 reveals that shorter repeats are often contained in longer repeats. To remove redundancy, we restrict to *maximal repeats*. A repeat is *maximal* if it is *left maximal* and *right maximal*, where these notions are defined as follows: A repeat $((i,j),(i',j'))$ is *left maximal* if and only if $i-1<0$ or $S_{i-1} \neq S_{i'-1}$. A repeat $((i,j),(i',j'))$ is *right maximal* if and only if $j'+1>n-1$ or $S_{j+1} \neq S_{j'+1}$. The maximality notion is illustrated in the following figure:



From now on we want to restrict ourselves to maximal repeats. All repeats which are not maximal can easily be obtained from the maximal repeats. In Example 26, the last four repeats can be extended to the left or to the right. Hence only the first repeat is maximal.

In the following we will present an algorithm to compute all maximal repeats. It works in two phases. In the first phase, the leaves of the suffix tree are annotated and in the second phase the repeats are output while simultaneously the branching nodes are annotated.

We will show how the algorithm works for the input string x$ggcgc$y$gcgcc$z. The corresponding suffix tree (with some unimportant edges left out) is shown in Figure 4.9.

Now suppose we have the suffix tree for some string S of length n over some alphabet $\mathcal{A}$ such that the first and the last character of S both occur exactly once in S (as in Figure 4.9). We ignore leaf edges from the root, since the root corresponds to repeats of length zero, and we are not interested in these. In the first phase the algorithm annotates each leaf of the suffix tree: if $v = S_i \dots S_n$, then the leaf $\overline{v}$ is annotated by the pair (a,i), where i is the position where the suffix v starts and $a = S_{i-1}$ is the character to the immediate left of that position. (a,i) is the *leaf annotation* of $\overline{v}$. We also write $A(\overline{v}, S_{i-1}) = \{i\}$ to denote the annotation, and assume $A(\overline{v}, c) = \emptyset$ (the empty set) for all characters $c \in \mathcal{A}$ different from S_{i-1}. The latter assumption holds in general (also for branching nodes), whenever

Figure 4.9: The suffix tree for x*ggcgc*y*gcgcc*z. Leaf edges from the root are not shown. These edges are not important for the algorithm.

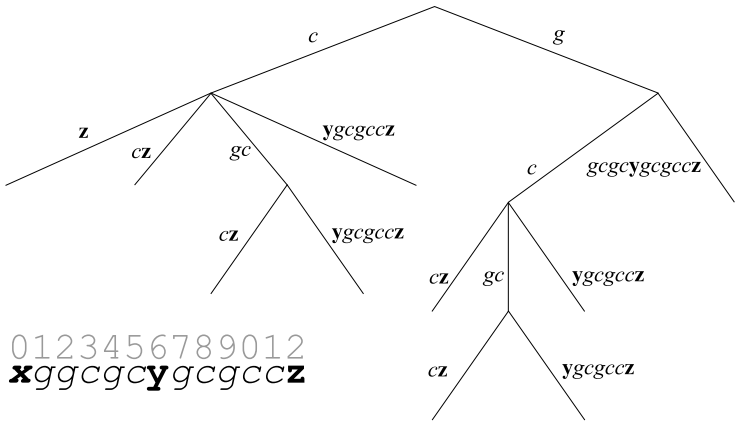
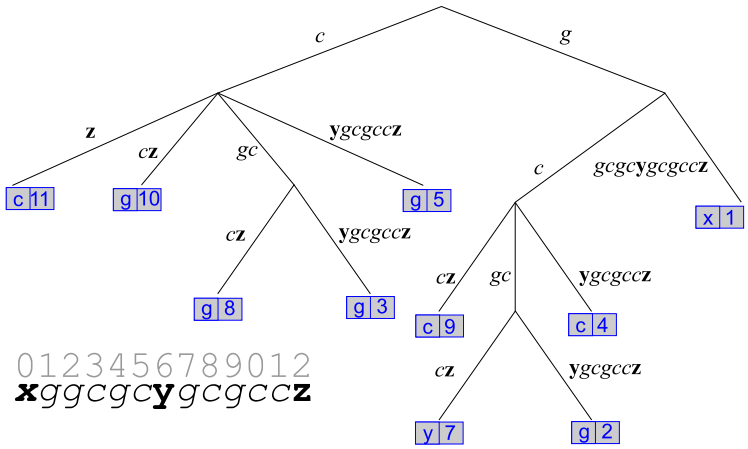


Figure 4.10: The suffix tree for x*ggcgc*y*gcgcc*z with leaf annotation.



there is no annotation (c, j) for some j. For the suffix tree of Figure 4.9, the leaf annotation is shown in Figure 4.10.

The leaf annotation gives us the character upon which we decide the left-maximality of a repeat, plus a position where a repeated string occurs. We only have to combine this information at the branching nodes appropriately. This is done in the second phase of the algorithm: In a bottom-up traversal the repeats are output and simultaneously the annotation for the branching nodes is computed. A bottom-up traversal means that a branching node is visited only after all nodes in the subtree below that node have been visited. Each edge, say $\overline{w} \xrightarrow{au} \overline{v}$, is processed as follows: At first repeats (for w) are output by combining the annotation already computed for node $\overline{w}$ with the complete annotation stored for $\overline{v}$ (this was already computed due to the bottom-up strategy). In particular, we output all pairs $((i, i+q-1), (j, j+q-1))$, where

