Heikki Hyyrö

Practical Methods for Approximate String Matching

ACADEMIC DISSERTATION

To be presented, with the permission of the Faculty of Information Sciences of the University of Tampere, for public discussion in the B1096 Auditorium of the University on December 5th, 2003, at 12 noon.

DEPARTMENT OF COMPUTER SCIENCES
UNIVERSITY OF TAMPERE

A-2003-4

TAMPERE 2003

Opponent:	Prof. Esko Ukkonen Department of Computer Science University of Helsinki
Reviewers:	Prof. Jorma Tarhio Laboratory of Information Processing Science Helsinki University of Technology
	Prof. Jukka Teuhola Department of Information Technology University of Turku

Department of Computer Sciences
FIN-33014 UNIVERSITY OF TAMPERE
Finland

Electronic dissertation
Acta Electronica Universitatis Tamperensis 308
ISBN 951-44-5840-0
ISSN 1456-954X
http://acta.uta.fi

ISBN 951-44-5818-4
ISSN 1459-6903

Tampereen yliopistopaino Oy
Tampere 2003

Abstract

Given a pattern string and a text, the task of approximate string matching is to find all locations in the text that are similar to the pattern. This type of search may be done for example in applications of spelling error correction or bioinformatics. Typically edit distance is used as the measure of similarity (or distance) between two strings. In this thesis we concentrate on unit-cost edit distance that defines the distance beween two strings as the minimum number of edit operations that are needed in transforming one of the strings into the other. More specifically, we discuss the Levenshtein and the Damerau edit distances.

Aproximate string matching algorithms can be divided into off-line and on-line algorithms depending on whether they may or may not, respectively, preprocess the text. In this thesis we propose practical algorithms for both types of approximate string matching as well as for computing edit distance.

Our main contributions are a new variant of the bit-parallel approximate string matching algorithm of Myers, a method that makes it easy to modify many existing Levenshtein edit distance algorithms into using the Damerau edit distance, a bit-parallel algorithm for computing edit distance, a more error tolerant version of the ABNDM algorithm, a two-phase filtering scheme, a tuned indexed approximate string matching method for genome searching, and an improved and extended version of the hybrid index of Navarro and Baeza-Yates.

To evaluate their practicality, we compare most of the proposed methods with previously existing algorithms. The test results support the claim of the title of this thesis that our proposed algorithms work well in practice.

Keywords:
Edit distance, approximate string matching, approximate pattern matching

Acknowledgements

This thesis is based on the work that was carried out between the years 2000 and 2003 at the Department of Computer Sciences, University of Tampere. Some early groundwork was done already in 1999 during the research done for my master's thesis on string matching. The following people had the most important role in making this work possible.

First of all I would like to thank my primary supervisor, Professor Martti Juhola. Ever since he recruited me in 1998 to do research under his supervision, he has offered me excellent work environment as well as various kinds of general support and advice. I am thankful for him for trusting me enough to let me work in a very autonomous and free manner. I am grateful that Martti remained supportive even when some of my decisions on the direction of the work and the contents of the thesis resulted in prolonging the preparation of the thesis.

The second person to thank is Professor Mauno Vihinen. He inroduced me in 1998 to an application in bioinformatics that involved exact and approximate string matching, and in that he is mainly responsible for the topic of the thesis. In fact, large parts of the thesis have been done in conjunction with that original application.

Without doubt the most influential person in terms of the eventual contents of the thesis has been Gonzalo Navarro, who at the moment of writing this is an Associate Professor at the University of Chile. We started to work together after meeting in the SPIRE conference in Chile in November 2001, and I now regard him to be not only a work partner but also a long-distance friend. He has been, perhaps unknowingly, an unofficial supervisor of this work. As one of the leading researchers in the field of string matching, Gonzalo has provided me with inspiration as well as excellent advice and insight into the field.

Throughout the work I have also enjoyed the general environment provided by the Department of Computer Sciences at the University of Tampere. I will not mention separately everyone involved in making the department a pleasant place. However, as one exception I would like to thank Professor Erkki Mäkinen. He has always been prepared to help for example by proof-reading article manuscripts.

Contents

Introduction 1

Basic Notation 2

I Edit Distance and Approximate String Matching 3

1 Levenshtein and Damerau Edit Distance 5

2 Dynamic Programming 7

3 Filling Only a Necessary Portion of the Dynamic Programming Matrix 11

4 Bit-parallel Methods 15

II Our Contributions 17

5 Our Variant of the Algorithm of Myers 19

6 Adding Transposition into the Bit-parallel Methods 27

7 Using Diagonal Tiling in Computing Edit Distance 37

8 Using BPM in ABNDM 43

9 On Using Two-Phase Filtering 59

10 A Practical Index for Genome Searching 73

11 An Improvement and an Extension on the Hybrid Index of Navarro & Baeza-Yates 87

Conclusion 99

Bibliography 101

Introduction

Finding the occurrences of a given query string (pattern) from a possibly very large text is an old and fundamental problem in computer science. It emerges in applications ranging from text processing and music retrieval to bioinformatics. This task, collectively known as string matching, has several different variations. The most natural and simple of these is exact string matching, in which, like the name suggests, one wishes to find only occurrences that are exactly identical to the pattern string. This type of search, however, may not be adequate in all applications if for example the pattern string or the text may contain typographical errors. Perhaps the most important applications of this kind arise in the field of bioinformatics, as small variations are fairly common in DNA or protein sequences. The field of approximate string matching, which has been a research subject since the 1960's, answers the problem of small variation by permitting some error between the pattern and its occurrences. Given an error threshold and a metric to measure the distance between two strings, the task of approximate string matching is to find all substrings of the text that are within (a distance of) the error threshold from the pattern.

In this work we concentrate on approximate string matching that uses so called unit-cost edit distance as the metric to measure the distance between two strings. We consider two different kinds of edit distances: the Levenshtein edit distance and the Damerau edit distance. These two, and especially the Levenshtein edit distance, are the most commonly used forms of unit-cost edit distance.

Most of the research underlying this thesis has been inclined towards practical results. The primary aim has been to develop methods that work well in practice. Therefore theoretical considerations have been given a slightly secondary role. A major reason for this choice is that much of the work has been done in conjunction with a real-life application: applying string matching in searching for unique oligonucleotides in a large DNA genome [19, 26, 18, 23]. The term *oligonucleotide* refers to a fairly short sequence of DNA.

In the first part we present a concise overview of edit distance and approximate string matching. This gives the basic background for part two, in which we present our primary contributions.

Basic Notation

We will use the following notation throughout the thesis.

String characters will be indexed with a subscript: P_i refers to the ith character of the string P, and $P_{i..j}$ to its *substring* that begins from the ith character and ends at the jth character. $|P|$ is the length of P. The first character has the index 1, and so $P = P_{1..|P|}$. We interpret the non-existing substring $P_{1..0}$ as the empty character ϵ. The superscript R denotes the reverse of the string. For example if $P =$ "abcd", then $P^R =$ "dcba", $P^R_{1..2} =$ "dc" and $(P_{1..2})^R =$ "ba". Note the last two examples that show how we may use parentheses to differentiate between a substring of the reversed string and a reversed substring. The notation $P \circ T$ denotes the concatenation of the strings P and T. For example if $P =$ "abc" and $T =$ "def", then $P \circ T =$ "abcdef".

The string B is a *subsequence* of the string A if $B_i = A_{x(i)}$ for $i = 1..|B|$, where $x(i)$ is a mapping that fulfills the conditions $1 \leq x(i) \leq |A|$ for $i = 1..|B|$ and $x(i-1) < x(i)$ for $i = 2..|B|$. Thus B is a subsequence of A if the characters $B_1, B_2, \ldots, B_{|B|}$ appear in the same order, but not necessarily consecutively, in A.

For sake of uniformity, the two compared strings in the context of computing edit distance are denoted by P and T. In the context of approximate string matching P is a *pattern* and T is the *text*. It is a standard practice in the literature to denote the length $|P|$ of P by m and the length $|T|$ of T by n. Throughout the text we assume that $m \leq n$.

Σ denotes the used alphabet and σ the size (number of different characters) of Σ. In addition k denotes the maximum allowed error in the context of thresholded edit distance or approximate string matching, and w is the size (number of bits) of the computer word. The Levenshtein edit distance between the strings P and T will be denoted by $ed_L(P,T)$ and the Damerau edit distance by $ed_D(P,T)$.

Bit-operations are described as follows: '&' denotes bitwise "AND", '|' denotes bitwise "OR", '^' denotes bitwise "XOR", '~' denotes bit complementation, and '<<' and '>>' denote shifting the bit-vector left and right, respectively, using zero filling in both directions. The ith bit of the bit vector V is referred to as $V[i]$ and bit-positions are assumed to grow from right to left. In addition we use a superscript to denote bit-repetition. As an example let $V = 1001110$ be a bit vector. Then $V[1] = V[5] = V[6] = 0$, $V[2] = V[3] = V[4] = V[7] = 1$, and we could also write $V = 10^2 1^3 0$.

Part I
Edit Distance and Approximate String Matching

The edit distance $ed(P, T)$ between the strings P and T is defined in general as the minimum cost of any sequence of edit operations that edits P into T or vice versa. Differing in their choices of the allowed set of edit operations and their costs, for example the following types of edit distance have appeared in the literature:

Levenshtein edit distance [31]: The allowed edit operations are insertion, deletion or substitution of a single character, and each operation has the cost 1. This type of edit distance is sometimes called *unit-cost edit distance*. Levenshtein edit distance is perhaps the most common form of edit distance, and often the term *edit distance* is assimilated to it.

Damerau edit distance [9]: Otherwise identical to the Levenshtein edit distance, but allows also the fourth operation of transposing two adjacent characters. A further condition is that the transposed characters must be adjacent before and after the edit operations are applied.

Weighted/generalized edit distance [56, 33]: Allows the same operations as the Levenshtein/Damerau edit distance, respectively, but each operation may have an arbitrary cost.

Hamming distance: Allows only the operation of substituting a character, and each substitution has the cost 1.

Longest common subsequence: Measures the similarity between P and T by the length of their longest common subsequence. This is in effect equivalent to allowing the edit operations of deleting or inserting a single character with the cost 1.

This work concentrates on the first two types of edit distance: unit-cost Levenshtein and Damerau edit distance.

Sometimes one wishes to check whether two strings are within some pre-determined distance threshold of each other. That is, given a distance threshold k and strings P and T, one wishes to check whether $ed(P, T) \leq k$.

In this thesis we will refer to this kind of "edit distance check" as *thresholded edit distance computation*.

Approximate string matching is closely related to edit distance. It refers to searching for approximate matches of a pattern string P from a usually much longer text string T, where edit distance is used as the measure of similarity between P and the substrings of T. Typically there is a predetermined maximum error threshold k that gives the maximum allowed edit distance for an approximate match. We will concentrate on this type of approximate string matching, which can be defined more formally as finding the text indices j for which $ed(P, T_{j-h..j}) \leq k$ for some $h \geq 0$. Other more rare variants of approximate string matching include for example searching for the best match(es) for P.

1 Levenshtein and Damerau Edit Distance

The unit-cost Levenshtein edit distance between the strings $P = P_{1..m}$ and $T = T_{1..n}$, denoted by $ed_L(P, T)$, can be defined as the minimum number of single-character insertions, deletions and substitutions needed in transforming P into T or vice versa. For example if P = "cat" and T = "act", then $ed_L(P, T) = 2$ as at least two single-character insertions, deletions or substitutions are needed in converting "cat" into "act". There are two ways to transform P into T with exactly two operations: either delete P_1 = 'c' and insert a 'c' between P_2 = 'a' and P_3 = 't', or substitute P_1 = 'c' with an 'a' and P_2 = 'a' with a 'c' (Fig. 1a).

a) delete 'c': **c**at → at, insert 'c': at → a**c**t
substitute 'c'→ 'a': **c**at → **a**at, substitute 'a'→ 'c': a**a**t → a**c**t

b) transpose "ca": **ca**t → **ac**t

Figure 1: The two rows in Figure a) show the two ways of editing the string "cat" into the string "act" that correspond to ed_L("cat","act") = 2. Figure b) shows how ed_D("cat","act") = 1 as the Damerau edit distance allows also the operation of transposing two adjacent characters.

In similar fashion, the unit-cost Damerau edit distance $ed_D(P, T)$ can be defined as the minimum number of single-character insertions, deletions or substitutions or transpositions between two permanently adjacent characters that are needed in transforming P into T or vice versa. Continuing with the same example of the strings P = "cat" and T = "act", we have that $ed_D(P, T) = 1$ as now a single transposition of the characters P_1 = 'c' and P_2 = 'a' is enough to convert P into T (Fig. 1b).

2 Dynamic Programming

Both the Levenshtein and the Damerau edit distance suit well the technique of dynamic programming. In the case of the Levenshtein edit distance basically the same way of using dynamic programming has been independently "invented" several times by different authors [41]. We begin by discussing the dynamic programming algorithm for the Levenshtein edit distance.

The dynamic programming algorithm computes the desired final solution $ed_L(P,T)$ by filling incrementally an $(m+1) \times (n+1)$ dynamic programming table D, in which each cell $D[i,j]$ will eventually hold the value $ed_L(P_{1..i}, T_{1..j})$ for $i = 0..m$ and $j = 0..n$. Initially the trivially known "boundary values" of form $D[i,0] = ed_L(P_{1..i}, T_{1..0}) = ed_L(P_{1..i}, \epsilon) = i$ and $D[0,j] = ed_L(P_{1..0}, T_{1..j}) = ed_L(\epsilon, T_{1..j}) = j$ are filled. Then the computation proceeds incrementally using the second row of the following well-known Recurrence 1.

Recurrence 1: Computing the Levenshtein edit distance.

$$D[i,0] = i, D[0,j] = j.$$
$$D[i,j] = \begin{cases} D[i-1,j-1], \text{ if } P_i = T_j. \\ 1 + \min(D[i-1,j-1], D[i-1,j], D[i,j-1]), \text{ otherwise.} \end{cases}$$

The choices for filling the value $D[i,j]$ enumerate the different possibilities of editing the string $T_{1..j}$ into $P_{1..i}$ (or vice versa) after $T_{1..j-1}$ has already been edited into $P_{1..i-1}$, $T_{1..j}$ into $P_{1..i-1}$ or $T_{1..j-1}$ into $P_{1..i}$ (or vice versa). The first choice corresponds to the situation where $P_i = T_j$, and thus it adds no further cost to the value $D[i-1,j-1]$. If we choose the point of view of editing T into P, then the three possibilities inside the min-clause correspond, respectively, to substituting P_i by T_j, inserting P_j between the characters T_{j-1} and T_j, and deleting T_j.

The form of Recurrence 1 requires that the cell values $D[i-1,j-1]$, $D[i-1,j]$ and $D[i,j-1]$ are known before computing the cell value $D[i,j]$. It is common to fill the table D in either a row-wise (first the cells $D[1,1]..D[1,n]$, then $D[2,1]..D[2,n]$, and so on) or a column-wise manner (first the cells $D[1,1]..D[m,1]$, then $D[1,2]..D[m,2]$, and so on). Fig. 2a shows the dynamic programming table D for computing the Levenshtein edit distance between the strings P = "cat" and T = "act".

The dynamic programming algorithm for the Damerau edit distance works in a similar manner to the above algorithm. The only difference is that now also a choice for doing a transposition is added into the recurrence. Recurrence 2 shows the version for computing the Damerau edit distance.

a)

		a	c	t
	0	1	2	3
c	1	1	1	2
a	2	1	2	2
t	3	2	2	**2**

b)

		a	c	t
	0	1	2	3
c	1	1	1	2
a	2	1	1	2
t	3	2	2	**1**

Figure 2: The dynamic programming matrix D for computing the edit distance between the strings P = "cat" and T = "act". Figure a) is for the Levenshtein edit distance and Figure b) for the Damerau edit distance.

This recurrence is derived from Du and Chang [10].

Recurrence 2: Computing the Damerau edit distance.

$$D[i,0] = i, D[0,j] = j.$$

$$D[i,j] = \begin{cases} D[i-1,j-1], \text{ if } P_i = T_j. \\ 1 + \min(D[i-2,j-2], D[i-1,j], D[i,j-1]), \text{ if } P_{i-1..i} = (T_{j-1..j})^R. \\ 1 + \min(D[i-1,j-1], D[i-1,j], D[i,j-1]), \text{ otherwise.} \end{cases}$$

Fig. 2b shows the dynamic programming table D for computing the Damerau edit distance between the strings P = "cat" and T = "act".

Instead of computing the edit distance between the strings P and T, the dynamic programming algorithm can be modified to find approximate occurrences of P somewhere inside T by changing the boundary condition $D[0,j] = j$ into $D[0,j] = 0$. In this case $D[i,j] = \min(ed_L(P_{1..i}, T_{h..j}), h \leq j)$ with the Levenshtein edit distance and $D[i,j] = \min(ed_D(P_{1..i}, T_{h..j}), h \leq j)$ with the Damerau edit distance. Thus the situation corresponds to the earlier definition of approximate string matching. Fig. 3 shows the dynamic programming table D for searching for approximate occurrences of the pattern string P = "cat" from the text string T = "abradacabra" when the maximum allowed distance k is 1 and the Levenshtein edit distance is used.

These basic dynamic programming algorithms clearly have a run time and space consumption of $O(mn)$, as they fill $O(mn)$ cells and filling a single cell takes a constant number of operations and space. It is simple to diminish the needed space into $O(m)$ when column-wise filling order of D is used: When column j is filled, only the cell values in one or two previous columns are needed, depending on whether the Levenshtein or the Damerau distance is used. This means that it is enough to have only column $(j-1)$

		a	b	r	a	d	a	c	a	b	r	a
	0	0	0	0	0	0	0	0	0	0	0	0
c	1	1	1	1	1	1	1	0	1	1	1	1
a	2	1	2	2	1	2	1	1	0	1	2	1
t	3	2	2	3	2	2	2	2	1	1	2	2

Figure 3: The dynamic programming matrix D for approximate matching the pattern $P =$ "cat" in the text $T =$ "abradacabra" under the Levenshtein edit distance.

or also column $(j - 2)$ in memory when computing column j, and so the needed space is $O(m)$.

It is straightforward to verify that the following properties hold for both the edit distance computation and approximate string matching versions of D under both the Levenshtein and the Damerau edit distance.

-The diagonal property:	$D[i, j] - D[i - 1, j - 1] = 0$ or 1.
-The adjacency property:	$D[i, j] - D[i, j - 1] = -1, 0,$ or 1, and
	$D[i, j] - D[i - 1, j] = -1, 0,$ or 1.

In the text we will sometimes refer to a *q-diagonal* or *diagonal* q. By this we mean the diagonal of D that consists of the cells $D[i, j]$ for which $j - i = q$. For example the 0-diagonal (or diagonal 0) consists of the cells $D[0, 0], D[1, 1], D[2, 2], \ldots$, and the -2-diagonal (or diagonal -2) of the cells $D[2, 0], D[3, 1], D[4, 2], \ldots$.

An *edit path* is a concept that is closely related to the dynamic programming matrix D. An edit path $EP_{P,T}$ between the strings P and T traces the cells $D[i, j]$ that correspond to an optimal sequence of edit operations that transforms P into T (or vice versa) with the minimum total cost (e.g. $ed_L(P, T)$ under the Levenshtein edit distance).

In the case of computing edit distance, $EP_{P,T}$ can be generated by first filling D and then backtracking from the cell $D[|P|, |T|]$ to the cell $D[0, 0]$ by tracing the recurrence backwards. This is done by always moving from $D[i, j]$ into any one of the cells $D[i - 1, j - 1]$, $D[i - 1, j]$ and $D[i, j - 1]$ (or also $D[i-2, j-2]$ with the Damerau edit distance) that has a legal value in terms of the dynamic programming recurrence. At each backtracking step the corresponding edit operation may be recorded. Note that there may be more than one edit path for P and T.

In the case of approximate string matching an edit path $EP_{P,T_{j-h..j}}$ that corresponds to P and a suffix $T_{j-h..j}$ of $T_{1..j}$ that matches best with P can be traced in a similar manner as above. The only difference is that now the backtracking starts from $D[m,j]$ and ends at some cell $D[0,j-h]$ with $h \geq 0$.

Fig. 4 shows in bold the cells that belong to some edit path in the dynamic programming tables of Fig. 2.

a)

		a	c	t
	0	**1**	2	3
c	**1**	**1**	**1**	2
a	2	**1**	**2**	2
t	3	2	2	**2**

b)

		a	c	t
	0	1	2	3
c	1	1	1	2
a	2	1	**1**	2
t	3	2	2	**1**

Figure 4: The dynamic programming matrix D for computing the edit distance between the strings P = "cat" and T = "act". Figure a) is for the Levenshtein edit distance and Figure b) for the Damerau edit distance. The cells that belong to some edit path are shown in bold. In Figure a) there are several edit paths. From the cell $D[3,3]$ we can backtrack only to the cell $D[2,2]$, which corresponds to the match $P_3 = T_3$, but from there on there are many options. We can for example continue to the cell $D[1,1]$, which corresponds to a substitution between P_2 and T_2, and from there to the cell $D[0,0]$, which corresponds to a substitution between P_1 and T_1. Another possibility would be to continue from the cell $D[2,2]$ to the cell $D[2,1]$, which corresponds to inserting T_2 before P_3 or deleting T_2 from T, then to the cell $D[1,0]$, which corresponds to the match $P_2 = T_1$, and finally to the cell $D[0,0]$, which corresponds to deleting P_1 from P or inserting P_1 before T_1. In Figure b) there is only one edit path. From the cell $D[3,3]$ we can backtrack only to the cell $D[2,2]$, which corresponds to matching the last characters of "act" and "cat", and from there we can continue only to the cell $D[0,0]$, which corresponds to transposing the first two characters.

3 Filling Only a Necessary Portion of the Dynamic Programming Matrix

In this section we review three basic methods for trying to restrict the number of cells that are filled in the dynamic programming table D. Each of these methods is based on the diagonal and/or the adjacency property. As these properties hold in both the case of the Levenshtein and the Damerau edit distance, the restriction methods work also with the Damerau edit distance although they were initially presented for the Levenshtein edit distance.

3.1 Diagonal Restriction in Computing Edit Distance

Ukkonen [53] presented the following method to try to cut down the number of cells filled in D when computing the Levenshtein edit distance between P and T. From the diagonal and adjacency properties Ukkonen concluded that if $ed_L(P,T) \leq t$ and $m \leq n$, then it is sufficient to fill only the cells in the diagonals $-\lfloor (t-n+m)/2 \rfloor, -\lfloor (t-n+m)/2 \rfloor + 1, \ldots, \lfloor (t+n-m)/2 \rfloor$ of the dynamic programming matrix. All other cell values are known to have a value larger than $D[m,n] = ed(P,T)$, and thus they can be ignored. Ukkonen used this rule by beginning with $t = (n-m)+1$ and filling the above-mentioned diagonal interval of the dynamic programming matrix. If the result is $D[m,n] > t$, t is doubled. Eventually $D[m,n] \leq t$, and in this case it is known that $D[m,n] = ed(P,T)$. The run time of this procedure is dominated by the computation involving the last value of t. As this value is $< 2 \times ed(P,T)$ and with each value of t the computation takes $O(t \times \min(m,n))$ time, the overall run time is $O(ed(P,T) \times \min(m,n))$.

This method of "guessing" a starting limit t for the edit distance and then doubling it if necessary is not really practical for actual edit distance computation. Even though the asymptotic run time is good, it involves a large constant factor whenever $ed(P,T)$ is large. But the method works well in practice in thresholded edit distance computation, as then one can immediately set $t = k$ and only a single pass is needed.

3.2 Cut-off

Ukkonen [54] has also proposed a dynamic *cut-off* method to allow filling less cells in the dynamic programming matrix when the problem involves a distance threshold k. The idea is very simple: The cells with a value $> k$ are redundant in terms of a task with an error threshold k. It follows from

the diagonal property that once $D[i,j] > k$, then the cells $D[i+h, j+h]$, where $h \geq 0$, are known to be redundant from that point on. Assume that column-wise order is used in filling D. Let r_u hold the row number of the upmost and r_l the row number of the lowest cell that is deemed to have to be filled in column j. Initially when $j = 1$, the boundary condition $D[i, 0] = i$ means that $r_u = 1$ and $r_l = k + 1$. At each column j the cells $D[i, j]$ are filled for $i = r_u..r_l$, and after this computation r_u is set to record the index of the upmost and r_l the index of the lowest row with a value $\leq k$ in the column. Finally both r_u and r_l should be incremented to reflect the fact that the cut-off actually restricts diagonals, not rows, and the next cell along a diagonal is one row down. If r_l and r_u become non-existing, the region that needs to be filled vanishes, which means that no more cells can have a value $\leq k$. Note that because $D[0, j] = 0$ for all j in the case of approximate string matching, one can only use the limit r_l in that case as then always $r_u = 1$.

It has been shown in [4, 2] that using the cut-off method leads into an expected run time of $O(kn)$ in the case of approximate string matching under the Levenshtein edit distance.

3.3 Greedy Filling Order

Another restriction method is to fill the cells of D in a greedy fashion: first all the cells that get the value 0 are filled, then the cells that get the value 1, then the cells that get the value 2, and so on until all necessary cells are filled. This is done in a diagonal-wise manner by taking advantage of the diagonal property. As the values along each q-diagonal are non-decreasing and moving from one diagonal to another costs one operation, all cells on a q-diagonal that get the value x can be filled if all cells with a value $x - 1$ have already been filled. Let us consider first Recurrence 1 for the Levenshtein edit distance. If the cell $D[i, j]$ has the value x, then also the cells $D[i + h, j + h]$, where $h = \max(z \mid (P_{i+1..i+z} = T_{j+1..j+z}) \vee (D[i + z, j - 1 + z] = x - 1) \vee (D[i - 1 + z, j + z] = x - 1))$, will get the value x. From the diagonal property it follows that $D[i + h + y, j + h + y] > x$ for $y > 0$. Note that in particular $D[i+h+1, j+h+1] = x+1$ if $i+h+1 \leq m$ and $j + h + 1 \leq n$. In the case of Recurrence 2 for the Damerau edit distance we need to include the option of a transposition, which results in the condition $h = \max(z \mid (P_{i+1..i+z} = T_{j+1..j+z}) \vee ((D[i-2+z, j-2+z] = x-1) \wedge (P_{i-1+z..i+z} = (T_{j-1+z..j+z})^R)) \vee (D[i+z, j-1+z] = x-1) \vee (D[i-1+z, j+z] = x-1))$.

The diagonal property allows us to represent the cell values on a q-

diagonal by recording for it each row i where the value increases by one, that is, those indices i where $D[i, i+q] - D[i-1, i+q-1] = 1$.

Ukkonen [53] initially proposed using the greedy algorithm in the case of computing edit distance. A straightforward implementation runs in $O(dm)$ worst-case and $O(m+d^2)$ expected time, where d is the edit distance between the compared strings. By using a suffix tree also a worst-case time of $O(m+k^2)$ is achievable.

Later Landau & Vishkin modified the greedy method for use in approximate string matching, achieving first $O(k^2n)$ [29] and later $O(kn)$ [30] worst-case run time. There have subsequently appeared also numerous other variants of the scheme (e.g. [37, 11, 5]).

4 Bit-parallel Methods

During the 1990's the technique of *bit-parallelism* broke through in the field of string matching. This type of algorithms are based on exploiting the fact that a computer manipulates data in chunks of w bits, where w is the computer word size (typically $w = 32$ or 64 in a modern computer). The idea is to encode multiple data items of an algorithm into a single computer word and thereby be able to handle many items in parallel during a single computer operation (thus the name "bit-parallelism").

