Filter Algorithms for Approximate String Matching

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich–Technischen Fakultät I
der Universität des Saarlandes

von

Stefan Burkhardt

Saarbrücken
2002

Datum des Kolloqiums:

Dekan der Naturwissenschaftlich–Technischen Fakultät:
Professor Dr. Slusallek

Gutachter:
Professor Dr. Hans-Peter Lenhof, Universität des Saarlandes, Saarbrücken
Professor Dr. Esko Ukkonen, Universität Helsinki, Finnland

Abstract

In this work we present new results and methods for approximate string matching with filter algorithms. We begin with the presentation of QUASAR, our efficient implementation of an improved version of the filter based on the q-gram lemma. The q-gram lemma provides a method based on matching substrings to quickly detect potential matches to a query in a subject or target. We improved and implemented an algorithm originally introduced in 1991. This resulted in a very efficient program for approximate string matching using an index. It was successfully applied to EST-clustering, a problem from computational biology.

The second part of our work introduces a new class of filters based on gapped q-grams. We analyze the potential of this somewhat more complicated approach for use in filters for approximate string matching with an index. We consider two important distance measures in approximate approximate string matching, the Hamming and the edit distance. For both problems we provide all the tools required to solve them using gapped q-grams. This includes threshold computation and the selection of good gapped q-grams using predictions of their speed and filtration efficiency. Furthermore we consider the potential of gapped q-grams for use in lossy filters. We support our findings with extensive experiments. Our results prove that gapped q-grams are superior to existing filter approaches with respect to speed, filtration efficiency and their potential for use in lossy filters.

Kurzzusammenfassung

In dieser Arbeit beschreiben wir neue Ergebnisse und Verfahren auf dem Gebiet der Filteralgorithmen für Ähnlichkeitssuche in Textdatenbanken. Im ersten Teil stellen wir QUASAR, die Implementierung eines verbesserten Filters basierend auf dem sogenannten q-gram Lemma, vor. Dieses Lemma basiert auf dem Vergleich von kurzen Teilwörtern und ermöglicht die effiziente Erkennung der Teile einer Textdatenbank, die einer bestimmten Anfrage ähneln. Wir haben einen 1991 vorgestellten Algorithmus der auf diesem Lemma aufbaut verbessert und implementiert. Daraus ergab sich ein sehr effizientes Programm zur Ähnlichkeitssuche in Textdatenbanken das in der Praxis für sogenanntes EST-clustering, ein Problem aus der Bioinformatik, eingesetzt wurde.

Der zweite Teil der Arbeit stellt eine neue Klasse von Filtern die q-grams mit Lücken, sogenannte 'gapped q-grams', benutzen vor. Wir untersuchen das Potential dieser komplexeren q-grams für die Nutzung in Filteralgorithmen für Index-basierte Ähnlichkeitssuche in Textdatenbanken. Dabei betrachten wir zwei wichtige Distanzmasse für Zeichenketten, die Hamming- und die Edit-Distanz. Wir beschreiben Verfahren, die alle für die erfolgreiche Anwendung notwendigen Vorraussetzungen, z.B. die Berechnung des sogenannten Threshold und die Auswahl guter q-grams, erfüllen. Ausserdem untersuchen wir die Eignung von q-grams mit Lücken für den Einsatz in nicht verlustfreien Filteralgorithmen. Wir ergänzen unsere Ergebnisse mit einer grossen Zahl von Experimenten die belegen, dass q-grams mit Lücken deutliche Vorteile in Bezug auf Geschwindigkeit, Qualität der Ergebnisse und Eignung fuer nicht verlustfreie Filter gegenüber herkömmlichen Filterverfahren bieten.

Acknowledgements

The work on this thesis began at the Max-Planck-Institute for Computer Science in the group of Prof. Dr. Kurt Mehlhorn under the supervision of Prof. Dr. Hans-Peter Lenhof. I finished it at the newly founded chair of Prof. Dr. Hans-Peter Lenhof.

I want to thank Prof. Dr. Kurt Mehlhorn and Prof. Dr. Hans-Peter Lenhof for the great work environment in their groups and the opportunity to pursue research in those areas that were and are the most interesting to me. I am grateful for the introduction into the field of computational biology and the new insights I gained in this area.

During my work on the thesis my main cooperation was with Dr. Eric Rivals during the development of QUASAR and Dr. Juha Kärkkäinen in the exploration of gapped q-grams. I very much enjoyed working with them and am thankful for their contributions to my work.

Last but not least I want to thank my family and Irene for their support, interest and understanding.

Contents

Chapter 1

Introduction

Motivation

Approximate string matching is one of the classic problems in computer science. In contrast to the simpler exact string matching problem, which consists of locating all exact matches between a query or pattern string and a target string or database, it includes recognizing all approximate matches with respect to a certain measure of similarity or distance.

There is a number of practical applications for approximate string matching. Examples include text retrieval, signal processing, optical character recognition and pattern recognition. One that has become particularly important recently is computational biology.

The advent of whole genome sequencing and increased genetic research resulted in the creation of huge genomic databases. The basic blueprint of life is a long, double stranded molecule, the DNA. Its structure can be represented by a simple, albeit very long string over a four-letter alphabet corresponding to the four different nucleic acids. The two main reasons why one wants to allow approximate instead of exact searches in genomic databases are the evolutionary mutations in the genomes and the errors introduced in the actual sequencing process. They invalidate methods that rely only on exact string matching. For example, the genomes of different but evolutionary related organisms will contain many regions where they still share a certain degree of similarity. However, in order to recognize these common genetic sequences one has to be able to cope with the mutations introduced over time.

In our case there was a particular application from computational biology that initiated our research in the field of approximate string matching. The so-called EST-clustering is a process where relations between a very large number of short DNA fragments are analyzed (EST = expressed sequence tag, piece of a gene). To achieve this, all fragments have to be compared with all other fragments. This is typically done with approximate string matching. An important part of this thesis was the development and implementation of QUASAR, an algorithm for efficient comparison of many short DNA sequences with a large EST database.

Dynamic Programming

Early methods for locating approximate matches of a pattern P in a string S were based on dynamic programming. They compute the entries of a matrix of size $|P| \times |S|$ and can be used to determine the similarity or distance of P and S in time $O(|P||S|)$. While these algorithms can detect even very distant approximate matches, they suffer from their comparatively slow runtime, especially for applications with very large databases or many queries.

Filter Algorithms

This problem led to the development of various faster methods for approximate string matching in general and for computational biology in particular. A common principle shared by many of these algorithms is the idea of filtering. Filter algorithms typically consist of two phases. Assuming a pattern or query string P and a database S, the first phase of a filter algorithm uses a simple and highly efficient filter criterion to analyze S. This filter criterion is a set of one or more conditions designed to recognize regions in S that contain potential approximate matches. Usually it will only mark small stretches of S that have a reasonable level of similarity or distance to P. In the second phase, the verification phase, a slower, conventional approximate string matching algorithm is used to analyze the potential candidates returned by the filtration phase and detect those that actually contain approximate matches. A filter criterion should be quick to evaluate, resulting in a short runtime for the filtration phase, and have a high filtration efficiency, i.e. discard as much of S as possible without loosing any true matches. A smaller number of potential matches will decrease the runtime of the verification phase.

On-line and Off-line Matching

There are two subclasses of approximate string matching. One instance – called the *on-line* approximate string matching problem – does not allow preprocessing of the database S before the actual search is conducted. The time spent for any examination of S during the search is considered part of the search time. In the other – the *off-line* version – one can examine the database S, gathering information about it and saving this information in index data structures in a preprocessing stage. The time required for preprocessing is not considered part of the runtime of a search in this model. The construction time of an index and the amount of space required for storing it are nevertheless quite important to the practical usefulness of such algorithms.

Off-line string matching is useful for databases that change slowly and where a large number of searches on the same database are necessary. Frequent changes of the database can only be handled if the preprocessing step is sufficiently fast. On-line string matching is more useful for rapidly changing databases and for cases where only a small number of searches in a given database have to be performed. In these cases the overhead for preprocessing the database becomes significant and reduces the overall efficiency of index-based approaches.

Reasons for Off-line Algorithms

The motivation for our initial work in this area came from a project at the German Cancer Research Center in Heidelberg. The goal was to analyze the set of all known Mouse ESTs in order to group them into *clusters* of similar fragments and to build consensus sequences out of these sets. The database containing all Mouse ESTs at that time featured roughly 200 000 fragments with an average length of about 400 nucleic acids resulting in a database containing 80 million characters. For the first step of the cluster project, an all vs. all comparison of the Mouse ESTs was required. This corresponded to executing 200 000 searches in the database S. An important point was that the desired level of similarity for grouping ESTs was relatively high. First attempts in using BLAST, a popular, efficient tool for approximate string matching in DNA databases, for a small subset of the searches resulted in an expected total runtime for all searches of roughly 8 days. Since it was also planned to cluster the Human EST database which at that time had a size of 200 million characters and would have required an estimated total search time of 7 weeks, plans for implementing a fast algorithm for high similarity off-line approximate string matching were made.

QUASAR

Our initial idea was to extend an approach described by Jokinen and Ukkonen. They introduced an algorithm called q-gram filtration and a simple index structure resulting in a very efficient filter algorithm. q-grams are short strings of length q, usually substrings of the query and/or the target. We implemented their proposal and improved several important parts. The implementation which we called QUASAR resulted in an executable that was able to solve the clustering about 20 times faster than BLAST. However, the nature of the q-gram lemma which is the foundation of the q-gram filtration limits this approach to approximate string matching with relatively high degrees of similarity.

Improving the classic q-gram Lemma

The next natural goal was therefore to search for filters that can benefit from efficient indices but also allow for higher sensitivity. We were able to explore the nature of filters based on exact matching of substrings and to increase their range of possible applications by enhancing their ability to cope with errors.

Gapped q-grams

To achieve these goals we developed two filters based on substrings with gaps, one for the Hamming and one for the edit or Levenshtein distance. Instead of looking for exact matches of a string of length q one can decide to only match certain letters of the string. The other characters in the gaps are basically 'don't-care' characters or wildcards, i.e. one can match them with any possible character. We call the arrangement of the gaps a shape. For example if one uses a substring of length four where one places a gap of size one at the third position this results in the following gapped q-gram:

`##-#`

where a '`#`' stands for a character that has to match and a '`-`' describes a gap or 'don't-care' character.

Developing filters that use gapped q-grams turned out to be fairly complicated. When using the Hamming distance, most problems could be overcome with reasonable effort. However, the case of the Levenshtein distance is much more complex. The two main problems were the selection of good shapes, i.e. arrangements of gaps, and the computation of the required number of matching q-grams for a region of S to be considered a potential match. In the case of the classic q-gram lemma this is simple but for gapped q-grams it requires several non-trivial algorithms. Another problem which is specific to the case of the Levenshtein distance is the question of handling insertions and deletions in potential approximate matches. We came up with a relatively simple method that does not have a very big impact on the runtime of our solution.

We implemented the algorithms based on gapped q-grams and analyzed their performance in a number of experiments. We use two well defined measures, the number of items that have to be retrieved from the index data structure and the number of potential matches returned by the filter to allow an unbiased comparison of different filter algorithms. These values allow characterizing the runtime of the filtration and verification phase. We then compared the filter approach of BLAST, the classic q-gram lemma and our two algorithms using gapped q-grams for the Hamming and the edit distance with each other. Since filter algorithms in general are only suitable for moderate to high similarity searches we only looked at these cases.

For the Hamming distance we were able to provide a wide range of improved filters. They reduced the amount of data to be processed in the filtration phase by up to three orders of magnitude and the amount of potential matches by up to six orders of magnitude when compared to the classic q-gram lemma. The differences to the BLAST filter criterion were even larger. Filters that offer a tradeoff between speeding up the filtration and the verification phase also exist. If using the edit distance, our filters based on gapped q-grams reduced the amount of data to be processed in the filtration phase and the amount of potential matches by up two oders of magnitude when compared to the classic q-gram lemma. Again we also provided filters that offer a tradeoff between speeding up the filtration and the verification phase.

In a second set of experiments we analysed the potential of filters based on gapped q-grams for use as *lossy* filters, i.e. algorithms that are faster but may miss a small number of actual matches in the filtration phase. The results imply that gapped q-grams are far better suited than conventional ungapped q-grams for use in lossy filters. A comparison with the filter criterion implemented in BLAST also yielded very encouraging results not only for high levels of string similarity but also for moderate levels.

Conclusion

In this work we describe new achievements in the field of off-line approximate string matching. The first part of our work consists of an improvement and a generalization of a previously known combination of a filter with a simple index structure including their implementation. In the second part we introduce two new filter algorithms based on gapped q-grams which are designed to be used for off-line approximate string matching and can make efficient use of an index structure. They offer increased speed, filtration efficiency or a combination of these two when compared to the original filter algorithm employed in the first section of our work. This extends the range of possible applications for filter-based approximate string matching algorithms.

Outline

Our work is structured into six chapters apart from the introduction. We begin with a brief description of basic terms and notions in Chapter 2. Chapter 3 features index structures for exact string matching. In Chapter 4 we introduce filter algorithms in general and provide specific examples of existing filters as well as a description of QUASAR. The following Chapter 5 is devoted to gapped q-grams. Finally we present extensive experimental results in Chapter 6 and conclude our work in Chapter 7.

Chapter 2

Preliminaries

In this chapter we will introduce some definitions and notions used in our work. Readers familiar with the topic of the thesis may skip this section and refer to it only when necessary.

2.1 Stringology

The notion of a *string* or *sequence* is of vital importance to this thesis.

Definition 2.1.1: A *sequence* or *string* of *length* n is a n-tuple $S = (s_1, s_2, \dots, s_n)$ over an alphabet Σ with $s_i \in \Sigma, \forall i \in \{1, \dots, n\}$. One can also write S as $s_1 s_2 \dots s_n$. The length n is denoted by $|S|$ and the set of all finite sequences by Σ^*. Th empty sequence has length 0 and is written as ϵ.

Definition 2.1.2: For a given sequence $S = s_1 \dots s_n$ the sequence $I = s_p \dots s_q$ with $1 \leq p \leq q \leq n$ is called the *infix* or *substring* from p to q of the sequence S and denoted by $S(p, q)$. An infix is called the i-th *prefix* of S if $p = 1$ and $q = i$; it is called the i-th *suffix* of S if $p = i$ and $q = n$. A single character at position i in S is denoted by $S(i)$.

Definition 2.1.3: Given a string $S = s_1 \dots s_n$, we define the *period* of S as

$$\min_{i \in \{1, \dots, n\}} \{i | \forall j \in \{1, \dots, n - i\} : S(j) = S(i + j)\}$$

The period is basically the smallest amount of characters by which one has to shift S to the right in order for the overlap between the shifted and the unshifted string to be equal.

2.2 Graphs and Trees

We also need the concept of *trees* which are a special instance of a *graph*. We will list a few important definitions here (taken from [CLR90]).

Definition 2.2.1: A *directed graph* G is a pair (V, E) where V is a finite set and E is a binary relation on V. The set V is called the *vertex set* of G, and its elements are called *vertices*. The set E is called the *edge set* of G, and its elements are called *edges*. An edge e that connects two vertices u and v with $u, v \in V$ is written as (u, v).

Definition 2.2.2: An *undirected graph* is a directed graph modified so that the edge set E consists of *unordered* pairs of vertices, rather than ordered pairs. That is, an edge is a pair (u, v) where $u, v \in V$ and $u \neq v$. In essence this means, that while in the directed case an edge (u, v) only represents a connection from u to v, in the undirected case it also represents a connection from v to u.

Some further graph terminology:

Definition 2.2.3: If (u, v) is an edge in a graph $G = (V, E)$, we say that vertex v is *adjacent* to vertex u. When the graph is undirected the adjacency relation is symmetric, when it is directed this is not necessarily the case.

Definition 2.2.4: The *degree* of a vertex v in an undirected graph is equal to the number of edges in E that contain v. In a directed graph the *out-degree* of a vertex is the number of edges leaving it, and the *in-degree* of a vertex is the number of edges entering it. The *degree* of a vertex in a directed graph is the sum of in-degree and out-degree.

Definition 2.2.5: A *path* of length k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $\langle v_0, v_1, \ldots, v_k \rangle$ of vertices such that $u = v_0, u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. A path is *simple* if all vertices in the path are distinct. In an undirected graph, a path $\langle v_0, v_1, \ldots, v_k \rangle$ forms a *cycle* if $v_0 = v_k$ and $v_1, \ldots, v_k$ are distinct. A graph without is *acyclic*. An undirected graph is *connected* if every pair of vertices is connected by a path.

Some graphs with specific properties are given special names. One instance is the so-called *tree*.

Definition 2.2.6: A *tree* is a connected, acyclic undirected graph. A *rooted tree* is a tree in which one of the vertices is distinguished from the others. This vertex is called the *root* of the tree. We refer to the vertices of rooted trees as *nodes* of the tree.

Rooted trees have several notions associated with them:

Definition 2.2.7: Consider a node x in a rooted tree T with root r. Any node y on the unique path from r to x is called an *ancestor* of x. If y is an ancestor of x, then x is called a *descendant* of y. Every node is both an ancestor and a descendant of itself. If $x \neq y$, then y is a *proper ancestor* of x and x is a *proper descendant* of y. If the last edge on the path from the root r of a tree T to a node x is (y, x), then y is the *parent* of x, and x is a *child* of y. The root is the only node in T with no parent. If two nodes have the same parent, they are *siblings*. A node with no children is an *external node* or *leaf*. A non-leaf node is an *internal node*.

The *degree* of a node in a rooted tree is defined differently than that of a vertex in a graph:

Definition 2.2.8: The number of children of a node x in a rooted tree T is called the *degree* of x. The length of the path from the root r to a node x is the *depth* of x in T. The largest depth of any node in T is the *height* of T.

2.3 Pattern Matching

In string matching as well as in other areas of pattern matching one often uses the following 4 descriptors to describe classes of results of a search. Assume a query P and a database S and a definition of a match between P and an entry in S as well as an algorithm that attempts to locate all matches according to this definition. Those matches that are true matches according to the match definition and are reported by the search algorithm are called *true positives*. Those that are true matches but are missed by the algorithm are referred to as *false negatives*. Matches that are not true matches but reported as such are *false positives* and those that are neither true nor reported matches are *true negatives*.

2.4 DNA Sequences

Parts of this thesis discuss problems originating in the field of genetics and computational biology, namely DNA sequence databases. DNA, Deoxyribonucleic acid, can be thought of for our purposes as a string in the four-letter alphabet of *nucleotides* or *nucleic acids* A, T, G and C which denote the molecules adenine, thymine, guanine and cytosine. The entire DNA of a living organism is called its *genome*. Genomes vary in length from a few million letters (bacteria) to a few billion letters (mammals). DNA forms a helix. It is usually double-stranded with each single strand being the so-called *Watson-Crick complement* of the other where T pairs with A and C pairs with G. Since the complement is well-defined it is sufficient to store only one strand of a given sequence. Typical sequence databases contain DNA in the form of one or more strings over the four-letter alphabet of nucleotides. For each sequence they usually store a header with additional information about this string. The length of a DNA sequence or DNA databases is often measured in *base pairs* (bp).

Chapter 3

Indexing – Data Structures for Exact String Matching

Our work in the field of approximate string matching is based on filter algorithms. When matching two strings, these algorithms use a filter criterion to search for potential approximate matches between the two strings. These potential matches are then verified with a conventional algorithm for approximate string matching. Many filter criteria rely on exact matching of short substrings of query and target. We will describe filter algorithms more closely in the next chapter, but before we want to introduce some data structures for solving the *off-line exact string matching* problem and explain their properties. In this context we speak of *off-line* matching since we allow preprocessing of the longer string. Since many filter algorithms only require matching *q-grams*, substrings of a fixed length q, we also introduce a restricted version of the off-line exact string matching problem. We call it *off-line exact string matching with a fixed string length* q and provide an efficient solution.

3.1 Exact String Matching

A vital basic operation for many methods and algorithms in approximate string matching and database search is the following:

Definition 3.1.1: For a given *pattern* or *query* $P = p_1 \dots p_m$ and a *database string* or *target* $S = s_1 \dots s_n$ the problem of *exact string matching* consists of finding all starting positions i in S where $S(i, i+m-1) = P$, i.e. P matches the substring of length m that starts at position i in S.

In most applications $|S| \gg |P|$.

There are two different versions of the exact string matching problem. One, the so-called *on-line* version, is limited to working on two previously unknown strings P and S. The *off-line* version allows the algorithm to collect and store information about the structure of S before solving the problem. These two different versions of the exact string matching problem have been examined very closely in the past and many different algorithms have been developed for solving them. It should be noted that there exist two exactly reversed naming schemes for these algorithms that switch the terms on-line and off-line [AG97, Nav01]. We will stick to the more intuitive one used in [Nav01].

3.2 On-line Exact String Matching

On-line exact string matching consist in solving the problem stated above without any previous knowledge about the two strings P and S. Therefore a trivial lower bound for the asymptotic runtime is $\Omega(n+m)$ since all characters in P and S may have to be checked in order to find all occurrences of P in S.

Various algorithms for the solution of the on-line exact string matching problem have been published. Examples include algorithms by Knuth, Morris and Pratt [KMP77], Boyer and Moore [BM77] or Apostolico and Giancarlo [AG86].

3.3 Off-line Exact String Matching

The advantage of the off-line exact string matching problem when compared to the on-line version is the fact that one is allowed to examine the string S and record information about it before searching for all approximate matches of a string P in S. Data structures that are used to collect information about S and make them readily available are commonly called *index structures* or *indexes*. The two main features of such index structures are their *space consumption* and their *access speed*. Both of these parameters are characterized in dependency on the size of S. The use of index structures allows a much faster solution of the exact string matching problem than the on-line version. In this section we will present a few efficient index structures that can be used for exact string matching and describe their performance.

Two other important properties of index structures are their *construction time* and *construction space*. In order to be of practical use, it is important that an index can be constructed within reasonable time and without consuming excessive amounts of space. Databases with frequent changes in S require faster construction algorithms than relatively static ones.

3.3.1 The Suffix Tree

A popular data structure for exact string matching that also provides other powerful operations is the *suffix tree* [Wei73].

Definition 3.3.1: A *suffix tree* $\mathcal{T}$ for a string S with $|S| = n$ is a rooted tree with exactly n leaves labeled 1 to n. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S. No two edges from a node to its children can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to i exactly match the suffix of S that starts at position i, i.e. it spells out $S(i, n)$.

This definition does not guarantee that a suffix tree exists for a given string S. The problem is that if one suffix of S matches a prefix of another suffix of S, no suffix tree obeying the above definition is possible, since the path for the first suffix would not end at a leaf (the tree would have to contain a node in which two outgoing edges share the same first character). To avoid this problem one assumes that the last character of S appears nowhere else in S. Then, no suffix of S can be a prefix of another suffix. In practice one concatenates a unique 'termination' character \$ to S before creating the suffix tree for S. Figure 3.1 shows the suffix tree $\mathcal{T}$ for the string `abbaba`

A suffix tree provides an efficient mechanism for solving the off-line exact string matching problem. It is possible to access all occurrences of a given string P by matching the characters in P along the unique edges of $\mathcal{T}$ until one either reaches the end of P or one can not find further matches in $\mathcal{T}$. In the first case all leaves in the subtree below the point of the last match are labeled with the starting positions of P in S. In the second case there is no occurrence of P in S.

