Universität Bielefeld

Technische Fakultät
Abteilung Informationstechnik
Forschungsberichte

Online Abelian Pattern Matching

Tahir Ejaz Sven Rahmann Jens Stoye

Report 2008-01

Impressum: Herausgeber:
Ellen Baake, Robert Giegerich, Ralf Hofestädt,
Franz Kummert, Peter Ladkin, Ralf Möller, Tim Nattkemper,
Helge Ritter, Gerhard Sagerer, Jens Stoye, Ipke Wachsmuth

Technische Fakultät der Universität Bielefeld,
Abteilung Informationstechnik, Postfach 10 01 31,
33501 Bielefeld, Germany

ISSN 0946-7831

Online Abelian Pattern Matching

Tahir Ejaz [*] Sven Rahmann [†] Jens Stoye [‡]

March 31, 2008

Abstract

An abelian pattern describes the set of strings that comprise of the same combination of characters. Given an abelian pattern P and a text $T \in \Sigma^n$, the task is to find all occurrences of P in T, i.e. all substrings $S = T_i ... T_j$ such that the frequency of each character in S matches the specified frequency of that character in P.

In this report we present simple online algorithms for abelian pattern matching, and give a lower bound for online algorithms which is $\Omega(n)$.

Key Words: Pattern Matching; String Matching; Abelian Patterns; Online Algorithms; Permutation Patterns; Compomers

1 Introduction

In the past few years, the abundance of completely sequenced genomes has led to the idea of comparison and analysis of whole genomes at gene level. Gene clustering is one approach for this type of comparison and analysis. It is believed that genes with similar functionality tend to occur close to each other, so gene clustering can help in finding the functionality of genes. Moreover, it can also help in inferring the phylogenetic distance between different organisms. Gene clustering aims at finding genes that are located in close proximity of each other, hence it assumes that the order of the occurrence of

[*]AG Genominformatik, Technische Fakultät, and Institut für Bioinformatik, CeBiTec, Universität Bielefeld, Germany. tahir@cebitec.uni-bielefeld.de

[†]Chair of Algorithm Engineering, Faculty of Computer Science, Technische Universität Dortmund, Germany. Sven.Rahmann@tu-dortmund.de

[‡]AG Genominformatik, Technische Fakultät, and Institut für Bioinformatik, CeBiTec, Universität Bielefeld, Germany. stoye@techfak.uni-bielefeld.de

these genes is irrelevant. This phenomenon can be approximately modeled by abelian pattern matching, as we are not interested in the order of the occurrence of characters in an abelian pattern, rather we want to find the substrings matching the specified frequencies of the characters.

Abelian patterns (also known as compomers [1] and permutation patterns [3]) have also been considered for DNA de-novo sequencing [1]. Abelian pattern matching also resembles weighted string matching [2]; however, the set of all the matching weighted strings (i.e. all those strings whose weights are the same as those of the given string) is a superset of all the abelian matches of the given string. Moreover, matching weighted strings can be of different lengths, but exactly matching abelian patterns are always of the same length.

2 Abelian Pattern Matching

The problem of *Abelian Pattern Matching* differs from *Classical Pattern Matching* in the sense that in case of classical pattern matching we seek for exact occurrences of a pattern substring in the given input string, and the order of characters in the pattern substring is preserved while looking for a match. In case of abelian pattern matching, however, the order of characters in the pattern substring does not matter. Hence *'abc'* and *'bac'* are considered matching (abelian) substrings. Here we are not looking for an exact (ordered) occurrence of a substring, *rather we want to find any* ***permutation*** *of a given* ***combination*** *of characters that forms our pattern substring.*

2.1 Formal Problem Definition

Formally, given an alphabet Σ, an *abelian pattern* is a function $P : \Sigma \to \mathbb{N}$ that assigns a multiplicity to each character in Σ. We set $\Sigma_P := \{c \in \Sigma : P(c) > 0\}$, the set of characters occurring in the pattern, and call $|\Sigma_P|$ the *size* of the pattern. We write the pattern symbolically as $P = \sum_{c \in \Sigma} m_c\, c$, where $m_c = P(c)$ denotes the multiplicity of character c in the pattern. We call $m := |P| := \sum_{c \in \Sigma_P} m_c$ the *length* of the pattern. For example, over the alphabet $\Sigma = \{a, b, c, d\}$, the strings *abcb* and *bbca* match the same abelian pattern $P = (1, 2, 1, 0)$ (function specification in lexicographic order) or $P = 1a + 2b + 1c + 0d = a + 2b + c$ (symbolic sum specification).

Given an abelian pattern P and a text $T \in \Sigma^n$, the *abelian pattern matching problem* is to find all occurrences of P in T, i.e. all positions of substrings $S = T_i ... T_j$ with $j - i + 1 = |P|$ such that the frequency of each character in S matches the specified frequency of that character in P. For $T = ababcccabaccbacdddba$, the pattern $P = 2a + b + 3c$ occurs at positions 3, 5 and 10.

2.2 Properties of Abelian Patterns

Abelian patterns are quite different from normal classical patterns. In this section we shed light on properties of abelian patterns.

- The number of abelian patterns/strings of length m over an alphabet Σ can be viewed as the number of integer solutions to the equation

  $$x_1 + \cdots + x_{|\Sigma|} = m$$

  under the condition that $x_i \geq 0$ for all $i = 1, \ldots, |\Sigma|$. This number is $\binom{|\Sigma|+m-1}{m}$ [6]. Note that, for large values of m, this number is significantly smaller than the number of classical patterns of length m over the alphabet Σ, which is $|\Sigma|^m$. This is because of the fact that an abelian pattern can be spelled by more than one strings.

- Let S_P be the set of all strings that match an abelian pattern P, then we call S_P the *pattern set* of P and $|S_P|$ the *size* of the pattern set of P. For an abelian pattern $P = \sum_{i=1}^{|\Sigma|} m_{c_i} c_i$ of length m, the size of its pattern set can be computed as the multinomial coefficient:

  $$|S_P| = \binom{m}{m_{c_1}, \ldots, m_{c_{|\Sigma|}}}$$

2.3 Some Definitions

In this section we give some definitions that we use later.

Definition 1. *An abelian pattern $P' = \sum_{i=1}^{|\Sigma|} m'_{c_i} c_i$ is an* abelian sub-pattern *of another abelian pattern $P = \sum_{i=1}^{|\Sigma|} m_{c_i} c_i$ if and only if $m'_{c_i} \leq m_{c_i}$ for all $i = 1, 2, \ldots, |\Sigma|$. Symmetrically, P is called an* abelian super-pattern *of P'.*

Definition 2. *Given an abelian pattern $P = \sum_{i=1}^{|\Sigma|} m_{c_i} c_i$ and its abelian sub-pattern $P' = \sum_{i=1}^{|\Sigma|} m'_{c_i} c_i$, the abelian pattern $P - P' := \sum_{i=1}^{|\Sigma|} (m_{c_i} - m'_{c_i}) c_i$ is called the* difference pattern *between P and P'.*

Definition 3. *Given an abelian pattern $P = \sum_{i=1}^{|\Sigma|} m_{c_i} c_i$, the multiset $\{m_{c_i} \mid c_i \in \Sigma_P\}$ denoted by M_P is called the* multiplicity set *of P.*

Observation 1. *The length-j abelian sub-patterns of an abelian pattern P of length m have a many-to-one relationship with the integer partitions of $m-j$. For each partition λ of $m-j$, there exists a distinct class C_λ comprising of (zero or more) length-j abelian sub-patterns of P such that the elements of $M_{P-P'}$ have a one-to-one correspondence with the elements of λ for each $P' \in C_\lambda$.*