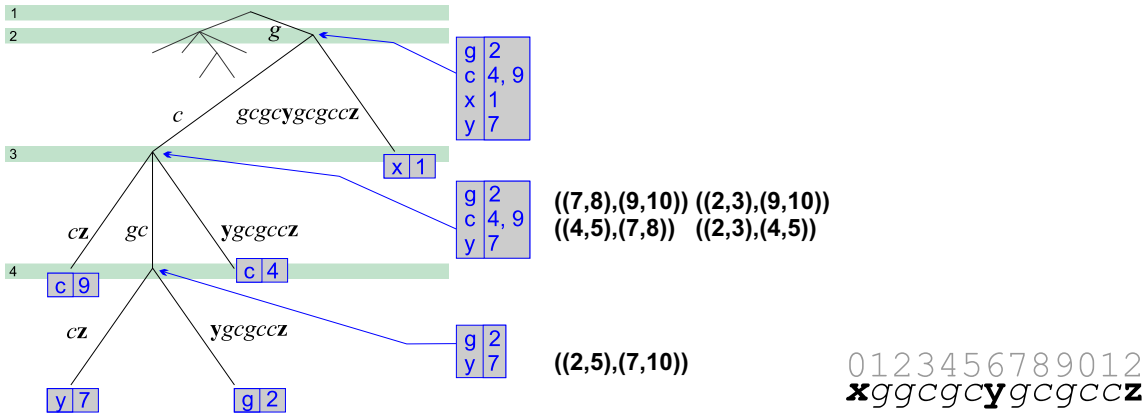
1. q is the depth of node $\overline{w}$, i.e. $q = |w|$,
2. $i \in A(\overline{w}, c)$ and $j \in A(\overline{v}, c')$ for some characters $c \neq c'$,
3. $A(\overline{w}, c)$ is the annotation already computed for $\overline{w}$ w.r.t. character c and $A(\overline{v}, c')$ is the annotation stored for node $\overline{v}$ w.r.t. character c'.

The second condition means that only those positions are combined which have different characters to the left. Thus it guarantees left-maximality of the repeats. Recall that we consider processing the edge $\overline{w} \xrightarrow{au} \overline{v}$. The annotation already computed for $\overline{w}$ was inherited along edges outgoing from $\overline{w}$, that are different from $\overline{w} \xrightarrow{au} \overline{v}$. Thus the first character of the label of such an edge, say c, is different from a. Now since w is the repeated substring, c and a are characters to the right of w. As a consequence, only those positions are combined which have different characters to the right. In other words, the algorithm also guarantees right maximality of the repeats.

As soon as for the current edge the repeats are output, the algorithm computes the union $A(\overline{w}, c) \cup A(\overline{v}, c)$ for all characters c, i.e. the annotation is inherited from node $\overline{v}$ to node $\overline{w}$. In this way, after processing all edges outgoing from $\overline{w}$, this node is annotated by the set of positions where w occurs, and this set is divided into (possibly empty) disjoint subsets $A(\overline{w}, c_1), \ldots, A(\overline{w}, c_r)$, where $\mathcal{A} = \{c_1, \ldots, c_r\}$.

Example 27 Figure 4.11 shows the annotation for a large part of the previous suffix tree, and some repeats. The bottom up traversal of the suffix tree for x*ggcgc*y*gcgcc*z begins with node $\overline{gcgc}$ of depth 4, before it visits node $\overline{gc}$ of depth 2. The maximal repeats for the string gc are computed as follows: The algorithm starts by processing the first edge outgoing from $\overline{gc}$. Since initially there is no annotation for $\overline{gc}$, no repeat is output, and $\overline{gc}$ is annotated by $(c, 9)$. Then the second edge is processed. This means that the annotation $(g, 2)$ and $(y, 7)$ for $\overline{gcgc}$ is combined with the annotation $(c, 9)$. This gives the repeats $((7, 8), (9, 10))$ and $((2, 3), (9, 10))$. The new annotation for $\overline{gc}$ becomes $(c, 9)$, $(y, 7)$, $(g, 2)$. Finally, the third edge is processed. $(c, 9)$ and $(c, 4)$ cannot be combined, see condition 2 above. So only the repeats $((4, 5), (7, 8))$ and $((2, 3), (4, 5))$ are output, resulting

Figure 4.11: The annotation for a large part of the suffix tree of Figure 4.10 and some repeats.



from the combination of $(y, 7)$ and $(g, 2)$ with $(c, 4)$. The final annotation for $\overline{gc}$ is $(c, 9)$, $(y, 7)$, $(g, 2)$, $(c, 4)$ which can also be read as $A(\overline{gc}, c) = \{4, 9\}$, $A(\overline{gc}, y) = \{7\}$, and $A(\overline{gc}, g) = \{2\}$. □

Let us now consider the running time of the algorithm. Traversing the suffix tree bottom-up can surely be done in time linear in the number of nodes, since each node is visited only once and we only have to follow the paths in the suffix tree. There are two operations performed during the traversal: Output of repeats and combination of annotations. If the annotation for each node is stored in linked lists, then the output operation can be implemented such it runs in time linear in the number of repeats. Combining the annotations only involves linking lists together, and this can be done in time linear in the number of nodes visited during the traversal. Recall that the suffix tree can be constructed in $O(n)$ time. Hence the algorithm requires $O(n + z)$ time where n is the length of the input string and z is the number of repeats.

To analyze the space consumption of the algorithm, first note that we do not have to store the annotations for all nodes all at once. As soon as a node and its father has been processed we no longer need the annotation. As a consequence, the annotation only requires $O(n)$ space. Hence the space consumption of the algorithm is $O(n)$.

Altogether the algorithm is optimal, since its space and time requirement is linear in the size of the input plus the output.