The runtime of this procedure is $O(m + o)$ assuming o occurrences of P in S and a finite alphabet. The key to understanding this runtime is to note that each node in $\mathcal{T}$ has at most $|\Sigma|$ outgoing edges (since no edges can have edge-labels beginning with the same character). Stepping through a node in the suffix tree therefore requires constant time (at least for a finite alphabet, where the size is considered constant). Since each outgoing edge of an internal node contains a non-empty substring the number of nodes one has to access in order to find an instance of P in the concatenated edge-labels

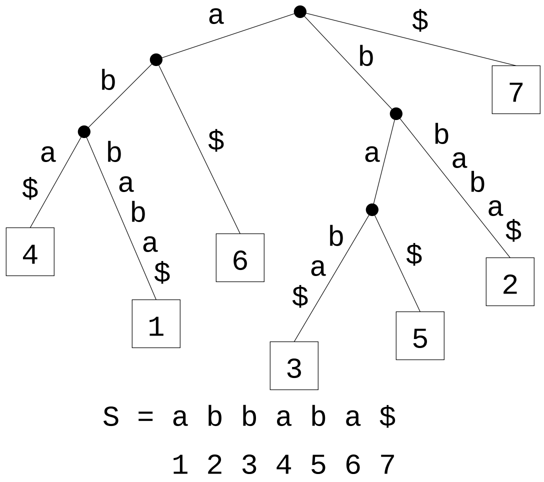


Figure 3.1: The suffix tree $\mathcal{T}$ for the string `abbaba`.

is at most m. If P does not exist in S this results in visiting at most m nodes and therefore requires $O(m)$ time. For the case where P has $o > 0$ occurrences in S one has to traverse the subtree at the end of the matching path using any linear-time traversal method (e.g. depth-first) and record the encountered leaves. Since every internal node has at least two children, the number of leaves accessed is proportional to the number of edges traversed, so the time for the traversal is $O(o)$. The total runtime is therefore $O(m + o)$.

The main drawback of suffix trees is their *space consumption*. With a properly chosen representation of the edge-labels in the tree a linear space bound can be achieved. The constants involved depend on the alphabet size. The reason for this is that any internal node can have up to $|\Sigma|$ outgoing edges and that the nodes have to be implemented to accommodate for this. There are many problems nicely solved in theory by suffix trees, where the typical string size is in the millions or even billions. For those problems a linear space bound in itself is not sufficient to ensure practicability. Consequently space-efficient implementations of suffix trees are of vital importance. Recent examples include work by Kurtz [Kur99] who presents an implementation that requires roughly 7 to 20 bytes per character in S with $|S|$ limited to 2^{27}. As a side note related to genomic databases we want to mention that the space consumption for DNA files that Kurtz tested was roughly 12.5 bytes per character. While this is already a considerable improvement over simpler implementations it can still be impractical for large texts.

The first linear time algorithm for constructing a suffix tree was introduced in Weiner's original paper from 1973 [Wei73], although he then called his tree a position tree. A different, more space efficient algorithm to build suffix trees in linear time was introduced by McCreight [McC76] in 1976. Ukkonen developed a less complex version in 1995 [Ukk95]. In 1997 Farach published an optimal suffix tree construction algorithm for integer alphabets [Far97].

Highly space-efficient implementations usually lead to increased construction times which, while still linear, can become an issue for large datasets. Also the amount of memory required to construct a suffix tree can be significantly higher than the actual space required for storing it.

As a conclusion it should be mentioned that suffix trees, while offering very efficient solutions to a wide range of problems in stringology, lack the required properties for our applications. The main problem is their space consumption. Therefore we looked for a more space efficient approach.

3.3.2 The Suffix Array

This data structure was introduced by Manber and Myers [MM93] in 1990. While not as versatile, it is more space efficient than the suffix tree and can be used to solve the off-line exact string matching problem with comparable efficiency.

Definition 3.3.2: Given a string S with $|S| = n$ the *suffix array* $\mathcal{A}$ for S is an array of integers in the range 1 to n , specifying the lexicographic order of the n suffixes of the string S. That is the suffix starting at position $\mathcal{A}[1]$ of S is the lexically smallest suffix, and in general suffix $\mathcal{A}[i]$ is lexically smaller than suffix $\mathcal{A}[i+1]$.

Again one uses a unique 'termination' character in order to distinguish suffixes where one is a prefix of the other. This termination character is usually defined to be either lexicographically smaller or greater than all characters in the alphabet. Figure 3.2 shows the suffix array $\mathcal{A}$ for the string `abbaba`.

```
1: a b b a b a $          4: a b a $
2: b b a b a $            1: a b b a b a $
3: b a b a $              6: a $
4: a b a $      ------>   3: b a b a $
5: b a $         SORT     5: b a $
6: a $                    2: b b a b a $
7: $                      7: $

            S = a b b a b a $
                1 2 3 4 5 6 7
   Suffix Array: 4 1 6 3 5 2 7
```

Figure 3.2: The suffix array $\mathcal{A}$ for the string `abbaba`.

In contrast to the suffix tree, the suffix array only contains integer pointers to the suffixes of S; its size is therefore completely independent of the alphabet size $|\Sigma|$. Hence it is possible to store a suffix array $\mathcal{A}$ for a given string S of length n in n pointers of size $\lceil \log_2 n \rceil$ bits. In practice this usually means that one uses the next larger available word size the hardware offers. For texts of sizes below 4 GB this means a pointer of size 4 bytes. The total space consumption of the suffix array would therefore be $4n$ bytes for such texts. The second consequence of the independence of alphabet size is that in general, suffix arrays are faster than suffix trees for large alphabets and slower for small alphabets [MM93]. This is related to the fact that the nodes of a suffix tree grow with the size of the alphabet, which leads to an increase in the time required for examining a node in the tree during a search.

In the original suffix array paper Manber and Myers also introduced two algorithms for efficient exact string matching. One of them uses only the suffix array, the other requires an auxiliary data structure containing the *longest common prefixes* of all suffixes with pointers that are adjacent to each other in the suffix array.

The algorithm using only the suffix array is very simple. One can conduct a *binary search* in $\mathcal{A}$ in order to find the two pointers $\mathcal{A}[s]$ and $\mathcal{A}[l]$ in $\mathcal{A}$ that point to the suffixes in S which are the largest/smallest suffix that is lexicographically smaller/larger than a query pattern P. In each step of the binary search one has to compare P with the suffix that $\mathcal{A}$ points to. This requires at most $|P| = m$ character comparisons. The binary search can be completed in $O(\log n)$ steps. The total number of character comparisons executed by this search algorithm is therefore $O(m \log n)$. In addition to the binary search one has to access the $l - s - 1$ pointers between $\mathcal{A}[s]$ and $\mathcal{A}[l]$ that point to the occurrences in the database string S. We call this number of occurrences o. The total time for locating and accessing all occurrences of P in S is therefore $O(m \log n + o)$.

By using a second data structure containing precomputed information about the length of the *common prefix* between certain entries in the suffix array one can speed up the search and it is possible to achieve a runtime of $O(m + \log n + o)$ character comparisons. A detailed description of this slightly more complicated algorithm can be found in the original suffix array paper as well [MM93]. The extra space required for the array with the longest common prefixes is n integer entries with a value of at most $|P| = m$. This results in a (theoretical) increase by $n \cdot \lceil \log_2 m \rceil$ bytes. However, unless one implements the array using a space optimal bit-manipulation approach it will usually result in adding $4 \cdot n$ bytes of extra storage on most systems.

Suffix arrays can in principle be constructed in linear time. This is done by first generating the suffix tree and then extracting all the leaves from it and storing their pointers in the suffix array. This of course means that one has to provide the space required for building the suffix tree of a given text. It also means that while the asymptotic runtime may be linear one will have to deal with quite large constants due to the complicated nature of the suffix tree construction. Therefore in practice one often uses other algorithms for the suffix array construction. Manber and Myers introduced an $O(n \log_2 n)$ time construction algorithm in their original work about suffix arrays [MM93]. In case one has to construct the suffix array in external memory, a widely used construction algorithm is the one by Baeza-Yates, Gonnet and Snider which constructs the suffix array in place [GBYS92]. A good survey of suffix array construction algorithms in external memory was published in 2002 [CF02].

When comparing suffix arrays with suffix trees with respect to their suitability for solving the exact string matching problem, it turns out that while suffix arrays are of comparable speed they offer savings in space consumption. Another important issue is the fact that suffix arrays are relatively simple to implement.

3.3.3 Other Indexes for Exact String Matching

There exist many more indexes for exact string matching like for example patricia trees [Mor68], prefix trees [Wei73] which are also known as position trees [AHU74] or string B-trees [FG99]. Because our main concern was the space consumption of the index and since the suffix array turned out to be the best in that respect we forego further, more detailed descriptions of these indexes.

We would like to mention that recently Ferragina and Manzini introduced the so-called FM-index [FM01], which is basically a compressed suffix array. It might be interesting to evaluate the suitability of this index for our problem. The tradeoff between access time and index size might be an interesting alternative for some problem instances. A second publication concerning compressed indices comes from Grossi and Vitter [GV00].

3.4 Exact String Matching with Fixed Length q

One can restrict the problem of exact string matching to strings of a predetermined, fixed length q. The algorithms we developed only require an efficient solution for this case since they use short q-grams to locate potential matches in S. Therefore we concentrated on exploiting this restriction and solving what we call the problem of *exact string matching with fixed length q*.

Definition 3.4.1: The problem of *exact string matching with a fixed length* q is a special case of the exact string matching problem. Again, for a query string P and a database string S one wants to locate all occurrences of P in S and report them. We call such occurrences *hits*. As a further restriction one requires the query to have a fixed length q.

This modified version of exact string matching allows further improvements of index structures with respect to access speed.

3.4.1 A Word Lookup Table

An example for such improvements is a simple word lookup table, namely an array of *hit lists*, employed for example by Jokinen and Ukkonen [JU91]. It requires a limited size alphabet Σ and allows improvements of access speed at the cost of additional space consumption compared to approaches for the general exact string matching problem. A drawback of this data structure is that it is only practical for relatively small values of q and moderate alphabet sizes.

Definition 3.4.2: Given a string S of length n over an alphabet Σ and a fixed string length q, the *word lookup table for q-grams* consist of an array W of size $|\Sigma|^q$ and $|\Sigma|^q$ *hit lists* $H(i), 0 \leq i < |\Sigma|^q$. One assigns each element $x \in \Sigma$ an integer ID $\mathcal{I}(x)$ between 0 and $|\Sigma| - 1$ and uses the canonical encoding

$$\mathcal{F}(s) = \sum_{i=0}^{q-1} \mathcal{I}(s(i+1))|\Sigma|^i, \qquad s \in \Sigma^q$$

of all strings of length q into integers of base Σ that assigns them numbers between 0 and $|\Sigma|^q - 1$ (see for example [KR87]). For each possible string s of length q one saves the list of all its occurrences in S in its corresponding hit list $H(\mathcal{F}(S))$. In order to be able to access this list efficiently one inserts a pointer to it into the array position $W[\mathcal{F}(S)]$.

To construct this data structure one uses the encoding of all strings of length q into integers of base $|\Sigma|$ to uniquely identify them. For each possible string one generates a list of all its occurrences in S. The second step is to generate an array of pointers to the lists of occurrences or *hit lists*. The array is indexed by the integer IDs of all possible q-grams.

With dynamic lists the hit lists can be generated in one pass and in time $O(n)$. Creating the array of pointers requires $O(|\Sigma|^q)$ time. Therefore the total construction time for this data structure is $O(n + |\Sigma|^q)$.

In practice a two-pass construction that stores the hit lists in one large array instead of in lists makes more sense. One generates two arrays W and H of sizes $|\Sigma|^q + 1$ and n.

W will later contain the starting positions (i.e. the offset) of the hit lists in the second array. In the first pass one scans S and uses W (which has to be initialized to contain only zeros) to count the number of occurrences of each possible string s of length q by incrementing $W[\mathcal{F}(s)]$ every time one finds the q-gram s. One then calculates the offsets of the hit lists in the second array by iterating over W setting $W[0]$ to zero and $W[i]$ to the number of hits in all hit lists from 0 to $i-1$. In the second pass one scans S again and for each q-gram s one looks up the offset of its hit list in $W[\mathcal{F}(s)]$, writes it position in S into H at that offset and then increments the offset in $W[\mathcal{F}(s)]$. This of course modifies W and in order to reset it to the original values one has to shift all values in W one array element to the right and reset $W[0]$ to 0 in a post-processing step. Figure 3.3 shows W and H for the sample string `abbaba`.

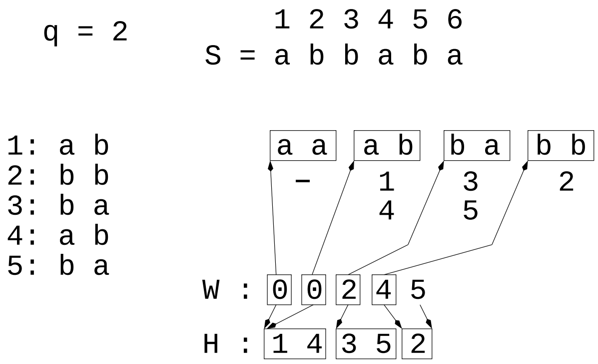


Figure 3.3: The offset array W and the hit list array H for the string `abbaba`.

Due to the nature of $\mathcal{F}$, the hit lists are stored consecutively and in ascending lexicographic order with respect to the substrings they encode.

These two arrays provide efficient access to all occurrences or hits of a given query P in S. One simply has to compute $\mathcal{F}(P)$, the integer ID of P, and access the start offset of the hit list in $W[\mathcal{F}(s)]$. The end of the hit list can be determined by looking up the start of the next hit list, $W[\mathcal{F}(s)+1]$ and subtracting one from it. Assuming there are o occurrences of P in S, this results in an access time of $O(|P|+o)$. While this is asymptotically the same as, for example, for a suffix tree, the constants involved are much smaller.

An additional benefit of this index is that one does not have to access the actual sequence of S anymore. All information is contained in H and W. This results in much better locality of memory access than that of other index structures and makes this index a good candidate for use in external memory algorithms.

3.4.2 Combining Suffix Array and Lookup Table

In the course of implementing an approximate string matching algorithm we wanted to be able to experiment with variable values of q. In order to avoid the necessity of constructing and storing several versions of the lookup table we used a suffix array instead of the hit list array H. This has the advantage, that the hits in the suffix array are sorted not only for one value of q, but for all. It allowed us to construct several lookup arrays W for the different values of q we were interested in. However, there is also a small drawback to this approach. Since a suffix array considers all suffixes, even those shorter than q, this will result in a few additional false hits for some queries. In our algorithms this effect was negligible due to the small number of such cases. Figure 3.4 shows the lookup table when combined with a suffix array instead of H as in Figure 3.3. The single suffix shorter than q is shaded gray to mark the difference to a conventional lookup table.

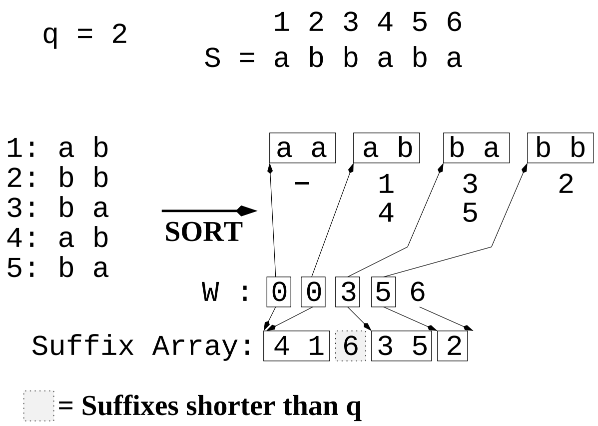


Figure 3.4: Using a suffix array instead of H for the string `abbaba` and $q = 2$.

3.4.3 Large Values of q – a Hybrid Approach

Of course the practicality of this index suffers when one wants to use larger values of q or large alphabets. Since the array of offsets W requires storing $|\Sigma|^q + 1$ integers, large alphabets and/or large values of q render this approach impractical. However, one can still combine this technique with a suffix array in order to achieve a certain speedup.

Again we assume a query string P of length q and a target string S. We first construct the suffix array for S. This suffix array is almost the same as the array of hit lists H for a certain value of q. There are only two differences. One is the fact that the suffix array contains pointers to suffixes that are shorter than q while H does not contain the offsets of these suffixes. It is trivial to remove these pointers from the suffix array to avoid any problems. The second difference between H and the suffix array is the way in which the hit lists are organized. While in H the hit list for a certain q-gram will be ordered by the occurrences of the q-grams in S (at least if one uses the construction algorithm that we described above), this is not the case for the suffix array. Here the hit list is ordered by the lexicographic order of the suffixes to which the entries in the hit list point to. One can make use of this fact when searching for exact matching strings with higher lengths q. In order to do so one builds the array of hit lists using the suffix array as H and using the largest practical value of q, q_{index}. Then the search for all occurrences of a string P of length $q > q_{index}$ can be conducted in two phases. In the first phase one searches for the sub-array of H that contains the hit list for the prefix of P with length q_{index}. This sub-array contains all pointers to occurrences of suffixes that have the first q_{index} characters in common with P. In order to actually locate the sub-array that contains pointers to matches between P and S, the same binary search approach on the current sub-array can be applied, that is used for searches in suffix arrays. This two-phase approach reduces the number of accesses to the text string S which may not make much of a difference for an internal memory index but can be quite interesting for an external one. In any case it increases the locality of the memory access pattern of the search.

Chapter 4

Filters for Approximate String Matching

Our primary research goal was the development of efficient algorithms for solving certain variants of the off-line version of the *approximate string matching* problem. The terms *on-line* and *off-line* have the same meaning as for exact string matching, i.e. in the off-line version of the problem it is possible to preprocess the database and store information about it in an index while in the on-line version it is not.

The variants of the off-line approximate string matching problem which we consider are vital for many scientific and commercial applications. Examples include text retrieval, signal processing, optical character recognition and pattern recognition. They have become increasingly important in recent years with the advent of large genomic databases since many applications in computational biology query these databases [Wal95]. The particular problem that initiated our work in the field of approximate string matching was EST clustering [KV98]. More examples include genome assembly [WM97, VAS+98, MSD+00, VAM+01], gene finding [BH00], or large-scale genome comparison[MAM+02].

In this chapter we want to introduce three different well-known filter criteria. We also describe QUASAR, our implementation of a filter algorithm that combines one of them with an index structure to provide a highly efficient solution of an important approximate string matching problem. The development of QUASAR lead to new results which were published at the RECOMB conference in 1999 [BCF+99]. Furthermore, in order to be able to measure and describe important properties of filter algorithms, we introduce methods for the theoretical and experimental analysis of filter algorithms. Of special interest in this regard is the *minimum coverage*, a new measure for analysing and predicting filter performance.

4.1 Some Approximate String Matching Problems

There are several variants of the *approximate string matching* problem. One reason for this are the many different definitions of *similarity* or *distance*. Which is used for a specific problem depends on the types of strings being matched as well as the time constraints for the approximate string matching algorithm. In general, more sophisticated measures of similarity or distance require more complicated algorithms and data structures for solving the problem and lead to an increase in runtime. In many cases compromises with respect to quality are made in order to speed up searches.

A second important property of approximate string matching problems is whether they are *global* or *local*. Global approximate string matching for a given query P and a target S is defined as locating approximate matches between P and substrings of S. In contrast to this, local approximate string matching also locates approximate matches between substrings of P and S. An important parameter for local matching is the minimum length of the matches since very short matches are usually extremely common. We call this minimum match length the *window length* and assign it the variable w. Local approximate string matching would of course be able to detect a match of the complete query against a substring of S as well. The choice between global and local matching depends on the type of problem that is being considered. Obviously, local matching is more powerful since it will also detect matches between the complete query P and substrings of S, but it also introduces additional complexity.

4.1.1 Distance Measures

A very common distance measure for strings is the so-called *edit distance*. It is sometimes referred to as *Levenshtein distance* in recognition of the paper by V. Levenshtein [Lev66] where it was probably first discussed.

Definition 4.1.1: The *edit distance* of two strings S_1 and S_2 is defined as the minimum number of *character-wise edit operations* necessary to transform S_1 into S_2. A single edit operation can either delete or replace a character in S_1 or insert a new character into S_2. We write the edit distance as $d_E(S_1, S_2)$.

One of the reasons for the popularity of the edit distance in computational biology is that it is a simple but fairly accurate measure for the evolutionary proximity of two DNA sequences that reasonably similar to each other.

A second, simpler distance measure is the so-called *Hamming distance* introduced by R. W. Hamming [Ham50].

Definition 4.1.2: The *Hamming distance* of two strings S_1 and S_2 is defined as the minimum number of *character-wise replacement operations* necessary to transform S_1 into S_2. A single operation can replace a character in S_1 with a different one in S_2. We write the Hamming distance as $d_H(S_1, S_2)$.

This distance is clearly not as powerful and not suitable for many applications. Nevertheless there exist practical problems for which using the Hamming distance is sufficient or even desirable.

4.1.2 Four Interesting Variants

Using the notions from before we define four important variants of the approximate string matching problem. The first is the so-called *k-differences problem* which uses the edit distance and is based on global matching.

Definition 4.1.3: For a given query string $P = p_1 \dots p_m$ and a database string $S = s_1 \dots s_n$ the *k-differences problem* is that of finding the set M of all substrings $S(i,j)$ of S which have an edit distance of at most k to P, i.e. where $d_E(P, S(i,j)) \leq k$. We call members of M *true matches* or short *matches* and M the *set of true matches.*

The local variant of this problem is called the *substring matching problem with k-differences.*

Definition 4.1.4: For a given query string $P = p_1 \dots p_m$ and a database string $S = s_1 \dots s_n$ the *substring matching problem with k-differences* is that of finding the set M of substrings $S(i,j)$ of S which have an edit distance of at most k to a substring $P(x,y)$ of P, i.e. where $d_E(P(x,y), S(i,j)) \leq k$. In practice one adds a second parameter, the *window length w*. One only searches for pairs of substrings where $y - x + 1 \geq w$, i.e. where the substring of P has a certain minimum length. This is necessary to avoid trivial matches between single characters or very short substrings of P and S. Again we call members of M *true matches* and M the *set of true matches.*

For $w = m$ this problem is equivalent to the k-differences problem.

Both of these problems can also be formulated using the Hamming distance. In this case they are called the *k-mismatches problem* and the *substring matching problem with k-mismatches* which are the global and local variants of approximate string matching using the Hamming distance.