Example: Given an abelian pattern $P = 3a + 2b + 2c$ with $m = 7$, the following are its length-4 abelian sub-patterns:

$$\begin{array}{rclcrcl} P'_1 &=& 2a+b+c & \qquad & P'_2 &=& 2a+2b \\ P'_3 &=& 2a+2c & & P'_4 &=& 3a+b \\ P'_5 &=& a+b+2c & & P'_6 &=& 3a+c \\ P'_7 &=& a+2b+c & & P'_8 &=& 2b+2c \end{array}$$

and the following are the integer partitions of $3 = 7 - 4$:

$$\begin{array}{rcll} 3 &=& 3 & \text{(call this partition } \lambda_1) \\ &=& 2+1 & \text{(call this partition } \lambda_2) \\ &=& 1+1+1 & \text{(call this partition } \lambda_3) \end{array}$$

The length-4 abelian sub-patterns of P are classified as follows:

$C_{\lambda_1} = \{P'_8\}$, as

$$\begin{array}{rcccl} \lambda_1 &=& & 3 & ;\ \text{and} \\ & & & \updownarrow & \\ M_{P-P'_8} &=& \{ & 3 & \} \end{array}$$

$C_{\lambda_2} = \{P'_2, P'_3, P'_4, P'_5, P'_6, P'_7\}$, as

$$\begin{array}{rcccccl} \lambda_2 &=& & 2 & + & 1 & \qquad ;\ \text{and} \\ & & & \updownarrow & & \updownarrow & \\ M_{P-P'_i} &=& \{ & 2 & , & 1 \ \} & \qquad \text{for } 2 \le i \le 7 \end{array}$$

$C_{\lambda_3} = \{P'_1\}$, as

$$\begin{array}{rcccccccl} \lambda_3 &=& & 1 & + & 1 & + & 1 & ;\ \text{and} \\ & & & \updownarrow & & \updownarrow & & \updownarrow & \\ M_{P-P'_1} &=& \{ & 1 & , & 1 & , & 1 & \} \end{array}$$

Note that in case of length-3 abelian sub-patterns of P, if λ specifies the partition $4 = 4$, then C_λ is empty.

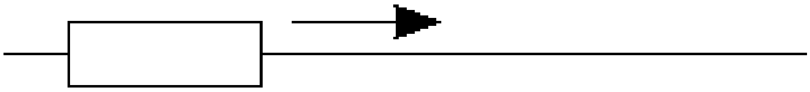

Figure 1: A window of length m is slided along the text

2.4 General Setting

In this report we discuss several algorithms for abelian pattern matching that do not require preprocessing of the text. In these algorithms, as in many other classical pattern matching algorithms [7], a sliding window of length m is moved along T and checked for a possible pattern match (Figure 1). We use three approaches for the procedure of checking for a possible pattern match inside the window:

Prefix based approach. In this approach we read the characters in the window one by one starting from the left end of the window. So at any time we have information about a prefix of the window.

Suffix based approach. Here we read the characters in the window one by one starting from the right end of the window. So at any time we have information about a suffix of the window. This approach may allow to skip some text characters from processing.

Parameterized suffix based approach. We employ the suffix based approach in a parameterized manner, and at any time we have information about at most two factors of the window.

In all the algorithms presented in this report, we use a frequency vector CFV *(current frequency vector)* which keeps the count of the characters read in the current window, and another frequency vector P *(pattern frequency vector)* which contains the count of the characters in the abelian pattern that is to be found. Both CFV and P can be implemented using linked lists, sorted arrays or directly accessible arrays. For a directly accessible array, the cost of query and increment/decrement operations in these vectors is $O(1)$ in the RAM model, and the memory requirement depends on the perfect hash function used for the direct accessibility feature; for a minimal perfect hash function, the memory requirement is $O(|\Sigma|)$. From now onwards we assume that there exists a minimal perfect hash function ρ for the characters in Σ, and both CFV and P are maintained as directly accessible arrays of size $|\Sigma|$. Note that for the alphabets of English language, ρ is quite simple; it just subtracts a constant from the ASCII values of the characters.

3 Prefix Based Algorithm

In the prefix based algorithm, we set a window of size m at the beginning of the input text T and process the characters in the window in a prefix based manner. After we have processed the last character in the window, we check the current window for a match with the given pattern P. After that, the window is slid towards the right by one position and checked again for a match. This way the window is slid through the whole text. As the $m-1$ length suffix of the current window equals the $m-1$ length prefix of the next window, we can construct the next window from the immediately preceding window in constant time. Pseudo code of this algorithm is presented in *Algorithm 1*.

In the first phase of this algorithm we initialize CFV with the first m characters of T. We also initialize the *mismatch* for this CFV, where *mismatch* counts the number of differences between CFV and P. If *mismatch* is zero, we output the first position of the text as starting position of a matching abelian pattern. In the next phase we proceed incrementally. We construct the new CFV by performing two operations on the previous CFV. We also update *mismatch* in constant time.

This algorithm reads and processes every character in T exactly twice; for the first time to increment its frequency in CFV, and for the second time to decrement its frequency in CFV. So the overall time complexity of this algorithm is $\Theta(n)$. At any point in time, this algorithm keeps in memory only two frequency vectors, P and CFV; and one integer variable *mismatch*. Hence the space complexity of this algorithm is $O(|\Sigma|)$, in addition to the space required for the input and the output.

4 Suffix Based Horspool Type Algorithm

This algorithm is an adaptation of Horspool [5] type algorithms. Instead of reading characters from left to right, here we read characters from right to left in the search window; thus using a suffix based approach. While reading characters from right to left inside the window, as soon as an *overflow* of frequency in CFV occurs (i.e. the frequency of a character in CFV exceeds its specified one in P), we stop reading further in the window, as this window cannot contain the given pattern. In fact, no substring that contains the so far read suffix of this window can be a matching pattern. So we can safely shift this window towards the right at the position of the second character of this suffix (as it was the first character of the suffix that caused the overflow). After the window shift, we reset the frequencies of all the characters that were

Algorithm 1 Prefix based Abelian Pattern Matching

Input: A pattern P of length m, a text stream $T = T[1] \dots T[n]$ and a hash function ρ

Output: Positions where the given abelian pattern starts in T

```
▷ Build current frequency vector (CFV) for the first m characters
for i = 1 to |Σ| do
    CFV[i] ← 0
for i = 1 to m do
    CFV[ρ(T[i])] ← CFV[ρ(T[i])] + 1
▷ Calculate the number of mismatching characters between the current window and the given pattern
mismatch ← 0
for i = 1 to |Σ| do
    if CFV[i] ≠ P[i] then
        mismatch ← mismatch + 1
if mismatch = 0 then
    output 1
i ←2
while i ≤ n − m + 1 do
    if T[i − 1] ≠ T[i + m − 1] then
        CFV[ρ(T[i − 1])] ← CFV[ρ(T[i − 1])] − 1
        if CFV[ρ(T[i − 1])] = P[ρ(T[i − 1])] then
            mismatch ← mismatch − 1
        else if CFV[ρ(T[i − 1])] = P[ρ(T[i − 1])] − 1 then
            mismatch ← mismatch + 1
        CFV[ρ(T[i + m − 1])] ← CFV[ρ(T[i + m − 1])] + 1
        if CFV[ρ(T[i + m − 1])] = P[ρ(T[i + m − 1])] then
            mismatch ← mismatch − 1
        else if CFV[ρ(T[i + m − 1])] = P[ρ(T[i + m − 1])] + 1 then
            mismatch ← mismatch + 1
    if mismatch = 0 then
        output i
    i ← i + 1
```

read previously. For this, we maintain a list *RCList* (*read characters list*) that holds all the characters read in the window. We also use a binary vector *RCV* (*read characters vector*) to avoid inserting the same character multiple times in *RCList*. Note that under the suffix based approach, the number of characters in *RCList* at any time is $O(|\Sigma_P|)$.