The first bit-parallel approximate string matching algorithm was given by Wu & Manber [60] and achieved a run time of $O(k\lceil m/w\rceil n)$. Then Wright [58] presented an $O(\lceil m\log(\sigma)/w\rceil n)$ algorithm and Baeza-Yates & Navarro [2] followed with an $O(\lceil km/w\rceil n)$ algorithm. Finally Myers [38] achieved the optimal speedup over the basic dynamic programming algorithm with his $O(\lceil m/w\rceil n)$ algorithm.

In a recent survey by Navarro [41], the bit-parallel algorithms of Wu & Manber [60], Baeza-Yates & Navarro [2] and Myers [38] dominated the other approximate string matching algorithms for most part when testing with English text and pattern lengths up to $m = 100$. The only case where a more "traditional" algorithm was faster was when both m and k were large. In that case the algorithm of Wu, Manber & Myers [61] was reported to be the fastest. This result, however, is not completely clear. To check this, we compared the bit-parallel algorithm of Myers against the algorithm of Wu, Manber & Myers when the error level k/m is large. The comparison involved patterns of lengths 10, 20,..., 150, and with each pattern length m the tested k values were $k = 0.5m$, $k = 0.7m$ and $k = 0.9m$. There were 50 randomly picked patterns for each (m, k)-combination, and the searched text was a $\approx$ 10 MB sample from Wall Street Journal articles taken from the TREC-collection [14]. The test was done on both a Sparc Ultra 2 workstation and a 600 MHz Pentium 3 to see how the underlying architecture affects the comparison. Navarro used a Sparc Ultra 1 in his tests. Fig. 5 shows the results. It can be seen that the bit-parallel algorithm of Myers is always faster, and the cases where the margin is quite small do not agree with Navarro's findings.

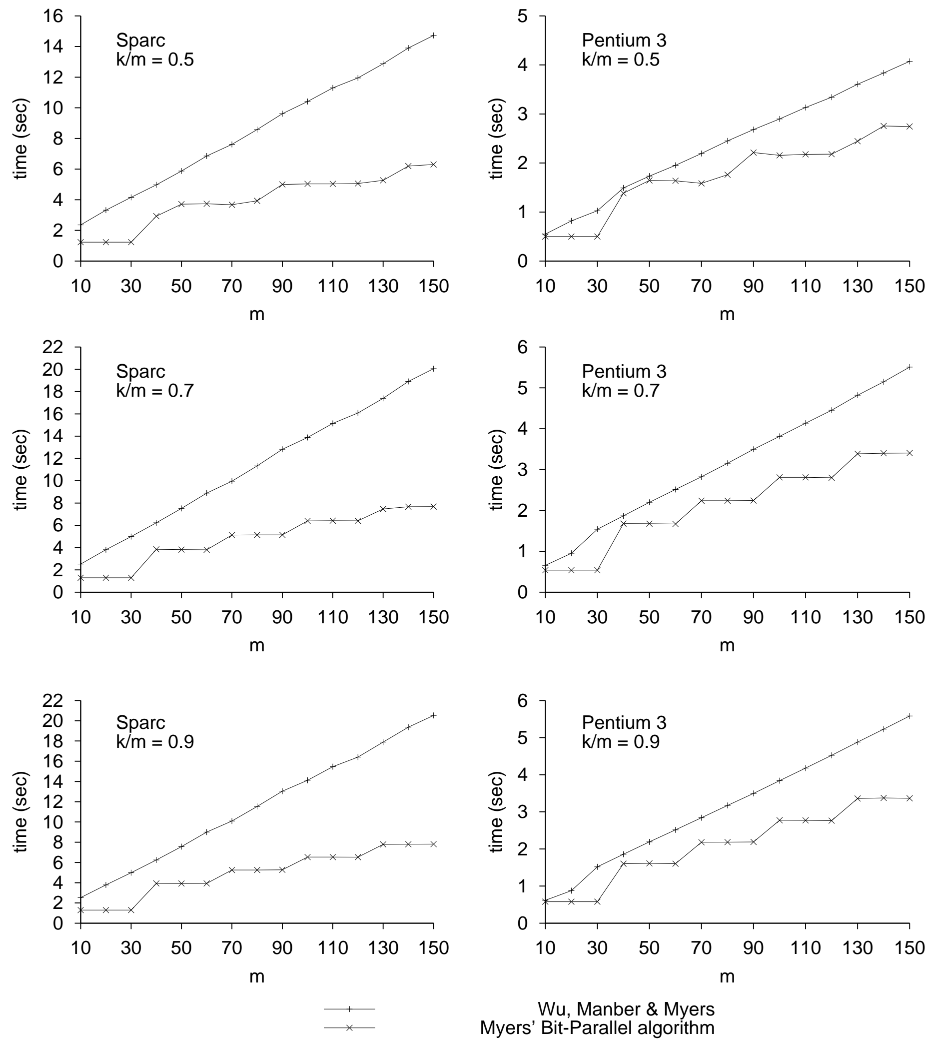


Figure 5: The plots show the average time for searching for a randomly picked pattern from a 10 MB sample of Wall Street Journal articles taken from the TREC-collection. The left column shows the results for Sparc Ultra 2 and the right column for Pentium 3.

Part II
Our Contributions

5 Our Variant of the Algorithm of Myers

In this section we derive a slightly modified version of the bit-vector algorithm of Myers. This version is easier to understand than the original ([49], page 158). It shares the same logic and thus essentially the same set of operations, but uses one variable less and is perhaps more convenient in terms of modifying the algorithm for other purposes (e.g. in [24]). This work has appeared as a part of the technical report [17].

To enable an efficient bit-parallel representation, the dynamic programming matrix D is stored by using delta encoding: Instead of the actual cell values, the differences between the values of adjacent cells are recorded. Because of the diagonal and adjacency properties, the following bit-vectors can be used in representing D:

-The vertical positive delta vector VP_j:

$$VP_j[i] = 1 \text{ iff } D[i,j] - D[i-1,j] = 1.$$

-The vertical negative delta vector VN_j:

$$VN_j[i] = 1 \text{ iff } D[i,j] - D[i-1,j] = -1.$$

-The horizontal positive delta vector HP_j:

$$HP_j[i] = 1 \text{ iff } D[i,j] - D[i,j-1] = 1.$$

-The horizontal negative delta vector HN_j:

$$HN_j[i] = 1 \text{ iff } D[i,j] - D[i,j-1] = -1.$$

-The diagonal zero delta vector $D0_j$:

$$D0_j[i] = 1 \text{ iff } D[i,j] = D[i-1,j-1].$$

Fig. 6 shows an example of these vectors.

The algorithm also uses the following pattern match vector PM_λ for each character λ.

-The match vector PM_λ:

$$PM_\lambda[i] = 1 \text{ iff } P_i = \lambda.$$

		o	n	c	e		u	p	o	n
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
o	**1**	0	1	1	1	1	1	1	0	1
n	**2**	1	0	1	2	2	2	2	1	0
e	**3**	2	1	1	1	2	3	3	2	1

VP_6	VN_6	HP_6	HN_6	$D0_6$
1	0	0	0	0
1	0	0	0	0
1	0	1	0	0

Figure 6: On the left is the dynamic programming matrix for searching for the pattern P = "one" from the text T = "once upon". On the right are the vectors VP_6, VN_6, HP_6, HN_6 and $D0_6$. Both row and column zero are shown in bold, and the sixth column, which the shown vectors correspond to, is shaded.

The algorithm imitates column-wise filling order of the dynamic programming matrix, and calculates explicitly only the values $D[m, j]$ for $j = 1..n$. All other cell values are represented implicitly by the earlier defined delta vectors. First VP_0 and VN_0 are initialized according to the boundary conditions for D. This means that $VP_0[i] = 1$ and $VN_0[i] = 0$ for $i = 1..m$. In addition $D[m, 0]$ is initialized to the value m. Then moving from column j-1 to column j involves the following four steps:

1. The diagonal vector $D0_j$ is computed by using PM_{T_j}, VP_{j-1} and VN_{j-1}.
2. The horizontal vectors HP_j and HN_j are computed by using $D0_j$, VP_{j-1} and VN_{j-1}.
3. The value $D[m, j]$ is calculated by using $D[m, j-1]$ and the horizontal delta values $HP_j[m]$ and $HN_j[m]$.
4. The vertical vectors VP_j and VN_j are computed by using $D0_j$, HP_j and HN_j.

An approximate occurrence of the pattern ends at text position j whenever $D[m, j] \leq k$ during the scan of the text.

Step 1: Computing $D0_j$

Assume that the values $VP_{j-1}[i]$, $VN_{j-1}[i]$ and $PM_{T_j}[i]$ are known. From Recurrence 1 for filling D we can see that there are the following three ways for $D0_j[i]$ to have a value 1.

1. $D[i, j-1] = D[i-1, j-1] - 1$, i.e. $VN_{j-1}[i] = 1$. This enables the zero-difference to propagate from the left by using the recurrence option $D[i, j] = D[i, j-1] + 1 = D[i-1, j-1]$.

2. $PM_{T_j}[i] = 1$. The zero-difference arises from the equality $P_i = T_j$, which sets $D[i, j] = D[i-1, j-1]$.

3. $D[i-1, j] = D[i-1, j-1] - 1$. This enables the zero-difference to propagate from above by using the recurrence option $D[i, j] = D[i-1, j] + 1 = D[i-1, j-1]$.

The first and second cases are easy to handle. All we need to do is to set $D0_j[i] = 1$ if $VN_{j-1}[i] = 1$ or/and $PM_{T_j}[i] = 1$. This means that the cases 1 and 2 can be treated for the whole vector $D0_j$ by OR-ing it with both VN_{j-1} and PM_{T_j}.

The third case, however, is the trickiest part of the algorithm. But Myers has presented a nice solution for it. Since the adjacency and the diagonal properties require that $D[i-1, j-1] - 1 \leq D[i-2, j-1] \leq D[i-1, j]$, it follows that $D[i-1, j] = D[i-1, j-1] - 1$ iff $D[i, j] = D[i-1, j-1]$ and $D[i-1, j] = D[i-2, j-1] = D[i, j]-1$. This translates into saying that $D[i-1, j] = D[i-1, j-1]-1$ iff $D0_j[i] = 1$, $D0_j[i-1] = 1$ and $VP_{j-1}[i-1] = 1$. On the other hand, the condition $VP_{j-1}[i-1] = 1$ implies that $VN_{j-1}[i-1] = 0$, and then $D0_j[i-1] = 1$ iff either the case 2 or the case 3 applies to the row $i-1$. This means that $D[i-1, j] = D[i-1, j-1]-1$ iff $VP_{j-1}[i-1] = 1$ and also $PM_{T_j}[i-1] = 1$ or $D[i-2, j] = D[i-2, j-1]-1$. By recursively applying the preceding reasoning for the second term, $D[i-2, j] = D[i-2, j-1]-1$, of the 'or', we have that $D[i-1, j] = D[i-1, j-1] - 1$ iff $VP_{j-1}[i-1] = 1$ and also $PM_{T_j}[i-1] = 1$ or $VP_{j-1}[i-2] = 1$ and also $PM_{T_j}[i-2] = 1$ or $D[i-3, j] = D[i-3, j-1] - 1$. When we continue in this manner, always expanding the last term of form $D[i-q, j] = D[i-q, j-1] - 1$, we arrive at some $h \leq i-1$ for which $PM_{T_j}[h] = 1$, and the recursion can stop. This must happen because the initial conditions on the dynamic programming matrix guarantee that $D[0, j] \neq D[0, j-1] - 1$. Thus we have the following rule for the case 3:

$D[i-1, j] = D[i-1, j-1] - 1$ iff $\exists h : PM_{T_j}[h] = 1$ and $VP_{j-1}[q] = 1$ for $q = h..i-1$.

The above rule states that $D[i-1, j] = D[i-1, j-1] - 1$ if and only if the $(i-1)$th bit of the vector VP_{j-1} belongs to such a run of consecutive 1-bits that there is also a match between the character T_j and some character P_h of

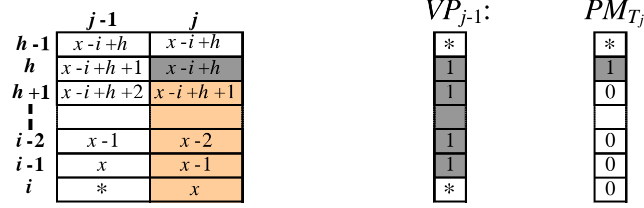


Figure 7: On the left are rows $h-1, h, .., i$ of columns $(j-1)$ and j of the matrix D. If $D[i-1, j] = x-1 = D[i-1, j-1]-1$, then there must be a match between T_j and some character P_h above the row i. The vertical delta is positive at least from row i up to row h. The corresponding segments of the vectors VP_{j-1} and PM_{T_j} are shown on the right. An asterisk indicates that the corresponding cell/bit may have different values depending on the situation.

the pattern that overlaps the run of consecutive bits at or above the $(i-1)$th bit (Fig. 7).

The run of consecutive 1-bits in VP_{j-1} propagates a diagonal zero-difference down in a way that resembles the carry-effect of integer addition. If $PM_{T_j}[h] = 1$ and $VP_{j-1}[q] = 1$ for $q = h..i-1$, then we know from the previous discussion that $D0_j[q] = 1$ for $q = h..i$. Now if we add the vectors $PM_{T_j}[h]$ and $VP_{j-1}[h..i-1]$ together, the carry effect causes the bits $h..i-1$ of VP_{j-1} to change from 1 to 0 and the ith bit to change either from 1 to 0 or from 0 to 1, depending on its original value. Suppose we XOR the bits $h..i$ of the result of the sum $PM_{T_j}[h] + VP_{j-1}[h..i-1]$ with the original bits $h..i$ of VP_{j-1}. Then the bits $h..i$ will all have the value 1, which is exactly the desired result. From PM_{T_j} we can extract only the bits i for which also $VP_{j-1}[i] = 1$ by AND-ing PM_{T_j} with VP_{j-1}. When we then add this vector $PM_{T_j} \& VP_{j-1}$ together with VP_{j-1} and XOR the resulting value $(PM_{T_j} \& VP_{j-1}) + VP_{j-1}$ with VP_{j-1}, we get almost the desired result for the whole vector. There are only two differences. One is the situation where there are several bits of PM_{T_j} that have the value 1 inside the same continuous run of ones in VP_{j-1}. This causes the XOR-operation to turn off some of these bits, because they will have a value 1 before and after the addition. The second is that the bit, which corresponds to the first match along a consecutive run of ones in VP_{j-1}, will also be set even though the horizontal delta value above it is not -1. But neither of these two is a problem in terms of the correctness of the vector $D0_j$ because the corresponding

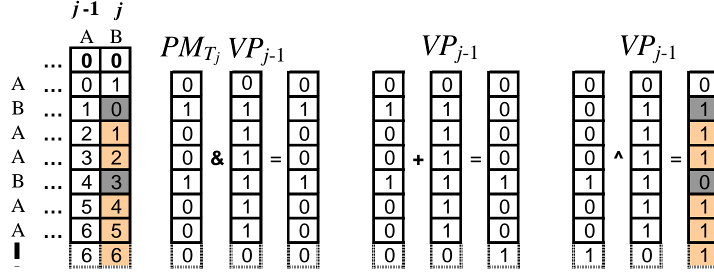


Figure 8: An example of handling the third case in computing $D0_j$ when $T_{j-1..j}$ = "AB" and $P_{1..7}$ = "ABAABAA". As can be seen from the filled column j, a match propagates diagonal zero deltas downwards as long as the vertical delta in the preceding column $j-1$ has a value +1. First the matching bits in PM_{T_j} that overlap a segment of ones in VP_{j-1} are extracted by AND-ing PM_{T_j} and VP_{j-1}. Then the resulting vector is added together with VP_{j-1} to change the bit value in the positions that get a diagonal zero delta from above. Finally these changed bits are set to 1 by XOR-ing the result of the addition with the original VP_{j-1}. The darker shading marks the locations where a match causes a diagonal zero delta, and the lighter shading the positions where a diagonal zero delta propagates from above.

bits will be set anyway when handling the case 2. Fig. 8 shows an example.

Putting together all the pieces for the cases 1, 2 and 3, we arrive at the following formula for computing $D0_j$:

$$D0_j = (((PM_{T_j} \ \& \ VP_{j-1}) + VP_{j-1}) \wedge VP_{j-1}) \mid PM_{T_j} \mid VN_{j-1}.$$

Step 2: Computing HP_j and HN_j

At this point we can assume that, in addition to the vectors VP_{j-1}, VN_{j-1} and PM_{T_j}, also the vector $D0_j$ is known.

It can be seen from the adjacency and diagonal properties that $HP_j[i] = 1$ iff $D[i,j-1] = D[i-1,j-1]-1$ (Fig. 9b), or $D[i,j] = D[i-1,j-1]+1$ and $D[i,j-1] = D[i-1,j-1]$ (Fig. 9a). In terms of the delta vectors this means that $HP_j[i] = 1$ iff $VN_{j-1}[i] = 1$, or $D0_j[i] = 0$ and $VP_{j-1}[i] = 0$ and $VN_{j-1} = 0$. Because the left side of the preceding 'or' has only the condition $VN_{j-1}[i] = 1$, the requirement $VN_{j-1}[i] = 0$ on the right side can be removed as it is implicitly expressed by the former. This results in the following formula for computing the vector $HP_j[i]$:

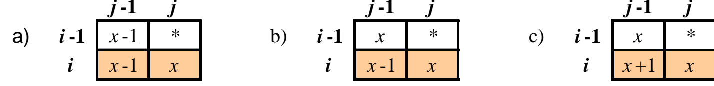


Figure 9: The figures a) and b) show the only possible combinations for the cells $D[i-1, j-1]$, $D[i, j-1]$ and $D[i, j]$ to have $D[i, j] = x = D[i, j-1]+1$. Similarly figure c) shows the only case where $D[i, j] = x = D[i, j] - 1$.

$$HP_j = VN_{j-1} \mid \sim (D0_j \mid VP_{j-1}).$$

In similar fashion as for HP_j, we can see that $HN_j[i] = 1$ iff $D[i, j] = D[i-1, j-1]$ and $D[i, j-1] = D[i-1, j-1] + 1$ (Fig. 9c). This results in the rule $VN_j[i] = 1$ iff $D0_j[i] = 1$ and $VP_{j-1}[i] = 1$, and so we have the following formula for computing the vector HN_j:

$$HN_j = D0_j \ \& \ VP_{j-1}.$$

Step 3: Computing the value $D[m, j]$

After computing the vectors HP_j and HN_j, the value $D[m, j]$ is easy to calculate from $D[m, j-1]$. If $HP_j[m] = 1$, then $D[m, j] = D[m, j-1] + 1$, and if $HN_j[m] = 1$, then $D[m, j] = D[m, j-1] - 1$. Otherwise $D[m, j] = D[m, j-1]$.

Step 4: Computing VP_j and VN_j

This step is diagonally symmetric with step 2 (computing HP_j and HN_j).

By imitating the case of HP_j, we have that $VP_j[i] = 1$ iff $HN_j[i-1] = 1$ or $D0_j[i] = 0$ and $HP_j[i-1] = 0$. Now we need to align the row $(i-1)$ bits $HN_j[i-1]$ and $HP_j[i-1]$ with the row i bit $VP_j[i]$. This means shifting the former two one step down (that is, to the left). After shifting these two vectors left, their first bits represent the values $HP_j[0]$ and $HNj[0]$, which are not explicitly present in the algorithm. These two values correspond to the difference $D[0, j]-D[0, j-1]$. Since we assume zero filling, shifting HN_j and HP_j one step to the left introduces a zero in their first positions. This is the same as using the values $HP_j[0] = 0$ and $HN_j[0] = 0$, which corresponds correctly to the boundary condition $D[0, j] = 0$ (i.e. $D[0, j] - D[0, j-1] = 0$) of approximate string matching. If we were to use this algorithm for computing edit distance, the newly introduced 0-bit of the vector HP_j would have to be changed into a 1-bit so that $HP_j[0] = 1$, which corresponds to

the initial condition $D[0,j] = j$ (i.e. $D[0,j] - D[0,j-1] = 1$) of computing edit distance. The resulting formula for computing the vector VP_j is then:

$$VP_j = (HN_j << 1) \mid \sim (D0_j \mid (HP_j << 1)).$$

By imitating this time the case of HN_j, we have that $VN_j[i] = 1$ iff $D0_j[i] = 1$ and $HP_j[i-1] = 1$. Again the row $(i-1)$ bit $HP_j[i-1]$ has to be shifted one step down to align it with row i. The same comment as above, about setting the newly introduced bit of HP_j into a 1-bit in the case of computing edit distance, applies also here. We get the following formula for computing the vector VN_j:

$$VN_j = D0_j \ \& \ (HP_j << 1).$$

The complete algorithm corresponding to steps 1 - 4 is given in Fig. 10. We follow the example of [49] and call it "BPM", which stands for **B**it **P**arallel **M**atrix. Apart from the different way of deriving the algorithm, the only difference between this version and the original algorithm of Myers is that he uses two vectors XV_j and XH_j instead of a single diagonal vector $D0_j$. In fact $D0_j = XV_j$ OR XH_j, and XV_j corresponds to the cases 1 and 2 and XH_j to the cases 2 and 3 of the computation of $D0_j$.

A practical version would avoid shifting the vector HP_j twice, make some of the vectors share the same variable and keep only the currently needed values of the difference vectors in memory in a similar fashion to what was discussed about saving space in the case of the dynamic programming matrix (Section 2).

```
ComputePM(P)
 1.   For i ← 0 to σ − 1 Do
 2.        PM_i ← 0^m
 3.   For i ← 1 to m Do
 4.        PM_{P_i} ← 0^{m−i}10^{i−1}

BPMSearch(P, T, k)
 5.   ComputePM(P)
 6.   VN_0 ← 0^m, VP_0 ← 1^m, currDist ← k
 7.   For j ← 1 to n Do
 8.        D0_j ← (((PM_{T_j} & VP_{j−1}) + VP_{j−1}) ^ VP_{j−1}) | PM_{T_j} | VN_{j−1}
 9.        HP_j ← VN_{j−1} | ~(D0_j | VP_{j−1})
10.        HN_j ← D0_j & VP_{j−1}
11.        If HP_j & 10^{m−1} = 10^{m−1} Then
12.             currDist ← currDist + 1
13.        Else If HN_j & 10^{m−1} = 10^{m−1} Then
14.             currDist ← currDist − 1
15.        If currDist ≤ k Then
16.             Report occurrence at T_j
17.        VP_j ← (HN_j << 1) | ~(D0_j | (HP_j << 1))
18.        VN_j ← D0_j & (HP_j << 1)
```

Figure 10: Our $D0_j$-based version of the bit-parallel matrix algorithm (BPM). We assume that the alphabet Σ is represented by the integers from 0 to $\sigma - 1$.

6 Adding Transposition into the Bit-parallel Methods

In this section we discuss how the three currently best bit-parallel approximate string matching algorithms (see Section 4) can be modified to use the Damerau edit distance. This work has appeared in [21], and we have previously shown a modification for BPM alone in [17].

6.1 A New Recurrence for the Damerau Edit Distance

We begin by reformulating Recurrence 2 into a form that is easier to use with bit-parallel algorithms. Our trick is to investigate how a transposition relates to a substitution. Consider comparing the strings $P =$ "abc" and $T =$ "acb". Then $D[2,2] = ed_D(P_{1..2}, T_{1..2}) = ed_D($"ab","ac"$) = 1$, where the one operation corresponds to substituting the first character of the transposable suffixes "bc" and "cb". When filling in the value $D[3,3] = ed_D($"abc","acb"$)$, the effect of having done a single transposition can be achieved by allowing a free substitution between the latter characters of the transposable suffixes. This is the same as declaring a match between them. In this way the cost for doing the transposition has already been paid for by the substitution of the preceding step. It turns out that this idea can be developed to work correctly in all cases. We find that the following Recurrence 3 for the Damerau edit distance is in effect equivalent with Recurrence 2. It uses an auxiliary $|P| \times (|T| + 1)$ Boolean table MT as it is convenient for bit-parallel algorithms. The value $MT[i,j]$ records whether there is the possibility to match or to make a free substitution when computing the value $D[i,j]$.

Recurrence 3

$$D[i,0] = i, D[0,j] = j, MT[i,0] = \textbf{false}.$$

$$MT[i,j] = \begin{cases} \textbf{true}, \text{ if } P_i = T_j \text{ or } & (MT[i-1,j-1] = \textbf{false} \text{ and} \\ & P_{i-1..i} = (T_{j-1..j})^R). \\ \textbf{false}, \text{ otherwise.} & \end{cases}$$

$$D[i,j] = \begin{cases} D[i-1,j-1], \text{ if } MT[i,j] = \textbf{true}. \\ 1 + \min(D[i-1,j-1], D[i-1,j], D[i,j-1]), \text{ otherwise.} \end{cases}$$

We prove by induction that Recurrence 2 and Recurrence 3 give the same values for $D[i,j]$ when $i \geq 0$ and $j \geq 0$.

Clearly both formulas give the same value for $D[i,j]$ when $i = 0$ or 1 or

$j = 0$ or 1. Consider now a cell $D[i, j]$ for some $j > 1$ and $i > 1$ and assume that all previous cells with nonnegative indices have been filled identically by both recurrences[1]. Let x be the value given to $D[i, j]$ by Recurrence 2 and y be the value given to it by Recurrence 3. The only situation in which the two formulas could possibly behave differently is when $P_i \neq T_j$ and $P_{i-1..i} = (T_{j-1..j})^R$. In the following two cases we assume that these two conditions hold.

If $D[i-1, j-1] = D[i-2, j-2] + 1$, then $MT[i-1, j-1] =$ **false** and $MT[i, j] =$ **true**, and thus $y = D[i-1, j-1]$. Since the diagonal property requires that $x \geq D[i-1, j-1]$ and now $x \leq D[i-2, j-2] + 1$, we have $x = D[i-2, j-2] + 1 = D[i-1, j-1] = y$.

Now consider the case $D[i-2, j-2] = D[i-1, j-1]$. Because $P_{i-1} = T_j \neq P_i = T_{j-1}$, this equality cannot result from a match. If it resulted from a free substitution, then $MT[i-1, j-1] =$ **true** in Recurrence 3. As $P_i \neq T_j$, the preceding means that $MT[i, j] =$ **false**. Therefore $y = 1 + \min(D[i-1, j-1], D[i-1, j], D[i, j-1])$ and $x = 1 + \min(D[i-2, j-2], D[i-1, j], D[i, j-1])$. Because $D[i-2, j-2] = D[i-1, j-1]$, the former means that $x = 1 + \min(D[i-1, j-1], D[i-1, j], D[i, j-1]) = y$. The last possibility is that the equality $D[i-2, j-2] = D[i-1, j-1]$ resulted from using the option $D[i-1, j-1] = 1 + \min(D[i-2, j-1], D[i-1, j-2])$. As $P_{i-1} = T_j$ and $P_i = T_{j-1}$, both recurrences must have set $D[i-1, j] = D[i-2, j-1]$ and $D[i, j-1] = D[i-1, j-2]$, and therefore $D[i-1, j-1] = 1 + \min(D[i-2, j-1], D[i-1, j-2]) = 1 + \min(D[i-1, j], D[i, j-1])$. Now both options in Recurrence 3 set the same value $y = D[i-1, j-1]$, and $x = 1 + \min(D[i-2, j-2], D[i-1, j], D[i, j-1]) = 1 + \min(D[i-1, j], D[i, j-1]) = D[i-1, j-1] = y$.