4.2 Lossless and Lossy Algorithms

When we talk about algorithms for the problems described above, we distinguish between two different classes, *lossless* and *lossy* algorithms. We speak of *lossless algorithms for approximate string matching* if they are able to locate all true matches. Some algorithms are not as perfect and may miss some true matches resulting in false negatives. We call them *lossy algorithms for approximate string matching.* They usually offer increased speed and can be, depending on the amount of false negatives, good alternatives for many applications. It is also possible that the same algorithm is a lossless algorithm for certain parameters but becomes a lossy algorithm for other parameters such as, for example, higher values of k in the above problems.

4.3 Algorithms for On-line Approximate String Matching

The on-line version of the k-differences and k-mismatches problem has been studied quite thoroughly (see for example [AG97, Nav01]). There are several variations of an algorithm based on dynamic programming for the so-called pairwise global alignment problem ([NW70, Sel74, SW81]) which can be used to solve them. They are lossless algorithms which require $O(nm)$ time and can therefore be impractical for conducting large numbers of searches on long target strings and databases. Myers introduced a bit-parallel version in 1999 [Mye99] which runs in time $O(nm/w)$ where w is the word-size in bits. Nevertheless the runtimes can be too large for certain applications.

This lead to the development of lossy algorithms for the on-line version of the k-differences problem. Two important algorithms from the late 1980s and early 1990s are FASTA[PL88, Pea90] and BLAST[AGM$^+$90, AMS$^+$97]. Their expected runtime is $O(mn)$ with BLAST having smaller constants for most applications.

Algorithms for the on-line versions of approximate string matching usually proceed by sequentially scanning the database. We call such algorithms *sequential algorithms.*

4.4 Filter Algorithms

One class of algorithms for the off-line versions of the 4 problems introduced above are the so-called *filter algorithms.* Filter algorithms consist of at least two phases. In the first pass they conduct a fast but cursory inspection of S with a *filter criterion* which is a set of one or more conditions used to decide whether a given region of S is a potential candidate for a match with P or a substring of P in the local case. We call this region a *match region.* Its exact form can differ. The first stage of the filter algorithm is the *filtration phase* and the candidates returned by it are *potential matches.* Evaluating the filter criterion for the database S and a given query P is usually significantly faster than using a conventional non-filtering algorithm to examine S. All substrings of S that are returned as potential matches by the filter are then further analyzed with such a conventional approximate string matching algorithm in order to detect the true matches between P and S. This second stage of the filter algorithm is called *verification phase.* Typically the filter criterion will return only a small subset of S as potential candidates for approximate matches with P or substrings of P thus the verification phase requires far less time than applying a conventional algorithm to the complete database. Figure 4.1 contains a flow chart of a typical filter algorithm.

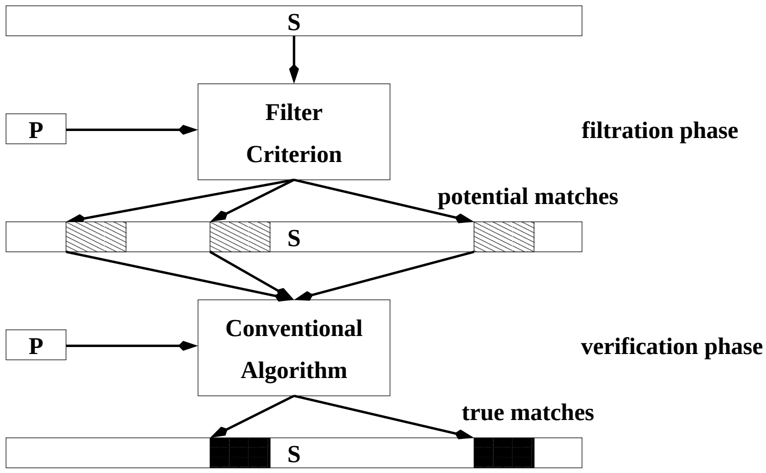


Figure 4.1: The structure of a typical filter algorithm.

In general, a good filter criterion will dramatically reduce the amount of data that has to be evaluated in the verification phase. This can result in a significant speedup when compared to conventional approximate string matching algorithms.

There are different types of filters. *Lossless filters* guarantee reporting all true positives in the set of potential candidates. Their only possible 'errors' are false positives, i.e. reporting sequences that are not an approximate match as potential matches. *Lossy filters* can also miss true matches which results in false negatives. They are typically even faster than lossless filters and offer a tradeoff between speed and reliability.

Some filter algorithms apply more than one filter either in parallel (by determining the intersection of their individual filter results) or in sequence (by iteratively applying different filters to the filter results of their predecessors).

When analysing the speed of a filter algorithm and the ability to distinguish between true and false positives, certain notions are important. Given a string S, we define any starting position of a substring of S that is a true match to the query P (or a substring of P in case of local matching) according to the problem definition as a *matching position*. It should be noted that even though there are usually many substrings of different lengths for a given starting position in S, in most cases only few of these substrings will constitute true matches. In this case we still consider this starting position a matching position. Those starting positions for which all substrings of S starting at that position do not constitute a true match with P or a substring of P in the local case are called *non-matching positions*. We define $P(S)$ to be the set of all matching positions and $N(S)$ as the set of non-matching positions. A filter criterion will return match regions that contain potential candidates and in doing so mark some starting positions of substrings in S as *potential matching positions*, the set of which we call $PM(S)$. The set of rejected starting positions is defined as $PN(S)$. We call the set of true positive starting positions $TP(S)$, the set of false positive starting positions $FP(S)$, the set of false negative starting positions $FN(S)$ and the set of true negative starting positions $TN(S)$. Figure 4.2 contains a graphical representation of all sets mentioned above for a typical lossless filter algorithm. The set of potential matches $PM(S)$ contains all matching positions $P(S)$ which form the set of true positive starting positions $TP(S)$ as well as some of the non-matching positions $N(S)$ which form the set of false positive starting positions $FP(S)$. The non-matching positions that are rejected by the filter form the set of true negative starting positions $TN(S)$. The set of false negative starting positions $FN(S)$ is empty since a lossless filter never misses true matches.

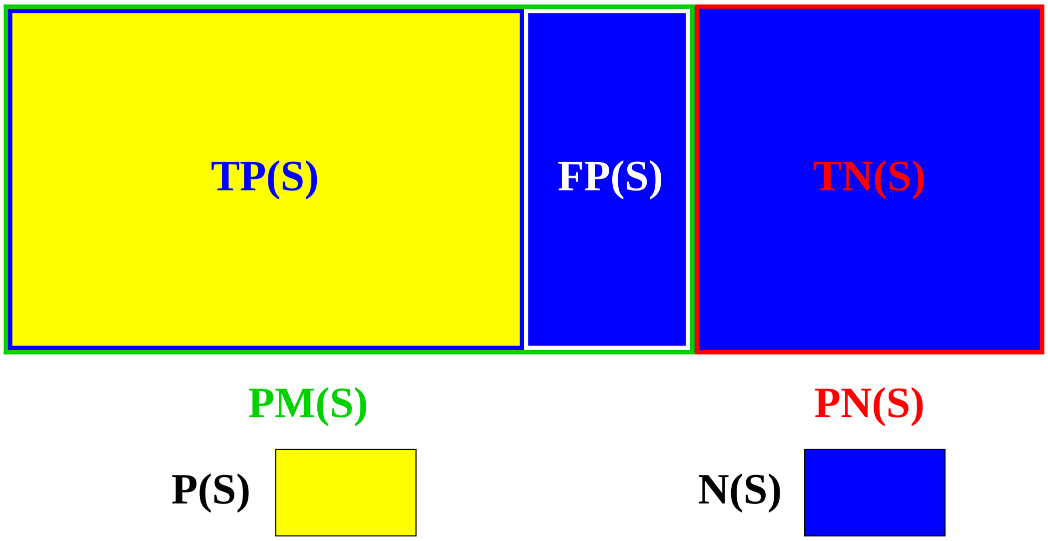


Figure 4.2: A typical lossless filter.

Figure 4.3 shows all the sets for a typical lossy filter algorithm. Here $FN(S)$ is not empty.

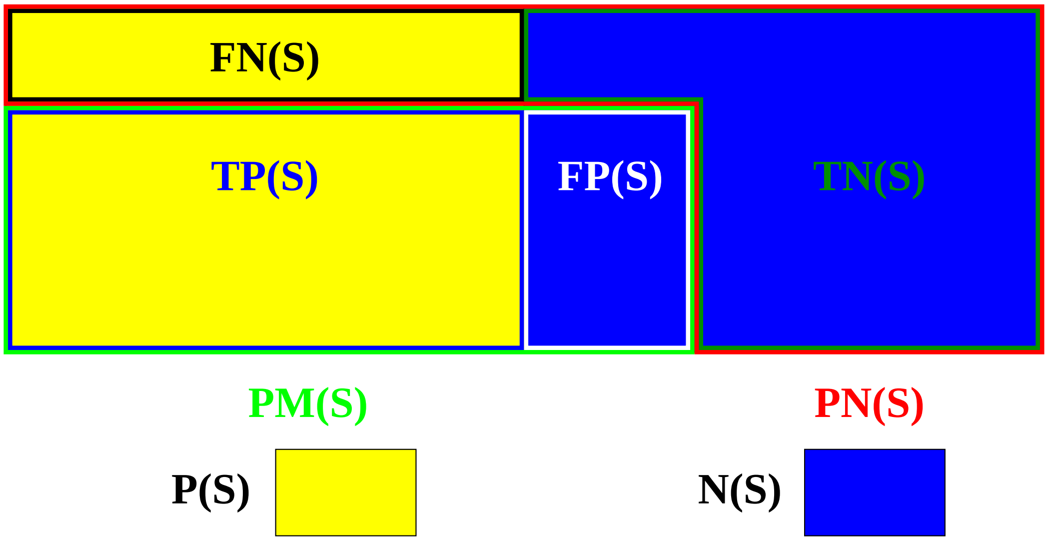


Figure 4.3: A typical lossy filter.

Using the sets defined above we can define an important concept that we call the *filtration efficiency*:

Definition 4.4.1: Given a filter algorithm and a query P as well as a target string S for which $N(S)$ is the set of non-matching positions with P and for which the filter algorithm rejects the starting positions $PN(S)$, we define the *filtration efficiency* f_e as

$$f_e = \frac{|PN(S)|}{|N(S)|},$$

the ratio between the rejected and non-matching starting positions.

The filtration efficiency is a good measure of the ability of a filter to discard uninteresting regions of S and reduce the amount of work necessary in the verification phase. We would like to mention that there exist other, different definitions of filter or filtration efficiency. However, we decided to use the one introduced above since it best suited our needs in describing this aspect of filter algorithms.

To illustrate the filtration efficiency we present a few special cases for which it has special values. If, for example a filter is capable of discarding all false positive starting positions, i.e. it reports only true matches, then it has a filtration efficiency of one or 100%. Another extreme case would be a filter that recognizes all false positive starting positions as potential matches. In this case the filtration efficiency would be zero or 0%.

In the following we analyze the speedup possible with filter algorithms and the dependencies between the speed of the filter criterion and the verification algorithm. We assume the evaluation of the filter criterion as well as the verification phase to have a runtime linear in the size of S. If one defines v_f as the throughput of the algorithm computing the filter criterion (e.g. in characters per second) and v_v as the throughput of the verification algorithm and neglects the influence of increased similarity of the potential matches with the query P one can approximate the speed increase for using a filter algorithm. The time t_c required to solve the problem solely with a conventional algorithm can be expressed as:

$$t_c = \frac{|S|}{v_v}$$

where v_v is the throughput of the conventional algorithm.

Assuming no true matches, the fraction of the database remaining after the filtration phase would be $1 - f_e$. Therefore, the time required by the filter algorithm using the same conventional algorithm for verification would be the sum of the filtration phase

$$t_f = \frac{|S|}{v_f}$$

and the verification phase

$$t_v = \frac{(1 - f_e) \cdot |S|}{v_v}$$

Therefore the speedup can be expressed as:

$$speedup = \frac{t_c}{t_f + t_v} = \frac{\frac{|S|}{v_v}}{\frac{|S|}{v_f} + \frac{(1-f_e)\cdot|S|}{v_v}} = \frac{1}{\frac{v_v}{v_f} + 1 - f_e}$$

This formula underlines the importance of the filtration efficiency of filter algorithms and the relation between the speed and efficiency of the filter criterion and the speed of the conventional algorithm used for verification.

In the following sections we will describe selected filter criteria. We introduce three different filter criteria of which two can use varying degrees of accuracy in describing the conditions under which they detect a potential match. How accurate they actually are depends on the definition of the match region. In general more stringent definitions of the match region will result in better filtration efficiency at the cost of increased runtime of the filtration phase. One has to pay for the increase in filter efficiency by spending more time on the evaluation of the filter criterion. In the case of the pigeonhole principle based filter and the q-gram lemma we will describe several different definitions of match regions which provide a tradeoff between speed and accuracy. For these filters we will also discuss the differences in the definitions of match regions depending on whether using the Hamming or edit distance. For the simpler case of the filter criterion employed by BLAST there is only one definition of a match region which is the same for both distance measures we consider.

All three filter algorithms are based on detecting exact matching substrings of a fixed length q between P and S. We will call such pairs of exact matching substrings *hits*. To fully describe a hit we use a two-tuple (i, j) where i corresponds to the starting position of the substring or q-gram in P and j to the starting position in S.

4.4.1 The q-gram Lemma

This filter was introduced by Jokinen and Ukkonen [JU91]. They also coined the term q-gram. The basic q-gram method works as follows. First, find all matching q-grams between the pattern and the text. That is, find all pairs (i, j) such that the q-gram at position i in the pattern is identical to the q-gram at position j in the text. Such a pair is a hit according to our previous definition. Second, identify the match regions in S that contain at least a certain minimum number t of hits. These form what we call the set of potential matches. We call the value t the *threshold*.

There are different ways of defining the match regions and counting the hits in them, see for example [JU91, HS94, BCF$^+$99]. However, all of them have the same threshold, the significant number of q-grams. This number is given by the *q-gram lemma*.

Lemma 4.4.1 (The q-gram lemma [JU91]): Let P and S be strings of length m and n. Then P and any substring of S with (Levenshtein or Hamming) distance k to P have at least $t = m - q + 1 - kq$ common q-grams.

The formula for the threshold t can be split into two parts. The first half, $m - q + 1$ equals the total amount of q-grams in P. The second half, kq represents the fact that in the worst case, a single error can invalidate up to q of the common q-grams, i.e. k errors can invalidate kq common q-grams.

The threshold t is tight in the sense that using any lower value might miss a true match and turn the filter into a lossy filter. We call such a tight threshold *optimal*.

For example, the strings `ACAGCTTA` and `ACACCTTA` have Hamming and Levenshtein distance one and have three common 3-grams: `ACA`, `CTT` and `TTA`. Assuming we allow at most $k = 1$ errors, the value of t for this example would be $|P| - q + 1 - q = 3$.

The following piece of pseudocode describes the general structure of the filtration phase which all filter algorithms based on the q-gram lemma have in common.

Algorithm 1: Filtering with the q-gram lemma

```
Set hit counters for match regions to 0
for all x such that x is a q-gram from P do
  Determine all hits of x in S
  for all hits h of x in S do
    Increment the hit counter of those match regions that contain h
  end for
end for
for all match regions y do
  if hit counter of y ≥ t then
    Report y as a potential match
  end if
end for
```

The above description of the q-gram method just provides a guideline for actual implementations of an algorithm based on the q-gram lemma. Different realizations of the method are described in [JU91, Ukk92, HS94, BCF$^+$99]. Our implementation called QUASAR which contains new ideas improving the approach described in [JU91] is discussed in Section 4.8.

There are several variations of the basic filter algorithm based on the q-gram lemma. We will briefly describe them in the following:

Sampling A popular way to reduce the runtime of the filtration phase is to consider only some of the q-grams of P and/or S [CM94, Tak94, Mye94, ST95, ST96, Shi96, NSTT00].

Approximate q-grams A method for improving filtration efficiency at the expense of increased runtime of the filtration phase is to allow one or more errors in the q-grams being matched between P and S [Mye94, CM94, ST95, NSTT00].

Gapped q-grams The use of q-grams with gaps can be used to improve runtime as well as filtration efficiency. This idea will be discussed in more detail in Chapter 5.

Multiple shapes This approach uses sets of different gapped q-grams [CR93, PW95].

Of course, various combinations of these methods are possible. For example, sampling and approximate q-grams have often been used together [Mye94, CM94, ST95, NSTT00].

Returning to the q-gram lemma there is one aspect that deserves closer scrutiny, the definition of 'common q-grams'. It is possible to come up with several more or less stringent definitions of this term. All of them are in principle descriptions of valid arrangements of t or more hits in P and S and define a match region. A naive approach would be to simply check whether there exist at least t hits between P and S. While easy to evaluate, for long strings S it would lead to almost certain recognition of S in its entirety as a potential match. A simple but quite efficient definition of match regions was introduced by Jokinen and Ukkonen. They observe that for any valid approximate match to P in S there is a substring of S with length $|P|$ and at least t common q-grams. Using this fact they proceed to define a match region that is very simple to evaluate. Instead of locating all substrings of S with length m and t or more hits, they check substrings of length $2(m-1)$. We call these substrings *blocks* and their length the *block size* b. For blocks of size $2(m-1)$ it is sufficient to examine every $(m-1)$-th substring. The blocks then overlap each other by $m-1$ characters and all substrings of length m are substrings of the actually checked blocks. One can therefore detect all potential matches by checking the amount of hits in only $\lceil \frac{n}{m-1} \rceil + 2$ substrings. The drawback of checking longer substrings is the increased probability of detecting t or more hits in a block. Nevertheless the strategy proposed by Jokinen and Ukkonen can decrease the overall runtime of filtration and verification phase. Figure 4.4 shows an example where one only has to examine the number of hits in 4 substrings of length $2(m-1)$ of S.

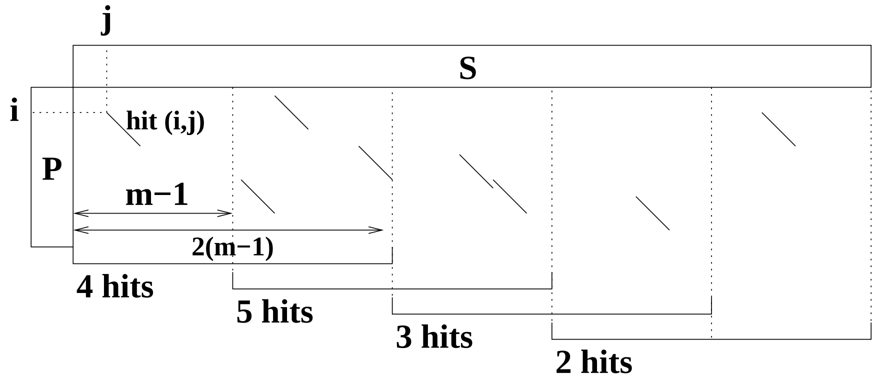


Figure 4.4: Blocks of length $2(m-1)$ as match regions.

A more stringent definition of the match region takes the ordering of the hits in P and S into account. It was introduced by Holsti and Sutinen in 1994 [HS94]. If one simply counts the number of hits in a given substring of S, it is possible that potential matches are found, where the hits in S have a different order than in P. To accurately describe this phenomen, one can define the *diagonal* d of a hit (i,j) with starting positions i in P and j in S as $d = j - i$.

One can think of the diagonal as corresponding to the diagonal in the dynamic programming matrix (used by the algorithms mentioned in Section 4.3) of the strings P and S. For the Hamming distance, the t or more hits of a true match between P and S have to be from the same diagonal since insertions or deletions are not allowed in an approximate match using the Hamming distance. For the case of the edit distance things are slightly more complicated. Here insertions and deletions are allowed and can lead to a shift in the diagonal of subsequent hits. Assuming k insertions or deletions, this can lead to the t or more hits being spread out over at most $k+1$ diagonals. Holsti and Sutinen observed this in their 1994 paper [HS94], so we can define a match region as a set of one (or $k+1$ respectively) diagonal(s) containing t or more hits. An important observation that helps to understand why this approach is more stringent than the one based on examining substrings is the fact that hits that are not in the same order in P and S will have significant differences in their diagonal. While this approach can increase the filtration efficiency, it requires more time to evaluate the filter criterion using this match region. It depends on the actual implementation of the filtration and verification algorithms whether it will lead to a speed-up in practice. Figure 4.5 shows an example for two strings P and S containing a total of 6 hits where all diagonals containing at least one hit are marked.

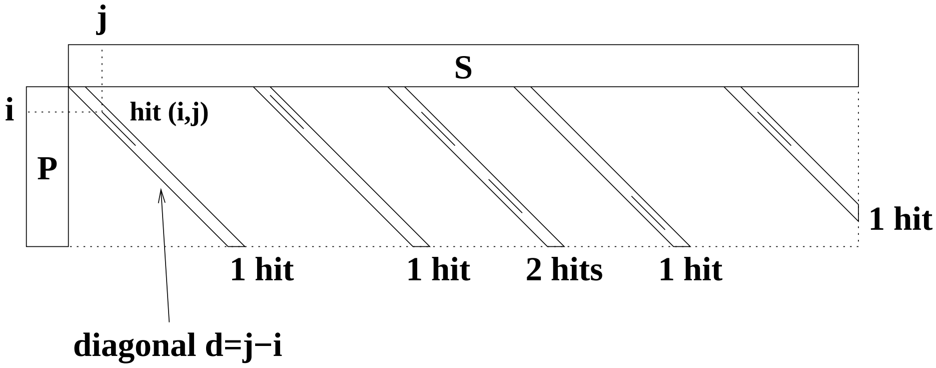


Figure 4.5: Single diagonals as match regions for the Hamming distance.

As for the case of checking substrings of S it is possible to consider larger sets of diagonals. Again this reduces the number of match regions to be checked but also reduces filtration efficiency. The best choice in practice depends on the implementation of the filter criterion and the verification algorithm.

We would like to point out a potential problem related to the actual characters in P and S that can affect both the likelihood and the structure of potential matches. If P, S or both strings contain substrings that are longer than q and have a low period and there is a matching q-gram in the respective other string, this can lead to degenerate sets of hits. Take for example the case of a query P containing only one letter from Σ.

Even if S only contains a single substring of length q that consists of the same letter as P, this will lead to $|P| - q + 1$ hits. While this example is an extreme case, repeats and low-periodicity strings can still cause serious degradation of filtration efficiency. In many practical applications this is not a problem though, since repeat sequences are often masked out or ignored. In computational biology for example, there exist repeat databases and tools to screen queries and/or databases for common repeats. It is also the case that more restrictive match regions will reduce the effects of such degenerate cases. If, for example, one uses sets of $k + 1$ diagonals, even a long repeat with period 1 that has a matching q-gram in the second string can only lead to at most $k + 1$ matching q-grams. To fully resolve the problems caused by repeats one can introduce an additional restriction on the hits in a potential match by treating groups of hits that share the same position in P or S as one hit. However, it can be quite time-consuming to check these further restrictions and it remains questionable whether this would increase the overall speed of an implementation. We conducted a small set of experiments that suggested the opposite.