By using the technique of safely shifting the window, we can skip some characters from processing, but at the same time there is a danger of reading several characters multiple times. This algorithm is only efficient if the *sparseness of matches* holds (i.e. only a few substrings of the input string match to a given abelian pattern), because if this is not the case (i.e. the number of matches is significant) then overflows will not occur frequently and this algorithm will not benefit much. Pseudo code of this algorithm is presented in *Algorithm 2*.

The worst case complexity of this algorithm is $O(nm)$, as we may need to read the same character m times. The best case occurs when, on average, we detect an overflow after reading a constant number of characters in each window; thus giving a best case time complexity of $\Omega(n/m)$.

The average case analysis of this algorithm depends heavily on the pattern. We begin with a lemma.

Lemma 1. *If on average we read ϵm characters in each window, then the time complexity of the suffix based abelian pattern matching is $O(\frac{n\epsilon}{1-\epsilon})$.*

Proof. We read ϵm characters in the window and advance the window by $(1-\epsilon)m+1$ positions. This gives us an $O(\frac{\epsilon}{1-\epsilon})$ cost for processing one character, and for the whole text this cost becomes $O(\frac{n\epsilon}{1-\epsilon})$. □

Theorem 1. *Let us assume that P is fixed and that the characters of the input text are independently and identically distributed, with probability $1/|\Sigma|$ for each character at each position. Then the average case time complexity of the suffix based abelian pattern matching algorithm is*

$$O\left(\frac{n\sum_{k=0}^{m-1}|ASP(P,k)|}{m|\Sigma|^k-\sum_{k=0}^{m-1}|ASP(P,k)|}\right)$$

where $ASP(P,k)$ denotes the set of strings of length k that match abelian sub-patterns of P.

Proof. If the overflow occurs after exactly k characters, we have read k characters and advanced the window by $m-k+1$ characters. Let J denote the random variable that describes the number of characters read in a window. Thus on average, in each iteration of the algorithm, the window is advanced by $m+1-\mathbb{E}[J]$ characters while examining $\mathbb{E}[J]$ characters.

Algorithm 2 Suffix based Abelian Pattern Matching

Input: A pattern P of length m, a text stream $T = T[1] \dots T[n]$ and a hash function ρ

Output: Positions where the given abelian pattern starts in T

for $i = 1$ to $|\Sigma|$ **do**
    $CFV[i] \leftarrow 0$
    $RCV[i] \leftarrow 0$
$RCList \leftarrow \emptyset$
$i \leftarrow 1$
while $i \leq n - m + 1$ **do**
    $overflow \leftarrow 0$
    for all $c \in RCList$ **do**
        $CFV[\rho(c)] \leftarrow 0$
        $RCV[\rho(c)] \leftarrow 0$
        remove c from $RCList$
    $j \leftarrow i + m - 1$
    while $j \geq i$ and $overflow = 0$ **do**
        $CFV[\rho(T[j])] \leftarrow CFV[\rho(T[j])] + 1$
        if $RCV[\rho(T[j])] = 0$ **then**
            insert $T[j]$ in $RCList$
            $RCV[\rho(T[j])] \leftarrow 1$
        if $CFV[\rho(T[j])] > P[\rho(T[j])]$ **then**
            $overflow \leftarrow 1$
        $j \leftarrow j - 1$
    if $overflow = 1$ **then**
        $i \leftarrow j + 2$
    else
        output i
        $i \leftarrow i + 1$

The probability that an overflow occurs after $\leq k$ characters equals the probability that the rightmost k characters in the window are not an abelian sub-pattern of P:

$$\mathbb{P}(J \leq k) = 1 - |\mathrm{ASP}(P, k)|/|\Sigma|^k$$

Since $\mathbb{E}[J] = \sum_{k=0}^{m} k\,\mathbb{P}(J = k) = \sum_{k=1}^{m} \mathbb{P}(J \geq k) = \sum_{k=1}^{m} [1 - \mathbb{P}(J \leq k - 1)] = \sum_{k=0}^{m-1} |\mathrm{ASP}(P, k)|/|\Sigma|^k$; by applying Lemma 1, the theorem is proved.

□

Now we show how $|\mathrm{ASP}(P, k)|$ can be computed using partitions of $\bar{k} := m - k$. We can generate all the partitions of $\bar{k}$ by using any algorithm for generating integer partitions [4, 8]. For a partition $\lambda := \langle 1^{\alpha_1}, 2^{\alpha_2}, \ldots, \bar{k}^{\alpha_{\bar{k}}} \rangle \vdash \bar{k}$ (that is, $\bar{k} = \alpha_1 1 + \alpha_2 2 + \cdots + \alpha_{\bar{k}} \bar{k}$), we construct the abelian sub-patterns belonging to C_λ, and sum $|S_{P'}|$ for all $P' \in C_\lambda$. By iterating this procedure over all the partitions of $\bar{k}$, we obtain $|\mathrm{ASP}(P, k)|$. The procedure for doing this is outlined in Algorithm 3.

The main processing of Algorithm 3 is done in the *Partition* sub-routine. In line 1 of this sub-routine, we select all those characters in P for which a value of l can be deducted from their multiplicities. If the number of such characters is less than α_l, we cannot decrement the multiplicities of the characters according to the given partion λ; hence cannot generate any length-k abelian sub-patterns of P corresponding to λ (i.e. C_λ is empty). At line 5 we have an abelian pattern of length m' ($m' = m$ for the first call of *Partition* sub-routine) and we fix exactly α_l characters from the characters that were selected at line 1. At line 6, we create a local copy of the abelian pattern received from the calling program and then decrement the multiplicities of each of the fixed characters by l in this copy; by doing so, we obtain an abelian pattern of length $m' - \alpha_l l$. If $l = 1$, we have obtained an abelian pattern of length $m - \bar{k} = k$, and in line 10, we compute the size of the pattern set corresponding to this length-k abelian pattern.

5 Lower Bounds

The following can be stated regarding the lower bounds for online abelian pattern matching.

Theorem 2. *A lower bound for* best *case time complexity of any oblivious algorithm of abelian pattern matching in a given text of length n with pattern size m is $\Omega(\lfloor n/m \rfloor)$.*

Algorithm 3 Algorithm for computing $|\mathrm{ASP}(P,k)|$

Main Algorithm

Input: $\bar{k} := m - k$; abelian pattern $P = \sum_{i=1}^{|\Sigma|} m_{c_i} c_i$ of length m

Output: Number of strings in Σ^k that match abelian sub-patterns of P

$n \leftarrow 0$
for each integer partition λ of $\bar{k}$ **do**
    $\mathcal{C} \leftarrow \{c_1, \ldots, c_{|\Sigma|}\}$
    $\mathcal{M} \leftarrow \{m_{c_1}, \ldots, m_{c_{|\Sigma|}}\}$
    $n \leftarrow n+$ Partition $(\bar{k}, \lambda, \mathcal{M}, \mathcal{C})$
return n

Partition (l, λ, M, C)

$C' \leftarrow \{c_i \in C \mid m_{c_i} \geq l\}$
if $|C'| < \alpha_l$ **then**
    return 0
$num \leftarrow 0$
for each distinct $C_{sub} = \{c_1, c_2, \ldots, c_{\alpha_l}\} \subseteq C'$ **do**
    $M' \leftarrow \{m'_{c_i}$; such that $m'_{c_i} = m_{c_i} \in M$ for $1 \leq i \leq |\Sigma|)$
    for each $c \in C_{sub}$ **do**
        $m'_c \leftarrow m'_c - l$
    if $l = 1$ **then**
        $num \leftarrow \binom{k}{m_{c'_1}, \ldots, m_{c'_{|\Sigma|}}}$
    else
        $num \leftarrow num+$ Partition$(l-1, \lambda, M', C \setminus C_{sub})$
return num

A lower bound for worst *case time complexity of any oblivious algorithm of abelian pattern matching in a given text of length n with pattern size m is $\Omega(n)$.*

Proof. The best case bound is straight forward using a classical adversary argument.