CHAPTER 5

Approximate String Matching

We consider the approximate string searching problem. Given a pattern $p \in \mathcal{A}^*$ of length m and an input string $t \in \mathcal{A}^*$ of length n, it consists of finding the positions in t where an approximate match ends. These positions are referred to as solutions of the approximate string searching problem. An approximate match is a substring p' of t such that $edist_\delta(p', p) \leq k$, for a given threshold value $k \in \mathbb{R}^+$.

The approximate string searching problem is of special interest in biological sequence analysis. For instance, when searching a DNA database (the input string) for a query (the pattern), a small but significant error must be allowed, to take into account experimental inaccuracies as well as small differences in DNA among individuals of the same or related species. Note that in biological context, especially when dealing with proteins, the cost function plays an important role. It provides a simple way to consider knowledge about biological phenomena on the amino acid level. That is, by choosing an appropriate cost function, one can select those matches which make biological sense, and reject others which do not.

Computer scientists have mainly focused on the k-differences problem. This is the approximate string searching problem, restricted to the unit cost function (each edit operation has cost 1). We will show two algorithms for this special problem in Sections 5.4, 5.3, and 5.5.

5.1 Sellers Algorithm

By a slight modification of the dynamic programming algorithm for computing the edit distance (see Section 3.2.2), Sellers obtained a simple method to solve the approximate string searching problem. Sellers' method is usually described by giving the recurrence for an $(m+1) \times (n+1)$-table. Our approach is slightly different. We specify Sellers algorithm by an initial distance column and a function

that transforms one distance column into the next distance column, according to some character b.

Definition 29 C denotes the set of functions $f : \{0, \ldots, m\} \to \mathbb{R}^+$ where $\mathbb{R}^+$ is the set of non-negative real numbers. The elements of C are *columns*. We define a function $nextdcol : C \times \mathcal{A} \to C$ as follows. For all $f \in C$ and all $b \in \mathcal{A}$, $nextdcol(f, b) = f_b$ where $f_b(0) = 0$ and

$$f_b(i+1) = \min \left\{ \begin{array}{l} f_b(i) + \delta(p_{i+1} \to \varepsilon), \\ f(i) + \delta(p_{i+1} \to b), \\ f(i+1) + \delta(\varepsilon \to b) \end{array} \right\}$$

Moreover, we define a function $dcol : \mathcal{A}^* \to C$ by $dcol(vb) = nextdcol(dcol(v), b)$ and $dcol(\varepsilon) = f_\varepsilon$ where $f_\varepsilon(0) = 0$ and $f_\varepsilon(i+1) = f_\varepsilon(i) + \delta(p_{i+1} \to \varepsilon)$. $dcol(v)$ is the *distance column* of v. $f \in C$ is a distance column if $f = dcol(v)$ for some $v \in \mathcal{A}^*$. □

Sellers algorithm evaluates for each character in t a "distance column" of $m + 1$ entries. If the last entry in the jth distance column is at most k, then an approximate match ending at position j in t is found.

Algorithm Compute $dcol(\varepsilon)$. For each $j \in [1, n]$, compute

$$dcol(t_1 \ldots t_j) = nextdcol(dcol(t_1 \ldots t_{j-1}), t_j).$$

If $dcol(t_1 \ldots t_j)(m) \leq k$, then output j. □

It is straightforward to show that

$$dcol(t_1 \ldots t_j)(i) = \min\{\delta(s, p_1 \ldots p_i) \mid s \text{ is a suffix of } t_1 \ldots t_j\}$$

for any $i \in [0, m]$ and any $j \in [0, n]$. This implies the correctness of Sellers algorithm. For each $j \in [0, n]$, the distance column $dcol(t_1 \ldots t_j)$ can be computed in $O(m)$ steps. This gives an overall time efficiency of $O(m \cdot n)$. Since in each step at most two columns have to be stored, Sellers algorithm needs $O(m)$ space.

5.2 Improving Sellers Algorithm

In this section, we show how to improve Sellers algorithm. The idea is to compute the distance column of $t_1 \ldots t_j$ modulo some equivalence.

Definition 30 An entry $f(i)$ of a distance column f is *essential* if $f(i) \leq k$. $lei(f) = \max\{i \in [0, m] \mid f(i) \leq k\}$ is the *last essential index* of f. The distance columns f and f' are equivalent, denoted by $f \equiv f'$, if for all $i \in [0, m]$, $f(i) = f'(i)$ whenever $f(i) \leq k$ or $f'(i) \leq k$. □

The relation $\equiv$ is preserved by $nextdcol$, as shown in the following observation.

Observation 16 Let $f \equiv f'$ and $b \in \mathcal{A}$. Then $nextdcol(f, b) \equiv nextdcol(f', b)$.

Proof: Let $f_b = nextdcol(f, b)$ and $f'_b = nextdcol(f', b)$. By induction on i, we show that $f_b(i) \leq k$ or $f'_b(i) \leq k$ implies $f_b(i) = f'_b(i)$. For $i = 0$ we have $f_b(i) = 0 = f'_b(i)$. Assume that $f_b(i+1) \leq k$ and consider the following cases:

- If $f_b(i+1) = f_b(i) + \delta(p_{i+1} \to \varepsilon)$, then $f_b(i) < k$ which implies $f_b(i) = f'_b(i)$ by induction hypothesis.
- If $f_b(i+1) = f(i) + \delta(p_{i+1} \to b)$, then $f(i) \leq k$ which implies $f(i) = f'(i)$ by assumption.
- If $f_b(i+1) = f(i+1) + \delta(\varepsilon \to b)$, then $f(i+1) < k$ which implies $f(i+1) = f'(i+1)$ by assumption.