As a conclusion we would like to mention that while further restriction of the match region may be possible, it appears questionable whether the increase in filtration efficiency justifies the increased complexity in evaluating the filter criterion.

4.4.2 Filtering with the Pigeonhole Principle

Another very simple filter algorithm uses a filter criterion based on the pigeonhole principle (see for example [NBY00]). If looking for approximate matches to a query P that have an edit or Hamming distance of at most k from P, one can cut P into $k + b$ segments and search for each of these pieces in the database S. Since only matches with an edit or Hamming distance of at most k from P are interesting, one can argue that in the worst case the k errors will be distributed across any k segments of q. This means that b of the $k + b$ pieces will be unaffected by errors. By locating all exact matches of the $k + b$ segments of P in the database S and searching for regions of the database that contain at least b of them in a valid arrangement it is possible to determine all locations of potential matches. There are multiple degrees of accuracy with which these valid arrangements can be described. Again they provide a tradeoff between time required for the evaluation of the filter criterion and the filtration efficiency. The value of b influences the length of the segments and therefore has a strong influence on the behavior of the filter.

The most straightforward definition of the match region for locating potential matches is a substring of S of length m for the k-mismatches and $m + k$ for the k-differences problem. As in the case of the q-gram lemma this would require examining a very large number of substrings. Another alternative is similar to the one introduced in the original q-gram paper by Jokinen and Ukkonen[JU91]. One only considers overlapping blocks in S in order to reduce the number of checks that have to be performed. The best choice of the block size again depends on the actual implementation.

A second method to define the match region more precisely is quite similar to the one described for the q-gram filtration. It takes into account that in order for the b matching segments to represent a true match, they have to be arranged in the order in which they appear in P. When using this filter for the k-mismatches problem this means that they all have to be located on the same hit diagonal and for the k-differences problem they can occupy up to $k+1$ adjacent hit diagonals.

Again one could potentially introduce further restrictions that increase the filtration efficiency at the cost of additional computational effort. It is also important to note that degenerate cases like those described for the q-gram lemma can again impact the filtration efficiency and may have to be taken into account in some applications.

As a side note it should be mentioned that this filter is a case of q-gram sampling as described in Section 4.4.1 since it only uses a subset of all q-grams in P.

4.4.3 BLAST

The program BLAST is an implementation of an approximate string matching algorithm that is very popular in computational biology. The underlying algorithm was proposed in 1990 [AGM+90] and refined for protein databases in 1997 [AMS+97].

There are two publicly available implementations of this algorithm, NCBI-BLAST and WU-BLAST. Both offer two different versions of BLAST for databases containing DNA (the 4 letter nucleotide alphabet) and amino acids (a 20 letter alphabet representing the different amino acids). While for DNA sequences the edit distance is a quite valuable similarity measure this is not the case for amino acids since there are several sets of amino acids which are quite similar in their chemical and physical properties. Amino acids from the same set will therefore be easier to match in an alignment than those from different sets. The simple edit distance can not take this special property of the amino acids into account, so more complicated similarity measures based on matrices have to be used. Therefore we will only describe the filter technique implemented in `blastn`, the program designed for matching DNA strings. For simplicity we will still refer to it as 'BLAST' in the following.

BLAST uses a very simple filter technique. Given a query P and a target S the user selects a value q that determines the *word length* used by BLAST. First, BLAST generates the set of all substrings of P with the length q in a similar way to the q-gram lemma. BLAST then proceeds to locate all hits, i.e. occurrences of substrings from this set in S. Sets of overlapping q-grams are collapsed into single, longer strings. All matching substrings remaining after this procedure are passed to a verification stage as potential matches. In principle the BLAST filter criterion consists of locating all hits of q-grams between P and S.

In order to locate the matching q-grams in S, BLAST constructs a *deterministic finite automaton* (DFA) [Mea55, HU79], which recognizes all q-grams in P. This DFA is used to conduct a serial scan of the string S creating a sequentially ordered list of matching q-grams while collapsing overlapping matches into single matches on the fly. This returns the list of all potential matches between P and S. In the verification stage BLAST uses an iterative extension technique to enlarge potential matches into alignments. In each iteration all current potential matches are extended in both directions by a fixed amount of characters. Those that do not fall below a cutoff similarity value are considered interesting and passed on to the next iteration of the extension. This continues until all remaining potential matches are true matches.

4.5 Using Index Structures with Filter Algorithms

Filter algorithms rely on different methods to evaluate their filter criterion across S. Current algorithms can be divided in two groups. The first group operates only on the two strings P and S. The two strings are not preprocessed in any way and in order to successfully compare both strings these algorithms may have to access all characters in both strings. The second group of methods preprocesses the string S in order to obtain information about its structure and contents. This information is then stored in a data structure which is called an *index structure* or short *index*. The preprocessing is done only once and is independent of P. When a query string P has to be compared with S these algorithms use the index to speed up the evaluation of the filter criterion and with it the detection of potential matches.

We call algorithms which use a preprocessing step to collect information about S and store it in an index structure *indexed algorithms*. Examples for such algorithms can be found in [JU91, Mye94, HS94, ST96, NBY98, NBYST01]. An index can often speed up the evaluation of the filter criterion by providing information about S in a readily available format. The downside of this approach is the time required for preprocessing as well as the extra space consumed by the index. Nevertheless several filter algorithms can benefit from using an index allowing them to run much faster. Of course, indexes only make sense when one wants to compare a larger number of queries with a string S and when the overhead for preprocessing is small compared to the time saved by using the indexed approach.

In the following sections we describe how the filters we introduced can benefit from an index. The basic idea behind all three filters is to locate match regions in S that contain one or more hits, i.e. pairs of exact matching substrings of P and S. While these hits can, of course, be located with a sequential scan of S, an index can be used to store information about them and provide fast access to all hits for a given q-gram in P. We name this set of hits a *hit list*. In the previous chapter we introduced different data structures for exact string matching and exact string matching with a fixed length q. They can be used for off-line filter algorithms and provide fast access to the hit lists of a given query P.

Processing the set of the hit lists of all necessary q-grams with the goal of identifying match regions that contain potential matches is a non-trivial problem. Its complexity depends on the type of filter and match region being used. We describe different approaches to solve this problem in the following section.

4.5.1 From Hits to Potential Matches

All three filter algorithms we described so far have to analyze the hit list of some or all q-grams in the query P in order to detect the potential matches in S. While this problem was not the main focus of our work it still represents an important point and therefore has to be considered in some depth.

BLAST uses a sequential scan to access the set of all hits between q-grams in P and S. The main drawback of this technique is the overhead introduced by examining every character in S. However, it also has some advantages. The sequential scan yields a list of all interesting hits that is sorted with respect to their position in S. This facilitates the identification of overlapping hits which can then be collapsed in order to speed up the verification process. It would also be possible to combine BLAST with an index that provides fast access to the hit lists of all q-grams in P (in fact the authors of the initial BLAST paper [AGM$^+$90] also evaluated this option but discarded it due to lower speed). However, this set of sorted lists of hits is somewhat harder to process than a single sorted list. For high values of q where the total number of hits is small, an index will be very useful, but for smaller values the sequential scan can be faster. The value of q for which one approach becomes better than the other depends on the actual implementation. BLAST uses a byte-compression technique to store 4 letters of DNA in one byte which allows a quick comparison of 4 characters at once and speeds up the scan by roughly a factor of 4 when compared to the trivial storage of one base per byte.

For the filters based on the q-gram lemma and the pigeonhole principle it is possible to locate all hits involving the respective q-grams in P with a sequential scan. In principle the general rule about larger values of q favoring indexes holds for these filters as well. For applications that only require a fairly low sensitivity one can use high values of q in both algorithms which leaves indexes as the obvious solution. In contrast to the BLAST filter criterion however, the fact that there is a different hit list for each of the required q-grams from P poses a somewhat bigger problem here since match regions containing at least t hits must be located. For $t > 1$ this introduces an additional level of complexity since all hit lists have to be considered as a whole. We would like to present two fundamentally different approaches in the following.

The first idea is similar to bucket sort. If we use fewer, longer substrings of S as match regions and only need to locate those that contain at least t hits, we can use a simple array that contains an integer counter for each of the substrings in S. We then proceed to process all hit lists and increment the counters of the block(s) in which a hit is located. This method is simple to implement and highly efficient. It also circumvents the problems caused by several independent hit lists. However, the block counters discard all information about the hits in the corresponding substring. This means that in verification phase it is necessary to locate the interesting regions in the substrings reported as potential matches a second time. It is possible to save additional information about the hits in each substring in a separate data structure in order to pass more information on to the verification phase but this is of course only interesting for verification algorithms that can actually use this information. It is also not necessarily the case that storing this additional information leads to a faster overall runtime than re-examining the substrings reported as potential matches. The runtime of the whole approach is linear in the total number of hits which have to be processed and is therefore a good candidate for large hit lists, i.e. smaller values of q.

A different method is based on sorting all hits with respect to either their position in S or their hit diagonal. This allows identifying potential matches in a sequential scan of the sorted list of all hits. While a conventional sorting technique for the hits leads to a higher asymptotic runtime than for the block counter based algorithm, the main advantage of this method is that one can retain all information about the hits and pass it on to the verification phase together with the potential matches. Depending on the algorithm used for verifying the potential matches this can lead to improvements in this stage. Since the time required for sorting the hits is the main drawback of this approach, it is better suited for smaller hit lists, i.e. higher values of q.

Again the most sensible approach is governed by many factors of the actual implementation that include the choice of the verification algorithm and the value of q as well as the chosen filter. In the Section 6.2 we provide some results that illustrate the behavior of the block array approach.

4.6 From Global to Local Matching

Until now we have only considered the k-mismatches and the k-differences problem. However, we are also interested in the local versions of these problems. Here we introduce a simple but efficient method to adapt solutions of the global problems to their local versions.

A naive solution would be the separate application of the global algorithm to all substrings of length w of the pattern P. The second alternative is to apply the global algorithm to P only once but use a threshold t which was computed for a shorter pattern length w, where w is not the length of P as it normally would be, but the minimum length of the substrings that are to be detected. Basically this means that we have to process all relevant q-grams from P not just those from a substring of length w. However, this also allows us to treat a complete pattern P as one query instead of several.

The second approach is obviously much faster but has two drawbacks. A longer query P will contain a larger number of different q-grams which leads to an increase in the number of hits that need to be processed. This will increase the number of potential matches due to the higher probability of detecting t hits in a match region. Our experiments showed however, that for reasonable differences between the substring length w and the pattern length P (in our case a factor between 6 and 12) the increase in the number of potential matches was acceptably low. For larger queries it would however be necessary to separate them in shorter segments and handle these separately. The optimal segment length in practice depends on the actual implementation. We did not conduct a thorough analysis of the segment length choice in this work but for queries with $|P|$ about 10 times higher than w this approach still worked very well if combined with the *sliding window* technique described in the next section.

4.6.1 A Sliding Window

Depending on the filter and on the method used for identifying the potential matches one can sometimes use hybrid strategies that require little extra effort but produce the same results as the naive approach. We will introduce a new technique based on sliding a 'window' of length w across the query P in this section.

For the q-gram Lemma used in conjunction with blocks as match regions we developed an idea that only introduces an overhead of a small constant factor. Instead of simply adding the hits contained in all hit lists of all q-grams in P to the block array, one proceeds in two stages. In the first stage one only processes the hit lists of the first $w - q + 1$ q-grams in P, i.e. those corresponding to the string $P(1, w)$, and records all blocks that contain at least t hits as potential matches. One can check this by examining every counter after incrementing it. Those that reach a value of exactly t correspond to potential matches and have to be stored as such. In the second stage one first removes all hits in the hit list of the first q-gram in P from the block array and then adds all hits in the hit list of the $(w-q+2)$-th q-gram. This has the effect that the block array now contains only the hits for $P(2, w+1)$. Again one can check for counters that reach the value of t when incrementing them and store the corresponding blocks as potential matches. To avoid storing a block more than once, we decided to never decrement counters that reach a value of t or more during the whole process. This ensures that no block ever gets assigned the value t more than once and will therefore only be recorded as a potential match once.

The technique we describe above requires accessing all hit lists twice. Once for adding the hits and once for removing them again. However, even this apparent drawback can also be used to avoid re-initializing the block counter array after each search. If all hits that are added at some point are also removed later on, all values of the block array will be zero again after the whole process. The only exceptions are those that correspond to blocks with potential matches. However, one can simply scan through the list of potential matches and reset the respective counters as well to get around this problem. For instances where the total amount of hits is smaller than the size of the block counter array this can even reduce the overall runtime of the whole algorithm. We implemented this technique in QUASAR, our software that was designed for EST-Clustering.

4.6.2 Modified Match Regions

Another idea that can improve the simple and efficient approach described above is the use of more specific match regions. One can for example assign separate block arrays to distinct substrings of P that overlap each other by $w-1$ characters. For long queries this is a solution to the problems caused by the large number of q-grams in P. If using one or $k+1$ diagonals as a match region, one can divide these into overlapping segments using the same idea. However, for the query lengths we used, the decrease in filtration efficiency caused by the additional q-grams was negligible which meant that these ideas remained untested in practice. It is definitely an interesting open question to determine good parameter ranges for different techniques either experimentally or theoretically.

4.7 Experimental Evaluation of Filter Algorithms

In order to compare different filters we had to find parameters which describe various aspects of a filter. The most important properties are of course the quality and the speed of a filter. Algorithms which employ an index structure should also be judged with respect to space efficiency of the index. In addition to the parameters described in Section 4.4 we will discuss the methods we used to experimentally evaluate the different filter algorithms in more detail. In the following we introduce the parameters we used to measure the quality and speed of filters experimentally.

4.7.1 Filter Speed

When it comes to comparing the speed of different filter algorithms, one can either implement them and measure the actual runtimes of a set of experiments with test databases or derive an implementation independent method to characterize the speed of a filter algorithm. We decided to use the second option for several reasons.

The main reason was the fact that all algorithms we examined share the same method of accessing their data, the hit lists of a certain amount of q-grams in P. They only differ in the number of hit lists and in the actual length of the hit lists. Apart from this, two of the algorithms face similar problems when it comes to processing the hit lists and detecting match regions that represent potential matches. In one respect however, the filter method used by BLAST is simpler than the filters based on the q-gram lemma or the pigeonhole principle. It does not require locating more than one hit in a certain region since every hit constitutes a potential match candidate. This gives the BLAST filter algorithm a certain edge with respect to speed that could be worth exploiting in a modified version of BLAST which uses an index instead of a sequential scan to locate all hits of q-grams from P in S. For comparing the filters with each other we decided to simply collect the information about the number of hit lists that needed to be accessed and the actual number of hits that were contained in the sum of all hit lists. We denote as $H(P)$ the set of all hits for all used q-grams from P. Since the access to a hit list can be achieved in time $O(1)$ given an ID and every hit has to be accessed at least once, the size of $H(P)$ provides useful information about the speed of the filter algorithm. The actual speed of an implementation may vary depending on the quality of the implementation and the algorithm used to actually process the hit lists. We were mostly interested in a relative comparison of different filters not in absolute values, so this was a good measure of filter speed. It also allowed us to conduct a comparison that was free of any bias induced by possible differences in the implementations of different filters. In Section 6.3, for each filter algorithm and each set of parameters for a given experiment we will provide $|H(P)|$, the sum of the length of all hit lists for a given query and use this number to judge the filter speeds.

There is a simple connection between the value of q and the expected number of hits for a certain q-gram. If we assume every q-gram to be equally likely to occur in our database S then the expected number of hits for any q-gram from P is

$$\frac{1}{|\Sigma|^q}(|S| - q + 1)$$

This is of course only an approximate value due to the uneven distribution of q-gram frequencies in typical databases. In fact for natural language texts it is a very bad estimate due to the highly uneven distribution of q-grams. While for DNA databases it is still only an estimate, in this case the word distribution is far more regular and the predicted values provide valuable information about the expected behavior of a filter. Furthermore, we only use the expected size of $H(P)$ for relative comparisons of different filters. Therefore the correctness of the absolute value is of lesser importance.

4.7.2 Filtration Efficiency

It can be quite difficult to accurately describe the behavior of a filter for a given set of parameters. These parameters include the length of P, w for the local problems, the similarity measure being used, the value of k which describes the maximum distance for which the filter is guaranteed to report all matches and the definition of the match region being used. While the value of k for which a filter is guaranteed to be lossless can be selected by the user, this number does not fully characterize the typical output of the filter.

A second very important aspect of filter quality is the filtration efficiency as defined on page 29. Ideally the set of potential matches returned by a filter would be identical to the set of true matches. Unfortunately this is usually not the case for the filters used in approximate string matching. Typically they also return a certain amount of false positives. This amount varies from filter to filter and it is connected to the selected value of k. We measure it by using the filtration efficiency. It has a big impact on the overall runtime of filtration and verification phase.

Therefore an important goal in defining the properties of different filters lies in predicting and measuring their filtration efficiency. Assuming a small number of true matches, this is nearly equivalent to predicting the percentage of the database that the filter will discard as irrelevant. This in turn is directly connected to the number of potential matches detected by the filter. While the experimental analysis of this value is straightforward, we also wanted to be able to predict filtration efficiency for a given filter in order to allow a theoretic evaluation of filters for the same reasons as stated in the analysis of filter speed.

4.7.3 The Minimum Coverage

In this section we want to present a property of a filter that is closely related to the filtration efficiency. We describe it with the purpose of using it as a guideline for predicting the filtration efficiency of a filter. While it is always possible to conduct experiments and measure the filtration efficiency of a filter, it is also of great interest to be able to predict the behavior of a given filter using a certain set of parameters. An important reason for this is the quick comparison of a large number of different filters. This will play an important role in the next chapter of this thesis.

We want to start with the observation that for any given potential match there exists a set H of at least t hits. This set of hits clearly defines which characters in P and S have to match. Each hit is by definition a pair of matching substrings in P and S and therefore the corresponding characters match as well. This leads us to the definition of the *cover* and the *coverage*.

Definition 4.7.1: Given two strings P and S, a value q for the length of the q-grams used, and a q-gram hit $h = (i, j)$ between P and S, we call the set of characters positions that are defined as matching by this hit, i.e. $i, \dots, i+q-1$ and $j, \dots, j+q-1$ the *cover* C of that hit for P and S respectively. We write it as:

$$C(h, q, P) = \{i, \dots, i+q-1\}$$

$$C(h, q, S) = \{j, \dots, j+q-1\}$$

We say that C *is the cover of* h *in* P (respectively S). We call the cardinality of these sets their *coverage* c:

$$c(h, q, P) = |C(h, q, P)|$$

$$c(h, q, S) = |C(h, q, S)|$$

One can expand this definition to sets of hits. Assuming a set H of hits $h_1, \dots, h_n$ between P and S we define the cover H as the union of the covers of all hits in H:

$$C(H, q, P) = \bigcup_{k=0}^{n} C(h_k, q, P)$$

$$C(H, q, S) = \bigcup_{k=0}^{n} C(h_k, q, S)$$

Again we call the cardinality of these sets their coverage c:

$$c(H, q, P) = |C(H, q, P)|$$

$$c(H, q, S) = |C(H, q, S)|$$

Depending on the actual strings P and S, the cover of a given set of hits H in P can differ significantly from the cover of H in S. The coverage can also vary.

Turning back to potential matches, i.e. sets of t hits, we now have a terminology for describing the matching characters in P and S for any such potential match since a potential match is nothing more than a set of hits. It is therefore also possible to talk about the cover and coverage of a potential match.

We now want to introduce a distinction between a practical and an abstract view of the strings P and S. In the first case, we work with actual, specific strings when talking, for example, about possible arrangements of hits. In the second case we assume that for both P and S all characters in these strings are pairwise different from all other characters in the same string. We do this to emulate the general case where we have no previous knowledge about the two strings and therefore have to consider all characters as potentially different from each other. This assumption has an important effect on the hits between two strings P and S. Since all characters in one string are considered distinct from each other, it is impossible that two hits share a starting position in either P or S. Of course this is not true in practice since substrings can occur repeatedly in either P or S.

The second view of strings simplifies some aspects of the analysis of the filtration efficiency because one can disregard certain degenerate special cases involving repeats. We base the following analysis of general filter behavior on it even though this will neglect the above mentioned special cases. However, we also discuss the aspects of the practical view of P and S that lead to the special cases and how they affect our general analysis.

A lossless filter will return a set of potential matches that contains all true matches but usually also some false positives. In order to quantify the expected number of false positives we need to analyze the cover of such false positives. The false positives are in principle sets of t or more hits that lie in a match region. These random potential matches are likely to have a low number of hits and a small coverage in S. The most likely case will usually be that of a potential match with exactly t hits that are arranged such that the cover in S is minimal, i.e. the set of characters in S that are 'fixed' are as low as possible.

We define the *minimum cover* and the *minimum coverage* as follows:

Definition 4.7.2: Given a value q for the length of the q-grams used, a threshold t, two arbitrary strings P and S for which

$$\forall i, j \in \{1, \ldots, |P|\} : p_i \neq p_j$$

$$\forall i, j \in \{1, \ldots, |S|\} : s_i \neq s_j$$

and the set

$$H^* = \{(i, j) | i \in \{1, \ldots, |P| - q + 1\}, j \in \{1, \ldots, |S| - q + 1\}\}$$

of all possible hits, we define the *minimum coverage* c_m by minimizing across all sets H with t hits as follows:

$$c_m(q, t, P) = \min_{H \subseteq H^*, |H| = t, \forall (i,j),(k,l) \in H : i \neq k, j \neq l} c(H, q, P)$$

$$c_m(q, t, S) = \min_{H \subseteq H^*, |H| = t, \forall (i,j),(k,l) \in H : i \neq k, j \neq l} c(H, q, S)$$

A *minimum cover* C_m of the two strings is a cover with minimal coverage.

Under the assumption of pairwise different characters within the strings P and S, a minimum cover is the arrangement of hits which requires the smallest number of matching characters. The size c_m of a minimum cover is an upper bound for the minimum coverage of all possible strings. For specific pairs of strings P and S the actual cover of t hits can be smaller than the minimum cover. However, there exist no pairs of string P and S for which no arrangement of t hits are possible that have a cover with size less than or equal to c_m.

For certain strings P and S and sets of hits H, the actual cover of H in P and/or in S can be smaller than the minimum cover. The reason for this lies in repeated sequences in either P, S or both strings. Such sequences that occur more than once in the same string are called *repeats*. A good example for a small cover in S would be the case where a certain q-gram is repeated t times in P but occurs only once in S. This would lead to t hits for which

$$\forall (i, j), (k, l) \in H : j = l$$

In this example the cover of H in S would be of size q (since the q-gram occurs only once) and the cover of H in P would be $q+t-1$ (since the q-gram occurs t times in P).