For the worst case bound, assume that there exists an abelian pattern matching algorithm A that processes less than n/k characters of the input text, where k is an arbitrary constant. Given an abelian pattern P, consider an input text T such that there are at least n/km non-overlapping matching substrings in T. Then there exists at least one matching substring S in T such that not all of its characters are processed by A. As the algorithm is claimed to be correct, it must have output the starting position of S. Now if

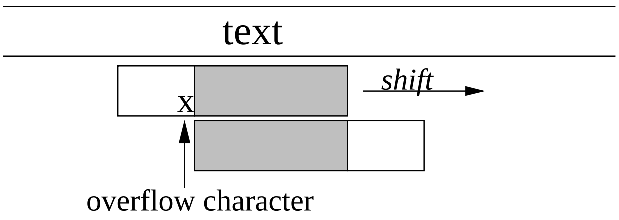

Figure 2: The gray area shows the information in CFV transferred from the previous window to the new window.

we replace any of the unread characters of S with an invalid character c (i.e. $c \notin \Sigma_P$) the output of A should remain unaffected; hence A is not a correct algorithm. $\square$

6 Parameterized Suffix based Algorithm

The main disadvantage of the suffix based algorithm is that it has to reset CFV after every overflow. In this section we present a parameterized suffix based algorithm that resets CFV only if the number of the characters read before an overflow does not exceed ϵm, where ϵ is a user defined parameter.

6.1 The Algorithm

Like the suffix based algorithm, we slide a search window of size m from left to right along the input text T and process the characters inside the window in a right to left manner. In case an overflow occurs in this process, we stop further processing the current window and decrease the frequency of the current character, call it x, by 1 in CFV, so that CFV again becomes compatible with P (i.e. $CFV[i] \leq P[i]$ for all i, $1 \leq i \leq |\Sigma|$). We also shift the window to the right such that its new starting position coincides with the character next to x. So far the processing of this algorithm is the same as that of the suffix based algorithm with the difference that we have decremented the frequency of x (which caused the overflow) by 1 in CFV in this algorithm. Note that CFV contains the information of the whole suffix (except x) that was read in the previous window, and this suffix is now a prefix of the current window (Figure 2).

In the parameterized suffix based algorithm, we do not reset CFV blindly after an overflow has occurred. Instead, we consider the amount of information contained in CFV, and if this information is less than or equal to ϵm (where ϵ is a user defined parameter) then we reset CFV, otherwise we keep

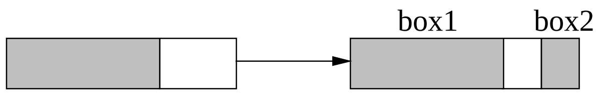

Figure 3: CFV contains collective information of a prefix and a suffix of the current window

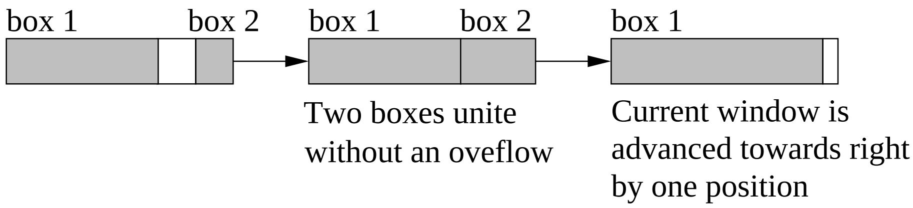

Figure 4: Box 2 unites with box 1 without an overflow. After reporting the current window as a matching substring, the current window is moved towards right by one position. The CFV contains information about an $m-1$ length prefix (representing *box 1*) of the new current window

the information in CFV and start reading characters from the end position of the new current window. This latter case is illustrated in Figure 3: We have two information boxes in the window, *box 1* contains the information of a prefix of the window and *box 2* contains the information of a suffix of the window, whereas CFV contains the collective information of both boxes. Note that every time we read a new character in the window, *box 2* is extended towards the left.

If in this process both boxes unite without an overflow, then the current window is a matching abelian substring and we output the starting position of the current window. We also decrement the frequency of the first character of the current window by 1 in CFV and advance the current window towards the right by one position (Figure 4). However, if an overflow occurs while reading characters in the window, then the current window does not contain a matching substring and we search for the leftmost occurrence of the overflown character in the current window. We start reading the characters in the current window from its left end, and decrement the frequency of each read character by 1 in CFV until we read the overflow character. We shift the new starting position of the current window next to the latest read character. Note that CFV now does not contain information about any character outside the new current window.

Figure 5 illustrates three possible positions of the leftmost occurrence of the overflow character in the current window. It also shows the resulting window when the current window is shifted next to the leftmost occurrence

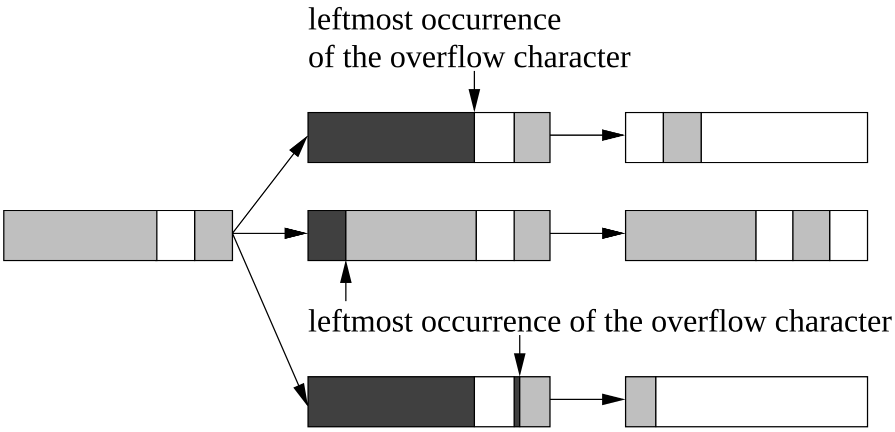

Figure 5: Three possible positions of the leftmost occurrence of the overflow character and the resulting windows after shifting the current window next to the overflow character

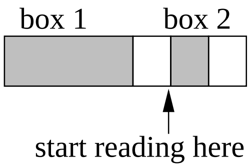

Figure 6: Filling the gap between two information boxes

of the overflow character. The dark gray regions in the figure show those characters whose count has been decremented in CFV. Note that after this step, *box 2* is no longer a suffix of the resulting current window.