Hence, we obtain

$$f_b(i+1) = \min \left\{ \begin{array}{l} f_b(i) + \delta(p_{i+1} \to \varepsilon) \\ f(i) + \delta(p_{i+1} \to b) \\ f(i+1) + \delta(\varepsilon \to b) \end{array} \right\} = \min \left\{ \begin{array}{l} f'_b(i) + \delta(p_{i+1} \to \varepsilon) \\ f'(i) + \delta(p_{i+1} \to b) \\ f'(i+1) + \delta(\varepsilon \to b) \end{array} \right\} = f'_b(i+1)$$

By an analogous argumentation, one shows $f'_b(i+1) = f_b(i+1)$ whenever $f'_b(i+1) \leq k$. □

Note that $f(m) \leq k$ if and only if the last essential index of f is m. Let $l = lei(f)$. The essential entries of $f_b = nextdcol(f, b)$ do not depend on the entries $f(l+1), f(l+2), \ldots, f(m)$ since these are larger than k. Hence it is not necessary to calculate f_b completely, as done by Sellers' algorithm. The calculation of f_b can be modified as follows. Compute $f_b(0), f_b(1), \ldots, f_b(l)$ according to Definition 29. If $l < m$, then compute

$$\begin{array}{rcl} f_b(l+1) & = & \min\{f_b(l) + \delta(p_{l+1} \to \varepsilon), f(l) + \delta(p_{l+1} \to b)\}, \\ f_b(l+2) & = & f_b(l+1) + \delta(p_{l+2} \to \varepsilon), \\ f_b(l+3) & = & f_b(l+2) + \delta(p_{l+3} \to \varepsilon), \\ & \vdots & \end{array}$$

until an entry $f_b(h)$ is reached such that either $h = m$ or $f_b(h) > k$ holds. Thus the computation of f_b is *cut off* at index h. The last essential index of f_b is the maximal $i \in [0, h]$ such that $f_b(i) \leq k$. This modification leads to a cutoff variation of Sellers' method. If δ is the unit cost function, then one can show that the expected running time of Seller's Algorithm is $O(k \cdot n)$. Empirical measurements suggest that this result holds for arbitrary cost functions, too. Note that the cutoff variation does not improve on the worst case efficiency of $O(m \cdot n)$.

5.3 Ukkonen's Cutoff Algorithm

The k-differences (approximate string searching) problem is the approximate string searching problem restricted to the case of the unit cost function. For the rest of this chapter, we consider this problem. That is, we assume that k is a non-negative integer and that δ is the unit cost function.

From the previous sections we know how to solve this problem by computing an $(m+1) \times (n+1)$ table D such that $D(i,j) = dcol(t_1 \ldots t_j)(i)$. Note that the only differences between table E_δ and D is the initialization of the first row, which is $0, 1, 2, \ldots, m$ for the former and constant 0 for the latter. This difference is not important for the Observations 7 and 8 of Section 3.2.5. That is, both observations

Table 5.1: The values of table D computed for $p = adbbc$, $t = abbdadcbc$, and $k = 2$. The solutions to the k-differences problem ar 3,4,7,8,9.

		j									
	D	0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
	1	1	0	1	1	1	0	1	1	1	1
i	2	2	1	1	2	1	1	0	1	2	2
	3		2	1	1	2	2	1	1	1	2
	4			2	1	2		2	2	1	2
	5				2	2			2	2	1

also hold for table D, when substituting p for u and t for v. This means that the cutoff variation of Sellers algorithm can further be optimized for the k-differences problem.

Algorithm Compute $D(i,0) = i$ for each $i \in [0,k]$, and set $l_0 = k$. For each $j \in [0, n-1]$ and each $i \in [0, l_j]$, perform the following steps:

(1) If $i < l_j$, then compute $D(i+1, j+1)$ according to Observation 8.

(2) Let $i = l_j < m$. If $p_{i+1} = t_{j+1}$ or $k > D(i, j+1)$, then set $D(i+1, j+1) = k$ and $l_{j+1} = l_j + 1$. Otherwise, set $l_{j+1} = \max\{l \in [0, l_j] \mid D(l, j+1) = k\}$.

For each $j \in [0, n]$ output j if $l_j = m$. □

This algorithm was first described by Ukkonen, hence we call *Ukkonen's cutoff algorithm*. Table 5.1 shows the values of table D computed for a particular input.

Theorem 8 Ukkonen's cutoff algorithm correctly solves the k-differences problem.

Proof: We show by induction on j that l_j is the last essential index of the jth column of table D, and that the values $D(i,j)$ are computed correctly for each $i \in [0, l_j]$. This implies the correctness. For $j = 0$ the claim easily follows. Suppose the claim holds for some $j \in [0, n-1]$. Let $i \in [0, l_j]$.

(1) If $i < l_j$, then $D(i+1, j+1)$ depends on $D(i, j+1)$, $D(i+1, j)$ and $D(i, j)$. These values are computed correctly. Hence, $D(i+1, j+1)$ is computed correctly according to Observation 8.

(2) Let $i = l_j < m$. Then $D(i,j) = k$ and $k \leq D(i+1, j+1) < D(i+2, j+1) < \cdots < D(m, j+1)$. If $p_{i+1} = t_{j+1}$ or $k > D(i, j+1)$, then $D(i+1, j+1) = k$ and $l_{j+1} = l_j + 1$ is the last essential index of column $j+1$. Otherwise, $D(i+1, j+1) > k$ and the last essential index of the column $j+1$ is $\leq l_j$. □

The algorithm computes $O(l_j)$ entries in column j of table D. In the worst case, $l_j = m$. Hence, the worst case running time is $O(m \cdot n)$. It can be shown that the expected value of l_j is $O(k)$. Thus, we can conclude that the expected running time of Ukkonen's cutoff algorithm is $O(k \cdot n)$.