In general, strings of length q with a short period can lead to overlapping repeats. Again a simple example would be a substring of length $q+t-1$ in P that consists of only one character from Σ and a substring of length q in S that also contains only this character. This would also lead to t hits. In practice such cases are usually either rare or irrelevant. In computational biology, for example, ubiquitous repeats are usually filtered out in a process called repeat masking in order to avoid large numbers of true matches that are basically meaningless due to their abundance in genetic sequences.

For the q-gram lemma the minimum coverage can be computed as

$$c_m = q + t - 1$$

For the pigeonhole principle

$$c_m = b * q$$

since the q-grams from the query are non-overlapping. For the BLAST filter criterion

$$c_m = q$$

If, for example, $P =$ `ACAGCTTA`, $S =$ `TCAGCTGC` and one uses the q-gram lemma with $q = 3$ to locate potential matches with at most one difference ($k = 1$) between P and S, one has to find at least $t = |P| - q + 1 - kq = 8 - 3 + 1 - 3 = 3$ matching 3-grams between P and a match region in S in order to consider it a potential match for P. The configuration of three 3-grams that minimizes the number of matching characters between two strings is the one where the starting positions of the three matching 3-grams in P are arranged right next to each other. In our example the substring `CAGCT` of P and S contains exactly three matching 3-grams with five matching characters.

We conjecture that the minimum coverage is a good number for approximate predictions about the filtration efficiency of the different filters. Similar to the expected amount of hits for a given q-gram we expect a certain amount of potential matches to occur by chance in a string S. For a minimum cover C_m of length $\max\{C_m\} - \min\{C_m\} + 1$ there are $|S| - \max\{C_m\} + \min\{C_m\}$ possible placements in S. At each of these positions a potential match with c_m matching characters is possible and its probability is proportional to $\frac{1}{|\Sigma|^{c_m}}$. Hence the expected number of false positives a filter will return is proportional to the following term:

$$\frac{1}{|\Sigma|^{c_m}}(|S| - \max\{C_m\} + \min\{C_m\})$$

We analyze the prediction quality of our conjecture in the experiments chapter in Section 6.3.2. It appears to be the case that even though we neglect false positives with a higher coverage than c_m and we also ignore repeats, the minimum coverage can be a valuable tool for the analysis of filtration efficiency of different filters and for their relative comparison.

4.7.4 A Closer Look – Match Zones

Apart from the filtration efficiency, the value of k plays a major role in describing the behavior of a filter algorithm. It defines an error level up to which a lossless filter reports all approximate matches in the string S. For many applications there is sufficient time available to use a lossless filter. However, it can also be interesting to use a lossy filter or use a lossless filter and also look at 'good' false positives, i.e. potential matches that have slightly more errors than k. While high values of k will result in slower algorithms and one may be unable to afford the increased runtime to report all matches for such a high value of k, one can instead use a lower value of k and still report all those potential matches that have slightly more than k errors. The filter will probably miss a certain percentage of those matches but the ones it finds may be valuable nevertheless. BLAST is an example for this strategy. While the values of k for which BLAST is guaranteed to report an approximate match are relatively low, there is still a high probability of finding matches with more than k errors. This probability of course decreases with the increase in the number of errors.

In order to examine the behavior of filter algorithms for error levels above k we decided to use the following descriptions of filter performance at different error levels. We defined three different *match zones*, error or distance ranges for which a filter algorithm behaves differently. Each of them is a pair of a lower and upper bound representing the smallest/largest number of errors/distance for which a filter functions according to the following definitions.

The first zone is defined as the range of errors or distance for which the filter will never miss any matches, i.e. where all true matches between P and S that lie within this range are reported by the filter as potential matches. We call this zone the *lossless zone*. For the lossless filters we introduced it represents the set of strings that are within a distance k from P and are always recognized as potential matches.

The next zone is the range of errors for which the filter will only report a certain fraction of all approximate matches between P and S that have a distance within this range. We call this zone the *lossy zone* and the percentage of all matches recognized by a filter the *recognition rate*. Depending on the filter, parts of this zone can still be interesting for practical use, especially if the recognition rate for a given error level e is very high. BLAST is typically used to search for matches in the lossy zone. While not all true matches will be found, a large portion will be and that alone can be sufficient for various applications.

The third zone is the range of errors for which a filter will not find any matches at all. We call this zone the *negative zone*.

Determining the recognition rate of a filter algorithm for a given error level or distance e is by no means trivial. While it is usually fairly simple to determine the borders between the zones using k and the minimum coverage, we were unable to make theoretic predictions about the behavior of a filter in the lossy zone. However, we developed an experimental set-up that allowed us to gain insight into aspect of a filter. The recognition rates in the lossy zone directly influence another property of the filter, its ability to distinguish between matches and non-matches. When comparing the filtration efficiency and the recognition rates of different filters it is necessary to take into account the parameters of the filter, i.e. the intended value of k and the query length.

4.7.5 Describing the Match Zones

Determining the amount of errors or the distance bound up to which a given filter will still include all true matches in the list of potential matches is fairly trivial for the filters described above. It is by definition the level of similarity that defines the border between the lossless and the lossy zone. For the filters we introduced in this chapter it is equal to k.

While for most filters it is quite trivial to come up with the similarity level that defines the border between the lossless and the lossy zone, things get a little trickier for the border between lossy and negative zone. In principle the minimum coverage yields a good approximation since for repeat-free strings the minimum coverage is the minimum number of matching characters required for a match. This in turn means that pairs of repeat-free strings with a distance higher than $|P| - c_m$ will never be recognized as potential matches. Of course it is possible, depending on the match region being used, that due to the existence of repeats, potential matches will be found with an even higher distance. See Section 4.4.1 for a discussion of such degenerate cases. Less stringent match regions will increase the tendency towards potential matches with more than $|P| - c_m$ errors.

Figure 4.6 shows a schematic containing the three match zones for a typical filter based on the q-gram Lemma.

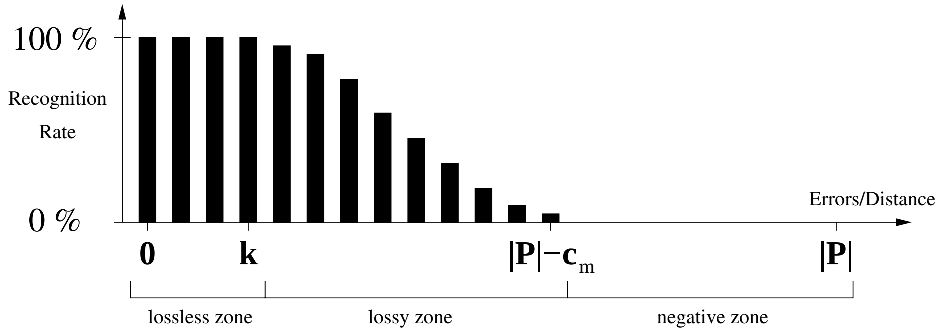


Figure 4.6: The three match zones with typical recognition rates for the q-gram Lemma.

We would like to make some observations to support the understanding of this graph. First of all, for each error value e and a given string P there is a set of strings with a distance of exactly e. Often this set is called the e-*neighborhood*. The recognition rate of a filter algorithm for a string P at this error rate is then the fraction of all strings in the e-neighborhood that would be recognized as potential matches by the filter. For $e \leq k$ this rate is 100% for a lossless filter. In the lossy zone it drops below 100% towards 0% at the border to the negative zone. The second observation is the size of the e-neighborhood. It grows nearly exponentially with e, i.e. the amount of strings in the neighborhood sets for high values of e is much higher. This in turn means that even the low recognition rate for values of e that are close or equal to $|P| - c_m$ multiplied with the size of the e-neihgborhood is far larger than the same product for lower values of e. In practice this means that even though the recognition rate for high values of e is relatively low, most randomly occurring false positives have c_m or close to c_m matching characters.

While principle it is possible to compute this figure given a specific string P and a filter algorithm, we are not aware of an efficient method (full enumeration of all possible strings in each e-neighborhood is only feasible for relatively small values of $|P|$ and/or e).

In addition to these theoretical considerations, we also wanted to conduct an experimental analysis of the match zones. For this purpose we constructed databases with a certain amount of sample sequences for each error level between 1 and $P - q$. We then used different filters for searching in these databases and recorded the percentage of the samples that were recognized for each error level e. This allowed us to produce approximate versions of the theoretical figure. The results of these experiments including a discussion can be found in Section 6.5.

4.8 QUASAR

An important part of our work was the development and implementation of QUASAR, which stands for **Q**-gram **A**lignments using a **S**uffix **AR**ray. It was specifically designed for efficient EST clustering and is based on the q-gram lemma and the word lookup table as well as the idea of a block array, all introduced in the paper by Jokinen and Ukkonen in 1991 [JU91]. However, we were able to develop several improvements of the approach described by the authors of this paper. The three main improvements were the introduction of a more flexible index structure, the use of an increased block size to decrease the runtime of the algorithm and the adaption to local approximate string matching using a sliding window. In the following we present an overview of the algorithm and recap the techniques used and some additional implementation details worth noting. We start with a pseudocode description of the implementation.

The use of a simple word lookup table to locate q-grams in the database S is sufficient for a fixed set of parameters. However, during the development of QUASAR we wanted to experiment with different values of q without having to reconstruct the whole index every time. Due to space constraints it was impossible to store separate indexes for all interesting values of q. This lead to the use of a suffix array as a word lookup table. We already described this idea in Section 3.4.1. The two main benefits are increased index flexibility and the extensibility to higher values of q using the hybrid approach described in Section 3.4.3.

The second improvement was prompted by the observation that using a block array with blocks larger than the $2(m-1)$ proposed by Jokinen and Ukkonen can speed up our algorithm significantly [BCF+99]. We stuck to the idea of Jokinen and Ukkonen of using blocks that overlap each other by half the block size in order to be able to locate potential matches that cross block boundaries. It would in principle be possible to use blocks that only overlap each other by $m-1$ characters. However, the increased simplicity of blocks overlapping by $\frac{b}{2}$ characters saved more time in practice than the increase in verification time resulting from the larger potential matches cost. The choice of the block size b allowed us to obtain a tradeoff between filter speed and filtration efficiency as larger values of b lead to a smaller number of blocks in S which in turn decreased the runtime of the filtration phase. The downside of this approach is that each potential match the filter returns becomes larger and requires more time for processing in the verification phase. Also the expected number of matching q-grams for one block increases and leads to a larger number of blocks that contain at least t matching q-grams. This tradeoff effect is very useful for optimizing the combination

Algorithm 2: Filtering with the QUASAR

```
Set all hit counters for blocks of the block array to 0
for i = 1 to w − q + 1 do
  for all hits h of the q-gram starting at position i in P do
    for all blocks y that contain h do
      Increment the block counter of y
    end for
  end for
end for
for i = w − q + 2 to |P| − q + 1 do
  for all hits h of the q-gram starting at position i − w + q − 1 in P do
    for all blocks y that contain h do
      if block counter of y < t then
        Decrement the block counter of y
      end if
    end for
  end for
  for all hits h of the q-gram starting at position i in P do
    for all blocks y that contain h do
      Increment the block counter of y
    end for
  end for
end for
for all blocks y do
  if block counter of y ≥ t then
    Consider block y a potential match
  end if
end for
for all potential matches p do
  Use BLAST to verify p
end for
```

of filtration and verification phase for a given set of parameters and even for different algorithms used in the verification phase. In Section 6.2.1 we take a closer look at the influence of the block size on the runtimes of the filtration and verification phase.

The third new idea we developed for our implementation was the use of a sliding window to locate local approximate matches. When matching EST sequences, one usually wants to detect terminal overlaps between two sequences. These overlaps are often just a fraction of the whole EST. This caused us to extend the initial approach to local approximate string matching. We created and implemented an algorithm that sequentially traverses the query P and at any point only considers a substring of length w of P. While this technique, described in more detail in Section 4.6.1, leads to a certain overhead, it allows us to use QUASAR for local approximate string matching and not just for the global variant.

For the verification phase of QUASAR we used BLAST to simply match P against the substrings of S that corresponded to the blocks with at least t hits. This solution turned out to be quite efficient in practice, even though the amount of time required to adapt BLAST for use as a C function within our own implementation was substantial.

Experimental results illustrating the performance of QUASAR can be found in Section 6.2. For our application QUASAR achieved a speed-up of a factor of roughly 30 compared to BLAST.

4.9 Concluding Remarks about Filter Algorithms

In this chapter we described three different filter criteria for off-line approximate string matching. We discussed several variants and details and introduced QUASAR, a powerful implementation of an algorithm based on the q-gram Lemma and a simple index that provides a highly efficient solution to the substring matching problem with k-differences. QUASAR also contains several new ideas to increase the overall performance of the implementation. Furthermore we defined measures, including the new minimum coverage, for analysing the behavior of different filters and comparing them with respect to filter speed and filtration efficiency.

Chapter 5

Introducing Gapped q-grams

After successfully combining several techniques into a practical algorithm for approximate string matching, we looked at all the components with the goal of improving them. While the approach we describe in the previous chapter is an efficient solution for relatively low error levels k, it quickly breaks down for higher values of k. The main reason for this is the connection between k and the values of q, the substrings used by the filter criterion. Higher values of k require lower values of q which has a direct impact on filtration speed since the expected length of a hit list grows exponentially with a decrease of q and the runtime of the filtration phase is at least linear in $|H(P)|$, the total number of hits for a query. We therefore decided to develop an improved filter criterion with increased speed and filtration efficiency.

In this chapter we will describe two new filter algorithms based on a modification of the q-gram lemma. We first introduce the basic concept behind the modification and then present a filter algorithm for the k-mismatches problem and one for the k-differences problem. Both algorithms can be applied to the local versions of these problems as well. We discuss the benefits of our algorithms and describe the modifications necessary to adapt existing algorithms to the new technique.

5.1 Gapped q-grams in Previous Work

A generalization of the q-gram method uses *gapped* q-grams. Basically they are patterns of length $s > q$ where instead of matching all characters one only matches q fixed characters and considers the gaps to be 'don't care' characters or wildcards. We call the arrangement of the gaps the *shape* of the gapped q-gram. For example, the gapped 3-grams of shape `##-#` in the string `ACAGCT` are `AC-G`, `CA-C` and `AG-T`. Gapped q-grams have been used by 3 groups so far [CR93, PW95, LST96]. In [CR93, PW95], the motivation is to increase the filtration efficiency of an approximate string matching algorithm by considering multiple shapes. Pevzner and Waterman [PW95] use 'regular' q-grams that match every $(k+1)$-st character together with ordinary contiguous q-grams for the Hamming distance. Califano and Rigoutsos [CR93] describe a lossy filter that uses as many as 40 different random shapes. Their approach is effective for high k but they need a huge index (18 GB for a 100 million nucleotide DNA database). The Grampse system of Lehtinen et al. [LST96] uses a shape containing every h-th character for some h (similar to [PW95]) for *exact* matching. Their motivation of using gapped q-grams is to reduce dependencies between the characters of a q-gram.

In contrast to the motivation of previous work on gapped q-grams, we studied gapped q-grams very thoroughly in order to show that they have a more fundamental advantage over *contiguous* (i.e. ungapped) q-grams. When used correctly, even single gapped q-grams can provide much faster and/or more efficient filtering. It is not at all necessary to use sets of q-grams to achieve this. In fact we have observed improvements of several orders of magnitude in our experiments.

Our approach to gapped q-grams differs from previous work in two important ways. First, while certain well-defined shapes are used in [PW95, LST96] and randomly selected shapes in [CR93], we carefully select the shapes for use in filter algorithms. We performed an exhaustive search of all feasible shapes and chose the best. Our approach allows us to choose the best shapes with respect to speed and filtration efficiency. Good shapes typically possess little or no regularity, but are rare and significantly better than randomly chosen shapes. Some regular shapes, including contiguous shapes, are actually among the worst shapes. A similar observation was made by Califano and Rigoutsos [CR93], who observed in their experiments that randomly generated sets of shapes outperform deterministically generated sets. They also mention the idea of using optimized (sets of) shapes but did not provide any method for choosing the shapes beyond experimental comparison.

The second deviation from previous work is that we use an *optimal* and *shape-dependent* threshold. Pevzner and Waterman [PW95] give equations for the threshold of their regular shapes that guarantee losslessness but are optimal only in certain special cases. Califano and Rigoutsos [CR93] determine the threshold based on probabilistic calculations and experiments. Their threshold does not depend on the actual shapes and may lead to false negatives turning their filter into a lossy filter. We have not found a closed form formula for the optimal threshold of arbitrary shapes like the one provided by the q-gram Lemma. However, we present a simple definition of the optimal threshold and an efficient dynamic programming algorithm for computing it.

After we published our results, another group presented work that improves BLAST by using gapped q-grams [MTL02]. They also use only one type of gapped q-grams and present a method to select good shapes. However, the scope of their work is limited since they only consider a BLAST-like approach with a threshold value of 1.

The rest of this chapter is organized as follows. We first formally define shapes as a way of characterizing gapped q-grams. Then we describe a filter algorithm for the k-mismatches problem based on gapped q-grams and introduce a method to compute the optimal threshold for a single gapped q-gram. We explain how to compute the minimum coverage for gapped q-grams and use it together with the value of q to predict the filtration efficiency and speed of shapes used in our filter. We discuss the importance of the minimum coverage in the context of gapped q-grams. In the second half of the chapter we characterize the problems arising when adapting gapped q-grams to the k-differences problem and provide solutions that lead to a second filter algorithm. We present methods to predict filtration efficiency and speed for this class of filters. Finally we proceed to discuss the adjustments that have to be made to existing filter algorithms in order to adapt them to gapped q-grams and discuss the overhead incurred by these modifications.

5.2 Shapes

We start by introducing some basic terminology and notation on shapes and gapped q-grams.

Definition 5.2.1: Let I be a set of integers. The *size* of I, denoted by $|I|$, is the cardinality of the set. The *span* of I is $\text{span}(I) = \max I - \min I + 1$. The *position* of I is $\min I$, and the *shape* of I is the set $\{i - \min I \mid i \in I\}$. An integer set Q with position zero is called a *shape*. A shape Q with size q and span s is called a q-shape or a (q, s)-shape if we also want to denote its span.

For any shape Q and integer i, let Q_i denote the set with shape Q and position i, i.e., $Q_i = \{i + j \mid j \in Q\}$. Let $Q_i = \{i_1, i_2, \ldots, i_q\}$, where $i = i_1 < i_2 < \cdots < i_q$, and let $S = s_1 s_2 \ldots s_m$ be a string. For $1 \leq i \leq m - \text{span}(Q) + 1$, the *$q$-gram* at position i in S, denoted by $S[Q_i]$, is the string $s_{i_1} s_{i_2} \ldots s_{i_q}$. A q-gram is also called a (q, s)-gram, where $q = |Q|$ and $s = \text{span}(Q)$.

As an example, let $Q = \{0, 1, 3, 6\}$ be a shape. Using the same notation as above, Q is the shape `##-#--#`. Its size is 4 and its span is 7. Thus Q is a 4-shape and a $(4, 7)$-

shape. The string $S =$ ACGGATTAC has three q-grams: $S[Q_1] = s_1s_2s_4s_7 =$ AC-G--T, $S[Q_2] =$ CG-A--A and $S[Q_3] =$ GG-T--C. This is illustrated below.

```
S = A C G G A T T A C
    A C - G - - T
      C G - A - - A
        G G - T - - C
```

We say that two strings P and S of length m *share* a q-gram or have a *common* q-gram at position i if $P[Q_i] = S[Q_i]$. The *q-gram similarity* $s_Q(P, S)$ of P and S is the number of their common q-grams, i.e.,

$$s_Q(P, S) = |\{i \in \{1, \ldots, m - \mathrm{span}(Q) + 1\} \mid P[Q_i] = S[Q_i]\}|$$

.

Let Q and S be as in the example above and let $P =$ ACTGACTAG. Then $s_Q(P, S) = 1$, since P and S share only the q-gram CG-A--A at position 2:

```
P = A C G T A C T A G
    A C - ? - - T
      C G - A - - A
        G ? - ? - - ?
S = A C G G A T T A C
```

5.3 Gapped q-grams and the k-mismatches Problem

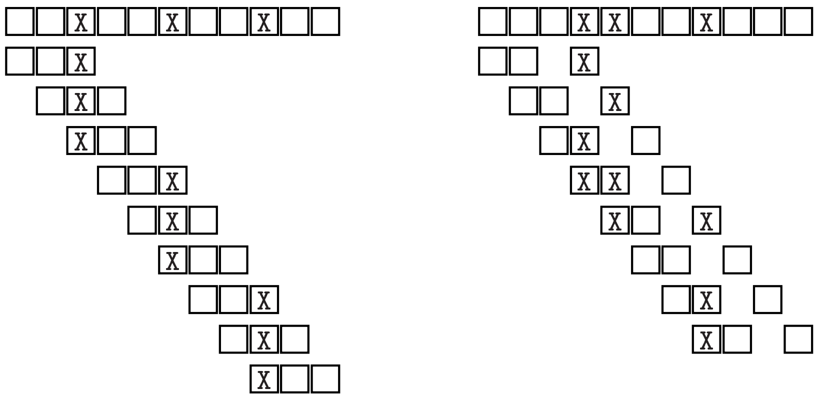
When we first took a closer look at gapped q-grams it quickly became obvious that while they are well-suited for approximate string matching using the Hamming distance, the case of the edit or Levenshtein distance is more complicated. The reason for this is the fact that a replacement in a gap will not affect a hit of a gapped q-gram but an insertion or deletion will. For this reason we first concentrated our efforts on the k-mismatches problem. We started with a simple analysis of the potential of gapped shapes in filters very similar to those based on the q-gram lemma. The only difference is the use of a certain gapped q-gram instead of an ungapped q-gram. This leads to complications in computing the threshold and the minimum coverage. We address these problems and present solutions in the following subsections. The contents of this section as well as some experimental results were published in 2001 [BK01].

5.3.1 The Threshold

The basic idea of a q-gram filter is to identify those match regions that contain a certain minimum number t, the threshold, of hits between P and S. The value of t is an important parameter for such a filter since it directly influences the number of potential matches that will be generated. It is desirable to use an optimal threshold, i.e. the highest value for t for which the filter remains lossless. In some applications lossy filters are interesting as well due to their increased speed. For these cases one can either use a value of k that is below the actually desired value and analyze the filter behavior in the lossy zone or one can simply increase t and evaluate the effects of the increase. We present a set of experiments concerning the lossy zone in Section 6.5, but restrict the scope of this chapter to lossless filters.

The q-gram lemma provides a simple formula for the optimal value of the threshold t for contiguous q-grams. A straightforward generalization for gapped q-grams with shape Q and at most k errors yields the formula $t = |P| - \mathrm{span}(Q) + 1 - |Q|k$ (see Lemma 5.3.3 below). Pevzner and Waterman [PW95] give this formula for their regular shapes. In addition to this they present a better formula for shapes with a span close to the length of the pattern. However, their general formula is not optimal for gapped q-grams as shown by the following example.