Here once again we have to decide whether or not to reset CFV. In case the collective information contents of both boxes (*box 1* possibly empty) are less than or equal to ϵm then we reset CFV, otherwise we keep the information in CFV. However, in the latter case, if we start reading from the end position of the window, then we could have to manage three information boxes in the situations where *box 2* is not a prefix of the current window. To avoid this, we start reading characters from the last position of the gap between *box 1* and *box 2* in these situations, so that CFV once again contains information about only a prefix of the current window (Figure 6).

After this gap is filled, CFV once again contains information about only a prefix of the current window and then we start reading from the right end of the window creating *box 2* to hold information for the rightmost characters of the window (Figure 3). However, an overflow can occur before the gap is filled and it can lead to a loop situation until the information in CFV

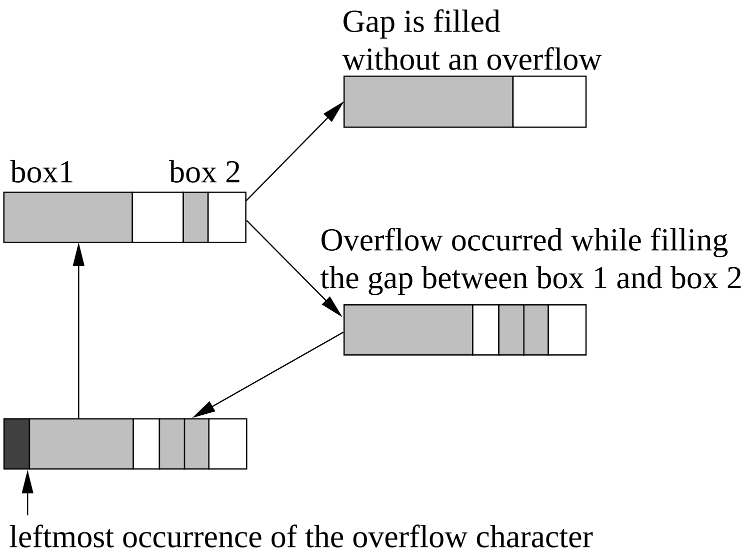

Figure 7: A loop situation while the gap between box 1 and box 2 is being filled

becomes less than ϵm (Figure 7). Nevertheless, we never have more than two information boxes at hand at any time.

In this way we keep on sliding the window along the input text until we reach the end of the text. Figure 8 illustrates this whole phenomenon.

6.2 Examples

To get a better understanding of the working of the parameterized suffix based algorithm, we present two examples and show how the algorithm works for each example using the transition graph presented in Figure 8.

In the following examples, we show different paths taken by the parameterized suffix based algorithm in the transition graph of Figure 8 for certain input strings and abelian patterns.

6.2.1 Example 1 ($1 \rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6$)

Consider an input string *abcccacbb* and an abelian pattern $a + b + 3c$. Figure 9 shows how the parameterized suffix based algorithm proceeds along the transition graph presented in Figure 8 to find the matching abelian patterns in the text.

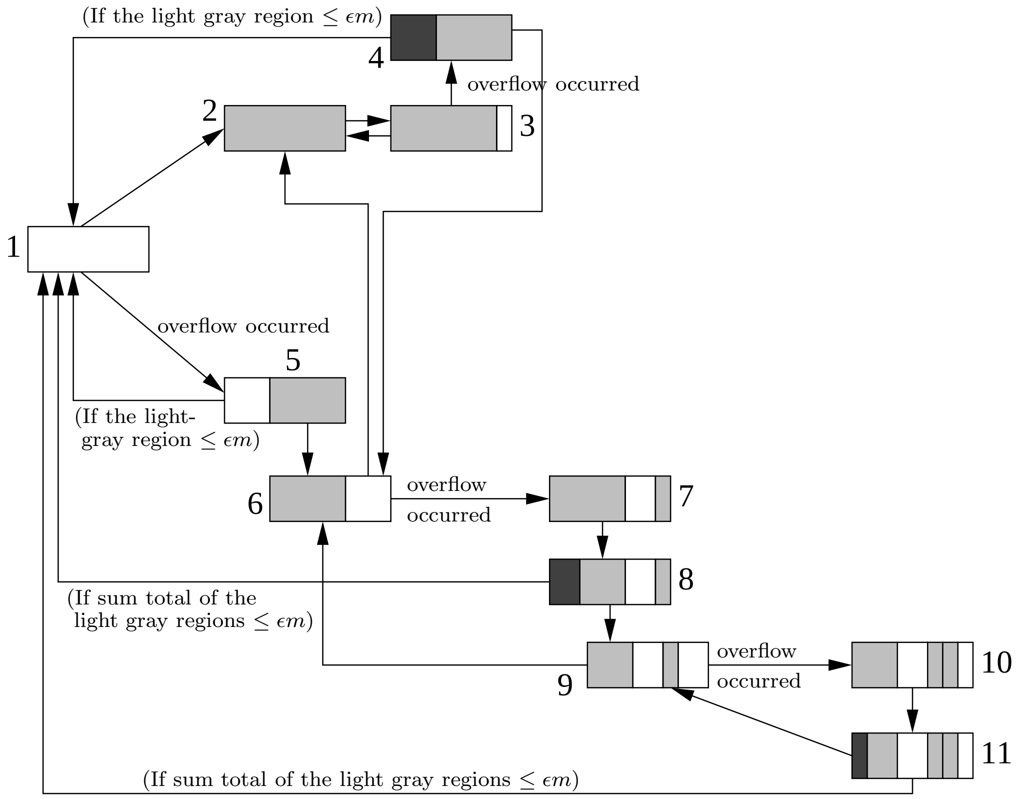

Figure 8: Complete transition graph of the parameterized suffix based algorithm with labeled states. In the figure, the light gray regions in a rectangle represent the characters read in the corresponding search window, hence the frequencies of these characters are incremented in CFV. The white regions in a rectangle represent the unread characters of the corresponding window. The dark gray region in a rectangle represents the characters that occur before the leftmost occurrence of the overflown character in the window, it also includes the overflown character; the frequencies of these characters are decremented in CFV, and the current window is shifted next to the leftmost occurrence of the overflown character.