In each case Recurrence 2 and Recurrence 3 assigned the same value for the cell $D[i, j]$. Therefore we can state by induction that the recurrences are in effect equivalent. □

The intuition behind the table MT in Recurrence 3 is that a free substitution is allowed at $D[i, j]$ if a transposition is possible at that location. But we cannot allow more than one free substitution in a row along a diagonal, as each corresponding transposition has to be paid for by a regular substitution. Therefore when a transposition has been possible at $D[i, j]$, another will not be allowed at $D[i+1, j+1]$. As shown above, this restriction on when to permit a transposition does not affect the correctness of the scheme.

[1] We assume that a legal filling order has been used, which means that the cells $D[i-1, j-1]$, $D[i-1, j]$ and $D[i, j-1]$ are always filled before the cell $D[i, j]$.

6.2 Modifying the Bit-parallel Algorithms

The Bit-parallel NFA of Wu & Manber

The bit-parallel approximate string matching algorithm of Wu & Manber [60] is based on representing a non-deterministic finite automaton (NFA) by using bit-vectors. The automaton has $(k+1)$ rows, numbered from 0 to k, and each row contains m states. Let us denote the automaton as R, its row d as R_d and the state i on its row d as $R_{d,i}$. The state $R_{d,i}$ is active after reading the text up to the jth character if and only if $ed(P_{1..i}, T_{h..j}) \leq d$ for some $h \leq j$. An occurrence of the pattern with at most k errors is found when the state $R_{k,m}$ is active. Assume for now that $w \leq m$. Wu & Manber represent each row R_d as a length-m bit-vector, where the ith bit tells whether the state $R_{d,i}$ is active or not. In addition they build a length-m match vector for each character in the alphabet. We denote the match vector for the character λ as PM_λ. The ith bit of PM_λ is set if and only if $P_i = \lambda$. Initially each vector R_d has the value $0^{m-d}1^d$ (this corresponds to the boundary conditions in Recurrence 1). The formula to compute the updated values R'_d for the row-vectors R_d at text position j is shown in Fig. 11.

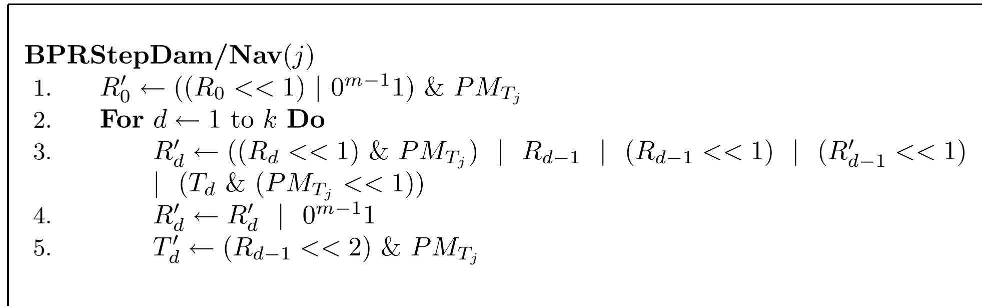
BPRStep(j)
1. $R'_0 \leftarrow ((R_0 << 1) \mid 0^{m-1}1) \ \& \ PM_{T_j}$
2. **For** $d \leftarrow 1$ to k **Do**
3. $\quad R'_d \leftarrow ((R_d << 1) \ \& \ PM_{T_j}) \mid R_{d-1} \mid (R_{d-1} << 1) \mid (R'_{d-1} << 1)$
4. $\quad R'_d \leftarrow R'_d \mid 0^{m-1}1$

Figure 11: The update formula of the bit-parallel algorithm of Wu & Manber at text position j.

The right side of the third row computes the disjunction of the different possibilities given by Recurrence 1 for a prefix of the pattern to match with d errors. The fourth row sets the first bit of R'_d for $d > 0$ to propagate the effect of the boundary condition $D[0, j] = 0$. At least [42, 41, 24, 49, 21] show an erroneous formula that does not take care of this. R_0 is different as it needs to consider only matching positions between P and the character T_j, and it also has to have its first bit set after the left-shift in order to let the first character match at the current position. When $m \leq w$, the run time of this algorithm is $O(kn)$ as there are $O(k)$ operations per text character. The

general run time is $O(kn\lceil m/w \rceil)$ as a vector of length m may be simulated in $O(\lceil m/w \rceil)$ time using $O(\lceil m/w \rceil)$ bit-vectors of length w. In this thesis we do not discuss the details of such a multi-word implementation for any of the bit-parallel algorithms.

Navarro [42] and Holub [16] have independently modified this algorithm to use the Damerau distance by essentially following Recurrence 2. Navarro did this by appending the automaton to have a temporary state vector T_d for each R_d to keep track of the positions where a transposition may occur. Initially each T_d has the value 0^m. Navarro's formula is shown in Fig. 12.



BPRStepDam/Nav(j)
1. $R'_0 \leftarrow ((R_0 << 1) \mid 0^{m-1}1)$ & PM_{T_j}
2. **For** $d \leftarrow 1$ to k **Do**
3. $R'_d \leftarrow ((R_d << 1)$ & $PM_{T_j}) \mid R_{d-1} \mid (R_{d-1} << 1) \mid (R'_{d-1} << 1)$ $\mid (T_d$ & $(PM_{T_j} << 1))$
4. $R'_d \leftarrow R'_d \mid 0^{m-1}1$
5. $T'_d \leftarrow (R_{d-1} << 2)$ & PM_{T_j}

Figure 12: Navarro's modification of the bit-parallel algorithm of Wu & Manber to use the Damerau edit distance.

Navarro's formula adds $6k$ operations into the basic version for the Levenshtein edit distance. The version of Holub is slightly different, but also it uses $6k$ additional operations.

Recurrence 3 suggests a simpler way to facilitate transposition. The only difference between it and Recurrence 1 is in the condition on when $D[i,j] = D[i-1,j-1]$: Instead of the condition $P_i = T_j$, Recurrence 3 sets the equal value if $MT[i,j] = \textbf{true}$. We use a length-m bit-vector TC in storing the last column of the auxiliary table MT. The ith bit of TC is set iff row i of the last column of MT has the value **true**. When we arrive at text position j, TC is updated to hold the values of column j. Initially $TC = 0^m$. Based on Recurrence 3, the vector TC may be updated with the formula $TC' = PM_{T_j} \mid (((\sim TC) << 1)$ & $(PM_{T_j} << 1)$ & $PM_{T_{j-1}})$. Here the right "AND" sets the bits in the pattern positions where $P_{i-1..i} = (T_{j-1..j})^R$, the left "AND" sets off the ith bit if row $(i-1)$ of MT had the value **true** in the previous column, and the "OR" sets the bits in the positions where $P_i = T_j$. By combining the two left-shifts we get the formula shown in Fig. 13 for updating the R_d vectors:

Our formula adds a total of 6 operations into the basic version for the

```
BPRStepDam/Ours(j)
1.    TC' ← PM_{T_j} | ((((~ TC) & PM_{T_j}) << 1) & PM_{T_{j-1}})
2.    R'_0 ← ((R_0 << 1) | 0^{m-1}1) & TC'
3.    For d ← 1 to k Do
4.        R'_d ← ((R_d << 1) & TC') | R_{d-1} | (R_{d-1} << 1) | (R'_{d-1} << 1)
5.        R'_d ← R'_d | 0^{m-1}1
```

Figure 13: Our modification of the bit-parallel algorithm of Wu & Manber to use the Damerau edit distance.

Levenshtein edit distance. Therefore it makes the same number of operations as Navarro's or Holub's version when $k = 1$, and wins when $k > 1$.

The Bit-parallel NFA of Baeza-Yates & Navarro

Also the bit-parallel algorithm of Baeza-Yates & Navarro [2] is based on simulating the NFA R. The first d states on row R_d are trivial in that they are always active. The last $k-d$ states will be active only if the state $R_{k,m}$ is active, and as we are only interested in knowing whether there is a match with at most k errors, having the state $R_{k,m}$ is enough. These facts enable Baeza-Yates & Navarro to include only the $m-k$ states $R_{d,d+1}..R_{d,m-k+d}$ on row R_d. A further difference is in the way the states are encoded into bit-vectors. They divide R into $m-k$ diagonals $D_1, .., D_{m-k}$, where D_i is a bit-sequence that describes the states $R_{d,d+i}$ for $d = 0..k$. If a state $R_{d,i}$ is active, then all states on the same diagonal that come after $R_{d,i}$ are active, that is, the states $R_{d+h,i+h}$ for $h \geq 1$. To describe the status of the ith diagonal it suffices to record the position of the first active state in it. If the first active state on the ith diagonal is f_i, then Baeza-Yates & Navarro represent the diagonal as the bit-sequence $D_i = 0^{k+1-f_i}1^{f_i}$. The value $f_i = k+1$ means that $f_i \geq k+1$, that is, that no states on the ith diagonal of R is active. A match with at most k errors is found whenever $f_{m-k} < k+1$. The D_i bit-sequences are stored consecutively with a single separator zero-bit between two consecutive states. Let RD denote the complete diagonal representation. Then RD is the length-$(k+2)(m-k)$ bit-sequence $0\ D_1\ 0\ D_2\ 0...0\ D_{m-k}$. We assume for now that $(k+2)(m-k) \leq w$ so that RD fits into a single bit-vector.

Baeza-Yates & Navarro encode also the pattern match vectors differently. Let PMD_λ be their pattern match vector for the character λ. The role of

the bits is reversed: a 0-bit denotes a match and a 1-bit a mismatch. To align the matches with the diagonals in RD, PMD_λ has the form
$0 \sim (PM_\lambda[1..k+1])\ 0 \sim (PM_\lambda[2..k+2])\ 0...0 \sim (PM_\lambda[m-k..m])$.

Initially no diagonal has active states and so $RD = (0\ 1^{k+1})^{m-k}$. The formula for updating RD at text position j is shown in Fig. 14. Again, we follow the example of [49] and call the algorithm "BPD", which stands for **B**it **P**arallel by **D**iagonals.

BPDStep(j)
1. $x \leftarrow (RD >> (k+2))\ |\ PMD_{T_j}$
2. $RD' \leftarrow ((RD << 1)\ \ |\ \ (0^{k+1}1)^{m-k}\ \&\ (((x + (0^{k+1}1)^{m-k})\ \wedge\ x) >> 1)$
3. $\quad\quad\&\ ((RD << (k+3))\ \ |\ \ (0^{k+1}1)^{m-k-1}01^{k+1})\ \&\ (0\ 1^{k+1})^{m-k}$

Figure 14: The update formula of the bit-parallel algorithm of Baeza-Yates & Navarro at text position j.

If $(k+2)(m-k) \leq w$, the run time of this algorithm is $O(n)$ as there is only a constant number of operations per text character. The general run time is $O(\lceil km/w \rceil n)$ as a vector of length $(k+2)(m-k)$ may be simulated in $O(\lceil km/w \rceil)$ time using $O(\lceil km/w \rceil)$ bit-vectors of length w.

Because of the different way of representing R, our way of modifying the algorithm of Wu & Manber to use the Damerau edit distance does not work here without some changes. Now we use a bit-vector TCD instead of the vector TC of the previous section. TCD has the same function as TC, but its form corresponds to the algorithm of Baeza-Yates & Navarro. First of all the meaning of the bit-values is reversed: now a 0-bit corresponds to the value **true** and a 1-bit to the value **false** in the table TR of Recurrence 3. The second change is in the way we compute the positions where $P_{i-1..i} = (T_{j-1..j})^R$. Because of the interleaving 0-bits in the pattern match vector PMD_λ, the formula $(PMD_{T_j} << 1)\ \ |\ \ PMD_{T_{j-1}}$ does not correctly set only those bits to zero that correspond to a transposable position (note that also the roles of "AND" and "OR" are reversed). But by inspecting the form of BPD_λ we notice that the desired effect is achieved by using the formula $(PMD_{T_j} >> (k+2))\ \ |\ \ PMD_{T_{j-1}}$. Shifting $(k+2)$ bits to the right causes diagonal $(i-1)$ to align with the ith diagonal, and this previous diagonal handles the matches one step to the left in the pattern. The only delicacy in doing this is the fact that now the first diagonal will have no match-data. Because we need to have made a substitution before making a

free substitution that corresponds to a transposition, a transposition will be possible only in diagonals $2..m-k$. Thus the missing data can be replaced with mismatches. Note that we do not need to consider the states not present in the reduced automaton of Baeza-Yates & Navarro. By similar reasoning also the previous values of TCD will be shifted $(k+2)$ bits to the right instead of 1 bit to the left, and its missing data can be replaced by '**false**' values. Initially TCD has only '**false**' values and so $TCD = (0\ 1^{k+1})^{m-k}$. The modified formula for updating RD at text position j is shown in Fig. 15.

BPDStepDam/Ours(j)

1. $TCD' \leftarrow PMD_{T_j}\ \&\ (((((\sim TCD)\ \mid\ PMD_{T_j}) >> (k+2))\ \mid\ PMD_{T_{j-1}})$
2. $\qquad\qquad \mid\ 01^{k+1}0^{(m-k-1)(k+2)})$
3. $x \leftarrow (RD >> (k+2))\ \mid TCD'$
4. $RD' \leftarrow ((RD << 1)\ \mid\ (0^{k+1}1)^{m-k}\ \&\ (((x + (0^{k+1}1)^{m-k})\ \wedge\ x) >> 1)$
5. $\qquad\qquad \&\ ((RD << (k+3))\ \mid\ (0^{k+1}1)^{m-k-1}01^{k+1})\ \&\ (0\ 1^{k+1})^{m-k}$

Figure 15: Our modification of the bit-parallel algorithm of Baeza-Yates & Navarro to use the Damerau edit distance.

Now the number of added operations is 7, as one "extra" operation arises from having to set the missing values (second row).

Myers' Bit-parallel Computation of D

Because the algorithm of Myers uses the same pattern match vectors as the algorithm of Wu & Manber, both of these algorithms can be modified to use the Damerau distance by using exactly the same method. Thus the formula to compute the updated vectors $D0', HP', HN', VP'$ and VN' at text position j is simply as shown in Fig. 16.

There is again 6 added operations.

6.3 Test Results

We implemented and tested a Damerau edit distance -version of each of the three discussed bit-parallel algorithms. The version of the algorithm of Wu & Manber was implemented from scratch by us, and the other two were modified using the original implementations from those authors. We compared also the versions for the Levenshtein edit distance to see how our modification affects the respective performance of the algorithms. The

BPMStepDam/Ours(j)

1. $TC' \leftarrow PM_{T_j} \mid ((((\sim TC) \ \& \ PM_{T_j}) << 1) \ \& \ PM_{T_{j-1}})$
2. $D0' \leftarrow (((TC' \ \& \ VP) + VP) \wedge VP) \mid TC' \mid VN$
3. $HP' \leftarrow VN \mid \sim (D0' \mid VP)$
4. $HN' \leftarrow VP \ \& \ D0'$
5. $VP' \leftarrow (HN' << 1) \mid \sim (D0' \mid (HP' << 1))$
6. $VN' \leftarrow (HP' << 1) \ \& \ D0'$

Figure 16: Our modification of the bit-parallel algorithm of Myers to use the Damerau edit distance.

computer used in the tests was a 600 MHz Pentium 3 with 256 MB RAM and Linux OS. All code was compiled with GCC 3.2.1 and full optimization switched on.

The tests involved patterns of lengths 10, 20, and 30, and with each pattern length m the tested k values were $1..\lfloor m/2 \rfloor$. There were 50 randomly picked patterns for each (m, k)-combination. The searched text was a 10 MB sample from Wall Street Journal articles taken from the TREC-collection [14].

The version of the algorithm of Baeza-Yates & Navarro was the one from [46], which includes a smart mechanism to keep only a required part of the automaton active when it needs several bit-vectors. As the pattern lengths were $\leq w = 32$, the other two algorithms did not need such a mechanism.

Fig. 17 shows the results. In general the algorithms compare quite similarly to each other with and without our modification to use the Damerau edit distance. It is seen that with the Levenshtein edit distance the algorithm of Wu & Manber becomes slowest when $k \geq 4$, whereas with the Damerau edit distance it becomes slowest already at $k = 3$. The algorithm of Baeza-Yates & Navarro is typically the fastest for low error levels irrespective of which of the two distances we use. But its advantage over the algorithm of Myers becomes smaller under the Damerau edit distance. The algorithm of Myers is affected very little by the modification, and it is the fastest algorithm when the error level k/m is large and the algorithm of Baeza-Yates & Navarro needs more bit-vectors in representing the automaton.

The algorithm of Baeza-Yates & Navarro behaved oddly with the Levenshtein edit distance in the case $m = 10$ and $k < 3$. We found no other reason than some intrinsic property of the compiler optimizer or the processor pipeline for the bad performance with these two values (even worse than

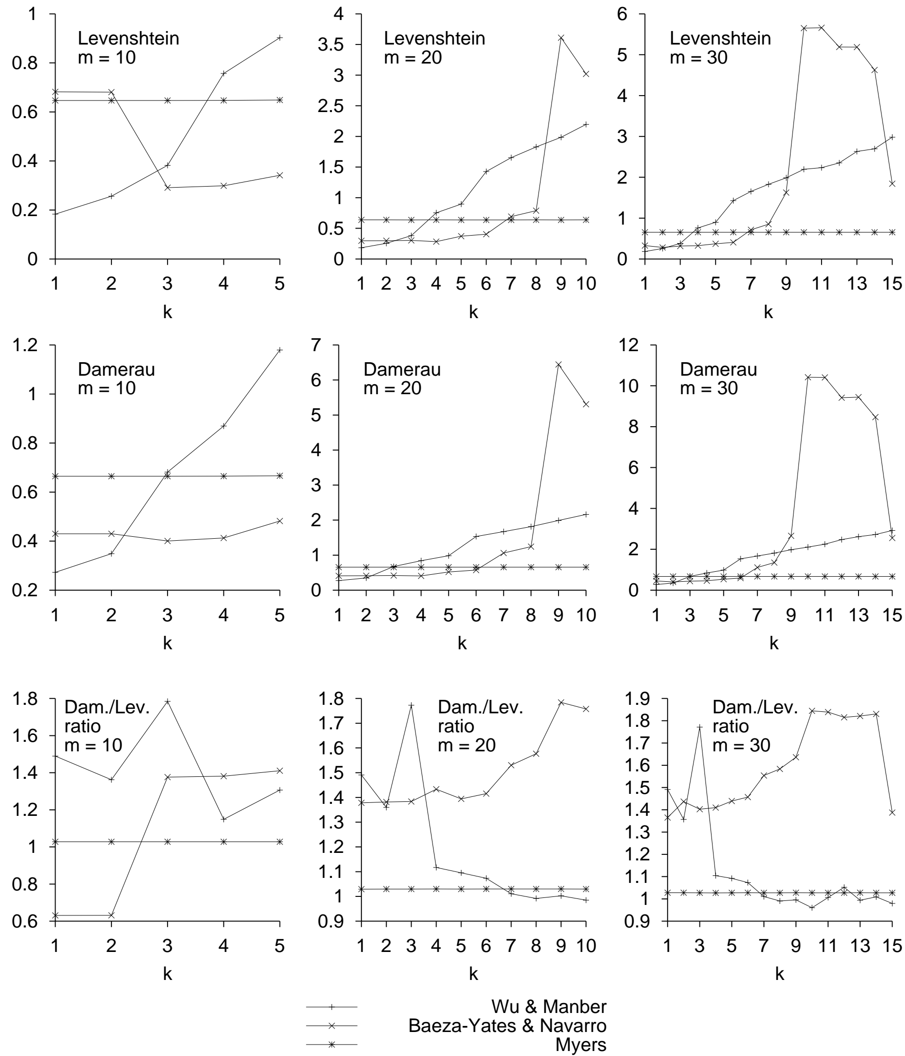


Figure 17: The two first rows show the average time for searching for a pattern from a 10 MB sample of Wall Street Journal articles taken from TREC-collection. The first row shows the results for the Levenshtein edit distance and the second row for the Damerau edit distance. The third row shows the ratio of the run times with and without the modification.

the version modified to use the Damerau edit distance).

7 Using Diagonal Tiling in Computing Edit Distance

In this section we present a bit-parallel version of the diagonal restriction scheme of Ukkonen, which was briefly discussed in Section 3. This work has appeared in [20, 22].

In the following we concentrate on the case where the computer word size w is large enough to cover the required diagonal region. Let l_v denote the length of the delta vectors. Then our assumption means that $w \geq l_v = \min(m, \lfloor (t-n+m)/2 \rfloor + \lfloor (t+n-m)/2 \rfloor + 1)$. Note that in this case each of the pattern match vectors PM_λ may have to be encoded with more than one bit vector: If $m > w$, then PM_λ consists of $\lceil m/w \rceil$ bit vectors.

The basic idea is to mimic the diagonal restriction method of Ukkonen by tiling the vertical delta vectors of BPM diagonally instead of horizontally (Fig. 18a). We achieve this by modifying slightly the way the vertical delta vectors VP_j and VN_j are used: Before processing column $j+1$ the vertical vectors VP_j and VN_j are shifted one step up (to the right in terms of the bit vector) (Fig. 18b). When the vertical vectors are shifted up, their new lowest bit-values $VP_j[l_v]$ and $VN_j[l_v]$ are not explicitly known. This turns out not to be a problem. From the diagonal and adjacency properties we can see that the only situation which could be troublesome is if we would incorrectly have a value $VN_j[l_v] = 1$. This is impossible, because it can happen only if $D0_j$ has an "extra" set bit at position l_v+1 and $HP_j[l_v] = 1$, and these two conditions cannot simultaneously be true.

In addition to the obvious way of first computing VP_j and VN_j in normal fashion and then shifting them up (to the right) when processing column $(j+1)$, we propose also a second option. It can be seen that essentially the same shifting effect can be achieved already when the vectors VP_j and VN_j are computed by making the following changes on lines 17-18 of the BPM algorithm in Fig. 10 (Section 5, page 26).

-The diagonal zero delta vector $D0_j$ is shifted one step to the right.
-The left shifts of the horizontal delta vectors are removed.

This second alternative uses less bit operations and does not require the extra modification of setting the first bit of HP_j after shifting it left in order to incorporate the boundary condition $D[0,j] = j$ (this modification is shown in Fig. 25 on page 46). But the choice between the two may depend on other practical issues. For example if several bit vectors have to be used in encoding $D0_j$, the column-wise top-to-bottom order may make it more difficult to shift $D0_j$ up than shifting both VP_j and VN_j down.

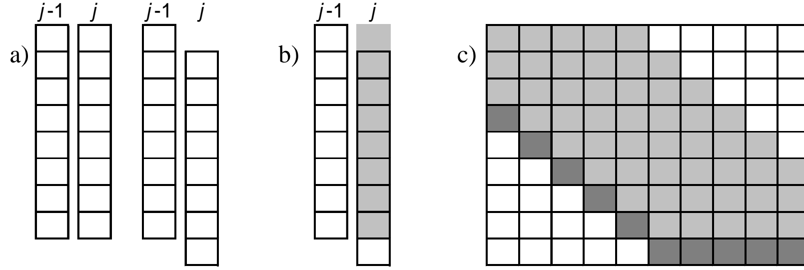


Figure 18: a) Horizontal tiling (left) and diagonal tiling (right). b) The figure shows how the diagonal step aligns the column $(j-1)$ vector one step above the column j vector. c) The figure depicts in gray the region of diagonals, which are filled according to Ukkonen's rule. The cells on the lower boundary are in darker tone.

We also modify the way some cell values are explicitly maintained. We choose to calculate the values along the lower boundary of the filled area of the dynamic programming matrix (Fig. 18c). For two diagonally consecutive cells $D[i-1, j-1]$ and $D[i, j]$ along the diagonal part of the boundary this means setting $D[i, j] = D[i-1, j-1]$ if $D0_j[l_v] = 1$, and $D[i, j] = D[i-1, j-1]+1$ otherwise. The horizontal part of the boundary is handled in similar fashion as in BPM: For horizontally consecutive cells $D[i, j-1]$ and $D[i, j]$ along the horizontal part of the boundary we set $D[i, j] = D[i, j-1]+1$ if $HP_j[l_v] = 1, D[i, j] = D[i, j-1]-1$ if $HN_j[l_v] = 1$, and $D[i, j] = D[i, j-1]$ otherwise. Here we assume that the vector length l_v is appropriately decremented as the diagonally shifted vectors would start to protrude below the lower boundary.

Another necessary modification is in the way the pattern match vector PM_{T_j} is used. Since we are gradually moving the delta vectors down, the match vector has to be aligned correctly. This is easily achieved in $O(1)$ time by shifting and OR-ing the corresponding at most two match vectors. Note that this requires us to keep track of the amount of misalignment between the current tile position and the match vectors.

If we are using the Damerau edit distance version of BPM (Section 6.2), we have to also modify the way the vector TC is computed in the first line of Fig. 16 (page 34). First of all the old TC vector is shifted down (left), which is not necessary when the vectors are tiled diagonally. Because of similar reason the vector PM_{j-1} has to be shifted one step up (to the right). This means that also the value $PM_{j-1}[l_v+1]$ will have to be present in the match vector PM_{j-1}. We do not deal with this separately, but assume from now

on that $l_v + 1 \leq w$ when dealing with the Damerau edit distance. Another option would be to set the last bit separately, which can be done in $O(1)$ time.

Figures 19 and 20 show the algorithms for computing the delta vectors at column j when diagonal tiling is used. The former figure is for the Levenshtein and the latter for the Damerau edit distance. We do not show the details of building the match vectors or updating the cell value at the lower boundary. These can be done as discussed above.

When $l_v \leq w$, each column of the dynamic programming matrix is computed in $O(1)$ time, which results in the total time being $O(\sigma + n)$ including also time for preprocessing the pattern match vectors. In the general case, in which $l_v > w$, each length-l_v vector can be simulated by using $\lceil l_v/w \rceil$ length-w vectors. This can be done in $O(\lceil l_v/w \rceil)$ time per operation, and therefore the algorithm has in general a run time $O(\sigma + \lceil l_v/w \rceil n)$, which is $O(\sigma + \lceil ed(P,T)/w \rceil \times n)$ as $l_v = O(ed(P,T))$. The slightly more favourable time complexity of $O(\sigma + \lceil ed(P,T)/w \rceil \times m)$ in the general case can be achieved by simply reversing the roles of the strings P and T: We still have that $l_v = O(ed(P,T))$, but now there are m columns instead of n. In this case the cost of preprocessing the match vectors is $O(\sigma + n)$, but the above complexities hold since $n = O(ed(P,T) + m)$ when $n > m$.

Computing column j in diagonal tiling (Levenshtein distance)
1. **Build the correct match vector into** PM_{T_j}
2. $D0_j \leftarrow (((PM_{T_j} \;\&\; VP_{j-1}) + VP_{j-1}) \wedge VP_{j-1}) \mid PM_{T_j} \mid VN_{j-1}$
3. $HP_j \leftarrow VN_{j-1} \mid \sim (D0_j \mid VP_{j-1})$
4. $HN_j \leftarrow D0_j \;\&\; VP_{j-1}$
5. **Update the appropriate cell value at the lower boundary.**
6. $VP_j \leftarrow HN_j \mid \sim ((D0_j >> 1) \mid HP_j)$
7. $VN_j \leftarrow (D0_j >> 1) \;\&\; HP_j$

Figure 19: Computation of column j with the Levenshtein edit distance and diagonal tiling (for the case $l_v \leq w$).