5.4 Ukkonen's Column DFA

Note that each column computed by Ukkonen's cutoff algorithm transforms one column into another column, according to a given character. (This view was already employed when describing Sellers Algorithm.) The idea now is to preprocess the column transformations into a deterministic finite automaton. Intuitively, each state of the automaton represents a possible column of table D and each transition represents the computation of a column from a previous column.

Let f be a distance column. If $f(i) > k$, then the *exact* value of $f(i)$ does not matter (see Observation 16). Hence, each value $f(i)$ larger than k can be set to $k+1$, without affecting the correctness of Sellers' algorithm. This motivates the following definition.

Definition 31 For each $v \in \mathcal{A}^*$ the column $ndcol(v)$ is defined as follows:

$$ndcol(v)(i) = \min\{dcol(v)(i), k+1\}$$

for all $i \in [0, m]$. $ndcol(v)$ is the *normalized distance column* of v. $f \in C$ is a normalized distance column if $f = ndcol(v)$ for some $v \in \mathcal{A}^*$. □

Note that a normalized distance column contains $m+1$ values, all of which are $\leq k+1$. Hence, there are only finitely many normalized distance columns.

Definition 32 The *column*-DFA for $\mathcal{A}$, k, and p is the deterministic finite automaton

$$(\mathcal{S}, \mathcal{F}, s_0, nextndcol)$$

which is specified as follows:

1. $\mathcal{S} = \{ndcol(v) \mid v \in \mathcal{A}^*\}$,
2. $\mathcal{F} = \{f \in \mathcal{S} \mid f(m) \leq k\}$,
3. $s_0 = ndcol(\varepsilon)$,
4. $nextndcol(ndcol(v), b) = ndcol(vb)$, for all $v \in \mathcal{A}^*$ and all $b \in \mathcal{A}$. □

Notice that one can define the column-DFA on *all* distance columns. However, using normalized distance columns reduces the size of the automaton considerably.

Example 28 Let $\mathcal{A} = \{a, b\}$, $k = 1$, and $p = abba$. The column-DFA for $\mathcal{A}$, k, and p is represented by the following table. The states are written as sequences of integers. The accepting states are underlined.

	01222	01122	01112	<u>01111</u>	<u>01101</u>	01012	<u>01011</u>	00122	<u>00121</u>	<u>00111</u>	<u>00110</u>
a	00122	00122	00121	00121	00110	00111	00111	00122	00122	00121	00121
b	01122	01112	01112	01112	01111	01101	01101	01012	01012	01012	01011

□

Algorithm CDFA: Preprocess the column-DFA $M = (\mathcal{S}, \mathcal{F}, s_0, nextndcol)$ for $\mathcal{A}$, k, and p. Let $s^0 = s_0$ and $s^j = nextndcol(s^{j-1}, t_j)$ for $j \in [1, n]$. Output j if $s^j \in \mathcal{F}$. □

The crucial point in Algorithm CDFA is the preprocessing phase. In order to compute a transition efficiently, the representation of $\mathcal{S}$ is very important. One can represent each $f \in \mathcal{S}$ by the sequence

$$(f(1) - f(0), f(2) - f(1), \ldots, f(m) - f(m-1)) \tag{5.1}$$

of differences. According to Observation 7, $f(i) - f(i-1) \in \{1, 0, -1\}$ for all $i \in [1, m]$. Hence, (5.1) can conveniently be stored in a *ternary tree* whose edge labels are 1, 0, and -1. Each membership test and each insertion into the ternary tree can be performed in $O(m)$ steps. Thus, a transition can be computed in $O(m)$ steps. Let $l = |\mathcal{A}|$. Since there are $|\mathcal{S}| \cdot l$ transitions, the column-DFA can be constructed in $O(|\mathcal{S}| \cdot l \cdot m)$ time. The space requirement for the ternary tree is $O(|\mathcal{S}| \cdot m)$.

By the cutoff technique described in section 5.3, one can improve the average case efficiency of the preprocessing. In particular, each $f \in \mathcal{S}$ can be uniquely represented by the sequence

$$(f(1) - f(0), f(2) - f(1), \ldots, f(lei(f)) - f(lei(f) - 1)) \tag{5.2}$$

Since the expected value of the last essential index is $O(k)$, a membership test and an insertion into the ternary tree takes $O(k)$ steps in the expected case. Hence, the construction time and the size of the ternary tree reduce in the expected case to $O(|\mathcal{S}| \cdot l \cdot k)$ and $O(|\mathcal{S}| \cdot k)$, respectively.

In every step of the preprocessing phase, the new states, that is, those for which the transitions have not been computed yet, must be stored. As all new states occur in the ternary tree, Ukkonen suggests to use a *queue* of pointers to the nodes representing the new states. This queue takes $O(|\mathcal{S}|)$ space, and deletions and insertions can be performed in constant time. Hence, the worst case preprocessing time is $O(|\mathcal{S}| \cdot l \cdot m)$ and the space requirement is $O(|\mathcal{S}| \cdot (l + m))$. To obtain the complexities for the average case, one substitutes m by k.

The representation of $\mathcal{S}$ by a ternary tree implies $|\mathcal{S}| \in O(3^m)$. Taking the threshold and the size of the input alphabet into consideration, Ukkonen derives a second upper bound. He shows that $|\mathcal{S}| \in O(2^k \cdot l^k \cdot m^{k+1})$. This gives the following result.