Let $|P| = 11$ and $k = 3$, and consider the 3-shapes `###` and `##-#`. The above formula gives thresholds zero and minus one, respectively, for the two shapes. Thus, neither shape would seem to be useful for lossless filtering in this case. However, the optimal threshold for the shape `##-#` is actually one. By an exhaustive search of all possible arrangements of three mismatches one can verify that at least one q-gram is always unaffected by the mismatches. The following figure contains an example of a worst possible arrangement of three mismatches for both shapes.



The above example already hints at how the optimal threshold value can be defined.

We formalize this next.

Let $P = p_1, \dots, p_m$ and $S = s_1, \dots, s_m$ be two strings of length m. Let $M(P,S)$ denote the set of positions where P and S contain the same character, i.e., $M(P,S) = \{i \in \{1, \dots, m\} \mid p_i = s_i\}$. Then $m - |M(P,S)|$ is the Hamming distance of P and S. Also, the q-gram similarity $s_Q(P,S)$ can be expressed using the set $M(P,S)$:

Lemma 5.3.1: $s_Q(P,S) = |\{i \in \{1, \dots, m - \mathrm{span}(Q) + 1\} \mid Q_i \subseteq M(P,S)\}|$.

The optimal threshold is by definition the largest value for which the filter is still lossless. In other words, it is the lowest q-gram similarity of two strings of length m with Hamming distance k. By Lemma 5.3.1, it is sufficient to consider the worst possible combination of the mismatches. Hence, we get the following general q-gram lemma.

Lemma 5.3.2: Let Q be a shape. For any two strings P and S of length m that are within Hamming distance k of each other, the q-gram similarity $s_Q(P,S)$ of P and S is at least

$$t_Q(m,k) = \min_{M \subseteq \{1,\dots,m\}, |M| = m-k} |\{i \in \{1, \dots, m - \mathrm{span}(Q) + 1\} \mid Q_i \subseteq M\}|.$$

Furthermore, for every string P of length m, there exists a string S that is within Hamming distance k of P, such that $s_Q(P,S)$ is exactly $t_Q(m,k)$.

We have not found a closed form for the optimal threshold in the general case. The following lower bound we already saw in the beginning of this section.

Lemma 5.3.3: $t_Q(m,k) \geq \max\{0, m - \mathrm{span}(Q) + 1 - |Q|k\}$.

Proof. Let M be the set minimizing the value of $t_Q(m,k)$. For each $j \notin M$ there is exactly $|Q|$ integers i such that $j \in Q_i$. Therefore, at most $k|Q|$ of the sets Q_i, $i \in \{1, \dots, m - \mathrm{span}(Q) + 1\}$, are not in M, and at least $m - \mathrm{span}(Q) + 1 - k|Q|$ are in M. $\square$

Tighter bounds can be provided for special cases. In particular, the classic q-gram lemma for contiguous q-grams yields the optimal threshold as shown by the following lemma.

Lemma 5.3.4: Let Q be a contiguous shape, i.e., $Q = \{0, \dots, q-1\}$. Then $t_Q(m,k) = \max\{0, m - \mathrm{span}(Q) + 1 - |Q|k\} = \max\{0, m - q + 1 - kq\}$.

Proof. The lower bound is shown by Lemma 5.3.3. Let $M = [1,m] \setminus \{q, 2q, \dots, kq\}$. Then Q_i is contained by M if and only if $i \in \{kq+1, \dots, m-q+1\}$. This shows the upper bound. $\square$

Using the lower bound of Lemma 5.3.3 (or any other lower bound) as the threshold guarantees the losslessness of the filter, but a non-optimal value reduces the filtration efficiency dramatically. In Section 5.3.6, we show that an increase of one in the threshold reduces the expected number of potential matches by a factor of at least $|\Sigma|$, where Σ is the alphabet, and in many cases by significantly more. The example on page 59 shows that the optimal threshold may be required to achieve any filtration at all.

In the next section we describe a dynamic programming algorithm for computing the optimal threshold. We have implemented the algorithm and used it, together with a pruning technique, to compute the optimal thresholds for a very large number of shapes with different values for m and k. For $m = 50$ and $k \in \{4, 5\}$ the tables 5.1 and 5.2 on pages 65 and 66 give the highest threshold among (q, s)-shapes for all combinations of q and s that have shapes with positive thresholds (except $q = s = 1$).

The tables show that our threshold example was not an isolated case: often, especially for higher values of q, the best gapped shapes have much higher thresholds than contiguous shapes of the same or even smaller size.

However, it is not sufficient just to have gaps; the shape has to be chosen carefully. For instance, for the parameters of the threshold example, $m = 11$, $k = 3$, $q = 3$, the shape `##-#` and its mirror image `#-##` are the only ones that have a positive threshold. As a more impressive example, for the parameter values $m = 50$, $k = 5$ and $q = 12$, there are just the two shapes, `###-#--###-#--###-#` and `#-#-#---#-----#-#-#---#-----#-#-#---#`, (and their mirror images) with a positive threshold. In most cases shown in Tables 5.1 and 5.2, only a few shapes achieve the highest threshold. This is also true for the minimum coverage. The distribution of threshold values and minimum coverage for a typical case is shown in Figure 6.7 on page 94. It is not always the case that the shapes with the highest minimum coverage have the highest threshold values and vice versa.

5.3.2 Computing the Optimal Threshold

The optimal threshold can be computed by an exhaustive search of all possible arrangements of k mismatches according to Lemma 5.3.2, but this is prohibitively expensive for large values of m and k. We have developed a significantly faster dynamic programming algorithm for computing the threshold.

Let us first define some notation. For a set I of integers let $I \oplus j$ denote the set $\{i + j \mid i \in I\}$, and define the operator $\ominus$ similarly. Let $[cond]$ be 1 if the condition $cond$ is true and 0 otherwise.

The computation is based on the following conditional threshold.

Definition 5.3.1: Let Q be a shape with span s. For non-negative integers m, $k \le m$, and a set $M \subseteq \{1, \dots, s-1\}$, $|M| \ge s-1-k$, let $t_Q(m, k, M)$ denote the optimal threshold for pattern length m and Hamming distance k under the additional condition that the matches in the last $s-1$ positions correspond to M. That is (cf. Lemma 5.3.2)

$$t_Q(m,k,M) = \min_{\substack{M' \subseteq \{1,\dots,m\}, |M'| = m-k \\ (M' \ominus (m-s+1)) \cap \{1,\dots,s-1\} = M}} |\{i \in \{1, \dots, m-s+1\} \mid Q_i \subseteq M'\}|.$$

The actual optimal threshold is easily computed by minimizing over the appropriate conditional thresholds:

Lemma 5.3.5: $t_Q(m, k) = \min_{M \subseteq \{1,\dots,s-1\}, |M| \ge s-1-k} t_Q(m, k, M)$.

The dynamic programming algorithm is based on the fact that the values $t_Q(i, *, *)$ can be computed from the values $t_Q(i-1, *, *)$. Note that it is not possible to compute $t_Q(i, *)$ from $t_Q(i-1, *)$. The matches in the last $s-1$ positions have to be recorded as well. The following lemma gives the detailed recursion.

Lemma 5.3.6: For $s \le i \le m$, $0 \le j \le k$ and $M \subseteq \{1, \dots, s-1\}$, $|M| \ge s-1-j$, we have

$$\begin{aligned} t_Q(s-1, j, M) &= 0 \\ t_Q(i, j, M) &= \min \begin{cases} t_Q(i-1, j-[s-1 \notin M], (M \cup \{0\} \setminus \{s-1\}) \oplus 1) \\ \qquad\qquad\qquad\qquad +[Q \subseteq (M \cup \{0\})] \\ t_Q(i-1, j-[s-1 \notin M], (M \setminus \{s-1\}) \oplus 1) \end{cases} \end{aligned}$$

Proof. If i is less than $s = \mathrm{span}(Q)$, $t_Q(i, *, *)$ is always zero since a string shorter than the span can not contain a matching q-gram. Let then $i \ge s$ and consider the value $t_Q(i, j, M)$. If we want to compute it from $t_Q(i-1, j^*, M^*)$ for some j^* and M^* we need to make two important observations. First, j^* is either equal to j or to $j-1$ depending on whether there is a character match at position i, i.e. whether M contains $s-1$. This is due to the fact that we only increase i by one and can therefore only introduce one additional mismatch. The second observation is that the threshold $t_Q(i, j, M)$ is either equal to $t_Q(i-1, j^*, M^*)$ (if we do not introduce an additional matching q-gram) or equal to $t_Q(i-1, j^*, M^*) + 1$ (if we do, i.e. if $Q \subseteq (M \cup \{0\})$). Furthermore the set M^* must agree with M on $s-2$ positions, i.e. $M^* \cap \{2, \dots, s-1\} = (M \cap \{1, \dots, s-2\}) \oplus 1$. The reason for this is that M overlaps M^* by $s-2$ characters. This leaves us only with the two possibilities given in the lemma. The use of the conditions combined with the minimization provides us with the correct choice. ☐

Given the recurrence from Lemma 5.3.6 and Lemma 5.3.5 we are now able to compute the optimal threshold. We achieve this by generating the values $t_Q(i,*,*)$ for all error values below or equal to k, for all possible sets M describing the matches in the last $s-1$ positions and for $i \leq m$. Out of all values $t_Q(m,k,*)$ we select the lowest threshold.

To speed up computation, all the values $t_Q(i,*,*)$ are stored in a single array of size $f(k,s) = \sum_{a=0}^{k} \binom{s-1}{a}(k-a+1)$. For each possible error value a between 0 and k it contains the threshold values $t_Q(i,a,M)$ for all possible sets M containing between 0 and a errors. One can therefore also write the array size as $f(k,s) = \sum_{a=0}^{k}\sum_{b=0}^{a}\binom{s-1}{b}$. The binomial term describes the amount of possible placements of b errors in a string of length $s-1$.

Updating the array from $t_Q(i-1,*,*)$ to contain the new values for $t_Q(i,*,*)$ requires evaluating the formula given in Lemma 5.3.6 for every entry of the array. This can be done in constant time since only two possibilities have to be considered. Thus one update can be computed in time $O(f(k,s))$. In order to compute $t_Q(m,*,*)$ we have to update the array $O(m)$ times. We arrive at the following result:

Theorem 5.3.1: The optimal threshold value $t_Q(m,k)$ can be computed in space $O(f(k,s))$, where $s = \text{span}(Q)$ and $f(k,s) = \sum_{a=0}^{k}\binom{s-1}{a}(k-a+1)$ and in time $O(mf(k,s))$.

The above dynamic programming algorithm is much faster than an exhaustive search but is still not fast enough to process all possible shapes. As shown in this thesis, there are useful shapes with large size and span. Even with the faster algorithm it would be impossible to compute the threshold for all shapes up to that size and span. To reduce the search space we used a pruning technique based on the following observation.

Lemma 5.3.7: If $Q' \subseteq Q$, then $t_{Q'}(m,k) \geq t_Q(m,k)$.

For each combination of q and s, we generated a set of *candidate* (q,s)-shapes from the set of $(q-1,s)$-shapes with positive threshold with the guarantee that every (q,s)-shape with positive threshold is among the candidates. To achieve this we first locate all sets of $(q-1,s)$-shapes which differ from all others in the same set only in the second to last position, i.e. any two shapes $Q' = \{i_1,\ldots,i_{q-1}\}$ and $Q'' = \{j_1,\ldots,j_{q-1}\}$ with $\forall k \in \{1,\ldots,q-3,q-1\} : i_k = j_k$. The following shapes could be in such a set for $q=5$ and $s=6$:

$$\{0,1,2,5\},\{0,1,3,5\},\{0,1,4,5\}$$

Given such a set of shapes we then generate a new shape

$$Q = Q' \cup Q'' = \{i_1, \dots, i_{q-2}, j_{q-2}, i_{q-1}\}$$

for all pairs Q', Q'' from the set. The new shape Q has size q since it contains all entries from Q' plus an extra entry from Q''. Returning to our example we would generate the shapes

$$\{0,1,2,3,5\}, \{0,1,2,4,5\}, \{0,1,3,4,5\}$$

Since we can assume that we generated all $(q-1,s)$-shapes with positive threshold and due to Lemma 5.3.7, all (q,s)-shapes with positive threshold have to be supersets of $(q-1,s)$-shapes with positive threshold. Pairing shapes which differ only in the second to last entry to produce new shapes is only one method of generating all (q,s)-shapes with positive t, however it sufficiently reduces the amount of shapes generated by this algorithm to make it practical for our purposes.

Finding the sets of $(q-1,s)$-shapes that differ only in the second to last position can be achieved very quickly in a lexicographically sorted list of all $(q-1,s)$-shapes with a positive threshold.

With this simple but effective pruning method it became feasible to analyze very many different shapes. For example it allowed us to find all shapes with a positive threshold for $m = 50$ and $k \in \{4,5\}$, a total of more than 25 million shapes.

The following Tables 5.1 and 5.2 contain the highest threshold and minimum coverage of all shapes with a positive threshold for $m = 50$ and $k \in \{4,5\}$. They show the highest minimum coverage and the highest threshold for all (q,s)-shapes and for all combinations of q and s.

5.3.3 Match Regions

For the filter we describe based on gapped q-grams, the same match regions can be used as for the case of the q-gram Lemma applied to the k-mismatches problem. Again one can divide S into overlapping blocks or use single diagonals and check whether a block or a single diagonal contains t or more hits. We can also use oversize blocks or sets of more than one diagonal to speed up the location of potential matches while introducing more of them. In practice the best match region for a given problem is determined by the choice of parameters and the actual implementations of the filtration and verification phase. The general statements about the choice of the match region in Section 4.4.1 for the q-gram lemma hold for gapped q-grams as well.

	q												
s	2	3	4	5	6	7	8	9	10	11	12	13	14
2	41/42												
3	40/41	36/38											
4	39/40	35/38	31/34										
5	38/39	34/38	30/34	26/30									
6	37/38	33/38	29/34	25/30	21/26								
7	36/37	32/38	28/34	24/30	20/26	16/22							
8	35/36	31/38	27/34	23/30	19/26	15/22	11/18						
9	34/35	30/38	26/34	22/30	18/26	14/22	10/18	6/14					
10	33/34	29/38	25/34	21/30	17/26	13/22	9/18	5/14	1/10				
11	32/33	28/36	24/34	20/30	17/27	14/24	10/20	7/17	4/14	0/0			
12	31/32	27/35	23/34	20/31	17/28	13/24	10/21	8/19	5/16	2/13	0/0		
13	30/31	26/34	22/34	19/31	16/28	13/25	10/22	8/20	6/18	3/15	1/12	0/0	
14	29/30	25/33	21/34	18/31	15/28	12/25	10/23	8/21	5/18	4/17	2/15	1/13	0/0
15	28/29	24/32	20/34	17/31	14/28	12/26	9/23	7/21	5/19	3/17	2/16	1/13	0/0
16	27/28	23/31	19/34	16/31	14/29	11/26	9/24	7/22	5/20	3/18	2/16	1/13	0/0
17	26/27	22/30	18/33	16/31[a]	13/29	11/27	9/25	7/23	5/21	4/20	2/17	1/13	0/0
18	25/26	21/28	17/32	15/31	12/29	10/27	8/25	6/23	5/22	3/20	2/18	1/13	0/0
19	24/25	20/27	17/30[a]	14/30	12/29	9/27	7/25	6/24	4/22	3/20	2/18	1/13	1/14
20	23/24	19/26	16/29[a]	13/30	11/29[a]	9/28	7/26	5/24	4/23	3/21	2/18	1/13	0/0
21	22/23	18/25	15/29	12/29	10/28	8/27	6/26	5/24	3/20	2/18	2/17	1/13	0/0
22	21/22	17/24	15/28[a]	12/29[a]	9/29	7/26	6/24	4/22	3/21	2/18	1/12	1/13	0/0
23	20/21	17/24	14/28	12/28[a]	9/28	7/26[a]	5/25	4/22	2/17	2/17	1/12	0/0	0/0
24	19/20	17/24	15/26	12/29[a]	10/27[a]	7/27	5/25	4/20[a]	2/17	1/11	1/12	0/0	0/0
25	20/21	17/24	15/28[a]	12/29[a]	10/29[a]	7/26	5/25	4/23	3/20	2/17	1/12	0/0	0/0
26	21/22	17/24	16/29[a]	12/30	11/31[a]	8/28	6/27	4/24	3/22	2/18	1/12	1/13	0/0
27	20/21	16/23	16/29[a]	12/30	12/30[a]	8/27	7/28[a]	5/26	4/24	3/21	2/18	1/13	0/0
28	19/20	15/21	15/29	11/30	11/30[a]	8/29	7/29[a]	6/25[a]	4/25	3/22	2/19	1/13	1/14
29	18/19	14/20	14/28	10/28	10/30	8/27[a]	7/27[a]	5/27	4/25	3/22	2/19	1/13	1/14
30	17/18	13/19	13/26	9/25	9/30	7/27	6/27	4/25	4/24	3/23	2/19	1/13	1/14
31	16/17	12/18	12/25	9/24	9/28[a]	6/27	6/27[a]	4/25	4/24	2/19	2/19	1/13	1/14
32	15/16	11/17	11/23	9/24[a]	7/25	5/25	5/25	3/22	3/22	2/18	2/18	1/13	1/14
33	14/15	12/17[a]	10/20	8/24	8/24[a]	5/25	4/24	3/22	2/18	2/18	1/12	1/13	0/0
34	13/14	12/18	9/19	8/24	8/24[a]	5/24	4/23	4/21[a]	2/18	2/17	1/12	1/13	0/0
35	12/13	12/18	8/18	8/24	8/25[a]	4/22	4/23	4/21	2/17	2/17	2/16	1/13	1/14
36	11/12	11/17	7/16	7/22	7/24	4/22	4/21	4/21	2/17	2/17	2/16	1/13	1/14
37	10/11	10/15	8/16[a]	6/19	6/21	4/22	4/21	3/18	2/16	2/17	2/16	0/0	0/0
38	9/10	9/14	8/16[a]	5/15	5/20	4/22	4/19[a]	3/18	2/17	2/14	1/12	0/0	0/0
39	8/9	8/13	8/16[a]	5/14[a]	4/18	4/21	4/19[a]	2/16	2/17	2/14	1/12	0/0	0/0
40	7/8	7/12	7/16	5/14	3/15	3/18	3/19	2/15	2/12	1/11	1/12	0/0	0/0
41	6/7	6/10	6/15	6/14[a]	4/14[a]	2/13	2/14	2/14	1/10	1/11	1/12	1/13	0/0
42	5/6	5/9	5/13	5/14	4/14[a]	2/12	2/13	2/14	1/10	0/0	0/0	0/0	0/0
43	4/5	4/8	4/10	4/14	4/14	3/12[a]	2/9	1/9	1/10	0/0	0/0	0/0	0/0
44	3/4	3/6	3/9	3/12	3/14	3/12[a]	2/10	1/9	1/10	0/0	0/0	0/0	0/0
45	2/3	2/5	2/7	2/9	2/11	2/12	2/11	1/9	0/0	0/0	0/0	0/0	0/0
46	1/2	1/3	1/4	1/5	1/6	1/7	1/8	1/9	1/10	0/0	0/0	0/0	0/0

[a]The shapes with the highest threshold and the highest minimum coverage are different.

Table 5.1: The best thresholds/minimum coverage for $m = 50$ and $k = 4$.

s	q: 2	3	4	5	6	7	8	9	10	11	12
2	39/40										
3	38/39	33/35									
4	37/38	32/35	27/30								
5	36/37	31/35	26/30	21/25							
6	35/36	30/35	25/30	20/25	15/20						
7	34/35	29/35	24/30	19/25	14/20	9/15					
8	33/34	28/35	23/30	18/25	13/20	8/15	3/10				
9	32/33	27/35	22/30	18/26	14/22	9/17	5/13	0/0			
10	31/32	26/34	21/30	18/27	13/22	10/19	6/15	3/12	0/0		
11	30/31	25/33	20/30	16/26	13/23	10/20	7/17	4/14	2/12	0/0	
12	29/30	24/32	19/30	16/27	12/23	9/20	7/18	4/15	2/13	0/0	0/0
13	28/29	23/31	19/30[a]	15/27	12/24	9/21	6/18	4/16	2/14	1/11	0/0
14	27/28	22/30	17/30	14/27	11/24	8/21	6/19	4/17	2/14	1/11	0/0
15	26/27	21/28	17/29[a]	13/27	10/24	8/22	5/19	3/17	2/15	1/11	0/0
16	25/26	20/27	16/29[a]	13/27	10/24	7/22	5/20	3/17	2/15	1/11	0/0
17	24/25	19/26	15/28	12/27	9/25	7/23	5/21	3/18	2/15	1/11	0/0
18	23/24	18/25	14/27	11/27	8/25	6/23	4/20	3/18	2/16	1/11	0/0
19	22/23	17/24	14/26[a]	11/26[a]	8/25	6/22	4/20	2/15	1/10	1/11	1/12
20	21/22	16/23	13/25[a]	10/24	7/24	5/22	3/18	2/15	1/10	1/11	0/0
21	20/21	15/21	12/25	9/23	7/22	5/21	3/18	2/15	1/10	0/0	0/0
22	19/20	15/21	12/23[a]	9/23	6/21	4/20	2/14	1/9	1/10	0/0	0/0
23	18/19	15/21	12/23[a]	9/23	6/21	4/19	2/14	1/9	0/0	0/0	0/0
24	18/19	15/21	13/24	9/23	7/20[a]	4/19	2/14	1/9	0/0	0/0	0/0
25	19/20	15/21	13/25[a]	9/23[a]	7/22	4/20	3/16	1/9	1/10	0/0	0/0
26	20/21	15/21	14/26[a]	9/24	8/24	5/23	3/19	2/15	1/10	0/0	0/0
27	19/20	14/20	14/26[a]	9/24	9/24[a]	6/23[a]	4/19	2/15	1/10	1/11	0/0
28	18/19	13/19	13/26	8/24	8/24	5/22	4/20	3/17	2/15	1/11	0/0
29	17/18	12/18	12/25	8/23	8/24[a]	5/22	5/22	2/15	2/15	1/11	0/0
30	16/17	11/17	11/23	7/21	6/21	4/20	4/20	2/15	2/15	1/11	0/0
31	15/16	10/15	10/20	7/21	7/20[a]	4/18	4/18[a]	2/14	2/14	1/11	0/0
32	14/15	10/15	9/19	7/19[a]	5/19	3/17	3/16	1/9	1/10	0/0	0/0
33	13/14	10/15[a]	8/18	6/19	6/19[a]	3/17	2/14	1/9	1/10	1/11	0/0
34	12/13	11/17	7/16	6/19	6/20[a]	3/17	3/14	2/12	1/10	1/11	0/0
35	11/12	11/17	6/15	6/19	6/20[a]	3/16	3/14	2/14	1/10	1/11	0/0
36	10/11	10/15	6/13[a]	5/15	5/20	2/13	2/13	2/14	1/10	0/0	0/0
37	9/10	9/14	7/15[a]	4/14	4/18	3/13[a]	2/13	2/13	1/10	1/11	1/12
38	8/9	8/13	7/15[a]	4/12[a]	3/15	2/13	2/12	1/9	1/10	1/11	0/0
39	7/8	7/12	7/15[a]	4/12	3/12	2/12	2/12	1/9	1/10	1/11	0/0
40	6/7	6/10	6/15	4/13	2/10	2/11	1/8	1/9	1/10	0/0	0/0
41	5/6	5/9	5/13	5/13[a]	3/10[a]	1/7	1/8	1/9	0/0	0/0	0/0
42	4/5	4/8	4/10	4/13	3/10	1/7	1/8	1/9	0/0	0/0	0/0
43	3/4	3/6	3/9	3/12	3/12	2/8	1/8	0/0	0/0	0/0	0/0
44	2/3	2/5	2/7	2/9	2/10	2/9	1/8	0/0	0/0	0/0	0/0
45	1/2	1/3	1/4	1/5	1/6	1/7	1/8	0/0	0/0	0/0	0/0

[a]The shapes with the highest threshold and the highest minimum coverage are different.