7.1 Test Results

In this section we present initial test results for our algorithm in the case of computing the Levenshtein edit distance. We concentrate only on the case where one wants to check whether the edit distance between two strings A

Computing column j in diagonal tiling (Damerau distance)

1. **Build the correct match vector into PM_{T_j}**
2. $TC_j \leftarrow PM_{T_j} \mid ((\sim TC_{j-1})\ \&\ (PM_{T_j} << 1)\ \&\ (PM_{j-1} >> 1))$
3. $D0_j \leftarrow (((TC_j\ \&\ VP_{j-1}) + VP_{j-1})\ \wedge\ VP_{j-1}) \mid TC_j \mid VN_{j-1}$
4. $HP_j \leftarrow VN_{j-1} \mid \sim (D0_j \mid VP_{j-1})$
5. $HN_j \leftarrow D0_j\ \&\ VP_{j-1}$
6. **Update the appropriate cell value at the lower boundary.**
7. $VP_j \leftarrow HN_j \mid \sim ((D0_j >> 1) \mid HP_j)$
8. $VN_j \leftarrow (D0_j >> 1)\ \&\ HP_j$

Figure 20: Computation of column j with the Damerau edit distance and diagonal tiling (for the case $l_v \leq w$).

and B is below some pre-determined error-threshold k. This is because the principle of the algorithm makes it in practice most suitable for this type of use. Therefore all tested algorithms used a scheme similar to the cut-off method briefly discussed in the end of Section 3.2. As a baseline we also show the runtime of using the original BPM, the $O(\lceil m/w \rceil n)$ bit-parallel algorithm of Myers.

The test consisted of repeatedly selecting two substrings in pseudo-random fashion from the DNA-sequence of the baker's yeast, and then testing whether their Levenshtein edit distance is at most k. The computer used in the tests was a 600 MHz Pentium 3 with 256 MB RAM and running Microsoft Windows 2000. The code was programmed in C and compiled with Microsoft Visual C++ 6.0 with full optimization. The tested algorithms were:

MYERS: The algorithm of Myers [38] modified to compute edit distance. The run time of the algorithm does not depend on the number of errors allowed. The underlying implementation is from the original author.

MYERS(cut-off): The algorithm of Myers using cut-off modified to compute edit distance. The underlying implementation (including the cut-off-mechanism) is from the original author.

UKKA(cut-off): The method of Ukkonen based on filling only a restricted region of diagonals in the dynamic programming matrix and using the cut-off method (Section 3.1).

UKKB(cut-off): The "greedy" method of Ukkonen [53] based on computing the values in the dynamic programming matrix in increasing order (Section 3.3).

OURS(cut-off): Our method of combining the diagonal restriction scheme of Ukkonen with BPM. We used a similar method of tracking the lowest active row as is used in ABNDM/BPM (Section 8.1).

The results are shown in Fig. 21. We tested sequence pairs with lengths 100, 1000 and 10000, and error thresholds of 10%, 20% and 50% of the sequence length (for example $k = 100$, 200 and 500 for the sequence length $m = n = 1000$). It can be seen that in the case of $k = 10$ and $m = 100$ UKKB(cut-off) is the fastest, but in all other tested cases our method becomes the fastest, being 8 % - 38 % faster than the original cut-off method of Myers that is modified to compute edit distance. The good performance of UKKB(cut-off) with low values of k is not too surprising as its expected run time has been shown to be $O(m+k^2)$ [36]. But we did investigate this a bit more and concluded that a major reason for the bit-parallel methods not to be always the fastest is that for low k the cost for preprocessing the match vectors (PM_λ) becomes significant.

	$m = n = 100$			$m = n = 1000$			$m = n = 10000$		
error limit (%)	10	20	50	10	20	50	10	20	50
UKKA(cut-off)	1,92	5,93	32,6	13,5	52,7	322	13,1	54,9	351
UKKB(cut-off)	1,23	3,02	14,9	6,17	22,9	139	5,57	22,4	146
MYERS(cut-off)	2,46	3,23	4,07	2,47	4,48	15,9	0,71	2,35	13,4
OURS(cut-off)	2,27	2,47	3,32	1,96	3,08	10,5	0,48	1,47	9,03
MYERS	4,24			17,0			14,5		

Figure 21: The results (in seconds) for thresholded edit distance computation between pairs of randomly chosen DNA-sequences from the genome of the baker's yeast. The error threshold is shown as the percentage of the pattern length (tested pattern pairs had equal length). The number of processed sequence pairs was 100000 for $m = n = 100$, 10000 for $m = n = 1000$, and 100 for $m = n = 10000$.

7.2 Further Considerations

We note that the diagonal area of the dynamic programming matrix could also be covered using the conventional horizontal tiling: compute the matrix

otherwise in a similar manner as the cut-off version of the original method of Myers [38], but now excluding all tiles that do not overlap the diagonal area. The run time would be $O(\sigma + (\lceil ed(P,T)/w \rceil + 1)m)$. This method could be competitive for large values of $ed(P,T)$ as then the additional work could be compensated by the benefit of not having to build the match vectors at each column.

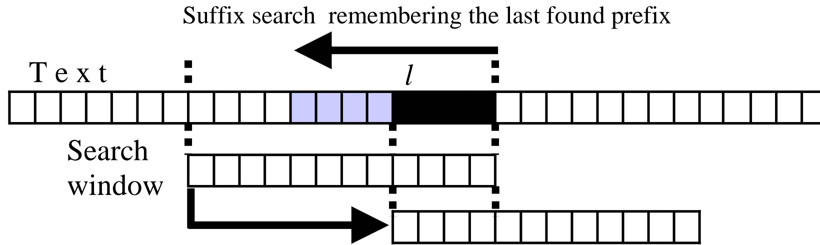


Figure 22: The BDM search scheme.

8 Using BPM in ABNDM

In this section we modify the ABNDM algorithm of Navarro and Raffinot [48] to use BPM (Section 5). The original version used BPR (see Section 6.2, page 29), which has the drawback that its run time scales directly with k. This work has appeared in [24].

In [8] Czumaj et al. proposed the BDM-heuristic for exact string matching. The method is based on the fact that if a pattern occurrence begins at position T_h inside a text window $T_{j..j+m-1}$ of length m, then the prefix $P_{1..j-h+1}$ of the pattern matches the text window suffix $T_{h..j}$. This simple fact can be exploited by using a suffix automaton (an automaton that matches all suffixes of the pattern, discussed for example in [7]) that is built for the reverse pattern P^R so that it detects prefixes of P. Initially $j = 1$, so that the text window is $T_{1..m}$, and the window is searched backwards starting from the last character $T_{j+m-1} = T_m$. The search continues until either the whole window has been searched or the automaton signals that there can no longer be further matches for any prefix of the pattern. If the suffix automaton reaches the beginning of the window and signals a match there, then $T_{j..j+m-1} = P$ and a match of the complete pattern has been found. During the search the last (smallest) l, for which $T_{j+l..j+m-1} = P_{1+l..m}$ and $l > 0$, is recorded. If no such position is found, then $l = m$. After the backward search stops, the window is moved l positions forward so that its first character is T_{j+l}, the next possible text position where a match for the whole pattern can begin. This ensures that no occurrence of the pattern can be missed. Fig. 22 illustrates.

The average run time of BDM is $O(n \log_\sigma(m)/m)$, which is optimal for exact string matching [62].

In [48] Navarro and Raffinot presented BNDM, a bit-parallel version of

BDM that is very fast for small or moderate m. It is optimal on average when $m \leq w$, and $O(n \log_\sigma(w)/w)$ otherwise. BNDM included, among other things, also an extension of the basic heuristic of BDM for approximate string matching under the Levenshtein edit distance. The approximate scheme has later been called ABNDM, and in [42] Navarro used it also under the Damerau edit distance. BNDM and the original ABNDM use modified versions of the bit-parallel NFA-simulations of Baeza-Yates and Gonnet [1] (the case of exact matching) and the BPR of Wu and Manber [60] (the case of approximate string matching).

The heuristic of ABNDM differs from that of BDM (and exact BNDM) in three ways.

The first difference is that the length of the text window is $m - k$, as that is the minimum length that an approximate match with k errors can have.

The second difference is that now the suffix automaton must detect *approximate* matches of the suffixes of P^R (= prefixes of P).

The third difference is that the possible matches must be verified separately. An approximate match between $T_{j..j+m-k-1}$ and some prefix of P does not guarantee an approximate match for the full pattern; it is a necessary but not a sufficient condition for a full match. This means that when the approximate suffix automaton reaches the beginning of the window and signals there a match with some prefix of the pattern, the current location is checked for a full match by running an edit distance computation between P and text from T_j forward until either an approximate match with the pattern is detected or the edit distance algorithm can decide (by using cut-off) that there is no match.

Fig. 23 illustrates the approximate heuristic and Fig. 24 shows a high-level pseudocode.

ABNDM-heuristic has been shown to work well with moderate $m \leq w$, fairly low error-level k/m and small σ. This is an interesting case, for example, in DNA searching. But as the run time of BPR scales directly with k, the original ABNDM can be competitive for only very small k regardless of the values of m and w.

We show how to modify BPM for use with the ABNDM heuristic.

To make the algorithm pseudo-codes in the following discussion simpler, we will enclose the basic logic of the BPM algorithm into the procedure **BPMEDStep** shown in Fig. 25. The shown version keeps the vectors in single bit-vectors. The procedure is shown only for the Levenshtein edit distance. We will not discuss separately the case of using the Damerau edit distance, but it can be handled by adding the modifications shown in

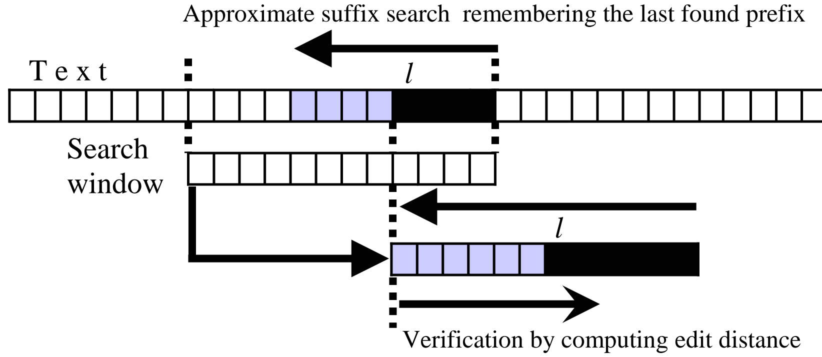
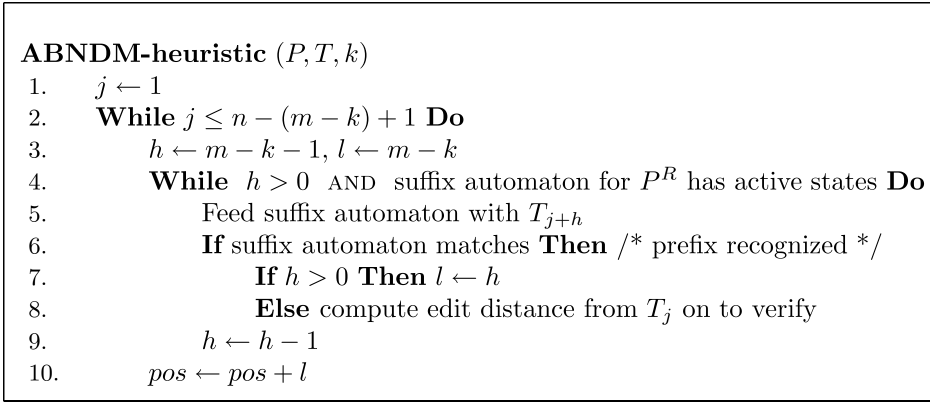


Figure 23: The ABNDM search scheme.



ABNDM-heuristic (P, T, k)
1. $j \leftarrow 1$
2. **While** $j \leq n - (m - k) + 1$ **Do**
3. $h \leftarrow m - k - 1$, $l \leftarrow m - k$
4. **While** $h > 0$ AND suffix automaton for P^R has active states **Do**
5. Feed suffix automaton with T_{j+h}
6. **If** suffix automaton matches **Then** /* prefix recognized */
7. **If** $h > 0$ **Then** $l \leftarrow h$
8. **Else** compute edit distance from T_j on to verify
9. $h \leftarrow h - 1$
10. $pos \leftarrow pos + l$

Figure 24: The generic **ABNDM** algorithm.

Fig. 16 in Section 6.2. In order to compute edit distance, lines 4 and 5 of **BPMEDStep** set the first bit of VP after shifting it (see Section 5).

8.1 Verifying with BPM

Let us start by discussing how to use BPM in the verification process.

Assume that we want to verify the position T_j for a possible approximate match, and consider the dynamic programming table D that corresponds to the edit distance computation between P and $T_{j..}$. We assume the same order as usual, that is, that the pattern is along the left side of D and $T_{j..}$ is on top. As the maximum length of an approximate match is $m + k$, D will have at most $(m+k+1)$ columns. By following the cut-off method (see

BPMEDStep(PM_λ)

1. $D0 \leftarrow (((PM_\lambda \ \& \ VP) + VP) \wedge VP) \mid PM_\lambda \mid VN$
2. $HP \leftarrow VN \mid \sim (D0 \mid VP)$
3. $HN \leftarrow D0 \ \& \ VP$
4. $VP \leftarrow (HN << 1) \mid \sim (D0 \mid ((HP << 1) \mid 0^{m-1}1))$
5. $VN \leftarrow D0 \ \& \ ((HP << 1) \mid 0^{m-1}1)$

Figure 25: The basic update logic of the BPM algorithm that is modified to compute edit distance.

Section 3.2), the verification can be abandoned if all cells in the currently computed column have values larger than k.

As mentioned in Section 5, it is straightforward to modify BPM to compute edit distance. But a problem is how to handle the cut-off heuristic. BPM knows only the value $D[m, j]$ explicitly at column j and the other cell values are encoded implicitly by the delta vectors (see Section 5). In order to use cut-off, we need to know if all cell values in the current column are larger than k. We propose to handle this problem by keeping track of the lowest active row (the lowest row with a value $\leq k$) in each column. We do this with a bit-vector L_j that is defined as follows:

-The lowest active row vector L_j:

$$L_j[i] = 1 \text{ iff } i = \max\{h \mid D[h, j] \leq k \text{ and } 1 \leq h \leq m \}.$$

Let ℓ_j denote the index of the lowest active row in column j. Then $L_j = 1 << (\ell_j - 1) = 10^{\ell_j - 1}$. Note that if there is no active row, then ℓ_j is undefined and $L_j = 0^m$.

From the boundary conditions on D we have that $\ell_j = k$ and $L_j = 0^{m-k}10^{k-1}$ initially when $j = 0$. It is clear from the diagonal property that $\ell_j \leq \ell_{j-1}+1$. Our strategy is to first assume that this increment $\ell_j = \ell_{j-1}+1$ happens, which corresponds to setting $L_j = L_{j-1} << 1$. This enables us to also handle the case $k = 0$ correctly: in that case L_0 would need to have only its non-existing 0th bit set. But as we now pre-shift L_j before text position j, we can start the algorithm by setting $L_1 = L_0 << 1 = 0^{m-k-1}10^k$, which works also when $k = 0$.

At text position j we check whether the pre-shift of L_j was correct by AND-ing $D0_j$ and L_j. If the result is zero, then $D[\ell_{j-1}+1, j] = D[\ell_{j-1}, j-1] + 1 = k+1$ and L_j needs to be recomputed. This is done by using the

vertical delta vectors VN_j and VP_j. We start from the value $D[\ell_{j-1}+1, j] = k+1$ and use VN_j, VP_j and L_j to compute the values $D[\ell_{j-1} - h, j]$ for $h = 0, 1, ..$ until either a value $\leq k$ is found or there are no more rows left ($h \geq \ell_{j-1}$). The vector L_j is updated during the computation.

A match is found if row m becomes active, and this is detected by checking if $L_j = 10^{m-1}$. Note that the condition $D[\ell_j, j] = k$ holds throughout the verification as it is stopped when a match is found.

Fig. 26 shows the algorithm. Here we keep/update also the current value of L_j in a single bit-vector L. The amortized cost of updating L_j is constant per processed text character as $\ell_j \leq \ell_{j-1} + 1$, initially $\ell_j = O(k)$, each increment or decrement takes constant time and at least $O(k)$ text characters need to be processed before ℓ_j becomes undefined.

BPMVerify(PM, j)
1. $VP \leftarrow 1^m$, $VN \leftarrow 0^m$
2. $L \leftarrow 0^{m-k-1}10^k$
3. **While** $j \leq n$ **Do**
4. **BPMEDStep**(PM_{T_j})
5. **If** $D0$ & $L = 0^m$ **Then**
6. $val \leftarrow k+1$
7. **While** $val > k$ **Do**
8. **If** $L = 0^{m-1}1$ **Then Return** FALSE
9. **If** VP & $L \neq 0^m$ **Then** $val \leftarrow val - 1$
10. **If** VN & $L \neq 0^m$ **Then** $val \leftarrow val + 1$
11. $L \leftarrow L >> 1$
12. **Else If** $L = 10^{m-1}$ **Then Return** TRUE
13. $L \leftarrow L << 1$
14. $j \leftarrow j+1$
15. **Return** FALSE

Figure 26: Adaptation of BPM to verify whether an occurrence of the pattern starts from T_j.

8.2 Suffix Matching with the BPM Simulation

The backward scanning phase of ABNDM uses a suffix automaton for P^R to match prefixes of P. Approximate matching suffixes of $A = P^R$ against a prefix of $B = (T_{j..j+m-k-1})^R$ can be defined as finding the indices j' for which $ed_L(A_{h..m}, B_{1..j'}) \leq k$. Since the edit distances we deal with are symmetric, we know from the discussion of approximate string matching

version of the dynamic programming table D (Section 2) that using the boundary conditions $D[i,0] = 0$ and $D[0,j] = j$ results in having $D[i,j] = \min(ed_L(A_{h..i}, B_{1..j}), h \leq i)$. In particular, we then have that $D[m,j] = \min(ed_L(A_{h..m}, B_{1..j}), h \leq m)$, which corresponds to matching suffixes of A against the prefix $B_{i..j}$: a suffix of A matches $B_{1..j}$ when $D[m,j] \leq k$. Thus the procedure **BPMEDStep** in Fig. 25 can be used also in this type of approximate suffix matching by using the initial settings $VN_j = 0^m$ and $VP_j = 0^{m-k}$. Note that we use the strings $A = P^R$ and $B = (T_{j..j+m-k-1})^R$ in order to match prefixes of P against a suffix of $T_{j..j+m-k-1}$.

The problem now is how to use cut-off efficiently with the backward suffix matching; this is crucial in terms of the performance of the ABNDM heuristic as it seeks to skip text characters by abandoning the current text window after scanning only relatively few characters from its end.

In principle we could use a similar method of keeping track of the lowest active row with L_j as was done in the previous section. But as the boundary condition is now $D[i,0] = 0$ and the backward scan proceeds at least as long as at least one cell in the current column is active, there are two changes. First of all the condition $D[\ell_j, j] = k$ does not hold any more: when $\ell_j = m$, the cell $D[\ell_j, j]$ may have any value between 0 and k. We can handle this by updating L_j as before only when $\ell_j < m$, and computing the value $D[m,j]$ explicitly as long as $\ell_j = m$. A second change is that now $\ell_0 = m$, which means that the cost of updating L_j is no more amortized to a constant time per text character. This makes this basic approach quite slow.

In the following sections we present two solutions to determine faster that all the $D[i,j]$ values at current position j have surpassed k.

Bit-parallel Counters

The original BPM computes explicitly only the value $D[m,j]$ at column j. To have more explicit information about the cell values at the current column j, we discuss in this section a scheme to compute many such values in parallel.

Our proposal is to put several counters into a single bit-vector MC so that they can be updated in parallel. Each counter will know the explicit value of some cell $D[i,j]$. Because the window is of length $m-k$, we know that $0 \leq D[i,j] \leq m-k$ during the backward scan. Thus each counter needs at least $\lceil \log_2(m-k) \rceil$ bits and there can be at most $O(m/\log m)$ of them (assuming $m \leq w$).

Assume that each counter takes Q bits and that there are $t = \lceil m/Q \rceil$ counters that know the values $D[m,j], D[m-Q,j], .., D[m-(t-1)Q,j]$

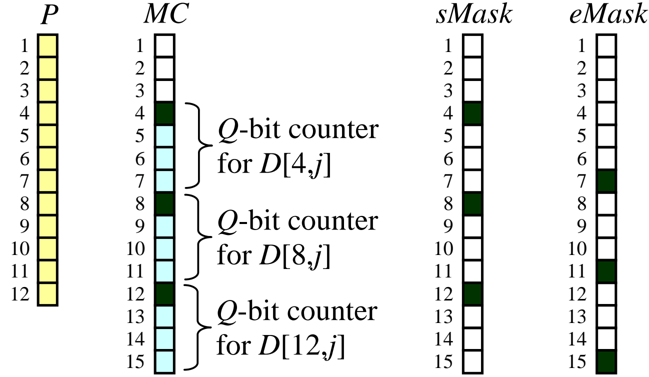


Figure 27: The structure of the bit-vectors MC, $sMask$ and $eMask$ when $m = 12$ and $Q = 4$.

at column j. In this situation the counters need $m + Q - 1$ bits, and so we assume from now on that $m + Q - 1 \leq w$. We compute a bit-vector $sMask = (0^{Q-1}1)^t 0^{m+Q-1-tQ}$ that has set bits in exactly those locations that are aligned with the first bits of the counters (see Fig. 27). By using $sMask$ we can update the counters of MC at position j by setting $MC \leftarrow MC - (HN_j \ \& \ sMask) + (HP_j \ \& \ sMask)$.

If we want to use only the values $D[m, j], D[m-Q, j], .., D[m-(t-1)Q, j]$ to know that all cells in the current column are $> k$, we need to have that $D[m - hQ, j] > k + \lfloor Q/2 \rfloor$ for $h = 0..(t-1)$. In the following we use the notation $k' = k + \lfloor Q/2 \rfloor$. The preceding rule is a consequence of the adjacency property (Section 2). Because $D[0, j] = j \geq D[i, j]$ for $i > 0$, we may think of having an implicit counter at row 0: If any counter has a value $> k'$, then also $D[0, j] > k'$. Each cell is at most at distance $\lfloor Q/2 \rfloor$ from a counter or row 0, and so for any i we have that $D[i, j] > k' - \lfloor Q/2 \rfloor = k$ when all counters have a value $> k'$.

To be able to check the counters fast, we set up each counter in such a way that its last bit is activated exactly when the value of the counter is $> k'$. This is done by adding a value b to the value of each counter. Given the previous discussion, the values Q and b have to fulfill the following rules:

(1) $b + k' + 1 = 2^{Q-1}$
(2) $b + m - k < 2^Q$
(3) $b \geq 0$

To check the last bits of the counters we use a bit-vector $eMask = sMask << (Q-1) = (10^{Q-1})^t 0^{m+Q-1-tQ}$. It has set bits at the positions that are aligned with the last bits of the counters (see Fig. 27). Now all counters have a value $> k' = k + \lfloor Q/2 \rfloor$ exactly when $MC \ \& \ eMask = eMask$.

The value of Q is determined so that it is the minimal number that fulfills the previous conditions (1)-(3). These rules can be transformed into requiring that $b = 2^{Q-1} - k' - 1$, $m - k - k' \leq 2^{Q-1}$ and $k' + 1 \leq 2^{Q-1}$. From this we get the solution $Q = 1 + \lceil \log_2(\max(m - k - k', k' + 1)) \rceil$ and $b = 2^{Q-1} - k' - 1$, which contains a recurrence for Q. Fortunately it is easy to solve. Since $(X+Y)/2 \leq \max(X,Y) \leq X + Y$ for any nonnegative X and Y, we have that $\log_2(((m-k-k')+(k'+1))/2) = \log_2((m-k+1)/2) \leq \log_2(\max(m-k-k', k'+1)) \leq \log_2((m-k-k')+(k'+1)) = \log_2(m-k+1)$. As $\log_2(m-k+1) = 1 + \log_2((m-k+1)/2)$, the result is that $\lceil \log_2(m-k+1) \rceil \leq Q \leq 1 + \lceil \log_2(m-k+1) \rceil$, a 2-integer range for the actual Q value. If $Q = \lceil \log_2(m-k+1) \rceil$ does not satisfy the conditions (1)-(3), we use $Q+1$. This scheme works correctly as long as $X, Y \geq 0$, that is, $\lfloor Q/2 \rfloor \leq m - 2k$, or $m - k \geq k'$. The preceding is not a true restriction because if $m - k < k'$, then none of the counters will ever get a value $> k'$ and they would thus be useless anyway.

It is also possible to use several interleaved sets of counters, each set in its own bit-parallel counter. This would make the distance between two adjacent counters smaller and hence decrease the limit k'. For example with two interleaved MC counters we could use the value $k' = k + \lfloor Q/4 \rfloor$, and in general with c counters the value $k' = k + \lfloor Q/2^c \rfloor$.

Because we traverse the window backwards until all the counters exceed k', we will examine a few more text characters than if we had known the exact values of all $D[i,j]$: The backward scan will behave as if we permitted $k' = k + \lfloor Q/2 \rfloor$ differences. Note that the amount of shifting in the ABNDM heuristic is not affected: we do know the exact value of $D[m,j]$.

Fig. 28 shows the algorithm for using bit-parallel counters. All the bit masks are of length m, except $sMask$, $eMask$ and MC, which are of length $m + Q - 1$. Now we use two different pattern match tables: PM for P and PMR for P^R.

Bit-parallel Cut-off

The technique of the previous section has the problem that it inspects more text characters than necessary. Now we propose a way to achieve a more accurate cut-off with the bit-parallel counters. The idea is to mix the bit-parallel counters with the technique of keeping track of the lowest active

row.

Assume that, given a value Q that we define later, the multiple counter vector MC is built in similar fashion as in the previous section. Thus it contains $t = \lceil m/Q \rceil$ counters that hold the values of the cells $D[m, j], D[m - Q, j], .., D[m - (t-1)Q, j]$. Now each counter is assigned a region of Q bits so that the counter for $D[m - hQ, j]$ handles the cells $D[m - (h+1)Q + 1, j], ..D[m - hQ, j]$. Note that the counter for $D[m - (t-1)Q, j]$ may have a smaller region, but this will not be a problem. We will keep track of the lowest active row within these regions. By this we mean that we will know the minimum x for which $D[m - hQ - x, j] \leq k$ with some $h \in [0..t-1]$. Let δ_j denote the value of x at column j, that is, the minimum distance between an active cell and the end of the corresponding counter region. As $D[m, 0] = 0$, we will initially have $\delta_0 = 0$.

Now the value of Q is the minimum number that fulfills the following conditions:

(1) $b + k + 1 = 2^{Q-1}$
(2) $b + m - k < 2^Q$
(3) $b \geq 0$

This time we can directly compute $Q = 1 + \lceil \log_2(max(m - 2k, k + 1)) \rceil$.