Theorem 9 Algorithm CDFA correctly solves the k-differences problem in $O(q \cdot l \cdot m + n)$ time and $O(q \cdot (l + m))$ space where $q = \min\{3^m, 2^k \cdot l^k \cdot m^{k+1}\}$ and $l = |\mathcal{A}|$. □

Theorem 9 shows that if m and k are not quite small, the large time and space requirements may limit the applicability of Algorithm CDFA.

5.5 Agrep

In this section we discuss the basic algorithms used in the popular string matching tool `agrep`. The name `agrep` is an abbreviation for "approximate general regular expression print". The user interface is similar to the well known programs `grep`, `egrep`, and `fgrep`. `Agrep` was developed in the early 1990's by

Manber and Wu. The main goal was to search for short patterns with few errors. The speed and versatility of the tool is mainly based on a new string matching approach exploiting bit-parallelism.

We assume a pattern p of length m. The bitvectors used in `agrep` are of length at least m. We assume that they are exactly of length m. This does not cause any problems, but the algorithms are more easy to explain. Agrep actually uses 4 byte unsigned integers to represent the bitvectors. So only words of length up to 32 can be searched.

5.5.1 Exact String Matching

For the exact string searching (i.e. $k = 0$), the `agrep`-algorithm computes bitvectors R_j^0 for $j \in [0, n]$ defined by:

$$R_j^0[i] = 1 \iff p_1 \dots p_i = t_{j-i+1} \dots t_j \tag{5.3}$$

for $i \in [1, m]$. Hence there is an exact match ending at position j in t if and only if $R_j^0[m] = 1$. Now consider how to compute the bitvectors R_j^0. For $j = 0$ we have $t_{j-i+1} \dots t_j = \varepsilon$ and hence $R_0^0[i] = 0$. Now suppose $j > 0$ and $i > 0$. We have $R_j^0[1] = 1 \iff p_1 = t_j$. Suppose $i > 1$. Then

$$\begin{aligned} R_j^0[i] = 1 &\iff p_1 \dots p_i = t_{j-i+1} \dots t_j \\ &\iff p_1 \dots p_{i-1} = t_{j-i+1} \dots t_{j-1} \text{ and } p_i = t_j \\ &\iff p_1 \dots p_{i-1} = t_{(j-1)-(i-1)+1} \dots t_{j-1} \text{ and } p_i = t_j \\ &\iff R_{j-1}^0[i-1] = 1 \text{ and } p_i = t_j \end{aligned}$$

Altogether we obtain

$$R_j^0[i] = \begin{cases} 0 & \text{if } j = 0 \\ 1 & \text{if } j > 0 \text{ and } (i = 1 \text{ or } R_{j-1}^0[i-1] = 1) \text{ and } p_i = t_j \\ 0 & \text{otherwise} \end{cases}$$

To compute the transition from R_{j-1}^0 to R_j^0 efficiently, we make use of bit parallelism. The idea is to preprocess for each character $a \in \mathcal{A}$ a bitvector S_a defined by $S_a[i] = 1 \iff p_i = a$. Then the result of the comparison $p_i = t_j$ is found in $S_{t_j}[i]$. Moreover, the dependence of R_j^0 on R_{j-1}^0 implies a right shift before combining the result with S_{t_j}. We have to fill with 1 after the right shift to consider the case $i = 1$. Altogether we obtain the following equation for R_j^0, $j \in [1, n]$:

$$R_j^0 = \mathit{rshift}(R_{j-1}^0) \,\&\, S_{t_j}$$

where

$$\mathit{rshift}(v)[i] = \begin{cases} 1 & \text{if } i = 1 \\ v[i-1] & \text{otherwise} \end{cases}$$

and & is the bitwise logical and-operation. Assuming $m \leq 32$ we can implement the expression $\mathit{rshift}(v)$ easily in C by

```
((v) >> 1) | (1 << 31))
```

Table 5.2 shows the sequence of bitvectors R_j^0 and S_a computed by the `agrep`-algorithm.

Now consider the efficiency. Let ω be the word length of the computer. Each R_j^0 can be stored $\lceil m/\omega \rceil$ computer words. We do not need to store R_{j-1}^0, once we have computed R_j^0. Moreover, we need $|\mathcal{A}|$ bitvectors of size m. Hence the space requirement is $O(|\mathcal{A}| \cdot \lceil m/\omega \rceil)$.

The precomputation of the bitvectors S_a for all $a \in \mathcal{A}$ requires $O(|\mathcal{A}| \cdot \lceil m/\omega \rceil)$ time. It is easy to see that each computation of R_j^0 from R_{j-1}^0 requires $3 \cdot \lceil m/\omega \rceil$ bit operations. Hence the time requirement is $O(n\lceil m/\omega \rceil)$. For the case $m \leq \omega$, the algorithm thus runs in linear time.

5.5.2 Allowing for Errors

Now we consider the case $k > 0$. Suppose D is the $(m+1) \times (n+1)$ table such that $D(i,j) = dcol(t_1 \dots t_j)(i)$, see Section 5.3. We compute $k+1$ bitvectors R_j^d for $d \in [0,k]$ and $j \in [0,n]$ satisfying:

$$R_j^d[i] = 1 \iff D(i,j) \leq d \tag{5.4}$$

for any $i \in [0,m]$. Hence there is an approximate match ending at position $j \in [0,n]$ if and only if $R_j^k[m] = 1$. Note that $D(i,j) \leq 0$ if and only if $p_1 \dots p_i = t_{j-i+1} \dots t_j$. Hence (5.3) coincides with (5.4) for $d = 0$.