Table 5.2: The best thresholds/minimum coverage for $m = 50$ and $k = 5$.

5.3.4 The Minimum Coverage of Gapped q-grams

When examining the huge amount of shapes with a positive threshold for a given set of parameters, one quickly arrives at an important observation. The threshold of a shape and the value of q are not very good parameters to measure the filtration efficiency of that shape. While f_e clearly depends on $t_Q(m, k)$, the correlation is not as strong as one might expect. In fact we were able to find a large number of instances where shapes with lower thresholds produce fewer potential matches. A simple example illustrating this phenomenon follows.

Let $m = 13$ and $k = 3$. Then both shapes `###` and `##-#` have a threshold of two. If two strings have four consecutive matching characters, they have two common 3-grams of shape `###`. In contrast, to have two common 3-grams of shape `##-#` two strings need to have at least five matching characters.

This is where the minimum coverage comes into play. While we were already aware of it during our work involving the q-gram lemma, it was examples like the one above together with the huge amount of possibly interesting shapes that convinced us of its importance. In contrast to the classic, ungapped q-grams where the set of available filters is very limited due to the fact that one can only modify the value of q, the choice of a good shape becomes much harder for gapped q-grams. For contiguous q-grams a relatively small set of experiments allowed us to pick good values of q but it is virtually impossible to experimentally evaluate all shapes with positive threshold for a given set of parameters. We therefore analyzed the potential of the minimum coverage as a measure for the filtration efficiency of gapped q-grams.

As a first step we decided to determine a method for computing the minimum coverage of a gapped shape. It turned out to be somewhat more difficult than for the ungapped case where the minimum coverage can be derived from a simple explicit formula.

5.3.5 Computing the Minimum Coverage

We have designed and implemented a branch-and-bound algorithm for computing the minimum coverage for any shape Q, threshold t and an arbitraty string S with $|S| \geq t \cdot \text{span}(Q)$. Given a specific shape Q, t and the length of S it uses sets of integers to store the starting positions of up to t matching shapes in S. The cover of such a set A is then defined as $\cup_{i \in A} Q_i$. The algorithm starts off by placing the first shape in a set A at position 1, i.e. $A = \{1\}$. It then recursively adds the remaining $t - 1$ shapes. For each extra shape the algorithm generates $\text{span}(Q) - 1$ many new versions of the set A by adding the new shape at positions $\{\max A + 1, \ldots, \max A + \text{span}(Q) - 1\}$.

However, it only considers those new sets A' for which $|\cup_{i\in A'} Q_i| + t - |A'|$ is smaller than the current minimum coverage. This cutoff condition stems from the fact that adding $t - |A'|$ more shapes will increase the cover of A' by at least $t - |A'|$. In order to quickly compute a good lower bound for the minimum coverage we use a depth-first approach that computes the coverage for the set $A = \{1, \ldots, t\}$ first. In most cases the coverage of this set is very close or equal to the minimum coverage and provides a good cutoff condition which reduces the amount of sets generated in each iteration. After computing all viable sets containing t shapes, we select the set with the smallest coverage which is then equal to the minimum coverage.

We would like to mention that the minimum coverage computation can easily be adapted to shorter strings S by only adding new shape positions to A if the shapes lie completely within S.

The efficiency of the above algorithm was sufficient to compute the minimum coverage of a large number of different of shapes in a reasonable amount of time (several million shapes in our case).

5.3.6 Some Observations on Gapped q-grams

When computing the minimum coverage for gapped q-grams we made two interesting observations.

One is the connection between the minimum coverage and the threshold of a shape. Given a contiguous shape and a threshold t, the minimum coverage of that shape will increase by exactly one if t is incremented by one. For gapped shapes an increase of t by one will often raise the minimum coverage by two or even more.

As an example, for $t = 1$, the shapes `###` and `##-#` have a minimum coverage of three. If we increase t to two, the contiguous shape has a minimum coverage of four and the gapped shape one of five. A second contiguous q-gram can be overlapped with the first one by adding one extra matching character, but for the gapped case the second q-gram adds one extra character at the end of the q-gram and one in the gap.

```
# # # -          # # - # -
- # # #          - # # - #

# # # #          # # # # #
```

This property of gapped q-grams explains why some of them have a much higher minimum coverage than ungapped q-grams and why they are better suited for filtering.

Given the same error level k, a filter based on a good gapped q-gram requires a potential match to have a much larger minimum cover, i.e. more matching characters between a pattern P and a text string S are necessary. This makes false positives far less likely.

A second observation was the fact that while for ungapped q-grams there exists exactly one minimum cover, this may not be the case for gapped q-grams. Here it is possible that several different minimum covers exist. This is not very common and usually the number of alternatives is fairly small. Nevertheless it can lead to a moderate increase in the expected number of potential matches. We will come back to this observation in the experiments section where we analyze how well the minimum coverage can be used to predict the filtration efficiency of a filter based on a certain shape.

After laying the groundwork for a thorough analysis of the suitability of gapped q-grams for solving the k-mismatches problem we conducted extensive experiments and compared the results with the predicted behavior of the different shapes. We used the same measures for speed and filtration efficiency as in the case of the contiguous q-grams. The experimental results and the correlation between predicted and measured behavior can be found in Chapter 6.

5.4 An Extension to the k-differences Problem

While the k-mismatches problem is quite interesting in itself, in many practical applications the k-differences problem is of much greater importance. We therefore wanted to overcome the problems described before concerning insertions or deletions in the gaps of a gapped q-gram. In this section we will introduce our approach. One of the main obstacles in realizing this algorithm was the computation of the threshold. We include a brief description of a solution by Juha Kärkkäinen [Kär02] in this work for the sake of completeness. The remaining contents of this section as well as some experimental results were published in 2002 [BK02].

5.4.1 The Filter Algorithm

The basic idea behind the gapped q-gram filters for the Levenshtein distance is the use of more than one shape to compensate for possible insertions or deletions in gaps of the shape. The simplest form of this technique is the application to shapes with only one gap. For these shapes we needed to find an approach that allows us to detect matching q-grams even if there was an insertion or deletion in the gap.

To achieve this goal we use three different shapes from the query to detect matches in the database string. A basic shape with only one gap and two other shapes formed from the basic shape by increasing and decreasing the length of the gap by one. For example, with the basic shape `##--#` we would also use the shapes `##---#` and `##-#`. For each of these shapes we compute the IDs of the corresponding q-grams in P. However, when searching for these q-grams in S we treat the three different shapes all as instances of the basic shape, basically transforming them into the basic shape. The filter therefore compares the q-grams of all three shapes in the pattern P to the q-grams of the basic shape in the text S. If S contains an insertion in the gap of the q-gram, the basic shape could be invalidated by this insertion, however the shape with reduced gap length from P would still result in a match with S when transformed into the basic shape. For a deletion, the shape with increased gap length matches with S at the same starting position when transformed into the basic shape. This ensures that matching q-grams are found even if there is an insertion or deletion in the gap.

In the following example we show which q-grams are generated in the pattern P.

```
P =  A C G T A C T A

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

It is important to point out that we transform all these q-grams with different gap lengths into q-grams of the basic shape when searching in S. A q-gram `GT-C` from P would be transformed into `GT--C` when searching for it in S. So except for the fact that we now use three shapes instead of one for each position in P, the basic approach remains the same. Again we have to detect match regions that contain at least t hits.

When creating the three distinct q-grams for a given starting position in P, it is possible that two or all three of them result in the same q-gram if the characters behind the gap are repeats. In this case we only count the two or three hits as one hit. In the next example, the first two q-grams would result in the same q-gram for search in S.

We would only search for the q-grams `AC--A` and `AC--C`.