Keeping track of the lowest active row within some region can be done in parallel. We will trace the lowest active row(s) by keeping track of the value δ_j and setting the counter-vector MC to hold the values of the cells $D[m - \delta_j, j], D[m - Q - \delta_j, j], .., D[m - (t-1)Q - \delta_j, j]$ at column j. The values of δ_j and MC can be computed by following a similar strategy as was done with ℓ_j and L_j in Section 8.1. If $\delta_{j-1} > 0$, we assume that $\delta_j = \delta_{j-1} - 1$, and otherwise that $\delta_j = \delta_{j-1} = 0$. Note that as δ_j measures the distance to the end of a region, decrementing it corresponds to incrementing ℓ_j. Then the values of the counters are updated. If we set initially $\delta_j = \delta_{j-1} - 1$, the counter values are updated by setting $MC \leftarrow MC + (\sim (D0_j << \delta_j) \ \& \ sMask)$, and otherwise by setting $MC \leftarrow MC - (HN_j \ \& \ sMask) + (HP_j \ \& \ sMask)$. The latter needed no shifting as then we initially assume that $\delta_j = \delta_{j-1} = 0$. If no counter has a value $\leq k$ after updating them, we start to increment δ_j and update MC correspondingly. This is done by first setting $MC \leftarrow MC + ((VN_j << \delta_j) \ \& \ sMask) - ((VP_j << \delta_j) \ \& \ sMask)$ and only then $\delta_j \leftarrow \delta_j + 1$. This order of computation comes from how we move "the wrong way" along the vertical delta vectors VN and VP. We keep incrementing δ_j until either some counter has a value $\leq k$ or we would be already trying the value $\delta_j = Q$. In the latter case we stop because then it

is known that all cells in the current column have a value $> k$ (each counter has already tried all remaining possibly active cells within its region).

It is possible that the value $m - (t-1)Q - \delta_j$ becomes smaller than 0 while incrementing δ_j, which means that the upmost counter would no more correspond to an existing cell $D[i, j]$. For this to happen, however, the whole region of the upmost counter must have values larger than k, and it is not possible that a cell in this area gets a value $\leq k$ later. So if the upmost counter goes out of bounds, we can safely stop using it by clearing its check bit from $eMask$. Note that ignoring this fact leads to inspecting slightly more characters (an almost negligible amount) but one instruction is saved, which is convenient in practice.

We also tried a practical version of using cut-off, in which the counters are not shifted. Instead they are updated in a similar fashion to the algorithm of Fig. 28, and when all counters have a value $> k$, we try to shift a *copy* of them up until either a cell with value $\leq k$ is found or $Q - 1$ consecutive shifts are made. In the latter case we can stop the search, since then we have covered checking the whole column j. This version turned out to be sometimes slightly faster than the present cut-off algorithm, but the difference between the two is *very* small.

Fig. 29 shows the algorithm. The counters are not shifted, we use δ instead.

8.3 Experimental Results

We compared our BPM-based ABNDM against the original BPR-based ABNDM, as well as those other algorithms that, according to a recent survey [41], are the best for moderate pattern lengths. We tested with random patterns and text over uniformly distributed alphabets. Each individual test run consisted of searching for 100 patterns from a text of size 10 Mb. We measured total elapsed times.

The computer used in the tests was a 64-bit Alphaserver ES45 with four 1 GHz Alpha EV68 processors, 4 GB of RAM and Tru64 UNIX 5.1A operating system. We used a 64-bit architecture in order to test our ABNDM with higher k values: As we test only with $m \leq w$ and the ABNDM heuristic works for only a fairly low error level k/m, the limit $m \leq w = 32$ would have restricted us to very low k values. All test programs were compiled with the DEC CC C-compiler and full optimization. There were no other active significant processes running on the computer during the tests. All algorithms were set to use a 64 KB text buffer. The tested algorithms were:

ABNDM/BPR(regular): ABNDM implemented on BPR [59], using a generic implementation for any k.

ABNDM/BPR(special code): The same as above, but especially coded for each value of k to avoid using an array of bit masks.

ABNDM/BPM(1 counter): ABDNM implemented using BPM and bit-parallel counters (Sec. 8.2). The implementation differed slightly from Fig. 28 due to optimizations.

ABNDM/BPM(2 counter): The same as above, but using two interleaved counter vectors.

ABNDM/BPM(cut-off): ABDNM implemented using BPM and cut-off (Sec 8.2). The implementation differed slightly from Fig. 29 due to optimizations.

ABNDM/BPM(static): The practical version of AMNDM/cutoff that does not actively shift the counters.

BPM: The sequential BPM algorithm [38]. The implementation was from us and used the slightly different (but practically equivalent in terms of performance) formulation from [17].

BPP: A combined heuristic using pattern partitioning, superimposition and hierarchical verification, together with a diagonally bit-parallelized NFA [2, 46]. The implementation was from the original authors.

EXP: Partitioning the pattern into $k + 1$ pieces and using hierachical verification with a diagonally bit-parallelized NFA in the checking phase [44]. The implementation was from the original authors.

Fig. 30 shows the test results for $\sigma = 4$, 13 and 52 and $m = 30$ and 55. This is only a small part of our complete tests, which included $\sigma = 4, 13, 20, 26$ and 52, and $m = 10, 15, 20, \ldots, 55$. We chose $\sigma = 4$ because it behaves like DNA, $\sigma = 13$ because it behaves like English, and $\sigma = 52$ to show that our algorithms are useful even on large alphabets.

Among our ABNDM/BPM variants the results show that using cut-off is always faster than using counters by a nonnegligible margin. ABNDM/BPM(cut-off) and ABNDM/BPM(static) are the fastest, and the difference between these two is very small.

It can be seen that our ABNDM/BPM versions are often faster than ABNDM/BPR(special code) when $k = 4$, and always when $k > 4$. Compared

to ABNDM/BPR(regular), our version is always faster for $k > 1$. We note that writing down a different procedure for every possible k value, as done for ABNDM/BPR(special code), is hardly a real alternative in practice.

With moderate pattern length $m = 30$, our ABNDM/BPM versions are competitive for low error levels. However, BPP is better for small alphabets and EXP is better for large alphabets. In the intermediate area $\sigma = 13$, we are the best for $k = 4 \ldots 6$. This area is rather interesting when searching natural language text.

When $m = 55$, our ABNDM/BPM versions become much more competitive, being the fastest in many cases: For $k = 5 \ldots 9$ with $\sigma = 4$, and for $k = 4 \ldots 11$ both with $\sigma = 13$ and $\sigma = 52$, with the single exception of the case $\sigma = 52$ and $k = 9$, where EXP is faster (this seems to be a variance problem, however).

ABNDMCounters(P, T, k)

1. `Preprocessing`
2. **For** $c \in \Sigma$ **Do** $PM[c] \leftarrow 0^m$, $PMR[c] \leftarrow 0^m$
3. **For** $i \in 1 \ldots m$ **Do**
4. $PM_{P_i} \leftarrow PM_{P_i} \mid 0^{m-i}10^{i-1}$
5. $PMR_{P_i} \leftarrow PMR_{P_i} \mid 0^{i-1}10^{m-i}$
6. $Q \leftarrow \lceil \log_2(m-k+1) \rceil$
7. **If** $2^{Q-1} < \max(m - 2k - \lfloor Q/2 \rfloor, k + 1 + \lfloor Q/2 \rfloor)$ **Then** $Q \leftarrow Q + 1$
8. $b \leftarrow 2^{Q-1} - k - \lfloor Q/2 \rfloor - 1$
9. $t \leftarrow \lceil m/Q \rceil$
10. $sMask \leftarrow (0^{Q-1}1)^t 0^{m+Q-1-tQ}$
11. $eMask \leftarrow (10^{Q-1})^t 0^{m+Q-1-tQ}$
12. `Searching`
13. $pos \leftarrow 0$
14. **While** $pos \leq n - (m-k)$ **Do**
15. $j \leftarrow m - k$, $last \leftarrow m - k$
16. $VP \leftarrow 0^m$, $VN \leftarrow 0^m$
17. $MC \leftarrow [b]_Q^t 0^{m+Q-1-tQ}$
18. **While** $j \neq 0$ AND $MC \ \& \ eMask \neq eMask$ **Do**
19. **BPMEDStep**$(PMR_{T_{pos+j}})$
20. $MC \leftarrow MC + (HP \ \& \ sMask) - (HN \ \& \ sMask)$
21. $j \leftarrow j - 1$
22. **If** $MC \ \& \ 10^{m+Q-2} \neq 0^{m+Q-1}$ **Then** /* prefix recognized */
23. **If** $j > 0$ **Then** $last \leftarrow j$
24. **Else If BPMVerify**$(PM, pos+1)$ **Then**
25. Report an occurrence at $pos + 1$
26. $pos \leftarrow pos + last$

Figure 28: The **ABNDM** algorithm using bit-parallel counters. The expression $[b]_Q$ denotes the number b seen as a bit mask of length Q. Note that **BPMVerify** can share its variables with the calling code because these are not needed any more at that point.

ABNDMCut-off $(P_{1\ldots m},\ T_{1\ldots n},\ k)$

1. `Preprocessing`
2.   **For** $c \in \Sigma$ **Do** $PM[c] \leftarrow 0^m$, $PMR[c] \leftarrow 0^m$
3.   **For** $i \in 1 \ldots m$ **Do**
4. $PM_{P_i} \leftarrow PM_{P_i} \mid 0^{m-i}10^{i-1}$
5. $PMR_{P_i} \leftarrow PMR_{P_i} \mid 0^{i-1}10^{m-i}$
6. $Q \leftarrow 1 + \lceil \log_2(\max(m-2k, k+1)) \rceil$
7. $b \leftarrow 2^{Q-1} - k - 1$
8. $t \leftarrow \lceil m/Q \rceil$
9. $sMask \leftarrow (0^{Q-1}1)^t 0^{m+Q-1-tQ}$
10. $eMask \leftarrow (10^{Q-1})^t 0^{m+Q-1-tQ}$
11. `Searching`
12. $pos \leftarrow 0$
13.   **While** $pos \leq n-(m-k)$ **Do**
14. $j \leftarrow m-k,\ last \leftarrow m-k$
15. $VP \leftarrow 0^m,\ VN \leftarrow 0^m$
16. $MC \leftarrow [b]_Q^t 0^{m+Q-1-tQ}$
17. $\delta \leftarrow 0$
18.   **While** $j \neq 0$ AND $\delta < Q$ **Do**
19.   **BPMEDStep** $(PMR_{T_{pos+j}})$
20.   **If** $\delta = 0$ **Then** $MC \leftarrow MC + ((HP\ \&\ sMask) - (HN\ \&\ sMask))$
21.   **Else**
22. $\delta \leftarrow \delta - 1$
23. $MC \leftarrow MC + (\sim (D0 << \delta)\ \&\ sMask)$
24.   **While** $\delta < Q$ AND $MC\ \&\ eMask = eMask$ **Do**
25. $MC \leftarrow MC - ((VP << \delta)\ \&\ sMask) + ((VN << \delta)\ \&\ sMask)$
26. $\delta \leftarrow \delta + 1$
27.   **If** $\delta = m-(t-1)Q$ **Then** $eMask \leftarrow eMask\ \&\ 1^{(t-1)Q}0^{m+2Q-1-tQ}$
28. $j \leftarrow j-1$
29.   **If** $\delta = 0$ AND $MC\ \&\ 10^{m+Q-2} \neq 0^{m+Q-1}$ **Then** /* prefix recognized */
30.   **If** $j > 0$ **Then** $last \leftarrow j$
31.   **Else If BPMVerify** $(PM, pos+1)$ **Then**
32.   Report an occurrence at $pos+1$
33. $pos \leftarrow pos + last$

Figure 29: The **ABNDM** algorithm using bit-parallel cut-off. The same comments of Fig. 28 apply.

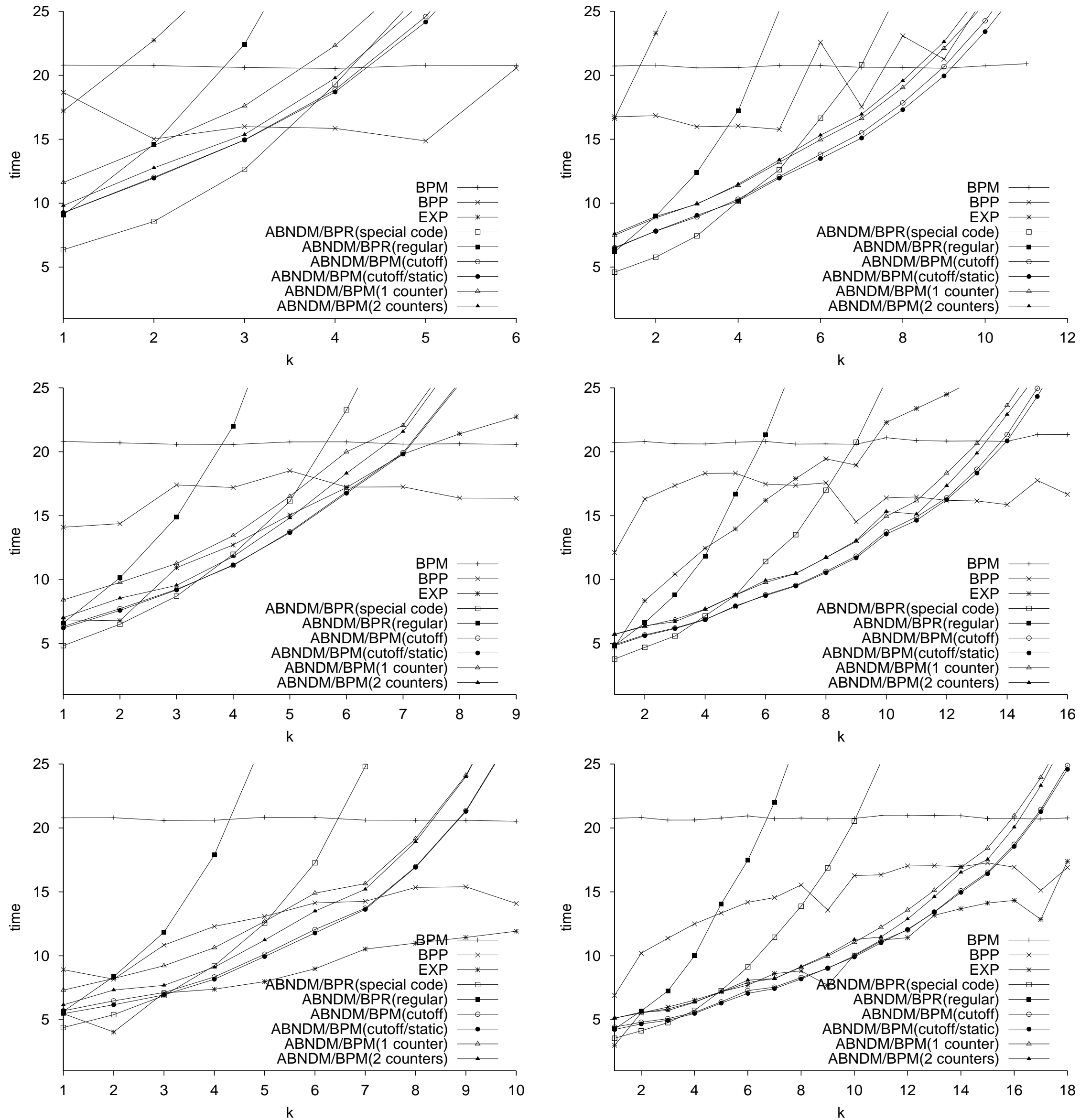


Figure 30: Comparison between algorithms, showing total elapsed time as a function of the number of differences permitted, k. From top to bottom row we show $\sigma = 4$, 13 and 52. On the left we show $m = 30$ and on the right $m = 55$.

9 On Using Two-Phase Filtering

In this section we first discuss using so-called *filtering conditions*, then proceed to propose a scheme of "two-phase filtering", and finally present some experimental results in conjuction with a specific application in bioinformatics. The section is largely based on the conference paper [19] and focuses only on the Levenshtein edit distance.

One way to accelerate approximate string matching is to try to minimize the amount of text that actually has to be searched with the abovementioned algorithms. This leads to dividing the overall searching process into two different procedures: filtering (or scanning) and checking. The former procedure tries to decide quickly which parts of the text possibly contain approximate matches of the searched string, and the latter procedure then uses one of the verification capable (can explicitly measure edit distance) approximate string matching algorithms to explicitly search only these areas of the text. Filtering is generally based on checking which portions of the text fulfill some minimal necessary criteria that any approximate match of the pattern must fulfill. The text locations that are not filtered out are often referred to as surviving regions. Some filtering criteria are discussed in more detail in the next subsection.

To be useful, the filtering process naturally has to have low-enough overhead and high-enough filtration efficiency to make the overall search faster than simply searching sequentially through the whole text. Achieving this goal depends a lot on whether it is plausible to preprocess the text before the search takes place. This type of searching is sometimes called off-line (as opposed to on-line) searching. We focus mainly on off-line searching where the text can be preprocessed. In this case we also assume that the text will be a target for searching for many different patterns since otherwise the preprocessing would be pointless due to its often considerable overhead. The goal of the preprocessing is generally to build an index, which can be queried to directly obtain the surviving regions, i.e. the locations in the text that still need further checking after applying the used filtering criteria.

Filtering can be viewed from two perspectives: We can think of it as pruning out text areas that are dissimilar to the pattern or as searching for text areas that are similar to the pattern. These views are complementary and affect only the nature of the discussion. We choose the latter perspective, searching for similar areas.

9.1 General Filtering Criteria

Practically all filtering criteria boil down to determining how much of the pattern must remain unaffected after it is altered with k arbitrarily chosen edit operations. This is usually treated as a combinatorial problem, which is dealt with application of the Pigeonhole Principle. A general approach is to choose a set of substrings from the pattern and count how they can be affected by k edit operations. For this purpose we propose Lemma 9.1, which can be seen as a generalization/combination of the filtering conditions presented in [15, 2].

Lemma 9.1 *Let S be a set of substrings from the pattern and o be an overlapping factor, which specifies the maximum number of substrings in S that overlap the same character. If the pattern has been edited with at most k edit operations, then there exists a subset S_g of S for which the following conditions hold:*

1) $|S_g| \geq g = |S| - \lfloor ko/(d_s + 1) \rfloor$.
2) Each string $s \in S_g$ has been altered by at most d_s edit operations.

Proof: Let b denote the number of substrings in S that have been altered by more than d_s edit operations. Clearly then the claim of the lemma is true if and only if $b \leq \lfloor ko/(d_s+1) \rfloor$. Let G be a graph with k vertices corresponding to the edit operations and $|S|$ vertices corresponding to the substrings of the set S. If for each k edit operations we add into G directed edges from its edit-vertex to each substring-vertex that corresponds to a substring altered by the operation, the graph will include at most ko edges. Now suppose that $b > \lfloor ko/(d_s + 1) \rfloor$. Because b is an integer, $b > ko/(d_s + 1)$. But this means that there must be at least $(d_s + 1)b > (d_s + 1)ko/(d_s + 1) > ko$ vertices in G, which is a contradiction. Thus $b \leq \lfloor ko/(d_s + 1) \rfloor$ and the claim of the lemma is true. □

Lemma 9.1 gives a simple and flexible basis to composing filtering schemes. It tells us that if we choose a set S of substrings from the pattern, then each approximate occurrence of the pattern must contain at least $g = |S| - \lfloor ko/(d_s + 1) \rfloor$ of them with at most d_s errors. Thus by varying the size of S as well as the level of overlap and the number of errors allowed within S, we can experiment with the lower limit for the number of substrings from S that any approximate match for the pattern must contain. In the following discussion we assume that Lemma 9.1 is used in such a manner that $g > 0$ and all strings in the set S are at least $d_s + 1$ characters long. This is

because otherwise the filtering criteria could include conditions that are in effect empty (satisfied by all locations of the text).

A natural additional condition is to require that the g substrings must occur in such locations with respect to each other that they could belong to an approximate match for the pattern. This requirement is formulated in Lemma 9.2.

Lemma 9.2 *Let S be a set of substrings from the pattern and for each $s_i \in S$, let u_i denote the position of the rightmost character of s_i in the original pattern and v_i denote the position of the rightmost character of s_i after the pattern has been edited. If the pattern has been edited by at most k edit operations, then the following conditions hold:*

1) If $s_i \in S$, then $|v_i - u_i| \leq k$.
2) If $s_i, s_j \in S$, then $|(v_i - u_i) - (v_j - u_j)| \leq k$.

Proof: A single edit operation can change character-position or -distance by at most one, and so k operations can result in a total change of at most k. Thus k is an upper limit for the left sides of the inequalities 1) and 2), which denote, respectively, the change in position of the rightmost character of a substring s_i, and the change in the distance between the rightmost ends of two substrings s_i and s_j. □

For non-overlapping substrings we can formulate the following stricter version of the condition 2) in Lemma 9.2.

Lemma 9.3 *If S is a set of non-overlapping substrings from the pattern, then the condition 2) in Lemma 9.2 can be replaced with the following condition 2'):*

2') If $s_i, s_{i+1}, .., s_{i+j} \in S$ and $u_{i+h-1} < u_{i+h}$ for $h = 1, .., j$, then $\sum_{h=1}^{j} |(v_{i+h} - u_{i+h}) - (v_{i+h-1} - u_{i+h-1})| \leq k$.

Proof: Initially $\sum_{h=1}^{j} |(v_{i+h} - u_{i+h}) - (v_{i+h-1} - u_{i+h-1})| = 0$. A single edit operation can increment only one of the distances $|(v_{i+h} - u_{i+h}) - (v_{i+h-1} - u_{i+h-1})|$, and the increment can be at most one. Thus k edit operations can increase the value of the whole summation by at most k. □

By combining Lemma 9.1 with either Lemma 9.2 or Lemma 9.3 we have that a text region can include an approximate match of the pattern only if

it contains at least g such substrings from a chosen pattern substring set S, that the locations of the substrings with respect to each other fulfill the conditions of Lemma 9.2 (or Lemma 9.3 if there is no overlap within S). The regions that fulfill the filtering criteria are the *surviving regions*. This term refers to the fact that we have not been able to rule out the possibility of them containing approximate matches, and therefore these regions require further checking.

After we have found a set of at least g substrings that fulfill the used filtering criteria, the boundaries for the surviving region implied by these substrings can be determined by using the condition 1) of Lemma 9.2. It is straightforward to see that the upper limit for the change in position of the last character of a string $s \in S$ is also an upper limit for the change in the distance between the last character of s_i and the left or the right end of the pattern.

One additional note is that due to the symmetry of the Levenshtein edit distance, Lemmas 9.2 and 9.3 can also be applied in the other direction: pick substrings from the text and consider how many of them must be contained in the pattern if the substrings are part of an approximate match. This would be reminiscent of the approach used in [50].

9.2 Overview of Some Filtering Schemes

In this section we briefly overview some of the filtering schemes we have seen in the literature. A more in-depth review on the subject is presented by Navarro in [40, 41], and an interested reader is strongly encouraged to read the referenced original papers.

Wu & Manber

The filtering condition proposed by Wu & Manber [60] corresponds to the setting $o = 1, d_s = 0$ and $|S| = k + 1$ in Lemma 9.1 and states, that if we divide the pattern into $k+1$ nonoverlapping substrings, then an approximate match must contain at least one exact copy of at least one of the substrings. This same rule has also been used in at least the methods of Baeza-Yates & Perleberg [3] and Navarro & Baeza-Yates [43]. The preceding methods did not take into account the location of the substrings in the pattern when defining the boundaries for the surviving regions. This has been improved for example by Navarro & Baeza-Yates [44] by using a requirement that corresponds to Lemma 9.2. Of these mentioned schemes [43] concerned the case of indexed searching, while the other two were focused on on-line

searching. The indexed method used a simple q-gram index, which contains for each possible string of length q a list of all its locations in the text.

Holsti & Sutinen

Setting $o = q, d_s = 0$ and $|S| = m - q + 1$ in Lemma 9.1, where q is a parameter defining the length of each substring, leads to the filtering condition used in the SSEQ filter of Holsti and Sutinen [15]. The method requires that at least $g = (m - q + 1) - kq = m + 1 - (k + 1)q$ substrings of length q from the pattern are present in an approximate match. The value $m + 1 - q$ is the number of all substrings of length q in the pattern. The method also imposes a requirement, which corresponds to Lemma 9.2, on the relative locations of the substrings. This is done by cumulating appropriate counters during a sequential scan over the whole text.

Myers

The indexed filtering method of Myers [35] is quite similar to the setting $o = 1$ and $g = 1$ in Lemma 9.1. It is essentially based on using recursively the fact that if for some strings A and B $ed_L(A, B) \leq k$ and A is split into two pieces A^0 and A^1 (i.e. $A = A^0 \circ A^1$, where $\circ$ denotes concatenation), then there must exist a partition $B = B^0 \circ B^1$ that satisfies the condition $ed_L(A^0, B^0) \leq \lfloor k/2 \rfloor$ or $ed_L(A^1, B^1) \leq \lfloor k/2 \rfloor$. The method uses a simple q-gram index and relies on generating so-called *condensed d-neighborhoods*. The condensed d-neighborhood $UC_d(A)$ of a string A is the set of all such strings B, that $ed_L(A, B) \leq d$ and for all prefixes B' of B $ed_L(A, B') > d$.

The method first splits the pattern recursively until the resulting pieces are short enough with relation to the index. Then the condensed $\lfloor k/2^r \rfloor$-neighborhoods, where r is the number of splitting iterations, are generated for all the pattern pieces. Locations of all strings contained in the generated sets are then found from the text by querying the index. Finally these occurrences are extended in a stepwise manner by merging the pattern pieces back together in the reverse order, which corresponds to backtracking the splitting phase. On each step it is checked whether the merged pattern piece is found from the current text location with at most $\lfloor k/2^i \rfloor$ errors, where i is the number of the splitting iteration in which the currently considered pattern piece was formed. If the test fails, the particular occurrence does not need to be extended any further.

Jokinen, Tarhio & Ukkonen

The *counting* method for filtering proposed by Jokinen, Tarhio & Ukkonen [27] (and later also Navarro [39]) is based on the fact that of the m characters in a region ("window") with length m, at least $m - k$ must still be present after making at most k edit operations. This leads to a rule that T_j can be the last character of an approximate match of the pattern only if the text area $T_{j-m+1..j}$ contains at least $m - k$ characters from the pattern.

Baeza-Yates & Navarro

In [2] Baeza-Yates & Navarro presented the method of *intermediate partitioning*, which corresponds to the setting $o = 1$ and $g = 1$ in Lemma 9.1. The rule is a generalization of the filtering condition of Wu & Manber and states that if we choose j non-overlapping substrings from the pattern, then an approximate match must contain at least one of them with at most $\lfloor k/j \rfloor$ errors. The original paper [2] considered on-line searching, but in [45] and [50] this filtering condition has also been used in indexed off-line searching. Both of these papers were based on running a bit-parallel approximate string matching algorithm [2] on a trie data structure composed from the text (a suffix tree or a suffix array in the former, and a trie of a chosen set of substrings of the text in the latter). This filtering condition is also a generalization of the basic filtering condition used by Myers [35], and in [46] Baeza-Yates & Navarro presented a slight generalization of the complete recursive splitting/checking method of Myers. They called it *hierarchical verification*. It works essentially in the same way as the method of Myers, the main difference being splitting the pattern and its pieces into an arbitrary number of substrings.

9.3 Two-Phase Filtering

The sought advantage of off-line searching over on-line searching is usually derived from being able to query the index to find the interesting locations in the text that would otherwise have to be located with often costly sequential search. Due to size inhibitions, the index is usually built in such a way that it can be used in locating certain substrings, but it cannot contain too much information about the surrounding area of the substring in the text. If the used filtering condition relies on finding multiple and/or short substrings in appropriate locations with respect to each other, the index may thus leave a relatively large amount of text that still has to be inspected.

Using a filtering scheme that consists of two phases may be helpful in reducing this problem. With the set of *initial hits* we refer to the set of locations in the text that we get from straightforward queries into the index. The idea is simply to always use such filtering criteria with relation to the index that the set of initial hits is as small as possible. After this, we may impose a second set of filtering criteria on the set of initial hits. This can be done by using a simple observation that if the pattern is edited with at most k edit operations, then also each subsequence of the pattern is edited with at most k edit operations.