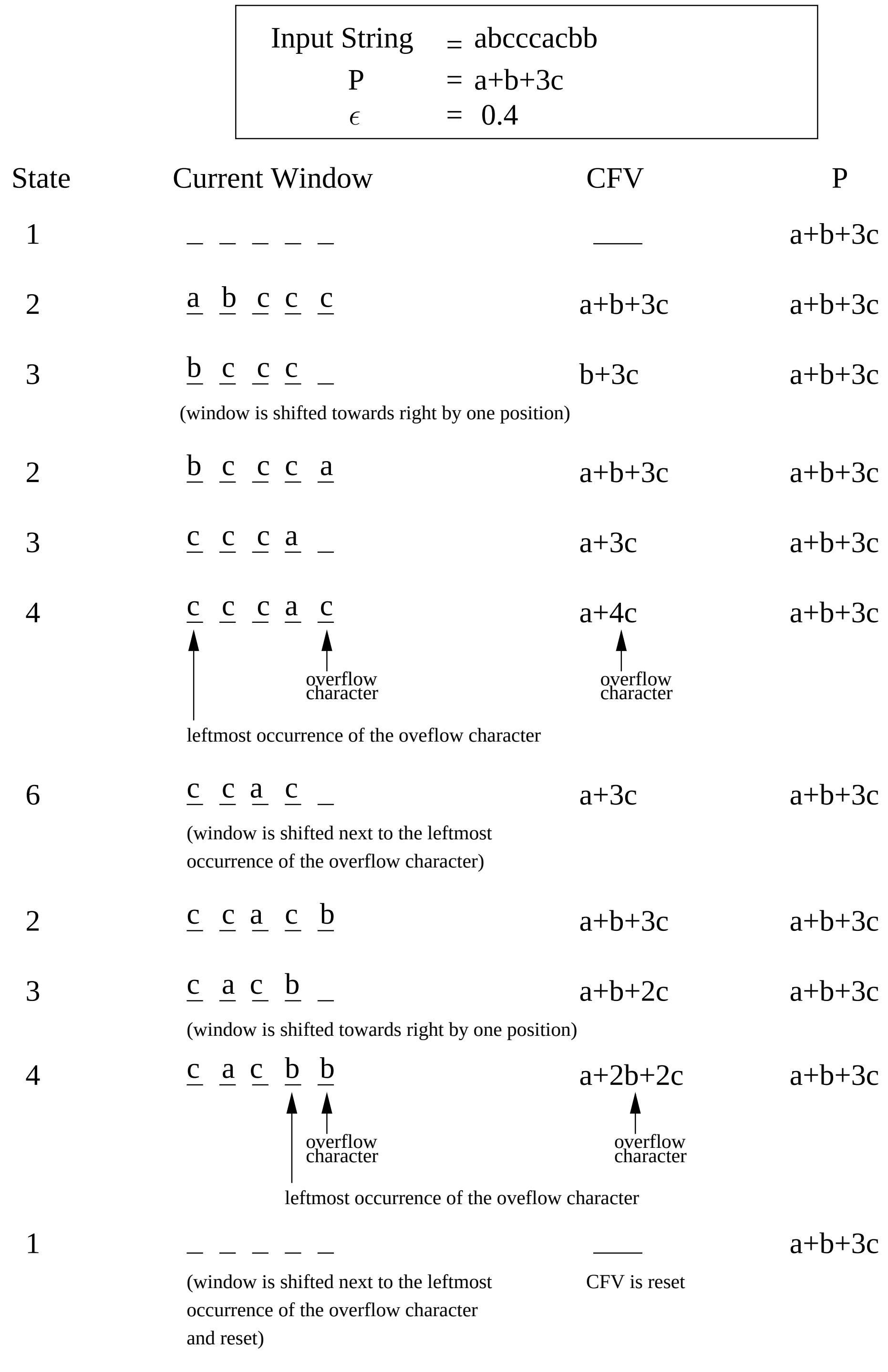

Figure 9: The path taken by the parameterized suffix based algorithm in the transition graph of Figure 8 for an input string *abcccacbb* and an abelian pattern $a + b + 3c$ with $\epsilon = 0.4$.

6.2.2 Example 2 ($1 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10 \to 11 \to 9 \to 6$)

Now consider a different input string *abcdeacabecabababcde* and an abelian pattern $2a + 3b + 3c + d + e$. Figure 10 shows the transitions between states made by the parameterized suffix based algorithm in an attempt to find the matching abelian patterns in the text.

6.3 Analysis

The parameterized suffix based algorithm has the same best case complexity as the suffix based algorithm which is $\Theta(n/m)$. However, its worst case complexity is better than that of the suffix based algorithm.

Theorem 3. *The upper bound for* worst *case time complexity of the parameterized suffix based algorithm for abelian pattern matching in a given text of length n with pattern size m is $O(n/(1-\epsilon))$.*

Proof. If, for a given input text, the parameterized suffix based algorithm operates in such a manner that the search window moves along the whole text without resetting the contents of CFV, then the time complexity of the algorithm on that input would be similar to the time complexity of the prefix based algorithm, which is $O(n)$.

However, if during the execution of the algorithm a window is reset after an overflow, then we would have to process the reset characters again. In the parameterized suffix based algorithm, two type of *resets* occur:

1. The resets corresponding to a transition from *state 5* to *state 1* (Figure 8), and

2. The resets corresponding to a transition from any of the *states 4,8, or 11* to *state 1* (Figure 8).

In the resets corresponding to the transition from *state 5* to *state 1*, we read at most ϵm characters and advance the window by at least $(1-\epsilon)m$ positions, thus giving us a cost of $O(\epsilon/(1-\epsilon))$ per character.

In the resets corresponding to transitions from *states 4,8, or 11* to *state 1*, the cost to process one character can be computed as follows:

We start with a search window with no entry in CFV. Now we read X characters in the window and advance the window by $m - X$ positions. Note that $X > \epsilon m$, otherwise a reset corresponding to the transition from *state 5* to *state 1* would have taken place. We continue executing the algorithm and let Y be the number of characters processed (in addition to X) before the algorithm decides to reset the window. Let Z be the amount of information