Now consider how to compute the bitvectors R_j^d. Recall that $D(i,0) = i$ for $i \in [0,m]$. Hence $R_0^d[i] = 1 \iff D(i,0) \leq d \iff i \leq d$. Let $j > 0$. For $d = 0$ we can resort to the equality $R_j^0 = rshift(R_{j-1}^0)$ & S_{t_j}. Now suppose $d > 0$, $i > 0$, and $j > 0$. Then

$$\begin{aligned}
R_j^d[i] = 1 &\iff D(i,j) \leq d \\
&\iff \min \left\{ \begin{array}{l} D(i-1,j-1) + \text{ if } p_i = t_j \text{ then 1 else 0} \\ D(i,j-1)+1 \\ D(i-1,j)+1 \end{array} \right\} \leq d \\
&\iff (\text{if } p_i = t_j \text{ then } (D(i-1,j-1) \leq d) \text{ else } (D(i-1,j-1)+1 \leq d)) \\
&\quad \text{or } D(i,j-1)+1 \leq d \\
&\quad \text{or } D(i-1,j)+1 \leq d \\
&\iff (\text{if } p_i = t_j \text{ then } (D(i-1,j-1) \leq d) \text{ else } (D(i-1,j-1) \leq d-1)) \\
&\quad \text{or } D(i,j-1) \leq d-1 \\
&\quad \text{or } D(i-1,j) \leq d-1 \\
&\iff (\text{if } p_i = t_j \text{ then } R_{j-1}^d[i-1] \text{ else } R_{j-1}^{d-1}[i-1]) \\
&\quad \text{or } R_{j-1}^{d-1}[i] \\
&\quad \text{or } R_j^{d-1}[i-1]
\end{aligned}$$

To apply these operations to the entire bitvectors in parallel, we have to shift right R_{j-1}^d, R_{j-1}^{d-1} and R_j^{d-1}, since they access the value at index $i-1$. The shift right always fills with the one bit to consider the case $i = 1$. Additionally we have to combine R_{j-1}^d with the bitvector S_{t_j} for the pairwise character comparison. We obtain the following expression to compute R_j^d for $d \in [1,k]$ and $j \in [1,n]$:

Table 5.2: The bitvectors R_j^0, R_j^1, S_a, S_b, and S_c computed by the `agrep`-algorithm for $p = aabac$, $t = aabaacaabacab$, and $k = 1$

		a	a	b	a	a	c	a	a	b	a	c	a	b			
	R_0^0	R_1^0	R_2^0	R_3^0	R_4^0	R_5^0	R_6^0	R_7^0	R_8^0	R_9^0	R_{10}^0	R_{11}^0	R_{12}^0	R_{13}^0	S_a	S_b	S_c
a	0	1	1	0	1	1	0	1	1	0	1	0	1	0	1	0	0
a	0	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0
b	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0
a	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0
c	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1

		a	a	b	a	a	c	a	a	b	a	c	a	b
	R_0^1	R_1^1	R_2^1	R_3^1	R_4^1	R_5^1	R_6^1	R_7^1	R_8^1	R_9^1	R_{10}^1	R_{11}^1	R_{12}^1	R_{13}^1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1
a	0	0	1	1	1	1	1	1	1	1	1	0	0	0
b	0	0	0	1	1	0	0	0	0	1	1	0	0	0
a	0	0	0	0	1	1	0	0	0	0	1	1	1	0
c	0	0	0	0	0	0	1	0	0	0	0	1	1	0

$$
\begin{aligned}
R_j^d \quad = \quad & rshift(R_{j-1}^d) \ \& \ S_{t_j} \\
| \quad & rshift(R_{j-1}^{d-1}) \\
| \quad & R_{j-1}^{d-1} \\
| \quad & rshift(R_j^{d-1})
\end{aligned}
$$

The algorithm computes the bitvectors in parallel. The precomputation of the bitvectors S_a for all $a \in \mathcal{A}$ requires $O(|\mathcal{A}| \cdot \lceil m/\omega \rceil)$ time. There are $k+1$ bitvectors R_0^d, $d \in [0, k]$ to initialize. This takes $O(k \cdot \lceil m/\omega \rceil)$ time. For computing each generation of bitvectors R_j^d, $d \in [0, k]$, we need $O(k \cdot \lceil m/\omega \rceil)$ time. Hence the running time is $O(n \cdot k \cdot \lceil m/\omega \rceil)$. If $m \leq \omega$, then the running time is $O(kn)$.

CHAPTER 6

Further Reading

There are now several text books on biological sequence analysis. The focus of the books are different. [Ste94] and [CR94] cover mostly exact and approximate string matching techniques. [Gus97] is more complete on the biologically relevant techniques. [Wat95] covers similar topics but it was written from a mathematician's viewpoint. [SM97] is also recommended. The most recent book [Pev00] covers some new topics.

The application areas in Section 1.1 are further described in [KS83]. The notion of edit operations dates back to [Ula72]. The introduction of the edit distance model is standard and it can be found in the text books mentioned above. The algorithm for computing the edit distance was described by [WF74], but others independently discovered the same algorithm. The methods for the fast computation of the (simple) Levenshtein distance have independently been described by [Ukk85a] and [Mye86]. The Smith-Waterman Algorithm is from [SW81].

The maximal matches model is from [EH88], while the q-gram Model was introduced by [Ukk92]. More information on FASTA can be found in [LP85, Pea90]. The version of BlastP discussed here is described in [AGM+90]. For some details on the current version of BlastP, see [AMS+97].