Based on the preceding discussion, we present the following high-level description of using what we call *two-phase filtering.*

1. Choose an indexed filtering method based on Lemma 9.1 in such a way that $g = 1$ and the expected size of the set of the initial hits is as small as practically achievable.

2. When an initial hit is located in the text, determine the boundaries of the corresponding surviving region.

3. Ignore from the surviving region that/those substring(s) that was/were located by the index query of step 1. Also ignore the corresponding substring(s) from the pattern.

4. Treat the resulting subsequences from step 3 as substrings and test a second filtering condition between them with k errors. The current surviving region is checked only if the second filtering condition holds.

For example the filtering condition of Wu & Manber could be applied twice as shown in Fig. 31.

In some cases the following Lemma 9.4 may improve the performance of the initial filtering criteria.

Lemma 9.4 *Let S be a set of nonoverlapping substrings of the pattern and suppose that each $s \in S$ has been altered by at least d_s edit operations. If the pattern has been edited with at most k edit operations, then there exists a subset S_g of S for which the following conditions hold:*

1) $|S_g| \geq g' = |S| - (k - |S|d_s)$.
2) Each string $s \in S_g$ has been altered by exactly d_s edit operations.

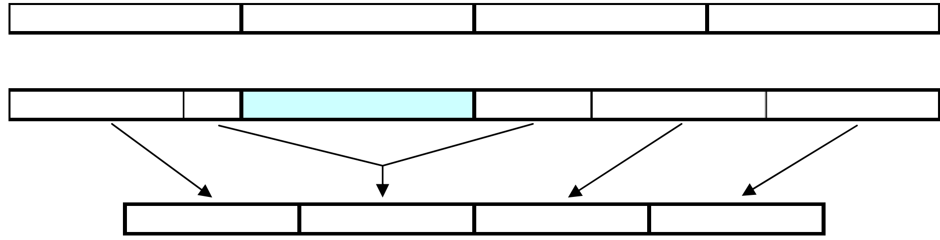


Figure 31: Applying the filtering condition of Wu and Manber twice. The first row shows the pattern partitioned into four pieces corresponding to a situation where $k = 3$. If we find the shaded piece on the second row in the text, we know that the rest of the pattern must occur close to the found piece with three errors. We can partition that part of the pattern again into four pieces, and check if one of those is present in the text. If the initially found substring breaks one of these secondary pieces, we check the occurrence of that secondary piece by trying to extend the initial match to cover also that piece without errors. Forming the secondary partition is shown on the second and the third rows.

Proof: Since the strings in the set S do not overlap, a single edit operation can alter at most one string in S. Thus we need at least $|S|d_s$ operations to alter each substring in the set S with exactly d_s edit operations. This leaves $k - |S|d_s$ edit operations, which means that at most $k - |S|d_s$ string can be altered by more than d_s operations. So there must be at least $g' = |S| - (k - |S|d_s)$ substrings that are altered by exactly d_s edit operations. □

An example of applying Lemma 9.4 will be presented in Section 9.5.

9.4 An application in bioinformatics: Searching for unique oligonucleotides

The notion *oligonucleotide* refers to a DNA-sequence of moderate length. Thus in terms of string matching, an oligonucleotide is simply a string from the alphabet $\Sigma = \{A, T, C, G\}$. There are several applications in bioinformatics that may benefit from using gene specific oligonucleotides. One example of such an application is the use of microarrays in analyzing gene expression. This procedure aims to determine whether or not a given sample contains certain genes by letting a large number of oligonucleotides, which are known to be present in the genes of interest, to interact with the sample. The presence of a particular gene is then inferred from the reactions

of the oligonucleotides corresponding to the gene. Thus it seems reasonable that the oligonucleotides representing a certain gene should be unique to it: no other genes should contain similar elements, since otherwise the results might be at least partially inconclusive.

Our approach to handle this problem is to use string matching in searching for appropriate unique oligonucleotides from a gene. We have discussed this earlier in [18, 26], which are two short practically oriented reports on searching for unique oligonucleotides from fourteen different small genomes.

Based on practical experience from molecular biology ([32, 57]), we chose 25 to be the length of the searched oligonucleotides. In order to permit some variation inherent in DNA, a minimum Levenshtein edit distance of five was decided to be the threshold that makes two oligonucleotides different enough. In addition there are also other desirable properties to take into account, but we do not consider them here since they have no real effect on the string matching problem. From the preceding we get the following formulation for the problem of finding unique oligonucleotides: Given a certain gene, we want to find from it all such oligonucleotides (substrings) of length 25, that no other gene in the genome contains an approximate match for any of them with the error threshold $k = 4$.

Since there are no really effective methods for conducting approximate search for more than a few strings at the same time, our current angle of attack for solving the task of finding unique oligonucleotides is to individually check the uniqueness of each oligonucleotide from a gene against all the other genes. Due to the static nature of the data in question, we can use indexing to accelerate the process. The eventual target genome for searching for unique oligonucleotides is the human genome. But due to its huge size of around 3×10^9 nucleotides, we used the genome of *Saccharomyces cerevisiae* (baker's yeast, length roughly 10^7 nucleotides) as a test genome for developing a suitable method.

9.5 Test results with the genome of *Saccharomyces cerevisiae*

In this section we present some practical results regarding the different filtering schemes presented in Sections 9.2 and 9.3.

All tests have been made in conjunction with searching for unique oligonucleotides, and are thus limited to searching DNA with certain values for the length of the pattern m and error level k.

The following implementations of filtering methods were tested with the genome of *Saccharomyces cerevisiae*. Due to its simplicity, we chose to

use the bit-parallel approximate string matching algorithm of Myers [38] in checking the surviving regions. Thus all implementations used it unless explicitly stated otherwise. We also briefly tested the algorithm of Baeza-Yates & Navarro [2] and saw no significant difference in the run times.

Indexed Wu & Manber (WM). Our implementation was quite similar to the method presented in [43]. The patterns (oligonucleotides) were partitioned equally into $k + 1 = 5$ parts of length 5. Also a simple 5-gram index was built, which contained for each possible 5-gram all their locations in the genome. Initially this was implemented by using lists, but later the use of arrays instead proved to improve the overall searching times by as much as roughly 25%. We treated the 5-grams as base-4 numbers and used the numerical values in querying the index. If $T_{j-q+1..j}$ was an occurrence of a pattern piece, the text area $T_{j-m-k+1..j+m+k-q}$ was checked.

Indexed Wu & Manber with location sensitivity (WMLS). The only difference to the preceding implementation was the use of location sensitivity in deciding the boundaries of the surviving regions. From Lemma 9.2 we know that the distance between the end of the pattern and the end of any substring of it can change by at most k when the pattern is edited with at most k edit operations. This meant that if $T_{j-q+1..j}$ was an occurrence of the substring $P_{i-q+1..i}$, then the area $T_{j-i-k..j+m-1-i+k}$ was checked.

Indexed Wu & Manber with location sensitivity and a second phase of using Holsti & Sutinen (WMLS-HS). A second phase of using the filtering criteria of Holsti & Sutinen with the setting $q = 3$ was used in conjunction with **WMLS**. We denote the concatenation of strings $T_{a..b}$ and $T_{c..d}$ by $T_{a..b} \circ T_{c..d}$. When an occurrence $T_{j-q+1..j}$ of the pattern piece $P_{i-q+1..i}$ was located, the substring $T_{j-i-k..j-q} \circ T_{j+1..j+m-1-i+k}$ was searched for 3-grams of the pattern. If the found piece was from either end of the pattern, we did not include the part of the subsequence that would correspond to edit operations outside the pattern. For example if $i - q + 1 = 1$, we consider only the substring $T_{j+1..j+m-1-i+k}$. The counter cumulation method presented in [15] was used in testing whether the found 3-grams are in such locations with respect to each other that an approximate match is possible.

Indexed Wu & Manber with location sensitivity and a second phase of using Jokinen, Tarhio & Ukkonen (WMLS-JTU). Now

a second phase of using the counting filter of Jokinen, Tarhio & Ukkonen was added into **WMLS**. If $T_{j-q+1..j}$ was an occurrence of the pattern piece $P_{i-q+1..i}$, we checked whether the substring $T_{j-i-k..j-} \circ T_{j+1..j+m-1-i+k}$ contains a substring of length $m-q$ that has at least $m-q-k$ characters from the corresponding pattern substring $P_{1..i-q} \circ P_{i+1..m}$.

Indexed Wu & Manber with location sensitivity and a second phase of using Wu & Manber with location sensitivity (WMLS-WMLS). This method was implemented in similar fashion to the Fig. 31. If an occurrence $T_{j-q+1..j}$ of the pattern piece $P_{i-q+1..i}$ was found, we checked whether the substring $T_{j-i-k..j-q} \circ T_{j+1..j+m-1-i+k}$ contained at least one piece from the partitioned substring $P_{1..i-q} \circ P_{i+1..m}$ so that the location of the substring has moved by at most k positions.

Generating 1-neighborhoods (G1N). In this method we chose to look for at least one non-overlapping substring of the pattern that contains at most 1 error. This corresponds to the setting $o = 1, d_s = 1$ and $g = 1$ in Lemma 9.1, which results in $|S| = 1 + \lfloor k/2 \rfloor = 3$. Thus we divided the pattern into three non-overlapping substrings knowing that each approximate match must contain at least one of them with at most one error. The pattern was partitioned evenly so that the substrings had equal lengths $q = 8$. We used a similar approach to the first part of the method of Myers [35] to locate the pattern pieces with at most one error. For all pattern pieces $P_{p_i-q+1..p_i}$ a set $M(1,q)$ of q-grams was generated so that any string from the condensed 1-neighborhood of $P_{p_i-q+1..p_i}$ must either contain or be contained in some string of $M(1,q)$. To be more accurate, $M(1,q)$ consisted of the pieces $P_{p_i-q+1..p_i}$ plus the set of all $q-$grams that can be derived from them by one of the following three choices:

-Deleting a character P_p, where $p \in [p_i - q + 1..p_i - 1]$, and appending any character of the alphabet to the end.

-Inserting a character from the alphabet between the characters P_p and P_{p+1}, where $p \in [p_i - q + 1..p_i - 2]$, and removing the last character.

-Replacing a character P_p, where $p \in [p_i - q + 1..p_i]$, with any character of the alphabet that is different from P_p.

The reason for limiting the deletion or the insertion operation to only a part of the substring is that the ignored cases are always covered by some replacement operation. It should be noticed that in order for appending

the substring in the case of a deletion to work correctly, the text must be appended with one extra character.

We used a simple 8-gram index and base-4 numerical representations of the 8-grams in querying the index. The boundaries of the checked areas were derived from the first condition of Lemma 9.2 in a similar way to what we discussed earlier. This time we just had to take into account the fact that containing one error can move the end of the pattern substring by one position in either direction. Therefore if the substring $T_{j-q+1..j}$ corresponded to a pattern substring $P_{i-q+1..i}$ (was either equal to it or generated from it), we checked the text area $T_{j-i-k-1..j+m-1-i+k+1}$.

This method of generating 1-neighborhoods could be described as a crossing between a special case of the pattern partition of Baeza-Yates & Navarro [2] and the indexing method of Myers [35]. The difference to the former is the use of neighborhood generation, and to the latter the details of neighborhood generation and the way the surviving regions are checked.

Generating 1-neighborhoods with a second phase of using Wu & Manber with location sensitivity (G1N-WMLS). Adding this method was implemented in the same way as when using Wu & Manber with location sensitivity as the first step. The only difference was the value of q.

Generating 1-neighborhoods with a second phase of using Jokinen, Tarhio & Ukkonen (G1N-JTU). Also adding this method was implemented in the same way as when using Wu and Manber with location sensitivity as the first step. The only difference was the value of q.

Generating 1-neighborhoods with a second phase of applying Lemma 9.4 (G1N-L4). If we apply Lemma 9.4 in the case of generating 1-neighborhoods we see that if all pattern pieces are changed by at least one edit operation, then at least two pattern pieces are changed by exactly one edit operation. This was used in such a way that if the substring $T_{j-q+1..j}$ was generated from a pattern substring $P_{i-q+1..i}$ (and was not equal to it), we checked the text area $T_{j-i-k-1..j+m-1-i+k+1}$ only if it contained a non-overlapping occurrence of at least one other string from the set $M(1, q)$. During the generation of the set $M(1, q)$ we recorded the base-4 numerical representations of the generated strings, and so the occurrence of any of these strings could be checked by incrementally computing the base-4 values of the q-grams of the surviving region during a single linear scan.

Generating 2-neighborhoods (G2N). Now we chose to look for at least one nonoverlapping substring of the pattern that contains at most 2 errors. This corresponds to the setting $o = 1, d_s = 2$ and $g = 1$ in Lemma 9.1, which results in $|S| = 1 + \lfloor k/3 \rfloor = 2$. The pattern was divided evenly, which resulted in two non-overlapping substrings with length $q = 12$. Now we generated a set $M(2, q)$ in a similar way to the generation of $M(1, q)$, but this time two alterations were made. In this case the text should be appended with two characters. The other parts of the method were almost identical with the method of generating 1-neighborhoods. We for example used once again the base-4 representations for the q-grams. Now we had to take into account the possibility of the end of the pattern substring moving two positions in either direction. When the substring $T_{j-q+1..j}$ corresponded to a pattern substring $P_{i-q+1..i}$ (was either equal to it or generated from it), we checked the text area $T_{j-i-k-2..j+m-1-i+k+2}$.

Navarro & Baeza-Yates (NBY). The implementation of this method was from the original authors. The method is given a parameter j, which specifies how many substrings the pattern is partitioned into. Based on the value of j as well as the other parameters m and k of the situation, the method then uses a suitable configuration of the bit-parallel approximate string matching automaton [2] in performing a depth-first search over a suffix array [34] of the text in order to locate the pattern pieces with at most $\lfloor k/j \rfloor$ errors. Once these are found, the method locates the occurrences from the text with the help of occurrence lists that have been added into the tree during preprocessing, and finally checks the surviving regions implied by them with the bit-parallel automaton. We tested the method with several different values for j. It should be noted that the tested implementation was not optimized for the case of searching DNA. This may have some effect on its results.

The tests

The tests were conducted with a set of 100 oligonucleotides of length 25. These oligonucleotides were chosen from random locations of random genes (all tests used exactly the same test set), and in the test runs each oligonucleotide was checked against gene other than its original one by searching for approximate matches of the oligonucleotide. Since we wanted to get a worst case estimate on the run time, we did not interrupt the checking phase of an oligonucleotide if it was found to be nonunique. The computer used in all tests was a 600 MHz Pentium III with 1024 MB RAM running on Windows

NT 4.0. The programming language of implementation was C++, and we used Microsoft Visual C++ 5.0 with optimization turned on in compiling the code. The results of the tests with 100 oligonucleotides are shown in Fig. 32.

Because of the slow run time we got with the combination of the methods of Wu & Manber and Holsti & Sutinen, we did not even test combining the latter with our 1-neighbourhood generation method. The special case involving the use of the 1-neighbourhood method in two phases proved to be the fastest choice. In addition it was also seen that using the methods of Wu & Manber and Jokinen, Tarhio & Ukkonen in the second filtering stage resulted in improved run times compared to the use of only one filtering phase.

Method	**Run time (s)**	**Total verified area (millions of chars)**
WM	25	394
WMLS	14.9	223
G1N	8.6	132
G2N	4.2	29
WMLSWMLS	15.1	141
WMLSHS	22.3	156
WMLSJTU	10.2	105
G1NWMLS	7.6	75
G1NJTU	8.6	106
G1NL4	3.5	11
NBY ($j = 1$)	61.5	NA
NBY ($j = 2$)	8.0	NA
NBY ($j = 3$)	29.3	NA
NBY ($j = 4$)	328.6	NA
NBY ($j = 5$)	82.4	NA

Figure 32: The test results of checking 100 randomly chosen oligonucleotides of length 25 in the genome of *Saccharomyces cerevisiae*.

10 A Practical Index for Genome Searching

In this section we propose an approximate string matching index for searching for patterns of moderate length in DNA. The index uses the Levenshtein edit distance and is basically an improved version of the index of Myers [35]. Large parts of this section appear in the conference paper [25].

10.1 The index of Myers

In the index of Myers [35] all the text q-grams, where $q = \lceil \log_\sigma n \rceil$, are stored together with their text positions. Then the pattern is recursively split (first the whole pattern into two or three pieces, then each piece into two or three smaller pieces, and so on), until a suitable partition is achieved (Fig. 33). During each split, the number of errors permitted in the pieces is also divided. If the pattern is split into j disjoint pieces, then an occurrence of it will contain at least one of them with at most $\lfloor k/j \rfloor$ errors, and so through recursive use of this rule the number of errors is divided by j (always rounding down) during a j-way split. The lengths of the pieces in the final partitioning depend on the pattern length m. If it is a multiple of q, the partitioning of Myers will set the pieces to have length q, and if $m \geq q^2$, the pattern will be partitioned into pieces of length q and $q+1$. Myers does not describe in detail how the partitioning is done when neither of the preceding two conditions hold, only that the pieces will be of length $q \pm c$ for some c "in a fashion that gives the best performance possible".

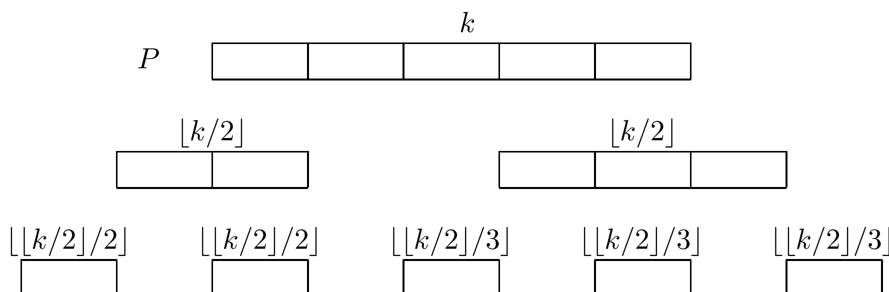


Figure 33: The pattern is recursively split into smaller and smaller pieces, also dividing the number of errors. Above each pattern piece we show with how many errors that piece is to be searched for.

Let d_i be the number of errors that are left for a given piece P^i in the partitioning. The occurrences of P^i in the text with at most d_i errors are

located by generating its *condensed d_i-neighborhood* $UC_{d_i}(P^i)$ (Section 9.2, page 63). Any occurrence of P^i in the text with at most d_i errors must have a prefix in $UC_{d_i}(P^i)$. All text occurrences of these generated substrings are located fast by using the q-gram index. These occurrences are then extended by going up the splitting hierarchy in stepwise manner. Each step consists of merging pieces back together and checking, with dynamic programming, whether the merged piece occurs in the text with as many errors as corresponded to it during the splitting phase. This recursive merge/check-process is continued until either some piece cannot be extended any further, or the merging results in finding an occurrence of the whole pattern. The condensed d-neighborhood generation algorithm presented by Myers runs in time $O(dz + q)$, where z is the number of strings in UC_d.

10.2 Index Structure

Our q-gram index is almost identical to that of Myers. The q-grams are ordered according to their base-σ numerical representation. For DNA, we use the coding a = 0, t = 1, c = 2 and g = 3, and each q-gram is treated as a base-4 number. For example the 4-gram "agct" has the base-4 representation 0321_4. The q-gram index has two tables: the header table and the occurrence location table. The header table H_q contains for each q-gram the start position of the interval in the location table, which holds in ascending order all the locations of the q-gram in the text. The location table L_q holds the intervals of location values one after the other in the order of growing numerical representation. For example if $q = 4$, the first interval holds the text positions in which the q-gram "aaaa" $= 0000_4$ occurs, the second interval the positions for the q-gram "aaat" $= 0001_4$, and so on until the last one, which is "gggg" $= 3333_4$. With this representation, the occurrences of the q-gram with numerical value x are located in the location table interval $L_q[H_q[x]..H_q[x + 1] - 1]$, except for the last q-gram. But we handle this in similar fashion by setting one extra value $H_q[\sigma^q + 1]$ to hold the location which is right after the last value of the last interval $H_q[\sigma^q]$.

The length of the q-grams in the index affects the length-range of the text substrings that can be retrieved most efficiently with the index. Having a "too large" q is not a problem in practice as long as the implementation of the table H_q, which has a size $O(\sigma^q)$, is not affected too much. This is because, as mentioned for example by Myers [35], a q-gram index can be used without much change also in finding shorter substrings. Let c be some positive integer, which is smaller than q. The locations of the $(q - c)$-gram with numerical representation x correspond to the interval $L_q[H_q[x\sigma^c]..H_q[(x +$

$1)\sigma^c] - 1]$, that is, to all q-grams that have the given $(q-c)$-gram as a prefix. This does not work with the $(q-c)$-grams that occur in the text after the character T_{n-q+1}, unless extra care is taken. One solution is to pad the end of the text with $q-1$ additional characters, and another option would be to check the last $q-1$ characters of the text separately during search time. Finding the locations of a $(q+c)$-gram A requires more work. We can for example take $A_{1..q}$, that is, the first q characters of A, and check one by one which locations of $A_{1..q}$ in the text are also locations of the whole A. As shown by Myers [35], this results in an $O(ch)$ additional overhead, where h is the number of locations of A in the text.

Using the setting $q = \lceil \log_\sigma n \rceil$ of Myers would result in the value $q = 16$ when indexing the human genome. In this case there would be $4^{16} \approx 4.3 \times 10^9$ different q-grams, which would result in a huge header table. Even though we permit secondary memory to be used in storing the index, we prefer to be able to keep the header table in primary memory (see Section 10.6). In our practical implementation we have currently opted to use $q = 12$, which results in a header table with $4^{12} \approx 16.8 \times 10^6$ entries. It is straightforward to build the above described q-gram index in $O(n + \sigma^q)$ time and space. With the 3 billion character human genome and our choice of $q = 12$, the size of the header table is 67 MB and the size of the location table is roughly 12 GB. These come from using 32-bit integers (= 4 bytes) for all entries.

In fact, it is possible to replace the location table by a suffix array, and still use the index header to accelerate the search for q-gram intervals. Pieces of length $q + c$ could then be searched by restricting the interval of their length-q prefix (obtained with H_q) with a binary search in secondary memory for the remaining c characters. Hence the $O(ch)$ cost of Myers would become $O(c \log n)$, although this time we refer to disk accesses. A drawback of this scheme is that text positions come totally unordered from the index, which increases the cost of merging occurrences and also worsens the prospects of compressing the index. This is the same effect of having a large q, in any case.

10.3 Building the Hierarchy in Bottom-up Order

Previous intermediate partitioning methods have used the rule that when the pattern P is partitioned into j disjoint pieces $P^1, .., P^j$, then each piece P^i is searched with $d_i = \lfloor k/j \rfloor$ errors. However, in [47] a more accurate rule was proposed. If a string A contains no pattern piece P^i with d_i errors, then $ed(A, P) \geq \sum_{i=1}^{j}(d_i + 1) = \sum_{i=1}^{j} d_i + j$ as each piece P^i needs at least $d_i + 1$ errors to match. We must have $\sum_{i=1}^{j} d_i + j \geq k + 1$ to ensure that

no approximate occurrence of P is missed, which can be rephrased as the condition $\sum_{i=1}^{j} d_i \geq (k+1) - j$ for the error values $d_1, .., d_j$ of the pieces. Naturally the best choice is to search for each piece with the least possible number of errors, and thus we use the strict requirement $\sum_{i=1}^{j} d_i = k - j + 1$.

Because a q-gram index is most suitable for retrieving substrings of length $\leq q$, we partition the pattern into pieces of at most length q. In addition let q_L be the minimum length for a pattern piece in the partition. We use the setting $q_L = q/2 = 6$ as it permits us to partition any pattern with $m > q$ into parts of at least size q_L.

Let d_M denote the maximum number of differences d that we will permit when searching for a pattern piece.

We have tested two partitioning methods. A simple scheme, that is somewhat similar in nature to previous methods, is to partition the pattern into $j = \lceil k/d_M \rceil$ pieces. This is the minimum number of pieces that permits each piece P^i to have $d_i \leq d_M$. To cover the whole pattern and have as equal lengths as possible, m modulo $\lfloor m/j \rfloor$ of the pieces will have the length $\lfloor m/j \rfloor + 1$ and the rest the length $\lfloor m/j \rfloor$. Finally all pieces longer than q will be set to have the length q. To enforce the strict error limit $\sum_{i=1}^{j} d_i = k - j + 1$, we set $d_i = \lfloor k/j \rfloor$ for (k modulo j) + 1 of the pieces (giving preference to the longest pieces), and $d_i = \lfloor k/j \rfloor - 1$ for the rest.

The second, and more sophisticated, approach is to precompute and store for each r-gram x, where $r \in q_L \ldots q$, and for each $d \in 0 \ldots \min(d_M, \lceil 0.25 \times r \rceil - 1)$, the number of text occurrences of all the r-grams in the d-neighborhood of x. This value, $C_{x,d}$, will be used in determining the optimal splitting.

Let us define $M_{i,t}$ as the minimum number of text positions to verify in order to search for $P_{i...m}$ with t errors. Then the following recurrence holds

$$
\begin{aligned}
M_{i,t} &= 0, \text{ if } t < 0, \\
M_{i,t} &= \infty, \text{ if } i + q_L - 1 > m \wedge t \geq 0, \\
M_{i,t} &= \min(M_{i+1,t}, \min(C_{P_{i...i+r-1},d} + M_{i+r,t-d-1} \\
&\qquad\qquad \mid d \in 0 \ldots \min(t, d_M) \wedge r \in q_L \ldots q)), \text{ otherwise.}
\end{aligned}
$$

The following lemma establishes the correctness of the recurrence.

Lemma 10.1 *Let $M_{i,t}$ be defined as in the above recurrence. Then, for $1 \leq i \leq m$ and $t \geq 0$, $M_{i,t}$ is the minimum number of occurrences of j disjoint pieces chosen from $P_{i...m}$ in such a way, that the number of errors permitted in matching any single piece is at most d_M and the total number of errors for matching all the pieces is $t - j + 1$. The minimum of an empty set is assumed to be ∞.*

Proof: We use induction on $m-i$. If $m-i < q-1$, then we cannot choose a q-gram from $P_{i...m}$ and the theorem refers to the minimum of an empty set, which is correctly set to ∞ on the second line of the recurrence. For larger $m-i$ values there are two possible cases: either one of the chosen pieces starts at the pattern position i, or not. In the second case, the solution is the same as for $P_{i+1...m}$, so by induction the minimum is given by $M_{i+1,t}$ (the number j of pieces in both solutions is the same). In the first case we can search for a piece $P_{i...i+r-1}$ with any number of errors between zero and $\min(t, d_M)$. The third line of the recurrence accounts both for the choice of the first and second case, as for the choice of the number of errors.

Once we have decided that the pattern piece $P_{i...i+r-1}$ is searched for with d errors, the rest of the pieces have to be chosen from $P_{i+r...m}$ in such a way that the pieces are disjoint and the total number of errors is $t-d-1$. By induction the minimum sum is given by $C_{P_{i...i+r-1},d} + M_{i+r,t-d-1}$.

It only remains to clarify the reason for the $t-d-1$ errors. Say that the solution for $M_{i+r,t-d-1}$ involves partitioning into j' pieces, so that the total number of errors, by inductive hypothesis, is $(t-d-1)-j'+1$. Then the solution for $M_{i,t}$, which involves one more piece searched with d errors, uses $j = j'+1$ pieces, and the total number of errors for all the pieces is $d+((t-d-1)-j'+1) = t-j' = t-j+1$, which is the right value. The first line of the recurrence is a guard for the case where we have already used all the errors and therefore do not need to choose more pieces. □

From the definition of $M_{i,t}$ it is clear that by computing the value $M_{i,k}$ we get the optimal partitioning of the pattern in terms of the number of text occurrences to verify. Once the $C_{x,d}$ values are pre-computed, the above algorithm takes time $O(qmk^2)$, which is rather low compared to the amount of verification and disk reads/seeks we might save, as we are ensuring that we will choose the pieces that yield the lowest overall number of positions in the text to verify.