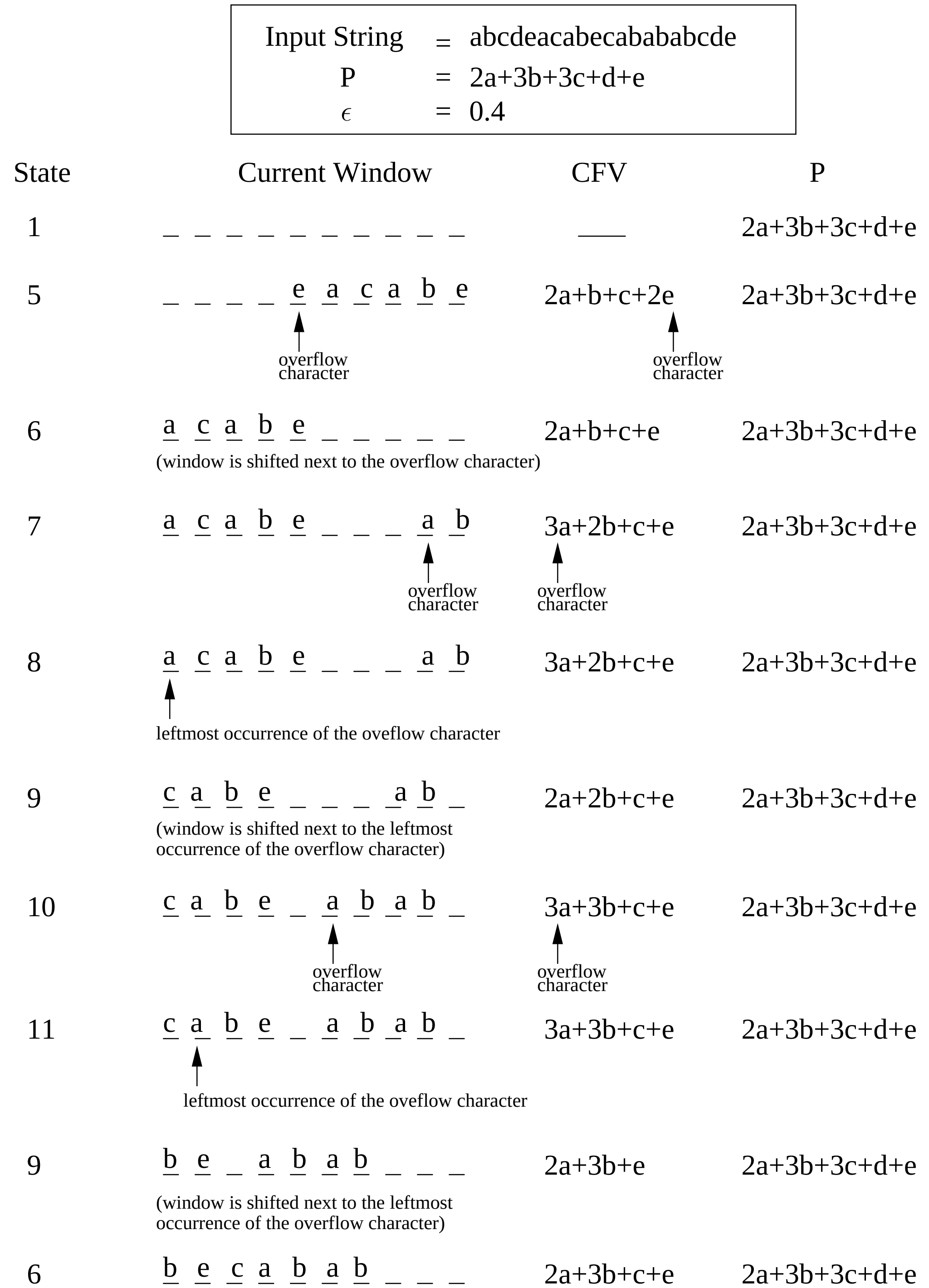

Figure 10: The path taken by the parameterized suffix based algorithm in the transition graph of Figure 8 for an input string *abcdeacabecabababcde* and an abelian pattern $2a + 3b + 3c + d + e$ with $\epsilon = 0.4$.

contained in CFV at the time of reset (clearly $Z \leq \epsilon m$). During this whole process, the window is advanced by $(m - X) + (X + Y - Z) = m + Y - Z$ positions along the input text. So we read $X + Y$ characters to advance the window by $m + Y - Z$ positions.

This gives the following cost per character:

$$\begin{array}{rll} & (X+Y)/(m+Y-Z) & \\ \leq & (m+Y)/(m+Y-Z) & \text{(since } m \geq X) \\ \leq & (m+Y)/(m+Y-\epsilon m) & \text{(since } Z \leq \epsilon m) \\ \leq & m/(m-\epsilon m) & \text{(since } (m/m-\epsilon m) > 1 \text{ and } Y > 0) \\ = & 1/(1-\epsilon) & \end{array}$$

Hence the complexity of the parameterized suffix based algorithm is bounded by $O(n/(1-\epsilon))$ in the worst case.

$\square$

7 Future Directions

Pattern matching in strings is an already established research area, however, abelian pattern matching is quite a new direction of research. The study of methods and algorithms for abelian pattern matching is still in its infancy and only little literature is available on this topic.

In this report we have presented two fundamental approaches to solve this problem and further showed how we can parameterize the suffix based approach to limit its disadvantage. We have also given a tight lower bound for this problem.

Now when we have algorithms for abelian pattern matching that run in linear time, the next step is to find algorithms that run sub-linearly with some preprocessing of the text. So we can think about indexing strategies for a given text in which we want to find abelian patterns.

References

[1] Sebastian Böcker. Sequencing from Compomers: The Puzzle. *Theory of Computing Systems*, 39(3):455–471, 2006.

[2] Mark Cieliebak, Thomas Erlebach, Zsuzsanna Lipták, Jens Stoye, and Emo Welzl. Algorithmic Complexity of Protein Identification: Combinatorics of Weighted Strings. *Discrete Applied Mathematics (DAM)*, 137(1):24–26, 2004.

[3] R. Eres, G. M. Landau, and L. Parida. Permutation Pattern Discovery in Biosequences. *Journal of Computational Biology*, 11(6):1050–1060, 2004.

[4] T.I. Fenner and G. Loizou. An Analysis of Two Related Loop-free Algorithms for Generating Integer Partitions. *Acta Informatica*, 16:237–252, 1981.

[5] R. N. Horspool. Practical Fast Searching in Strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

[6] S. Jukna. *Extremal Combinatorics With Applications in Computer Science*. Springer, 2001.

[7] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.

[8] A. Zoghbi and I. Stojmenovic. Fast Algorithms for Generating Integer Partitions. *International Journal of Computer Mathematics*, 70:319–332, 1998.

Bisher erschienene Reports an der Technischen Fakultät
Stand: 2008-05-07

94-01 Modular Properties of Composable Term Rewriting Systems
(Enno Ohlebusch)

94-02 Analysis and Applications of the Direct Cascade Architecture
(Enno Littmann, Helge Ritter)

94-03 From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction
(Robert Giegerich, Stefan Kurtz)

94-04 Die Verwendung unscharfer Maße zur Korrespondenzanalyse in Stereo Farbbildern
(André Wolfram, Alois Knoll)

94-05 Searching Correspondences in Colour Stereo Images – Recent Results Using the Fuzzy Integral
(André Wolfram, Alois Knoll)

94-06 A Basic Semantics for Computer Arithmetic
(Markus Freericks, A. Fauth, Alois Knoll)

94-07 Reverse Restructuring: Another Method of Solving Algebraic Equations
(Bernd Bütow, Stephan Thesing)

95-01 PaNaMa User Manual V1.3
(Bernd Bütow, Stephan Thesing)

95-02 Computer Based Training-Software: ein interaktiver Sequenzierkurs
(Frank Meier, Garrit Skrock, Robert Giegerich)

95-03 Fundamental Algorithms for a Declarative Pattern Matching System
(Stefan Kurtz)

95-04 On the Equivalence of E-Pattern Languages
(Enno Ohlebusch, Esko Ukkonen)

96-01 Static and Dynamic Filtering Methods for Approximate String Matching
(Robert Giegerich, Frank Hischke, Stefan Kurtz, Enno Ohlebusch)

96-02 Instructing Cooperating Assembly Robots through Situated Dialogues in Natural Language
(Alois Knoll, Bernd Hildebrand, Jianwei Zhang)

96-03 Correctness in System Engineering
(Peter Ladkin)

96-04 An Algebraic Approach to General Boolean Constraint Problems
(Hans-Werner Güsgen, Peter Ladkin)

96-05 Future University Computing Resources
(Peter Ladkin)

96-06 Lazy Cache Implements Complete Cache
(Peter Ladkin)

96-07 Formal but Lively Buffers in TLA+
(Peter Ladkin)

96-08 The X-31 and A320 Warsaw Crashes: Whodunnit?
(Peter Ladkin)

96-09 Reasons and Causes
(Peter Ladkin)

96-10 Comments on Confusing Conversation at Cali
(Dafydd Gibbon, Peter Ladkin)

96-11 On Needing Models
(Peter Ladkin)

96-12 Formalism Helps in Describing Accidents
(Peter Ladkin)

96-13 Explaining Failure with Tense Logic
(Peter Ladkin)

96-14 Some Dubious Theses in the Tense Logic of Accidents
(Peter Ladkin)

96-15 A Note on a Note on a Lemma of Ladkin
(Peter Ladkin)

96-16 News and Comment on the AeroPeru B757 Accident
(Peter Ladkin)

97-01 Analysing the Cali Accident With a WB-Graph
(Peter Ladkin)

97-02 Divide-and-Conquer Multiple Sequence Alignment
(Jens Stoye)

97-03 A System for the Content-Based Retrieval of Textual and Non-Textual Documents Based on Natural Language Queries
(Alois Knoll, Ingo Glöckner, Hermann Helbig, Sven Hartrumpf)

97-04 Rose: Generating Sequence Families
(Jens Stoye, Dirk Evers, Folker Meyer)

97-05 Fuzzy Quantifiers for Processing Natural Language Queries in Content-Based Multimedia Retrieval Systems
(Ingo Glöckner, Alois Knoll)

97-06 DFS – An Axiomatic Approach to Fuzzy Quantification
(Ingo Glöckner)