For a comprehensive overview of suffix trees, see [Gus97]. The Write-Only-Top-Down suffix tree construction was developed in [GK95]. Efficient implementation techniques are described in [GKS99]. Ukkonen's online construction is from [Ukk95]. The algorithm of McCreight can be found in [McC76]. The presentations of both algorithms follow [Kur95]. [GK97] show that McCreight's Algorithm is an optimization of Ukkonen's Algorithm, although both algorithms rely on different intuitive ideas. The implementation techniques for suffix trees are described in [Kur99]. The method to compute maximal unique matches stems from [DKF+99]. The algorithm to compute maximal repeats is from [Gus97]. The first efficient implementation was in the REPuter program [KS99, KOS+00], see

als `http://www.genomes.de`

Seller's algorithm is from [Sel80]. The improvement was probably known by many people, but it was written up in [Kur96]. The cutoff technique of Ukkonen and the column DFA were introduced in [Ukk85b]. Agrep is described in [WM92]. The source code can be obtained from

`ftp://ftp.cs.arizona.edu/agrep/agrep-2.04.tar.Z`.

Agrep is also part of the SUSE-Linux distribution.

Bibliography

[AGM+90] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. A Basic Local Alignment Search Tool. *J. Mol. Biol.*, **215**:403–410, 1990.

[AMS+97] S.F. Altschul, T.L. Madden, A.A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs. *Nucleic Acids Res.*, **25**(17):3389–3402, 1997.

[CR94] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, Oxford, 1994.

[DKF+99] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. Alignment of Whole Genomes. *Nucleic Acids Res.*, **27**:2369–2376, 1999.

[EH88] A. Ehrenfeucht and D. Haussler. A New Distance Metric on Strings Computable in Linear Time. *Discrete Applied Mathematics*, **20**:191–203, 1988.

[GK95] R. Giegerich and S. Kurtz. A Comparison of Imperative and Purely Functional Suffix Tree Constructions. *Science of Computer Programming*, 25(2-3):187–218, 1995.

[GK97] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*, 19:331–353, 1997.

[GKS99] R. Giegerich, S. Kurtz, and J. Stoye. Efficient Implementation of Lazy Suffix Trees. In *Proc. of the Third Workshop on Algorithmic Engineering (WAE99)*, pages 30–42. Lecture Notes in Computer Science 1668, 1999.

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

[KOS+00] S. Kurtz, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. Computation and Visualization of Degenerate Repeats in Complete Genomes. In *Proc. of the International Conference on Intelligent Systems for Molecular Biology*, pages 228–238, Menlo Park, CA, 2000. AAAI Press.

[KS83] J.B. Kruskal and D. Sankoff. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, Reading, MA, 1983.

[KS99] S. Kurtz and C. Schleiermacher. REPuter: Fast Computation of Maximal Repeats in Complete Genomes. *Bioinformatics*, 15(5):426–427, 1999.

[Kur95] S. Kurtz. *Fundamental Algorithms for a Declarative Pattern Matching System*. Dissertation, Technische Fakultät, Universität Bielefeld, available as Report 95-03, July 1995.

[Kur96] S. Kurtz. Approximate String Searching under Weighted Edit Distance. In *Proc. of Third South American Workshop on String Processing*, pages 156–170, 1996.

[Kur99] S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software—Practice and Experience*, 29(13):1149–1171, 1999.

[LP85] D.J. Lipman and W.R. Pearson. Rapid and Sensitive Protein Similarity Search. *Science*, **227**:1435–1441, 1985.

[McC76] E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, **23**(2):262–272, 1976.

[Mye86] E.W. Myers. An $O(ND)$ Differences Algorithm. *Algorithmica*, **2**(1):251–266, 1986.

[Pea90] W.R. Pearson. Rapid and Sensitive Sequence Comparison with FASTP and FASTA. In Doolittle, R., editor, *Methods in Enzymology*, volume **183**, pages 63–98. Academic Press, San Diego, CA, 1990.

[Pev00] P. A. Pevzner, editor. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, Cambridge, MA, 2000.

[Sel80] P.H. Sellers. The Theory and Computation of Evolutionary Distances: Pattern Recognition. *Journal of Algorithms*, **1**:359–373, 1980.

[SM97] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing, Boston, M.A., 1997.

[Ste94] G.A. Stephen. *String Searching Algorithms*. World Scientific, Singapore, 1994.

[SW81] T.F. Smith and M.S. Waterman. Identification of Common Molecular Subsequences. *J. Mol. Biol.*, **147**:195–197, 1981.

[Ukk85a] E. Ukkonen. Algorithms for Approximate String Matching. *Information and Control*, **64**:100–118, 1985.

[Ukk85b] E. Ukkonen. Finding Approximate Patterns in Strings. *Journal of Algorithms*, **6**:132–137, 1985.

[Ukk92] E. Ukkonen. Approximate String-Matching with q-Grams and Maximal Matches. *Theoretical Computer Science*, **92**(1):191–211, 1992.

[Ukk95] E. Ukkonen. On-line Construction of Suffix-Trees. *Algorithmica*, **14**(3), 1995.

[Ula72] S.M. Ulam. Some Combinatorial Problems Studied Experimentally on Computing Machines. In Zaremba, S.K., editor, *Applications of Number Theory to Numerical Analysis*, pages 1–3. Academic Press, 1972.

[Wat95] M.S. Waterman. *Introduction to Computational Biology: Maps, Sequences and Genomes*. Chapman Hall, 1995.

[WF74] R.A. Wagner and M.J. Fischer. The String to String Correction Problem. *Journal of the ACM*, **21**(1):168–173, 1974.

[WM92] S. Wu and U. Manber. Fast Text Searching Allowing Errors. *Communications of the ACM*, **10**(35):83–91, 1992.