The cost of precomputing $C_{x,d}$ is also not prohibitive. What is more relevant is the amount of memory necessary to store $C_{x,d}$. Since the information for $d = 0$ has to be kept anyway (because it is the length of the list of occurrences of x, and it is known also for every $r \leq q$), the price is two more numbers for each different r-gram. A way to alleviate this is to use fewer bits than necessary and reduce the precision of the numbers stored, since even an approximation of the true values will be enough to choose an almost optimal strategy.

If the partitioning that was finally chosen contains at least four pieces, we form a hierarchy on the pattern pieces similar to that of Myers (see

Fig. 33). But as we begin by optimizing the pieces at the lowest level of the partitioning-hierarchy, we form the hierarchy in bottom-up order.

Let j_h denote the number of pieces and $P^{i,h}$ the ith piece at the hth level of the partitioning-hierarchy. Also let $d_{i,h}$ denote the number of errors associated with piece $P^{i,h}$. The first (up-most) level corresponds to the whole pattern at the root of the hierarchy and therefore $j_1 = 1$, $P^{1,1} = P$ and $d_{1,1} = k$. Assume that computing the value $M_{1,k}$ leads into an ℓth level partitioning with j_ℓ pieces $P^{1,\ell}, .., P^{j_\ell,\ell}$. In general the $(h-1)$th level partitioning is formed by pairing together two adjacent pieces from the hth level so that $P^{i,h-1} = P^{2i-1,h} \circ P^{2i,h}$. If j_h is not even, the last piece $P^{j_h,h}$ will be joined together with the last pair, in which case $P^{j_{h-1},h-1} = P^{2j_{h-1}-1,h} \circ P^{2j_{h-1},h} \circ P^{2j_{h-1}+1,h}$. Note that we will always have $j_{h-1} = \lfloor j_h/2 \rfloor$.

The number of errors for the piece $P^{i,h-1}$ is determined by enforcing the partitioning rule $\sum_{i=1}^{j} d_i = k - j + 1$ locally. Using this rule for the piece $P^{i,h-1}$ means replacing the k with $d_{i,h-1}$, the sum $\sum_{i=1}^{j} d_i$ with $d_{2i-1,h} + d_{2i,h}$ or $d_{2i-1,h} + d_{2i,h} + d_{2i+1,h}$, and j with 2 or 3. The last two choices depend on whether $P^{i,h-1}$ is involved in a two- or a three-way merger. Thus in the case of a two-way merger we have $d_{i,h-1} = d_{2i-1,h} + d_{2i,h} + 1$, and in the case of a three-way merger $d_{i,h-1} = d_{2i-1,h} + d_{2i,h} + d_{2i+1,h} + 2$.

Here we assume that the possible gaps between or next to the pieces on the ℓth level of the hierarchy are padded by stretching the pieces on both sides of the gap (or one side, if the gap is at the beginning or the end of the pattern) when the $(\ell - 1)$th level partitioning is formed. The lowest level pieces $P^{1,\ell}, .., P^{j_\ell,\ell}$ are still searched as such, and so possibly having gaps in between them, but the pieces on all upper levels will cover the whole pattern.

The partitioning process goes up from the hth level to the $(h-1)$th level until only one piece, the whole pattern P, is left.

Lemma 10.2 *The above described bottom-up method produces a correct pattern partitioning hierarchy.*

Proof: First we note by induction that the condition $\sum_{i=1}^{j_h} d_{i,h} = k - j_h + 1$ holds for any $h \in 1 \ldots \ell$.

From the definition of $M_{i,k}$ we have that $\sum_{i=1}^{j_\ell} d_{i,\ell} = k - j_\ell + 1$, so the condition holds for $h = \ell$. Now assume the condition holds for h, that is, $\sum_{i=1}^{j_h} d_{i,h} = k - j_h + 1$. When the bottom-up method merges two or three pieces together into a $(h-1)$th level piece $P^{i,h-1}$, the corresponding number of errors $d_{i,h-1}$ is the sum of the errors in the merged pieces plus the difference between the number of pieces before and after the particular

merging. In this process each piece on the hth level is merged exactly once into a piece on the $(h-1)$th level. Thus the total number of errors on the $(h-1)$th level $= \sum_{i=1}^{j_{h-1}} d_{i,h-1} =$ the total number of errors on the hth level $+ j_h - j_{h-1} = \sum_{i=1}^{j_h} d_{i,h} + j_h - j_{h-1} = k - j_h + 1 + j_h - j_{h-1} = k - j_{h-1} + 1$. This concludes the inductive proof.

The bottom-up process of forming the partitioning hierarchy will end up in a situation in which there is only one piece left, and that piece is the whole pattern. This is because the number of pieces decreases when moving to an upper level, the possible gaps in the lowest level partitioning are removed so that the pieces on the upper levels cover the whole pattern, and clearly there will always be at least one piece. The corresponding number of errors will be k, because then $\sum_{i=1}^{j_1} d_{i,1} = d_{1,1} = k - j_1 + 1 = k$. Thus the first level of the partitioning hierarchy will be correct: it looks for the whole pattern with k errors. From the way the partitioning rule is enforced locally during each merger, it is quite obvious that the second level partitioning is correct: as the condition $\sum_{i=1}^{j} d_2 = k - j_2 + 1$ holds, we know that no occurrence of the pattern P with at most k errors is missed if we inspect only such areas of the text that contain some piece $P^{i,2}$ with at most $d_{i,2}$ errors. The same argument can now be repeated for each of the pieces on the second level, on the third level, and so on until the $(\ell - 1)$th level: no occurrence of an hth level piece $P^{i,h}$ is missed if we inspect only such areas of the text that contain some piece $P^{i,h+1}$, where $P^{i,h+1}$ is part of $P^{i,h}$, with at most $d_{i,h+1}$ errors. □

10.4 Generating d-neighborhoods

Our way of generating the d-neighborhoods differs slightly from the approach of Myers. Let A be the q-gram for which we wish to find all text substrings that are within d errors from A. Instead of generating the condensed d-neighborhood $UC_d(A)$, we generate an "artificial prefix-stripped length-q" d-neighborhood $UQ_d(A)$. This is done by enumerating all different strings that result from applying d errors into the string A in all possible combinations with the following restrictions:

1) All errors are applied only within the window of the first q characters.
2) A character is only replaced with a different character.
3) No characters are inserted before or after the first or the last character.
4) The string is aligned to the left side of the length-q window. That is, upon deletion the characters to the right of the deleted character are moved one position to the left, and upon insertion the characters after

the inserted characters are moved one position to the right.

5) A character introduced either by an insertion or a replacement is not removed or replaced.

It is difficult to accurately estimate the size of the set $UQ_d(A)$. According to Ukkonen's analysis [55], we have an upper bound $O((|A|\sigma)^d)$ for both $UC_d(A)$ and $UQ_d(A)$. However, in practice we have noted that using the set $UQ_d(A)$ often results in a slightly smaller set of q-grams to locate from the text than by using the set $UC_d(A)$ of Myers. This is because if for example A = "atcg" and $d = 1$, all the strings "aatcg", "tatcg", "catcg" and "gatcg" would belong to $UC_d(A)$, but of these only "aatcg" belongs to $UQ_d(A)$. This discarding of strings that include insertions made at the front is what we refer to as "artificial prefix stripping". On the other hand, there are strings that belong to $UQ_d(A)$ but not to $UC_d(A)$. For example if B = "attaa" and $d = 2$, then "ataaa" belongs to $UQ_d(A)$ but not to $UC_d(A)$, as its prefix "ataa" is in $UC_d(A)$. But when using the q-gram index, also the method of Myers will fetch all q-grams with the prefix "ataaa" provided that $q \geq 5$. A straightforward and crude analysis gives time complexity $O((3q\sigma)^d)$ for generating the set $UQ_d(A)$.

10.5 Faster Verification on Average

In [35] they used an $O(kn)$ dynamic programming approximate string matching algorithm in the stepwise merging/checking process, where it is checked whether a pattern piece occurrence can be extended to an occurrence of the whole pattern. In addition, they group pattern piece occurrence locations that were close to each other into a single interval. The idea was to process the whole interval in a single pass, and thus avoid checking the same text positions multiple times. In [45] they used a faster bit-parallel algorithm, but their approach was more crude: as any approximate occurrence of the pattern is at most of length $m + k$, they checked the text between the positions $j - m - k..j + m + k$ with an approximate string matching algorithm whenever a pattern piece occurrence ends at position j of the text. Also they merged checking of adjacent piece occurrences into a single interval.

Our approach is to check each piece occurrence separately on the bottom-level of the verification hierarchy. This is done by using an algorithm for computing edit distance instead of one for approximate string matching. We have found this verification method, a slight modification of the one used in *nrgrep* [42], to be considerably faster than the previous ones (see Section 10.7). On the upper levels we use interval merging and an approximate

string matching algorithm. This is a safeguard against a possible (although unlikely) event that a significant amount of verifications is done in areas where a complete match is found: In this case checking each piece separately could lead into quite a large overhead, as a single match could be checked as many times as it contains a match to a piece P^i with at most d_i errors.

The bottom-level verification works as follows. Let $P^i = P_{i..i+b}$ be a pattern piece, string A belong to the set $UQ_d(P^i)$, and A occur in the text starting from character T_j. Also let the substring $P^f = P_{i-u..i+v}$ be the "parent" of P^i in the hierarchy of the pattern pieces, that is, the piece P^i was one of the parts that the string P^f was split into. Note that if the number of pieces $j < 4$, then $P^f = P$. Initially we set $d = d_f + 1$, where d_f is the number of errors for P^f in the hierarchy. The value d can be interpreted to be the number of errors in the best match for P^f found so far. If P^i is not the rightmost piece in P^f, the edit distance $ed(T_{j..j+a}, P_{i..i+v})$ is computed for $a = 0, 1, 2, \ldots$ until either all distances of form $ed(T_{j..j+a}, P_{i..i+c})$, where $c \in [1..v]$, have a value $\geq d$, or we arrive at a value $ed(T_{j..j+a}, P_{i..i+v}) = 0$. Whenever a value $ed(T_{j..j+a}, P_{i..i+v}) = d - 1$ is found, we set $d = d - 1$. This forward edit distance computation will process at most $v + d_f + 2$ characters, as after that the first stopping condition must be true. If $d = d_f + 1$ after stopping, we know that the current setting does not correspond to an occurrence of the pattern. If $d \leq d_f$, we start computing the edit distance $ed(T_{j-a..j-1}, P_{i-u..i-1})$ for $a = 1, 2, \ldots$ in similar fashion as above, but this time setting initially $d = d_f - d + 1$ and without updating the value after that. If this backward computation finds a value $ed(T_{j-a..j-1}, P_{i-u..i-1}) < d$, we have found an occurrence of the substring P^f with at most d_f errors. If this does not happen, the current piece occurrence can be abandoned. Otherwise the position is recorded. In this case if $P^f = P$, a pattern occurrence was found (this type of occurrences will have to be sorted as they are not found in order and there may be multiple occurrences for the same position), and otherwise the position will be verified later on the upper level of the verification hierarchy using interval merging.

In the bi-directional verification phase we use the diagonal tiling edit distance computation algorithm [22] (Section 7), and the merged intervals are checked using the original BPM [38].

10.6 Secondary Memory Issues

In this section we discuss how to handle the situation in which the whole index cannot be stored in primary memory. In terms of secondary memory,

perhaps the biggest disadvantage compared to primary memory is slow seek time. That is, even though the secondary storage devices are in principle capable of reading data at rates of several megabytes per second, the situation worsens significantly if the data is not read from a moderate number of continuous locations. In terms of accessing the q-gram index from secondary memory, it is obvious that the queries will typically occur from more or less scattered positions. When d-neighborhood generation is used, the number of q-grams to be queried from the index grows almost exponentially with d. As each different q-gram means potentially a seek operation in secondary memory, we wish to limit the number of q-grams. This is done by setting an appropriate value, which is based on practical experience, for the limit d_M of generating the d-neighborhoods. In addition we choose to store the header table H_q in primary memory. Otherwise querying a single q-gram would involve two secondary memory seeks instead of one. This is feasible since the size of the table H_q is not prohibitively large for primary memory if a moderate value for q is used.

If we use the estimate $|UQ_d(A)| \approx (|A|\sigma)^d$ with the value $q = |A| = 12$, the size of the set $UQ_d(A)$ is ≈ 48 for $d = 1$, ≈ 2304 for $d = 2$, and ≈ 110592 for $d = 3$. If the average seek time of secondary memory were 1 ms, the seek time for the case $d = 3$ could be almost 2 minutes! We tested that with $q = 12$ and $d = 2$ the time for reading the d-neighborhood occurrences from disk was already around 17 seconds on our computer. Based on this, we have currently chosen to use the limit $d_M = 1$ in generating the d-neighborhoods.

The effect of the slow performance of secondary memory can also be taken into account in choosing the partitioning. The way we do this is to weight the value $C_{x,d}$ of the occurrence table (Section 10.3) with an estimated cost for querying the q-gram index with all the strings in $UQ_d(A)$. Specifically, if $C^w_{x,d}$ is the weighted cost for the substring x and d errors, we use the formula

$$\begin{aligned} C^w_{x,d} &= C_{x,d} \times (\textit{verification-cost} + \textit{disk-transfer-cost}) \\ &+ \textit{d-neighborhood-size}(x,d) \times \textit{disk-seek-cost} \end{aligned}$$

normalized to the form $C_{x,d} + c \times \textit{d-neighborhood-size}(x, d)$. The appropriate weight value c depends on the actual type of memory used in storing the index, and thus it should be based on empirical tests.

10.7 Test Results

As the test results in [45] found the index of Myers to be the best method in the case of DNA, we have compared the performance between it and our

index. The code for that index was from the original author. We used the $\approx$ 10 MB genome of *Saccharomyces cerevisiae* (baker's yeast) from [51] as the text, as it allows the index to be completely in memory. These tests were run on a 600 MHz Pentium 3 with 256 MB RAM and Linux OS, and all code was compiled with GCC 3.2.1 using full optimization. The implementation by Myers is only a limited prototype that is constrained to the pattern lengths of form $q \times 2^x$, where x is a non-negative integer and $q = \lceil \log_\sigma n \rceil$. In the case of *S. cerevisiae* this means that we may use pattern lengths 12, 24, 48, and so on.

The left side of Fig. 34 shows the results we got with searching for a pattern with lengths 24, 96, and 384 and a sampling of k-values having the error level $k/m < 0.25$. The test included three different variants of our index. All three used the simple partitioning scheme described in Section 10.3 with value $d_M = 2$. The first method used bidirectional verification and the second conventional interval-merging combined with approximate string matching. Both of these used the d-neighborhood generation method of Section 10.4. The third method used bidirectional verification, but combined with a d-neighborhood generation method closer to that of Myers (backtracking with edit distance computation over the trie of all strings). The first method was a clear winner, being 2 to 12 (and typically around 4 or more) times faster than the index of Myers. In many cases a large part of our advantage is explained by the strict rule $\sum_{i=1}^{j} d_i = k - j + 1$. This shows most evidently in the figures when k moves above $m/6$: at this point the index of Myers sets all pieces to have $d_i = 2$, whereas our index increases the number of errors in a more steady manner. The difference between the search mechanisms themselves is seen when $k = m/6 - 1$ or $k = m/4 - 1$, as at these points both indexes set $d_i = 1$ or $d_i = 2$, respectively, for all of the pieces. In these cases our fastest version is always roughly 4 times faster than the index of Myers.

We chose our fastest scheme from the tests with *S. cerevisiae* to be the core method for searching the human genome. Differences were that now the index was on disk, we used the value $d_M = 1$ and the text was encoded using 2 bits per nucleotide. In testing we used the August 8th 2001 draft of the human genome (obtained from [52]), which consists of roughly 2.85 billion nucleotides. The computer was a 1.33 GHz AMD Athlon XP with 1 GB RAM, 40 GB IBM Deskstar 60GXP hard disk and Windows 2000 OS, and the code was compiled using Microsoft Visual C++ 6.0 with full optimization. We compared the two different partitioning schemes from Section 10.3: the simple (as with *S. cerevisiae*) and the optimized one. The results are shown on the right side of Fig. 34. In the majority of cases

using optimized partitioning had a non-negative gain, which varied between 0-300%. There were also some cases where the effect was negative, but they were most probably due to the still immature calibration of our cost function. We also made a quick test to compare our disk-based index with the sequential bit-parallel approximate string matching algorithm of Myers [38]. For example in the case $m = 384$ and $k = 95$ our index was still about 6 times faster.

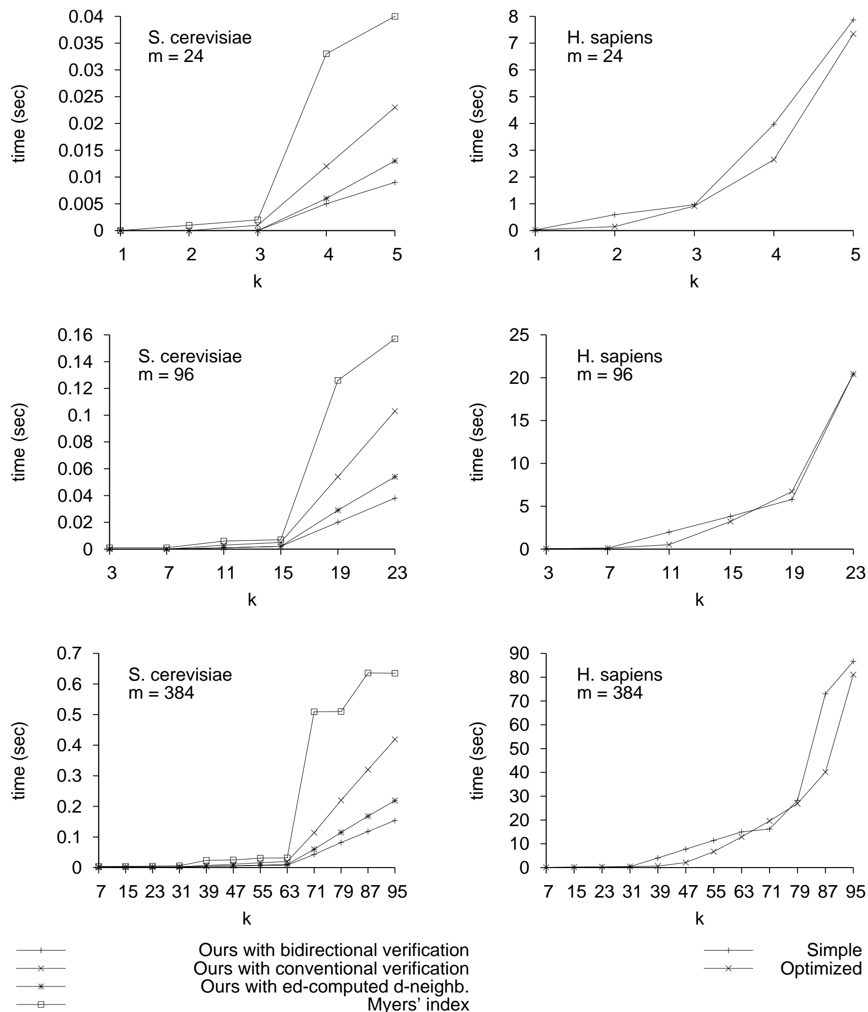


Figure 34: The figures in the left column show the comparison between the index of Myers and three variants of our index in searching the genome of *S. cerevisiae*. The figures on the right show the comparison between our two different partitioning schemes in the case of searching the human genome and using the best combination of verification/d-neighborhood generation from the tests with *S. cerevisiae*.

11 An Improvement and an Extension on the Hybrid Index of Navarro & Baeza-Yates

In this section we propose an improvement on the hybrid index of Navarro & Baeza-Yates for the Levenshtein edit distance [45]. In their tests that index was found to be the best in the case of searching English text. Our improved version is roughly 2-3 times faster than the original when the pattern is not partitioned. Then we also discuss how to extend the index for the Damerau edit distance.

11.1 The Hybrid Index

The so-called "hybrid index" of Navarro & Baeza-Yates [45] is based on *intermediate partitioning*, where the pattern is partitioned into j pieces $P^1, .., P^j$ and then each piece P^i is searched for with $d_i = \lfloor k/j \rfloor$ errors. When a hit $T_{j-h..j}$ is found so that $ed_L(P^i, T_{j-h..j}) \leq d_i$, there are two cases. If $j > 1$, the text area $T_{j-m-k..j+m+k}$ will be recorded to be checked for a complete match of P with k errors. The original implementation uses plain dynamic programming in verifying the hit surroundings. Otherwise if $j = 1$, the found hit is reported as a match for the pattern as then $P^i = P$ and $d_i = k$.

The hits for each piece P^i are found by a depth-first search (DFS) over a suffix tree[2] built for the text and filling an edit distance computing version of the dynamic programming table D during the DFS. When the DFS arrives at a node that corresponds to the text substring $T_{j+1..j+l}$, the distance $ed_L(P^i, T_{j+1..j+l})$ is computed. This is done incrementally by using column-wise filling order in filling D: moving from a depth-l node to a depth-$(l+1)$ node is similar to moving from column l to column $l+1$. We may assume that columns $0..l$ of D have been filled previously to enable computing the value $T_{j+1..j+l}$ at depth l, and so one needs to compute only column $l+1$ at depth $l+1$ to get the value $ed_L(P^i, T_{j+1..j+l+1})$. Here we assume that all columns of D are kept in memory. After column l has been computed there are the following three cases:

1. $D[m, l] \leq d_i$, that is, $ed_L(P^i, T_{j+1..j+l}) \leq d_i$ and a matching text substring has been found. The corresponding text positions are then recorded from the leaf nodes that descend from the current node. Now all matches with the current text substring $T_{j+1..j+l}$ as a prefix have been found, and so the DFS backtracks.

[2] A trie of all suffixes of the text in which each suffix has its own leaf node and the position of each suffix is recorded into the corresponding leaf.

2. $D[i,l] > d_i$ for $i = 1..m$, that is, there are no active cells in column l. Now no string with the current substring $T_{j+1..j+l}$ as a prefix can be a match, and so the DFS backtracks.

3. $D[m,l] > d_i$ but $D[i,l] \leq d_i$ for some $i \in [1, m-1]$, that is, a matching text substring is not found but there is at least one active cell. Now there may be some text substring with length $> l$ that has $T_{j+1..j+l}$ as a prefix, and so the DFS tries to continue to a level-$l+1$ child of the current node. If the current node has no children, the DFS backtracks.

When the edit distance computation backtracks from depth l to depth $l-1$, the current columns $0..l-2$ of D can still be used. When another character is tried at depth $l-1$, the previously computed values $D[i, l-1]$ are overwritten.

Navarro & Baeza-Yates used a faster bit-parallel algorithm [2] instead of dynamic programming in computing the Levenshtein edit distance during the DFS.

A suffix array [34, 13] is a list of the starting positions of all n text suffixes that are sorted in lexicographic order. It permits the most typical searching operations that can be done with a suffix tree, but there is an additional cost factor of $O(\log n)$ as the strings in a suffix array are searched by using binary searches. The hybrid index of Baeza-Yates uses a suffix array instead of a suffix tree. One major reason for this is space: a suffix array takes typically $4n$ bytes whereas a suffix tree takes $9n$ bytes or more [12]. In their tests Navarro & Baeza-Yates also found that a suffix array works faster in practice despite the additional $O(\log n)$ penalty.

It is not straightforward to select the number of pieces j in an optimal manner. Navarro & Baeza-Yates proved that the best choice is to have $j = \Theta(m/\log_\sigma n)$, which results in the run time $O(n^\kappa)$ where $\kappa < 1$ when $k/m < 1 - c/\sqrt{\sigma}$. Here c is a constant that is proven to be $\leq e$ and empirically known to be close to 1. They also proposed that using the simple rule $j = (m+k)/\log_\sigma n$ works reasonably well even though it does not always give the best choice for j.

11.2 Pruning the DFS

Based on our experience with the work in Section 10.3, an immediate way to improve the original hybrid index of Baeza-Yates & Navarro would be to use some of the smarter intermediate partitioning methods that were discussed in Section 10.3 and to use BPM in verifying the hits instead of the slow dynamic programming algorithm. But in the present work we

focus on improving the DFS search over a suffix array and leave aside the way the partitioning and verification is done. Thus we will consider only the case where $j = 1$. This work has value also for the general case as it corresponds to the initial phase where a pattern piece P^i is searched. There have been attempts to improve the performance of the DFS in [55, 6], but their practical value has been questioned in [45, 47].

In our experiments with the hybrid index we found that the DFS over the suffix array takes a considerable amount of time. By this we refer to the time that it takes to traverse the nodes: the speed of the algorithm used in computing edit distance during the DFS seemed to have very little significance to the overall time. When we modified the algorithm to compute column l twice at each level-l node, there was a very negligible change in the overall run time of the DFS (less than 1%).

Early Cut-off

Now we propose a way to process less nodes in the suffix array during the DFS. It is based on the same idea that Myers used in the d-neighborhood generation algorithm of his index [35]. Assume that the DFS is looking for matches to a pattern string P with d errors. The basic idea is that when the DFS reaches a depth-l node that corresponds to the text substring $T_{j+1..j+l}$ and where $D[i, l] \geq d$ for $i = 1..m$, the only strings that have $T_{j+1..j+l}$ as a prefix and match P with d errors are the strings of form $T_{j+1..j+l} \circ P_{r+1..m}$, where r fulfills the condition $D[r, l] = d$. This is a simple consequence of the diagonal property (page 9) and the form of the Levenshtein edit distance: once all cells (and thus diagonals) of column l have a value $\geq d$, the only[3] ways to have a value $D[m, l+h] \leq d$, where $h \geq 0$, is to have the rest of the cells along a still active diagonal to correspond to a match. And this requires exactly that the next traversed characters in the suffix tree/array form a suffix $P_{r+1..m}$ where the corresponding cell, $D[r, l]$, in the current column is active. Thus in this kind of situation we can try extending the current text substring $T_{j+1..j+l}$ with all such suffixes $P_{r+1..m}$ for which $D[r, l] = d$. A natural exception is the case where $D[m, j] = d$, as in that case the whole text substring is a match in itself and the extensions would be redundant. Fig. 35 illustrates.