98-01 Kognitive Aspekte bei der Realisierung eines robusten Robotersystems für Konstruktionsaufgaben
(Alois Knoll, Bernd Hildebrandt)

98-02 A Declarative Approach to the Development of Dynamic Programming Algorithms, applied to RNA Folding
(Robert Giegerich)

98-03 Reducing the Space Requirement of Suffix Trees
(Stefan Kurtz)

99-01 Entscheidungskalküle
(Axel Saalbach, Christian Lange, Sascha Wendt, Mathias Katzer, Guillaume Dubois, Michael Höhl, Oliver Kuhn, Sven Wachsmuth, Gerhard Sagerer)

99-02 Transforming Conditional Rewrite Systems with Extra Variables into Unconditional Systems
(Enno Ohlebusch)

99-03 A Framework for Evaluating Approaches to Fuzzy Quantification
(Ingo Glöckner)

99-04 Towards Evaluation of Docking Hypotheses using elastic Matching
(Steffen Neumann, Stefan Posch, Gerhard Sagerer)

99-05 A Systematic Approach to Dynamic Programming in Bioinformatics. Part 1 and 2: Sequence Comparison and RNA Folding
(Robert Giegerich)

99-06 Autonomie für situierte Robotersysteme – Stand und Entwicklungslinien
(Alois Knoll)

2000-01 Advances in DFS Theory
(Ingo Glöckner)

2000-02 A Broad Class of DFS Models
(Ingo Glöckner)

2000-03 An Axiomatic Theory of Fuzzy Quantifiers in Natural Languages
(Ingo Glöckner)

2000-04 Affix Trees
(Jens Stoye)

2000-05 Computergestützte Auswertung von Spektren organischer Verbindungen
(Annika Büscher, Michaela Hohenner, Sascha Wendt, Markus Wiesecke, Frank Zöllner, Arne Wegener, Frank Bettenworth, Thorsten Twellmann, Jan Kleinlützum, Mathias Katzer, Sven Wachsmuth, Gerhard Sagerer)

2000-06 The Syntax and Semantics of a Language for Describing Complex Patterns in Biological Sequences
(Dirk Strothmann, Stefan Kurtz, Stefan Gräf, Gerhard Steger)

2000-07 Systematic Dynamic Programming in Bioinformatics (ISMB 2000 Tutorial Notes)
(Dirk J. Evers, Robert Giegerich)

2000-08 Difficulties when Aligning Structure Based RNAs with the Standard Edit Distance Method
(Christian Büschking)

2001-01 Standard Models of Fuzzy Quantification
(Ingo Glöckner)

2001-02 Causal System Analysis
(Peter B. Ladkin)

2001-03 A Rotamer Library for Protein-Protein Docking Using Energy Calculations and Statistics
(Kerstin Koch, Frank Zöllner, Gerhard Sagerer)

2001-04 Eine asynchrone Implementierung eines Microprozessors auf einem FPGA
(Marco Balke, Thomas Dettbarn, Robert Homann, Sebastian Jaenicke, Tim Köhler, Henning Mersch, Holger Weiss)

2001-05 Hierarchical Termination Revisited
(Enno Ohlebusch)

2002-01 Persistent Objects with O2DBI
(Jörn Clausen)

2002-02 Simulation von Phasenübergängen in Proteinmonoschichten
(Johanna Alichniewicz, Gabriele Holzschneider, Morris Michael, Ulf Schiller, Jan Stallkamp)

2002-03 Lecture Notes on Algebraic Dynamic Programming 2002
(Robert Giegerich)

2002-04 Side chain flexibility for 1:n protein-protein docking
(Kerstin Koch, Steffen Neumann, Frank Zöllner, Gerhard Sagerer)

2002-05 ElMaR: A Protein Docking System using Flexibility Information
(Frank Zöllner, Steffen Neumann, Kerstin Koch, Franz Kummert, Gerhard Sagerer)

2002-06 Calculating Residue Flexibility Information from Statistics and Energy based Prediction
(Frank Zöllner, Steffen Neumann, Kerstin Koch, Franz Kummert, Gerhard Sagerer)

2002-07 Fundamentals of Fuzzy Quantification: Plausible Models, Constructive Principles, and Efficient Implementation
(Ingo Glöckner)

2002-08 Branching of Fuzzy Quantifiers and Multiple Variable Binding: An Extension of DFS Theory
(Ingo Glöckner)

2003-01 On the Similarity of Sets of Permutations and its Applications to Genome Comparison
(Anne Bergeron, Jens Stoye)

2003-02 SNP and mutation discovery using base-specific cleavage and MALDI-TOF mass spectrometry
(Sebastian Böcker)

2003-03 From RNA Folding to Thermodynamic Matching, including Pseudoknots
(Robert Giegerich, Jens Reeder)

2003-04 Sequencing from compomers: Using mass spectrometry for DNA de-novo sequencing of 200+ nt
(Sebastian Böcker)

2003-05 Systematic Investigation of Jumping Alignments
(Constantin Bannert)

2003-06 Suffix Tree Construction and Storage with Limited Main Memory
(Klaus-Bernd Schürmann, Jens Stoye)

2003-07 Sequencing from compomers in thepresence of false negative peaks
(Sebastian Böcker)

2003-08 Genalyzer: An Interactive Visualisation Tool for Large-Scale Sequence Matching – Biological Applications and User Manual
(Jomuna V. Choudhuri, Chris Schleiermacher)

2004-01 Sequencing From Compomers is NP-hard
(Sebastian Böcker)

2004-02 The Money Changing Problem revisited: Computing the Frobenius number in time
$O(k\,a_1)$
(Sebastian Böcker, Zsuzsanna Lipták)

2004-03 Accelerating the Evaluation of Profile HMMs by Pruning Techniques
(Thomas Plötz, Gernot A. Fink)

2004-04 Optimal Group Testing Strategies with Interval Queries and Their Application to Splice Site Detection
(Ferdinando Cicalese, Peter Damaschke, Ugo Vaccaro)

2004-05 Compressed Representation of Sequences and Full-Text Indexes
(Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, Gonzalo Navarro)

2005-01 Overlaps Help: Improved Bounds for Group Testing with Interval Queries
(Ferdinando Cicalese, Peter Damaschke, Libertad Tansini, Sören Werth)

2005-02 Two batch Fault-tolerant search with error cost constraints: An application to learning
(Ferdinando Cicalese)

2005-03 Searching for the Shortest Common Supersequence
(Sergio A. de Carvalho Jr., Sven Rahmann)

2005-04 Counting Suffix Arrays and Strings
(Klaus-Bernd Schürmann, Jens Stoye)

2005-05 Alignment of Tandem Repeats with Excision, Duplication, Substitution and Indels (EDSI)
(Michael Sammeth, Jens Stoye)

2005-06 Statistics of Cleavage Fragments in Random Weighted Strings
(Hans-Michael Kaltenbach, Henner Sudek, Sebastian Böcker, Sven Rahmann)

2006-01 Decomposing metabolomic isotope patterns
(Sebastian Böcker, Zsuzsanna Lipták, Anton Pervukhin)

2006-02 On Common Intervals with Errors
(Cedric Chauve, Yoan Diekmann, Steffen Heber, Julia Mixtacki, Sven Rahmann, Jens Stoye)

2007-01 Identifying metabolites with integer decomposition techniques, using only their mass spectrometric isotope patterns
(Sebastian Böcker, Matthias C. Letzel, Zsuzsanna Lipták, Anton Pervukhin)

2007-02 2-Stage Fault Tolerant Interval Group Testing
(Ferdinando Cicalese, José Augusto Amgarten Quitzau)