The DFS searches for only the shortest prefixes that match with P, and we would like to keep that property to minimize the number of hits. This means that when we try to extend the current text substring with the

[3]Under the Levenshtein edit distance.

		t	h	e	r	e			
	0	1	2	3	4	5	6	7	8
t	1	0	1	2	3	4			
h	2	1	0	1	2	3			
e	3	2	1	0	1	**2**			
s	4	3	2	1	1	**2**	$\underline{2}$		
i	5	4	3	2	2	**2**	$\underline{2}$	$\underline{2}$	
s	6	5	4	3	3	3	$\underline{2}$	$\underline{2}$	$\underline{2}$

Figure 35: The dynamic programming matrix D for computing the Levenshtein edit distance between the pattern $P =$ "thesis" and the text substring $T_{j+1..} =$ "there..". Assume that $d = 2$. Column 5 is the first column whose each cell has a value ≥ 2. Now the only way to reach a cell value $D[m, x] = 2$, $x > 5$, is to have only matches at the remaining parts of the diagonals with the value $D[h, 4] = 2$. The cells in these diagonal extensions have the value $d = 2$ underlined, and the pattern suffixes corresponding to the cell values $D[3, 5]$, $D[4, 5]$ and $D[5, 5]$ (shown in bold) are $P_{4..6} =$ "sis", $P_{5..6} =$ "is" and $P_6 =$ "s", respectively. Thus at this point we can check directly whether the current path can be continued to correspond to $T_{j+1..j+5} \circ P_{4..6} =$ "theresis", $T_{j+1..j+5} \circ P_{5..6} =$ "thereis" and/or $T_{j+1..j+5} \circ P_6 =$ "theres", and then backtrack in the DFS.

suffixes of P, we should not use a suffix $P_{r+1..m}$ if also some shorter prefix $P_{r+1+h..m}$, $h > 0$, of it can be used. That is, if $P_{r+1+h..m}$ is a prefix of $P_{r+1..m}$, $D[r+h, l] = d$ and $D[r, l] = d$, then we should not use the longer suffix $P_{r+1..m}$ because the resulting string $T_{j+1..j+l} \circ P_{r+1..m}$ has the shorter prefix $T_{j+1..j+l} \circ P_{r+1+h..m}$ that matches with P. For example in Fig. 35 the possible suffix extension $P_{4..6}$ = "sis" has the shorter possible suffix extension P_6 = "s" as its prefix, which means that $T_{j+1..j+5} \circ P_6$ = "theres" is a prefix of $T_{j+1..j+5} \circ P_{4..6}$ = "theresis". Thus the latter is redundant and can be discarded.

As noted already by Myers, the failure-function f of the well-known exact string matching algorithm of Knuth, Morris & Pratt [28] can be used in avoiding the preceding problem. Also we use it, but in a slightly different manner than Myers. Let f_P denote the failure function for the pattern P. It is defined as follows:

$$f_P(i) = \max(h \mid (h = 0) \vee ((P_{1..h} = P_{i-h+1..i}) \wedge (0 < h < i)))$$

Thus the value $f_P(i)$ gives the length of the longest prefix of $P_{1..i-1}$ that is also a suffix of $P_{1..i}$. We will use f', a "reverse-indexed" version of f_P^R that is defined as follows:

$$f'_P(i) = f_P^R(m - i + 1)$$

As $(P_{i..m})^R = P^R_{1..m-i+1}$, the value $f'_P(i)$ gives the length of the longest suffix of $P_{i..m}$ that is also a prefix of $P_{i+1..m}$. Together with f' we use a check table $extTab$ that is initialized by setting $extTab[i] = 0$ for $i = 1..m$. When $D[i, l] \geq d$ for $i = 1..m$ and $D[m, l] > d$, we go through the suffixes $P_{h..m}$ in the order $h = d + 1..m$. Note how we do not try extending with the suffixes $P_{1..m}, .., P_{d..m}$ as they match P as such and will be handled by the DFS in other branches of the tree. When considering the suffix $P_{h..m}$, we have one of the following five cases:

1. $D[h-1, l] = d$ and $f'_P(h) = x > 0$. In this case the suffix $P_{h..m}$ is a possible extension with a shorter prefix $P_{h..h+x-1} = P_{m-x+1..m}$ that could also be a possible extension. We do not try to extend with $P_{h..m}$ yet, but record the possibility by setting $extTab[m-x+1] = h$ so that it can be handled when $h = m - x + 1$ (note that now $h < m - x + 1$).

2. $D[h-1, l] > d$, $f'_P(h) = x > 0$ and $extTab[h] = y > 0$. In this case the suffix $P_{y..m}$ is a possible extension that was recorded using Step 1 because $P_{h..h+x-1}$ is its prefix. Now $P_{h..h+x-1}$ turned out not to be a possible extension, but it has a yet shorter prefix $P_{h..h+x-1} =$

$P_{m-x+1..m}$ that could be one. Thus we record the earlier possible extension $P_{y..m}$ to be handled when $h = m - x + 1$, that is, we set $extTab[m - x + 1] = y$.

3. $D[h-1, l] = d$ and $f'_P(h) = 0$. In this case the suffix $P_{h..m}$ is a possible extension that has no prefix that could be a possible extension. Thus we can check if there is a descending path that corresponds to the extended text substring $T_{j+1..j+l} \circ P_{h..m}$.

4. $D[h-1, l] > d$, $extTab[h] = y > 0$ and $f'_P(h) = 0$. In this case the suffix $P_{h..m}$ is not a possible extension, but the previously recorded suffix $P_{y..m}$ is. Now all prefixes of $P_{y..m}$ that could be possible extensions have been checked, and none of them actually were possible extensions. Thus we can check if there is a descending path that corresponds to the extended text substring $T_{j+1..j+l} \circ P_{y..m}$.

5. $D[h-1, l] > d$ and $extTab[h] = 0$. In this case $P_{h..m}$ is neither a possible extension nor a prefix of some longer possible extension. Thus we do nothing.

One difference between our and Myers' way of suffix extension is that our version accesses the values $D[i, l]$ in sequential order. This is convenient if we use BPM in computing edit distance as it encodes the values $D[i, l]$ incrementally.

Further Reduction of Traversed Nodes

The original hybrid index included a simple heuristic to decrease the number of visited nodes and the number of redundant matches found. It is enough to find only those matches to P that begin by matching one of its first $d+1$ characters. Therefore the hybrid index prunes the DFS at the root node by entering only to those nodes that correspond to one of these characters. We take this idea further by requiring that the chosen depth-1 character P_h actually matches with the character P_h in the corresponding edit path. We do this by computing the edit distance for only the suffix $P_{h..m}$ of P and preventing the edit distance computation from matching the character P_h at depths > 1. Moreover, since assuming that the first matching character is P_h means that there must be at least $h-1$ errors before it in the corresponding match, we allow only $d_i - h + 1$ errors during the part of the DFS that enters the tree with the character P_h. If $P_x = P_h = \lambda$ with some $1 \leq x < h \leq d+1$, the preceding restrictions are determined by the smallest such value x when

we enter the tree with the character λ. This ensures that the scheme works correctly even though we enter the tree only once for each different character.

Test Results with Our DFS

In this section we present test results to compare the DFS of the original hybrid index with our version that includes both the early cut-off and the prefix restrictions on matching pattern characters. Our implementation was built on the original implementation of Navarro & Baeza-Yates, so that the basic scheme of handling the suffix array nodes was the same. We used BPM in computing edit distance and the early cut-off was implemented by keeping track of the lowest active row (see Section 8.1). As noted earlier, the overall run time of the DFS is affected little by even fairly substantial differences in the performance of the edit distance computation algorithm. Therefore the fact that the two compared methods used different edit distance algorithms should not be a significant factor in the comparison. Each (m, k)-combination was tested by searching for 100 patterns that were picked randomly from the text. The tested pattern lengths were $m = 5$, $m = 10$ and $m = 15$. We used two texts of size ≈ 10 MB. The first was a 10 MB sample from Wall Street Journal articles taken from the TREC-collection [14]. The second was the DNA genome of *S. cerevisiae* (baker's yeast) obtained from [51]. As we wanted to test the DFS, each pattern was searched with the option $j = 1$ so that no pattern partitioning was involved. Fig. 36 shows the results. It can be seen that our DFS version is considerably faster with these parameters.

11.3 Using the Damerau Edit Distance

Now we discuss how to extend the hybrid index for the Damerau edit distance. This requires us to modify both the DFS and the intermediate partitioning scheme.

DFS Under the Damerau Edit Distance

Consider again the case of looking for matches to P with d errors. A trivial change is that the edit distance computation during the DFS should use the Damerau edit distance. This alone is clearly sufficient with the original DFS. But with the early cut-off scheme there may be more possible suffix extensions. Now $D[h+1, l+1] = D[h, l] = d$ if $D[h-1, l-1] = d-1$ and $P_{h..h+1} = (T_{j+l..j+l+1})^R$, and in this case also $T_{j+1..j+l-1} \circ (P_{h..h+1})^R \circ P_{h+2..m}$ is a possible extended text substring that matches P with d errors.

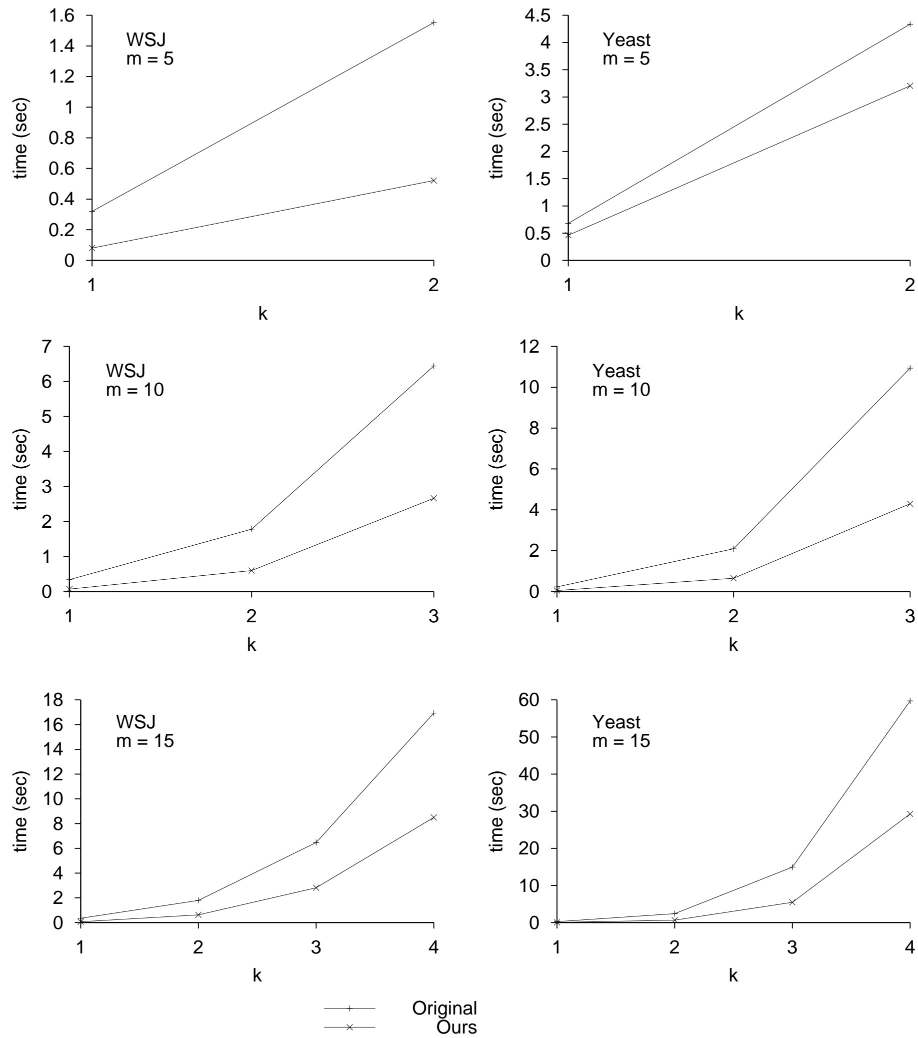


Figure 36: The left column shows the average time for searching for 100 patterns from a 10 MB sample of Wall Street Journal articles taken from TREC-collection. The right column shows the corresponding times in the case of searching the DNA genome of *S. cerevisiae* (baker's yeast).

Therefore in the case where we want to check the extended text substring $T_{j+1..j+l} \circ P_{h..m}$, with the Damerau edit distance we will at the same step also check the extended text substring $T_{j+1..j+l-1} \circ (P_{h..h+1})^R \circ P_{h+2..m}$. We do not need to change any other details in the process. For example using the function f' in avoiding redundant extensions works without any change.

Intermediate Partitioning for the Damerau Edit Distance

In [47] the condition $\sum_{i=1}^{j} d_i + j \geq k + 1$ was given for determining the number of errors d_i for each non-overlapping piece P^i under the Levenshtein distance. And as discussed in Section 10.3, it is natural to use the strict requirement $\sum_{i=1}^{j} d_i = k - j + 1$. We formulate similar rules for the Damerau edit distance. In the following we assume that we are searching for P with at most k errors and have j non-overlapping pattern substrings $P^1, .., P^j$.

With the Damerau edit distance one edit operation can cause one error into two consecutive pieces P^{i-1} and P^i by transposing the last character of P^{i-1} and the first character of P^i. It is clear that no other kind of edit operation can affect more than a single piece by more than a single error. Moreover, as there are j pieces, there can be at most $j - 1$ different boundaries between consecutive pieces P^{i-1} and P^i.

Assume that P is edited by some sequence of k edit operations under the Damerau edit distance and let n_i denote the number of edit operations in that sequence that affect exactly i pieces. Naturally $n_1 + n_2 \leq k$. Now the total number of errors in the pieces P^i, $i = 1..j$, is $n_1 + 2n_2 \leq k - n_2 + 2n_2 = k + n_2 \leq k + j - 1$. By using a similar argument as was used in [47], the previous discussion allows us to give the following lemma for determining the values d_i.

Lemma 11.1 *Let P^i, $i = 1..j$, be j non-overlapping substrings of the pattern P and B some string for which $ed_D(P, B) \leq k$. Also let each P^i be associated with the corresponding number of errors d_i. If $\sum_{i=1}^{j} d_i \geq k$, then one of the pattern substrings P^i matches inside B with at most d_i errors.*

Proof: We use contraposition. Assume that $\sum_{i=1}^{j} d_i \geq k$ and $ed_D(P, B) \leq k$ but no substring P^i matches in B with d_i errors. This means that we need at least $d_i + 1$ errors to match each P^i in B. Thus the total number of errors in the occurrences of $P^1, .., P^j$ in B is at least $\sum_{i=1}^{j}(d_i+1) = \sum_{i=1}^{j} d_i + j \geq k + j$. This is a contradiction because, as discussed above, the total number of errors in the pieces can be at most $k + j - 1$. $\square$

As noted by Navarro [42, 41], the problem that a transposition can alter two non-overlapping pieces may be avoided by requiring that the pieces are not consecutive, that is, that they have at least one character between them. Navarro used this fact with the filtering condition of Wu & Manber [60] where the pattern is partitioned into $k+1$ pieces and each is searched without errors. The following lemma is a natural extension of this for the general case of intermediate partitioning. It is almost identical to the one given for the Levenshtein edit distance in [47].

Lemma 11.2 *Let P^i, $i = 1..j$, be j non-overlapping substrings of the pattern P so that no two pieces P^x and P^y, $x, y \in [1..j]$, are consecutive. Also let B be some string for which $ed_D(P, B) \leq k$ and let each P^i be associated with the corresponding number of errors d_i. If $\sum_{i=1}^{j} d_i \geq k - j + 1$, then one of the pattern substrings P^i matches inside B with at most d_i errors.*

Proof: As now a single edit operation can alter at most one piece, the total number of errors is k. Assume that $\sum_{i=1}^{j} d_i \geq k - j + 1$ and $ed_D(P, B) \leq k$ but no substring P^i matches in B with d_i errors. This means that we need at least $d_i + 1$ errors to match each P^i in B. Thus the total number of errors in the occurrences of $P^1, .., P^j$ in B is at least $\sum_{i=1}^{j}(d_i + 1) = \sum_{i=1}^{j} d_i + j \geq k - j + 1 + j = k + 1$. This is a contradiction because the total number of errors in the pieces can be at most k. □

Having gaps between the pieces in Lemma 11.2 loses a bit of information about the pattern. To prevent this we propose the following lemma, which is slightly stronger. It uses *classes of characters*, which refers to permitting a pattern character to match with any character in a given *set* of characters. A set of possible matching characters is typically denoted by enumerating its characters inside square brackets. For example the pattern $P =$ "thes[ei]s" matches with the strings "theses" and "thesis" as its second-last character is allowed to match with an 'e' or an 'i'.

Lemma 11.3 *Let P^i, $i = 1..j$, be j non-overlapping substrings of the pattern P that are ordered so that P^{i+1} occurs on the right side of P^i in P. Also let B be some string for which $ed_D(P, B) \leq k$, let each P^i be associated with the corresponding number of errors d_i and let strings $\bar{P}^i$, $i = 1..j$, be defined as follows:*

$\bar{P}^i = P^i$, *if $i = j$ or P^i and P^{i+1} do not occur consecutively in P.*

$\bar{P}^i = P^i_{1..|P^i|-1} \circ [P^i_{|P^i|} P^{i+1}_1]$, *otherwise.*

If $\sum_{i=1}^{j} d_i \geq k - j + 1$, then one of the strings $\bar{P}^i$ matches inside B with at most d_i errors.

Proof: Let us interpret a transposition of the character pair $P_{i-1..i}$ as an edit operation that is associated with the single character P_i. With this interpretation each edit operation is associated with at most one piece P^i. From the proof of Lemma 11.2 we know that if $\sum_{i=1}^{j} d_i \geq k - j + 1$ then at least one piece P^i will be associated with at most d_i edit operations in the string B. The only case where this does not lead into P^i matching with at most d_i errors inside B is when P^i has been modified also by a transposition that is not associated with it. The only possible way for this to happen is that the character $P^i_{|P^i|}$ has been transposed with the character P^{i+1}_1, and in this case $P^i_{1..|P^i|-1} \circ P^{i+1}_1$ matches with at most d_i errors inside B. The preceding is the same as saying that $\bar{P}^i$ matches with at most d_i errors inside B. □

A Note on the Asymptotic Behaviour

In this section we assume that the pattern and the text follow the *symmetric Bernoulli model* where the probability of occurrence is $1/\sigma$ for each character. Typically all string matching algorithms are analysed under this assumption.

The counting filter (Section 9.2) for the Levenshtein edit distance states that if an approximate match of the pattern P ends at text position j, then the text substring $T_{j-m+1..j}$ contains at least $m-k$ characters of the pattern. The condition of the counting filter can be formulated into a more strict manner by saying that $lcs(T_{j-m+1..j}, P) \geq m - k$, where $lcs(A, B)$ denotes the length of the *longest common subsequence* of the strings A and B. It is straightforward to verify that this rule works also with the Damerau edit distance: $lcs(P, P) = m$, and a single edit operation can affect the length of the longest common subsequence within a window by at most 1.

Let $f_D(m, k)$ be the probability that the pattern P matches at a given text position j with at most k errors under the Damerau edit distance. From the preceding discussion it follows that $f_D(m, k)$ can be upper-bounded by assuming that P matches at text position j whenever $lcs(T_{j-m+1..j}, P) \geq m-k$. The probability of the latter happening is $\leq \sigma^k \binom{m}{k}^2 / \sigma^m = \binom{m}{k}^2 / \sigma^{m-k}$. Here σ^m is the number of all strings of length m, and $\sigma^k \binom{m}{k}^2$ counts in how many ways we can map a length-$(m - k)$ subsequence from the window of length m to some length-$(m - k)$ subsequence of the pattern and then fill the remaining k unmapped characters from the window with any characters of the alphabet. Thus we have that $f_D(m, k) \leq \binom{m}{k}^2 / \sigma^{m-k}$.

The asymptotic analysis of the original hybrid index [45] for the Levenshtein edit distance is based on an upper-bounding model that is more complicated but still inherently similar to our model. In fact, a fundamental part of their analysis is based on analyzing exactly the same term $\binom{m}{k}^2/\sigma^{m-k}$. They proved that $\binom{m}{k}^2/\sigma^{m-k} = O(\gamma^m)$, where $\gamma < 1$ if $k/m < 1 - e/\sqrt{\sigma}$. Based on this result we may conclude that $f_D(m,k) = O(\gamma^m)$, where $\gamma < 1$ if $k/m < 1 - e/\sqrt{n}$.

As in addition Lemmas 11.1 and 11.3 enable us to use intermediate partitioning where $d_i = k/j$, there is not much difference between our scenario and the model that Navarro & Baeza-Yates analysed. If we use Lemma 11.1, a difference is that we can have $d_i = \lceil k/j \rceil$ whereas the original hybrid index uses $d_i = \lfloor k/j \rfloor$. We can use $d_i = \lfloor k/j \rfloor$ under Lemma 11.3, but in that case the DFS has to permit the last characters of $P^1, .., P^{i-1}$ to match with two different characters. This does not change the asymptotic cost of the search as it is clear that this can at most double the cost of the DFS. Here we have excluded the fact that we use heuristic solutions to accelerate the DFS, but this does not affect the situation. We can hence claim that all analytic results stated in [45] hold also in the case of our Damerau edit distance version of the hybrid index. The most important of these is that the overall search cost can be made $O(n^\kappa)$, where $\kappa < 1$ if $k/m < 1 - e/\sqrt{\sigma}$ and $j = \Theta(m/\log_\sigma(n))$.

Conclusion

In this thesis we presented a variety of methods for the tasks of computing edit distance and approximate string mathing under the Levenshtein and the Damerau edit distances. For most of our methods we also showed empirical test results to verify that the methods are competitive in practice. As the obtained results were often quite satisfactory, we believe that the current work provides a worthwhile contribution.

The current work is by no means complete in that much remains to be done. We for example plan to compose a more elaborate analysis of some of the used heuristics, such as two-phase filtering (Section 9.3) and bi-directional verification (Section 10.5).

Bibliography

[1] R. Baeza-Yates and G. Gonnet. A new approach to text searching. *Comm. of the ACM*, 35(10):74–82, October 1992.

[2] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[3] R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. *Information Processing Letters*, 59:21–27, 1996.

[4] W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, LNCS 644, pages 172–181, 1992.

[5] W. Chang and E. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, Oct/Nov 1994.

[6] A. Cobbs. Fast approximate matching using suffix trees. In *Proc. CPM'95*, pages 41–54, 1995. LNCS 937.

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

[8] A. Czumaj, M. Crochemore, L. Gasieniec, S. Jarominek, Thierry Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12:247–267, 1994.

[9] F. Damerau. A technique for computer detection and correction of spelling errors. *Comm. of the ACM*, 7(3):171–176, 1964.

[10] M. W. Du and S. C. Chang. A model and a fast algorithm for multiple errors spelling correction. *Acta Informatica*, 29:281–302, 1992.

[11] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. on Computing*, 19(6):989–999, 1990.

[12] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proc. WAE'99*, LNCS 1668, pages 30–42, 1999.

[13] G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zuerich, Switzerland, 1992.

[14] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[15] N. Holsti and E. Sutinen. Approximate string matching using q-gram places. In *Proc. 7th Finnish Symposium on Computer Science*, pages 23–32. University of Joensuu, 1994.

[16] J. Holub. *Simulation of Nondeterministic Finite Automata in Pattern Matching.* PhD thesis, Dept. of Computer Science and Engineering, Czech Technical University, Prague, February 2000.

[17] H. Hyyrö. Explaining and extending the bit-parallel approximate string matching algorithm of Myers. Technical Report A-2001-10, Dept. of Computer and Information Sciences, University of Tampere, Tampere, Finland, 2001.

[18] H. Hyyrö. On applying string matching in searching unique oligonucleotides. In *Proc. METMBS 2001.* CSREA Press, 2001.

[19] H. Hyyrö. On using two-phase filtering in indexed approximate string matching with application to searching unique oligonucleotides. In *Proc. SPIRE'01*, pages 84–95. IEEE CS Press, 2001.

[20] H. Hyyrö. A bit-vector algorithm for computing Levenshtein and Damerau edit distances. In *Proc. Prague Stringology Conference '02 (PSC'02)*, pages "44–54", 2002.

[21] H. Hyyrö. Bit-parallel approximate string matching with transposition. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE'2003)*, LNCS, 2003.

[22] H. Hyyrö. A bit-vector algorithm for computing Levenshtein and Damerau edit distances. *Nordic Journal of Computing*, 10:1–11, 2003.

[23] H. Hyyrö, M. Juhola, and M. Vihinen. On exact string matching of unique oligonucleotides. *Computers in Biology and Medicine.* To appear.

[24] H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. 13th Combinatorial Pattern Matching (CPM'2002)*, LNCS 2373, pages "203–224", 2002.

[25] H. Hyyrö and G. Navarro. A practical index for genome searching. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE'2003)*, LNCS, 2003.

[26] H. Hyyrö, M. Vihinen, and M. Juhola. On approximate string matching of unique oligonucleotides. In *Proc. Medinfo 2001.* IOS Press, 2001.

[27] P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Software Practice and Experience*, 26(12):1439–1458, 1996.

[28] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. on Computing*, 6(1):323–350, 1977.

[29] G. Landau and U. Vishkin. Fast string matching with k differences. *J. of Computer and Systems Science*, 37:63–78, 1988.

[30] G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. of Algorithms*, 10:157–169, 1989.

[31] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966. Original in Russian in *Doklady Akademii Nauk SSSR, 163(4):845–848, 1965.*

[32] D. J. Lockhart, H. Dong, M. C. Byrne, M. T. Follettie, M. V. Gallo, M. S. Chee, M. Mittmann, C. Wang, M. Kobayashi, H. Horton, and E. L. Brown. Expression monitoring by hybridization to high-density oligonucleotide arrays. *Nature Biotechnology*, 14:1675–1680, 1996.

[33] R. Lowrance and R. Wagner. An extension of the string-to-string correction problem. *J. of the ACM*, 22:177–183, 1975.

[34] U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.

[35] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct/Nov 1994.

[36] G. Myers. Incremental alignment algorithms and their applications. Technical Report 86–22, Dept. of Computer Science, Univ. of Arizona, 1986.

[37] G. Myers. An $O(ND)$ difference algorithm and its variants. *Algorithmica*, 1:251–266, 1986.

[38] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.

[39] G. Navarro. Multiple approximate string matching by counting. In *Proc. WSP'97*, pages 125–139. Carleton University Press, 1997.

[40] G. Navarro. *Approximate Text Searching*. PhD thesis, Dept. of Computer Science, Univ. of Chile, December 1998. Technical Report TR/DCC-98-14. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/-thesis98.ps.gz`.

[41] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[42] G. Navarro. NR-grep: a fast and flexible pattern matching tool. *Software Practice and Experience (SPE)*, 31:1265–1312, 2001.

[43] G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998. `http://-www.clei.cl`.

[44] G. Navarro and R. Baeza-Yates. Very fast and simple approximate string matching. *Information Processing Letters*, (72):65–70, 1999.

[45] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms (JDA)*, 1(1):205–239, 2000. Special issue on Matching Patterns.

[46] G. Navarro and R. Baeza-Yates. Improving an algorithm for approximate pattern matching. *Algorithmica*, 30(4):473–502, 2001.

[47] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

[48] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5(4), 2000.

[49] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.

[50] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Proceedings of the 11st Annual Symposium on Combinatorial Pattern Matching (CPM'2000)*, LNCS 1848, pages 350–363, 2000.

[51] National center for biotechnology information. website: http://www.ncbi.nlm.nih.gov/.

[52] Ucsc human genome project working draft. website: http://genome.cse.ucsc.edu/.

[53] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[54] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

[55] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. CPM'93*, pages 228–242, 1993.

[56] R. Wagner and M. Fisher. The string to string correction problem. *J. of the ACM*, 21:168–178, 1974.

[57] L. Wodicka, H. Dong, M. Mittmann, M. Ho, and D. J. Lockhart. Genome-wide expression monitoring in saccharomyces cerevisiae. *Nature Biotechnology*, 15:1359–1367, 1997.

[58] A. Wright. Approximate string matching using within-word parallelism. *Software Practice and Experience*, 24(4):337–362, April 1994.

[59] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX Technical Conference*, pages 153–162, 1992.

[60] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.

[61] S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

[62] A. Yao. The complexity of pattern matching for a random string. *SIAM J. on Computing*, 8:368–387, 1979.