```
P =  A C G A A C
     A C - A
     A C - - A
     A C - - - C
```

5.4.2 Computing the Optimal Threshold

The new approach based on the use of three shapes for generating q-grams from the pattern P complicates threshold computation even further. To solve this problem we first needed to find a definition of the threshold under the new circumstances. Furthermore it was necessary to develop an algorithm to actually compute the threshold given this definition. Juha Kärkkäinen [Kär02] developed a two-stage algorithm that provides a general framework for computing thresholds. However, the algorithm still requires a so-called *profit function* for a specific filter in order to actually compute the threshold. Deriving this profit function can be quite difficult depending on the complexity of the filter.

We now present the definition of the threshold we used for our filter algorithm. We then proceed to define profit functions in general and the $Q \pm 1$-*profit* for the filter based on three one-gapped q-grams. This profit functions is required by the algorithm for the threshold computation. We also briefly sketch the actual algorithm in the last third of this section.

Following [Gus97] we define an *edit transcript* as a string over the alphabet of M(atch), R(eplace), I(nsert) and D(elete) operations, describing a sequential character-by-character transformation of one string into another. In such a transcript an M stands for a character that matches between both strings. An R stands for a character replacement in, an I for an insertion into and a D for a deletion from the first of the two strings.

For two strings P and S, let $\mathcal{T}(P, S)$ denote the set of all transcripts transforming P to S. For example, $\mathcal{T}(\texttt{ACTG}, \texttt{ACCT})$ contains `MMRR`, `MMIMD`, `MIMMD`, `IRMMD`, `IDIMDID`, etc.

For a transcript $\tau \in \mathcal{T}(P, S)$, the source length $slen(\tau)$ of τ is the length of P, i.e., the number of non-insertions in τ.

The Edit cost $c_E(\tau)$ is the number of non-matches in the edit transcript. The Hamming cost $c_H(\tau)$ is infinite if τ contains insertions or deletions, otherwise it is the number of replacements in the edit transcript. The Edit and Hamming distance of P and S are

$$d_E(P,S) = \min_{\tau \in \mathcal{T}(P,S)} c_E(\tau)$$

and

$$d_H(P,S) = \min_{\tau \in \mathcal{T}(P,S)} c_H(\tau)$$

respectively.

After having defined distance measures for strings using cost functions for edit transcripts we can also define the q-gram similarity measure for strings using *profit functions* for edit transcripts. An example for such a profit function is the Q-profit defined below. We introduce the concept of profit functions with the simple case of only one shape.

A *match alignment* M_τ of a transcript τ is the set of pairs of positions that are matched to each other. For example, $M_{\mathtt{IIMMIRDMR}} = \{(1,3),(2,4),(5,7)\}$. For a set I of integers, let $M_\tau(I)$ be the set to which M_τ maps I, i.e.,

$$M_\tau(I) = \{j \mid i \in I \text{ and } (i,j) \in M_\tau\}.$$

A *Q-hit* in a transcript τ is a pair (i,j) of integers such that $M_\tau(Q_i) = Q_j$. Note that, if $\tau \in \mathcal{T}(P,S)$, then a Q-hit (i,j) in τ implies $P[Q_i] = S[Q_j]$.

The *Q-profit* $p_Q(\tau)$ of a transcript τ is the number of its Q-hits, i.e., $p_Q(\tau) = |\{(i,j) \mid M_\tau(Q_i) = Q_j\}|$. Using p_Q as the profit function defines the *Q-similarity* of two strings P and S as $s_Q(P,S) = \max_{\tau \in \mathcal{T}(P,S)} p_Q(\tau)$.

We would like to point out that the corresponding similarity measure s_Q is not exactly the one used by the filter algorithms. However, there is a close connection. If $\tau \in \mathcal{T}(P,S)$, then a Q-hit (i,j) in τ implies $P[Q_i] = S[Q_j]$. Therefore, the number of matching q-grams between P and S is at least $s_Q(P,S)$, and thus at least t if $d(P,S) \leq k$. The number of matching q-grams may be higher than $s_Q(P,S)$. For the 1-gram $Q = \{0\}$, AB and BA have two matching 1-grams (A and B) even though $s_{\{0\}}(\mathtt{AB},\mathtt{BA}) = 1$. The exact value of $s_Q(P,S)$ could be computed by a more careful analysis of the matching q-grams, but it may not be worth the computational effort. It again boils down to the definition of the match region and choosing the right balance between getting as close as possible to s_Q and spending additional time to do so.

We can now define the threshold using edit transcripts as follows:

Definition 5.4.1: Given a pattern length m and a maximum distance k, the *threshold* for a cost function c and a profit function p of edit transcripts is

$$t_p^c(m,k) = \min_{\tau}\{p(\tau) \mid slen(\tau) = m, c(\tau) \leq k\}.$$

This leads to the following lemma for the filter criterion.

Lemma 5.4.1: Let c be a cost function and p a profit function for edit transcripts. Define a distance d of two strings P and S as $d(P,S) = \min_{\tau \in \mathcal{T}(P,S)} c(\tau)$ and a similarity s as $s(P,S) = \max_{\tau \in \mathcal{T}(P,S)} p(\tau)$. Now, if $d(P,S) \leq k$, then $s(P,S) \geq t_p^c(|P|,k)$.

The lemma holds for any choice of cost c and profit p.

For $c = c_H$ and $p = p_Q$, Definition 5.4.1 results in the thresholds that were used for the k-mismatches problem. If Q is the contiguous shape $\{0,1,\ldots,q-1\}$, the threshold is the same as given by the q-gram lemma.

However, we still lack a profit function that defines the q-gram based similarity for the case of one-gapped q-grams when using 3 shapes to compensate for insertions and deletions. Let $b_1, g, b_2 > 0$ and (b_1, g, b_2) denote the *one-gap shape* $\{0,\ldots,b_1-1, b_1+g,\ldots,b_1+g+b_2-1\}$. For a one-gap shape $Q = (b_1,g,b_2)$, let $Q^{+1} = (b_1, g+1, b_2)$ and $Q^{-1} = (b_1, g-1, b_2)$ (or $Q^{-1} = \{0,\ldots,b_1+b_2-1\}$ if $g = 1$). Then, a $Q \pm 1$-*hit* in a transcript τ is a pair (i,j) of integers such that $Q_j \in \{M_\tau(Q_i^{-1}), M_\tau(Q_i), M_\tau(Q_i^{+1})\}$. The $Q \pm 1$-*profit* of τ, denoted by $p_{Q\pm1}(\tau)$, is the number $Q \pm 1$-hits in τ, i.e., $p_{Q\pm1}(\tau) = |\{(i,j) \mid Q_j \in \{M_\tau(Q_i^{-1}), M_\tau(Q_i), M_\tau(Q_i^{+1})\}\}|$. The $Q \pm 1$-*similarity* of two strings P and S is $s_{Q\pm1}(P,S) = \max_{\tau\in\mathcal{T}(P,S)} p_{Q\pm1}(\tau)$.

For $c = c_E$ and $p = p_{Q\pm1}$, Definition 5.4.1 provides the thresholds used by the one-gapped q-gram filters for the Levenshtein distance. What was said about the relation of the filter and the similarity measure s_Q above, holds for this case as well.

We now have a definition of the threshold for the filter we want to use for the k-differences problem. To actually compute the threshold from this formula we need the two-stage algorithm by Juha Kärkkäinen [Kär02]. It is based on automata for computing the values $slen(\tau)$, $c(\tau)$ and $p(\tau)$ for any transcript τ. These automata have an initial state representing the empty transcript and four transitions out of each state labeled with M,R,I and D. Each transition has three output values, which represent the changes in the computed values $slen(\tau)$, $c(\tau)$ and $p(\tau)$ caused by this transition. For a given transcript τ, the computed value is the sum of the output values of the path representing τ in the automaton. Such automata are more generally known as *weighted finite automata*[BGW00] or *string-to-weight transducers*[Moh97]. Constructing such automata is a non-trivial task for all but the simplest filter algorithms. However, it requires far less effort than developing a method for computing the threshold of a given filter from scratch since one can rely on a well-founded framework for generating the automata.

To actually compute the threshold one combines the three automata for $slen(\tau)$, $c(\tau)$ and $p(\tau)$ into one larger automaton. The state set of the combined automaton is the Cartesian product of the state sets of the three simple automata, but since the automata for source length and Levenshtein or Hamming cost have just one state the combined automaton is simply the profit automaton with three output values on each edge.

One can think of this automaton as a directed graph $G = (V, E)$ in which each edge $e \in E$ is labeled with three non-negative values $l(e)$, $c(e)$ and $p(e)$ corresponding to the output value for source length, cost and profit of that edge. Now the threshold computation problem can be stated as follows:

Find the path τ_{min} in G starting from the initial state with the empty edit transcript that has the lowest profit $p(\tau_{min})$ and satisfies $c(\tau_{min}) \leq k$ and $l(\tau_{min}) = w$, i.e. has the lowest number of matching q-grams (profit), k errors (cost) and a length of w.

The problem stated above is very similar to the classic *constrained shortest path* (CSP) problem[Zie01]. There exist pseudo-polynomial algorithms, i.e., algorithms that work in polynomial time when the edge values are polynomial, which can be adapted to this problem.

We implemented the whole algorithm in C++ using LEDA [MN95] with a graph construction algorithm based on dynamic programming that was designed specifically for one-gap shapes and the Levenshtein distance. With this implementation we were able to conduct an exhaustive analysis of all shapes for the parameter sets used in our experiments within a reasonable amount of time.

5.4.3 Match Regions

For the filter based on one-gapped q-grams, the types of match regions that can be used are the same as for the case of the q-gram Lemma applied to the k-differences problem. Again one can use a division of S into overlapping blocks or use sets of $k + 1$ diagonals and check whether such a block or set of diagonals contains t or more hits. We can also use oversized blocks or sets of more than $k + 1$ diagonals to speed up locating potential matches while introducing more of them. In practice the best match region for a given problem is determined by the choice of parameters and the implementations of the filtration and verification phase. The statements made about the choice of the match region in Section 4.4.1 for the q-gram lemma hold for gapped q-grams as well.

5.4.4 The Minimum Coverage

When using one-gap shapes as filters in the way outlined above, the question of shape choice arises again. However, there are some important differences to k-mismatches problem. Since we only allow one gap, the total number of shapes with positive threshold, i.e. the set of potentially interesting shapes, is far lower. For many sets of parameters m, q and k it is actually small enough to conduct an experimental analysis of speed and filtration efficiency for all possible shapes. Nevertheless we decided to evaluate how well one can predict the filtration efficiency of this filter class with the minimum coverage.

Our first observation was that the minimum coverage with respect to the target string S remains the same. Since one only uses the basic shape and not all three on S, the minimum cover remains unchanged when compared to using only one shape and can be computed with the algorithm described in Section 5.3.5.

However, there are some effects on the expected number of potential hits that stem from the use of three shapes in the pattern P. While the size of the minimum cover remains the same, there are more distinct character patterns for the minimum cover since they can be constructed from the three different shapes at each position. A simple example would be the basic shape `##--#` with a threshold of two. The minimum coverage for this example is five. However, potential matches with minimal cover can be caused by several different arrangements of matching characters in P like, for example, `#####` (two of the pattern shapes with gap size reduced by one), `###-##` (two basic shapes), `###--##` (two of the pattern shapes with increased gap size) or `####-#` (the first shape with reduced gap length, the second a basic shape). This leads to more potential matches than expected.

Another effect that is most pronounced for small alphabets and shapes for which one of the two contiguous matching pieces is very short is also caused by using three alternate shapes. The generation of three usually different 'tail-ends' of the shape increases the likelihood of a hit. Take for example the shape `##--#` and the string $P =$ `ACAGCT`. For the first position in P we will look for the q-grams `AC--G`, `AC--C` and `AC--T` in the text S. Combined with a small alphabet size this will make a hit with that starting position in P more likely than when using only a single shape.

It is very hard to quantify the influence of these effects on the expected number of potential matches but our experiments still showed the minimum coverage to be a reasonable measure for predicting the filtration efficiency. For $m = 50$ and $k \in \{4, 5\}$ we calculated the minimum coverage of all shapes with positive threshold and $q \geq 8$ and ran experiments for these shapes in order to compare expected and actual values for filtration efficiency. The results can be found in Section 6.3.

5.4.5 Evaluating One-gap q-grams for the k-differences Problem

After laying the groundwork for a thorough analysis of the suitability of one-gap q-grams for solving the k-mismatches problem we conducted extensive experiments and compared the results with the predicted behavior of the different shapes. We used the same measures for speed and filtration efficiency as in the case of the contiguous q-grams. It turns out to be the case that our filter based on one-gap q-grams can be significantly better than previously published filters. However, the advantages are not as big as for the the k-mismatches problem. It is also the case that the correlation between predicted and measured filter behavior is not as good as for the k-mismatches problem. This is not a real drawback though, since it is usually viable to conduct an experimental analysis of all interesting shapes for a given set of parameters which was not the case for arbitrary gapped shapes. More information on the experimental data and the correlation between predicted and actual behavior can be found in Section 6.3.

5.5 Moving from Ungapped to Gapped q-grams

So far we have not talked about the modifications to existing algorithms and implementations required for the use of gapped q-grams. Filter algorithms typically rely on matching q-grams between a pattern P and a string S. To locate these hits, some form of index for S is used. For ungapped q-grams a lookup table is a very simple yet efficient solution. It has size $O(|S| + |\Sigma|^q)$, where Σ is the alphabet. It can be constructed in time $O(|S| + |\Sigma|^q)$ and all hits of a pattern can be found in time $O(|P| + h)$ where h is the number of hits. It is very simple to modify the lookup table to work for gapped q-grams as well by simply creating hit lists for all possible gapped q-grams for a given shape and value of q. Gapped q-grams increase the construction time of the index to $O(|S|q + |\Sigma|^q)$ and the query time to $O(|P|q + h)$ due to the more complex nature of computing unique IDs for the q-grams in S and P respectively. In practice the difference in construction time is not very important since it only adds a small constant factor to the linear time preprocessing phase. The difference in query time is not much of an issue for large databases since the number of hits typically dominates the query time. The use of a suffix array for contiguous q-grams of varying length is of course impossible for gapped q-grams. However, it is feasible to adapt the hybrid approach for large values of q to gapped q-grams. It is therefore quite simple to modify existing implementations to use gapped q-grams efficiently. The main problems lie in the choice of good shapes and in the computation of their threshold. However, these are one-time costs and we provide efficient algorithms and their implementations for this task.

5.6 Concluding Remarks about Gapped q-grams

The most important results of our work are the general superiority of gapped q-grams for filter algorithms in approximate string matching. While we have so far only developed two algorithms based on their use, there is still potential for additional development for which we provide the basic tools including, for example, the development of filters based on sets of shapes or the sequential or parallel use of filters using different shapes. Other areas of potential interest are the use of gapped q-grams in lossy filters and the combination of gapped q-grams with approximate q-grams.

Chapter 6

An Experimental Analysis of Filter Algorithms

This chapter presents the results of various experiments with filter algorithms discussed in this thesis. The experiments have two main goals – to evaluate and compare speed and filtration efficiency of different filters and to analyze their behavior in the different match zones. For gapped q-grams we present some results illucidating the distribution of shape quality that emphasize the importance of careful shape selection. Furthermore we compare experimental results with the predicted values for speed and filtration efficiency provided by computations based on the value of q and the minimum coverage c_m in order to analyze the quality of our predictions.

We begin this chapter with some experimental results for QUASAR in Section 6.2. Then we proceed to a comparative analysis of ungapped and gapped q-grams for the k-mismatches and the k-differences problem in Section 6.3. In Section 6.4 we illustrate the large variance in the quality of different gapped q-grams and underscore the importance of careful shape selection. Finally, Section 6.5 covers our experiments concerning the three match zones and the potential of the different algorithms for use as lossy algorithms.

Since one of the main applications of approximate string matching is computational biology with its huge genomic databases, we used either randomly generated or actual DNA data for our experiments.

6.1 Parameter Ranges

Part of the motivation for our research were problems from computational biology, namely EST clustering and shotgun sequencing. We therefore used values for the parameters w and k that are typical for these areas. In both applications one searches for local matches that have a minimum length between 40 and 60 base pairs. We set $w = 50$. The second parameter, k depends on the quality of the sequences one works with. In EST clustering the data quality is often quite low and values of k between 3 and 5, i.e. an error rate between 6 and 10%, are of potential interest. Most data produced for shotgun sequencing however is very recent and contains less errors. This means that one can usually get away with a value of $k = 3$ which corresponds to 6% error rate for $w = 50$.

In the applications described, queries are basically fragments of DNA that have been sequenced in one piece. Current sequencing technology limits the length of such fragments to roughly 500 base pairs. We therefore only conducted experiments with query lengths between 300 and 600 base pairs.

For large databases lower values of q result in huge hit lists and therefore very large sets $H(P)$. This leads to an explosion of the runtimes. For this reason we only ran few experiments with $q \leq 7$. On the other hand a lookup table becomes less practical for higher values of q so we did not conduct any experiments with $q > 14$.

6.2 Results for QUASAR

We will begin this chapter with some results for QUASAR, our implementation of a filter algorithm based on the q-gram lemma combined with a suffix array and a lookup table. This software was designed for EST clustering in 1999 and used to cluster the Mouse EST database. In EST clustering one basically has to run an all vs. all comparison of all EST sequences in a large sequence database. It is possible to think of the sequences as nodes in a graph and the approximate matches which meet certain criteria as edges connecting nodes that potentially belong to the same gene. Our 1999 version of the Mouse EST database had a size of 75.2 million base pairs and contained 198323 sequences. It was preprocessed for clustering by removing sequences with common repeats and sequence tails containing poly-X (X= $\{A, C, G, T, N\}$) which are frequently introduced by sequencing inaccuracies and are usually clipped by the sequencing centers. We also removed poly-AT signals at the beginning or end of EST sequences. For additional tests we used the Human EST database which at that time contained 279.5 million base pairs in 723675 sequences. It was preprocessed in a similar fashion.

Our implementation used a modified version of NCBI BLAST 2.0.3 for the verification phase. We spent a considerable amount of time to incorporate NCBI BLAST code into our implementation in order to be able to pass the potential matches reported in the filtration phase to BLAST for verification via main memory. This was necessary to avoid the overhead of writing data into a file and calling BLAST to operate on that file. Coding the direct interface turned out to be quite difficult and memory leaks caused by sloppy memory management in the BLAST became apparent. This resulted in memory consumption continuously increasing with every query and limited the amount of queries we were able to search for with one call of the program.

We conducted the searches for EST sequences in batches of 1000 queries against the complete database. The machine we used was a single SUN Ultra Sparc II processor with 333 MHz in a dedicated Sun Enterprise 10000 that was able to store all necessary data structures in main memory. For reference purposes we ran the same batch of queries using NCBI BLAST 2.0.3 against the EST database. In both cases we did not include the time required to read data from disk in the runtimes.

For QUASAR we used the q-gram lemma with a window length $w = 50$, the q-grams had length $q = 11$ and an error level of $k = 3$. This results in a threshold of $t = 7$. BLAST was executed with an E-value of $E = 10^{-5}$ and a word length of $q = 11$. The remaining parameters were left unchanged.

We conducted two sets of experiments. In the first we used different values for the block size b in which we divided the database to determine the optimal block size. The second set of experiments is a speed comparison between QUASAR and BLAST using the optimized block size.

6.2.1 Determining Optimal Block Size

One result of the development of QUASAR was the use of blocks with block sizes b that are greater than $2(m-1)$, the value proposed by Jokinen and Ukkonen. Depending on the filter parameters and the implementation this can significantly speed up the overall matching. In order to determine the best value for b we conducted experiments with different values of b. Due to the use of shift operations in our code we were restricted to powers of two. In Figure 6.1 we present the results of these experiments. We analyzed the time consumed in the filtration and verification phase as well as the total time. We also plot the percentage of the blocks that were recognized as potential matches. This is roughly equivalent to $1 - f_e$

The main result of our experiments was the optimal value for b, which turned out to

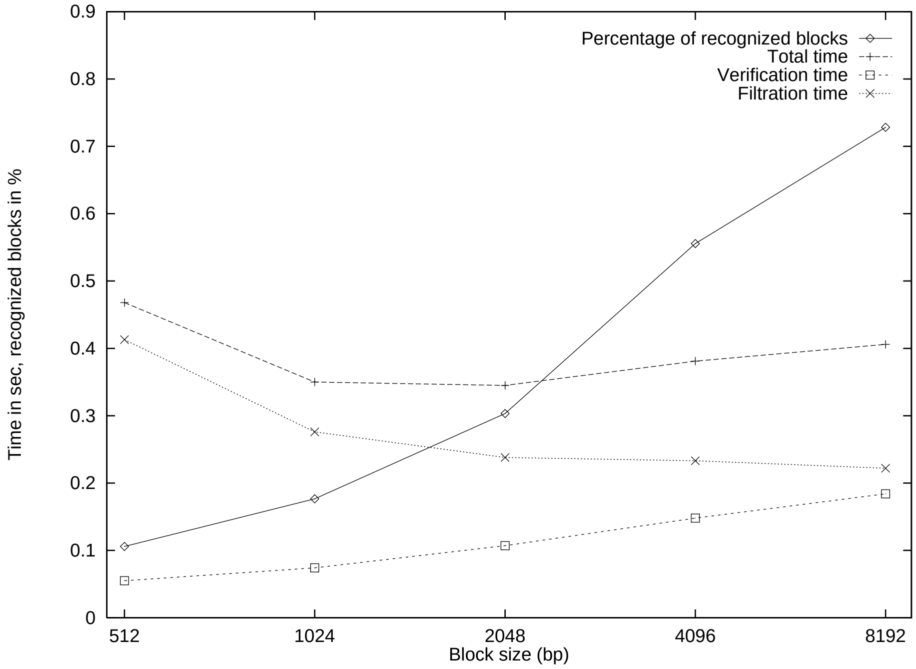


Figure 6.1: Running times and percentage of recognized blocks for various block sizes.

be 2048 for local matching with substrings of length 50 and $k = 3$. Another interesting observation is the fact that while the correct choice of b is important, the range of values for which one achieves good overall speed is fairly broad. The restriction of b to powers of two is therefore only minor.

6.2.2 QUASAR vs. BLAST

After determining the optimal value for b, we ran experiments to compare the speed of our algorithm with that of standard BLAST. We used the Mouse and Human EST databases available in 1999. The results are shown in Table 6.1. At the time we did not use the notion of filtration efficiency or actually counted potential matches. However, we protocolled the number of blocks marked as potential matches and compared it to the total amount of blocks in the database. We used the percentage of blocks with a threshold of t or more, i.e. the percentage of recognized blocks to illustrate the filtration efficiency.

For our applications QUASAR turned out to be up to 35 times faster than BLAST which allowed us to complete the all vs. all comparison of the Mouse EST database in roughly six hours, a process that would have required about a week using BLAST.

DB	Size Mbps	Query Size(bps)	Recognized Blocks(in %)	CPU times in seconds QUASAR	BLAST
Mouse	73.5	368	0.38%	0.121	3.37
Human	279.5	393	0.33%	0.37	13.27

Table 6.1: Running times of searches in Mouse and Human EST databases with block size of 2048bps. From left to right: database and its size, average query size, percentage of recognized blocks and the average CPU times per query for QUASAR and BLAST. All results were averaged over a batch of 1000 queries.

We would like to point out that the filtration phase accounts for roughly two thirds of the total running time (for a block size of 2048bps) even though it uses the simplest possible match region. Thus, the speed of QUASAR would benefit most from improvements in the filtration phase. One problem of the implementation was the crude interface between our code and BLAST which lead to a substantial overhead and was virtually impossible to keep up to date with newer versions of NCBI BLAST due to continued changes in their function structure. A second problem of the interface was that we had to pass all sequences from the database that were part of potential matches to BLAST in their entirety. We were also unable to provide additional information about the potential matches to BLAST via this interface so BLAST had to locate the q-grams causing the potential matches a second time by constructing and using it's automaton. For large scale application it would probably make more sense to implement an algorithm similar to BLAST from scratch. Due to the magnitude of this software engineering feat we decided against this within the scope of this thesis.

6.3 Speed and Filtration Efficiency of Different Filters

The purpose of this section is the presentation and discussion of experimental results for various filter algorithms. In order to allow a meaningful comparison between different filters we decided to measure speed and filtration efficiency using the two parameters described in Section 4.7. They are $|H(P)|$, the total amount of hits for a query and $|PM(S)|$, the total number of potential matches returned for a given filter, a query P and a database S. While we have provided means to predict these parameters for a given filter we also wanted to measure them directly in an experimental setting.

We implemented a universal filter algorithm which can use different filter criteria and apply them to a query P and a database S. We set up a test environment in which we searched for a large number of different queries P in synthetic and actual DNA databases. The length of these queries was 500 base pairs each. They are randomly generated DNA sequences with an independent and equal distribution of the four nucleic acids. We used two databases, one contains 50 million base pairs of sequences from the Human EST database, the second is an equally sized, randomly generated DNA database with equal and independent distribution of the four nucleic acids. We will refer to these two databases as the EST and the random database. We would like to point out that, even though the queries were randomly generated, none of them had actual matches in either database for any error level used in the experiments. This means that all potential matches are false positives and that the filtration efficiency can be simplified to

$$f_e = \frac{|PM(S)|}{|S|}$$

There were several reasons against using actual DNA as queries. The most important is our interest in the amount of potential matches that are generated by a filter. It is usually the case that the amount of false positives with c_m or slightly more matching characters by far outweighs the amount of relatively similar false positives. Only in the rare instances where the query is a repeat with a very high copy number and the filter has a high minimum coverage will this relation be reversed. The amount of meaningful but unwanted potential matches can then be higher than the amount of low similarity false positives occurring by chance. For such cases, while better filter algorithms with a higher filtration efficiency will still perform better, the disproportionate amount of relatively close false positives will lead to smaller than expected differences between the filters. Usually however, such queries are filtered out with repeat masking. The second reason for using random queries was the goal of comparing the results for the EST and random database.

The single most important goal of the following experiments was the *relative* analysis of different filter algorithms with respect to speed and filtration efficiency. While the absolute numbers for the amount of hits and potential matches may be somewhat different in practice depending on the actual data being used, the results regarding the relative performance of the different algorithms retain their significance. This is important for two reasons. We of course wanted to compare our developments with previously known filters and determine by how much we were able to improve their performance. Indirectly however, it is also a prerequisite for the selection of good gapped shapes. We decided to evaluate the performance of filters based on the q-gram lemma, the filter criterion used by BLAST and compare them with filters using gapped q-grams. We look at both the k-mismatches and the k-differences problem but present their results separately. We used single diagonals and sets of $k + 1$ diagonals as match regions for our tests. All results were averaged across the set of queries.

6.3.1 Choice of Gapped q-grams

For BLAST and the classic q-gram filter the set of possible parameter choices was very small. In all cases there exists exactly one filter for a given value of q. This allowed us to evaluate all potentially interesting filters in a reasonable amount of time. However, for gapped q-grams things are different. For the k-mismatches problem where we have to consider a very large amount of shapes, shape choice becomes very important due to the huge variance in shape quality between the best and the worst shapes. We will illustrate this variance further in Section 6.4

In principle one could envision two approaches to arrive at good shapes for filtering with gapped q-grams. One would be the development of a method to construct or design good or even optimal shapes. The second is an exhaustive search of all possible shapes with either a theoretical or practical analysis of their properties. We have not been able to come up with a solution for the first idea and therefore decided to follow the second approach.

6.3.2 q and the Minimum Coverage

In order to select the best gapped shapes for the use in filter algorithms, we had to conduct an exhaustive search of all possible shapes that includes either a theoretical or practical analysis of the different shapes and leads to a selection of optimal or near-optimal shapes. Due to the large number of shapes a method for quick judgement of the expected filter speed and filtration efficiency of a given gapped shape is vital. The selected value of q and the minimum coverage, defined in Section 4.7.3 are used to predict the performance of a shape. In order to support our claims regarding the accuracy of our predictions we conducted several tests in which we used different gapped q-grams to match a set of 1000 random query strings of length 500 against the EST and the random database. For some tests with very high runtimes we restricted our experiments to only 100 of the 1000 queries.

Figure 6.2 contains the results of the experiments concerning the expected filtration efficiency. The x-axis is used to represent the minimum coverage of the filters and the y-axis is used for the average number of potential matches per query. As a comparison the figure also contains a line which represents the function used for predicting the number of potential matches with a constant factor chosen to place the line amidst the experimental data points. Two important results are displayed in this plot. The first is the good correlation between the predicted and the expected behavior. While by no means exact, the prediction based on the minimum coverage is nevertheless a good method to quickly compare different gapped shapes and select those which promise to have good filtration efficiency. The second result is the increased variance for higher

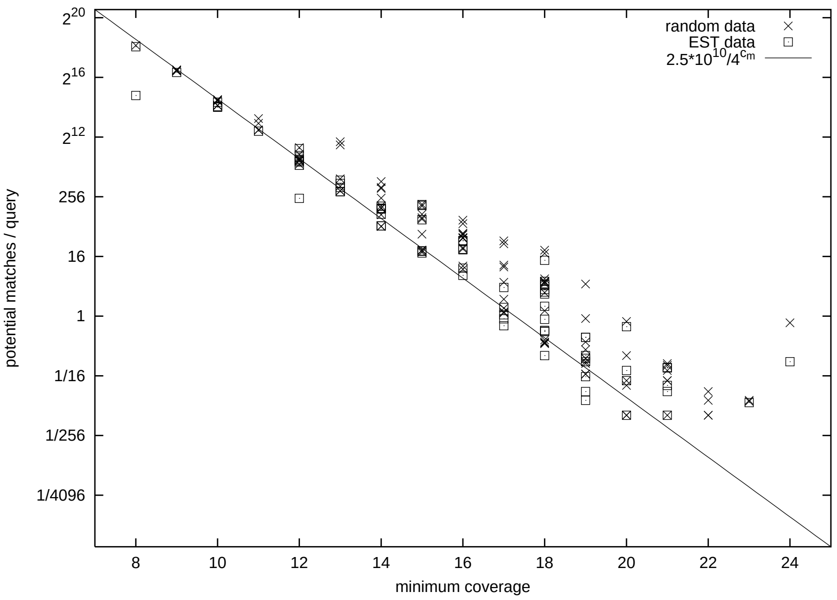


Figure 6.2: Correlation between expected and actual number of potential matches

values of the minimum coverage. These two combined suggest that a more advanced formula for predicting the filtration efficiency or an experimental analysis for a set of the best shapes with respect to minimum coverage could further improve the process of shape choice. As a first step in this direction we implemented code that counts the number of covers that are minimal for a given shape. For some gapped q-grams there exist more than one cover with minimal size. In this case the filtration efficiency is reduced due to the increased number of possible arrangements of the matching characters. Taking the number of different arrangements into account improves the prediction quality but also requires more time. It also might be interesting to analyze the covers of size $c_m + 1$ or maybe even $c_m + 2$.

To give an impression about the number of different shapes for a given set of parameters we would like to mention that for the case of the Hamming distance our exhaustive search considered roughly 25 million different gapped shapes. This number shows how important a simple and efficient method to judge shape quality is.

For shapes with only one gap and reasonable parameter values it is possible to experimentally evaluate all available shapes. While somewhat time-consuming this method provides an alternative to computing the minimum coverage of all shapes and using it as a guideline for filter selection. For the k-differences problem we analyzed all possible shapes for $w = 50$ and $k \in \{3, 4, 5\}$, in total about 300. We then examined the minimum coverage of the shapes with the best experimental results. In all but one case these shapes had maximal minimum coverage in the class of shapes with the same parameters. The single exception was a shape with the best experimental result but a minimum coverage one below the maximal minimum coverage of all shapes. However, in this case the shape with the highest minimum coverage only produced 10 % more potential matches than the shape with the best experimental results. This also supports the value of the minimum coverage as a measure of a shapes filtration efficiency.

Our measure for predicting filter speed based on the expected size of $H(P)$ turned out to be even more accurate. This is apparent, for example, in Figures 6.3 and 6.4. All filters with the same value of q end up producing sets of hits $H(P)$ of nearly the same size. In the figures this results in data points that have the same x-position. Also the data points of filters using one-gapped q-grams for the k-differences problem are shifted along the x-axis by a factor of roughly three which was expected due to the use of the three shapes from P.

After establishing two good measures for filter speed and filtration efficiency that can be evaluated quickly, we were able to select interesting shapes in our exhaustive search of all possible shapes. For each value of q we selected the shape with the highest minimum coverage for our experiments. For the k-mismatches problem we decided to include an 'average quality' shape, i.e. one with median minimum coverage, in our experiments. The purpose of this second class of shapes was to analyze and underscore the importance of a careful shape choice. For the k-differences problem, where we only use one-gapped shapes, the number of available shapes was much lower. In fact, in this case there were few enough shapes to allow a full experimental analysis of all shapes. We also used the minimum coverage to find good shapes. As we mentioned before in all but one case the shapes with the best experimental results had maximal minimum coverage among all shapes. We define three shape classes for the different values of q and k to which we will refer in our plots and in the discussion of the results:

- *Best Hamming* For the k-mismatches problem this is the shape with the highest minimum coverage. To choose between multiple shapes with the same minimum coverage we used the number of distinct covers of minimal size, the number of distinct covers of the minimum size plus one, and the threshold as secondary keys (in this order).

- *Median Hamming* For the k-mismatches problem and for each span s, all (q, s)-shapes (without mirror images but including those with threshold zero) were ordered by the minimum coverage, and by the same secondary keys as for the best shape, and the median shape in this order was identified. Of these shapes (one for each s) the best one was used in the experiments. The chance that a randomly chosen shape is better than this shape is at most half. We added this class of shapes to illustrate the importance of a good shape choice. The results for theses shapes make it clear that the best shapes are highly likely to be far better than randomly selected shapes.

- *Best Edit* This is the shape with the best experimental results of all one-gap shapes used for the k-differences problem. In all cases except one, this shape also had maximal minimum coverage. The single exception was a case where the shape with the highest minimum coverage had a large number of distinct arrangements with minimal cover.

For $k = 4, 5$ and $w = 50$ the chosen shapes are shown in Table 6.2. A missing shape means that the shape in that category had a threshold of zero. This also implies that for this value of q there will be no experiment with this class of shapes. For higher values of q it is quite common that more than half of all possible shapes have a threshold of zero. In this case the median shape also has a threshold of zero and no median data point is shown in the plots displaying our experimental results.

$k = 4$			
q	best Hamming	median	best edit
8	##--#------#-------#----##-#	#-#--#-#--##---#-#	#####---###
9	##--#-#--------#------##--#-#	####-#-#------###	#######---##
10	##--#-#-----#------#--##--#-#	###-###--#-----###	#######---###
11	###-#--#------#------#-#-#--##	######-###-#-#	#######--####
12	##-#-#---##---------#-##-#---##	##-##-####-###-#	#########--###
13	#########-####		#########-####
14	####-##---####-##		
$k = 5$			
q	best Hamming	median	best edit
8	##--#-#---------------##--#-#	#-#--####---#-#	######---##
9	###--#--#-#---#-##	######--#-#-#	######---###
10	###--#--#-#--###-#	##-##--#-#-###-#	#######-###
11	#######-##-##		
12	###-#--###-#--###-#		

Table 6.2: Some of the shapes used in Figures 6.3 and 6.4.

6.3.3 Results for the Best Shapes

After determining the best shapes according to our quality measures, we were able to conduct experiments with two main goals. The first was a comparison between different filters. The second was to support our predicted filter properties by additional experimental results. Both goals were achieved and the results of the experiments are presented in this section. We first discuss the k-mismatches problem and then proceed to the k-differences problem.

For the k-mismatches problem we display the results of our experiments in several plots contained in Figures 6.3 and 6.4. We present data for $k = 4$ and $k = 5$. In order to avoid side effects caused by repeats in actual DNA data we used the random database for these tests. However, we also conducted some tests on the EST database. The results of those experiments are shown in Figure 6.5. For each value of k there are two different plots. One shows the values of q and the minimum coverage c_m for all shapes we considered, the other contains the experimental results, namely the average number of hits and potential matches per query. In both cases we used the same axis and scaled the plots to visualize the correlation between the predicted and the experimental values. For the case of $k = 4$ some data points for gapped q-grams are missing since they had zero potential matches and could not be displayed on the logarithmic scale.

There are several things of interest in Figures 6.3 and 6.4. First, there is a good correlation between the predicted behavior and the actual experimental results. In most cases the correlation is very good, only for $k = 5$ and $q = 8$ we see a moderate difference between the two values. A possible reason for this lies in the low absolute number of potential matches. The total amount of potential matches recorded for all 1000 queries was only 12.

Second, the performance of the median shapes shows that while a randomly chosen shape is likely to be better than a contiguous one, much better results can be achieved with a careful shape selection. In many cases there are huge differences between the best shape and the median shape. Also, for high values of q it is quite unlikely that a randomly picked shape is one of those with a positive threshold.

Finally we would like to point out that given a fixed value of q the best gapped shapes can reduce the amount of potential matches by several orders of magnitude if compared to ungapped shapes. It is also worth noting that due to the higher values of q that are possible with gapped shapes, the size of $H(P)$ can be reduced by a factor of roughly 2^8 for both $k = 4$ and $k = 5$.

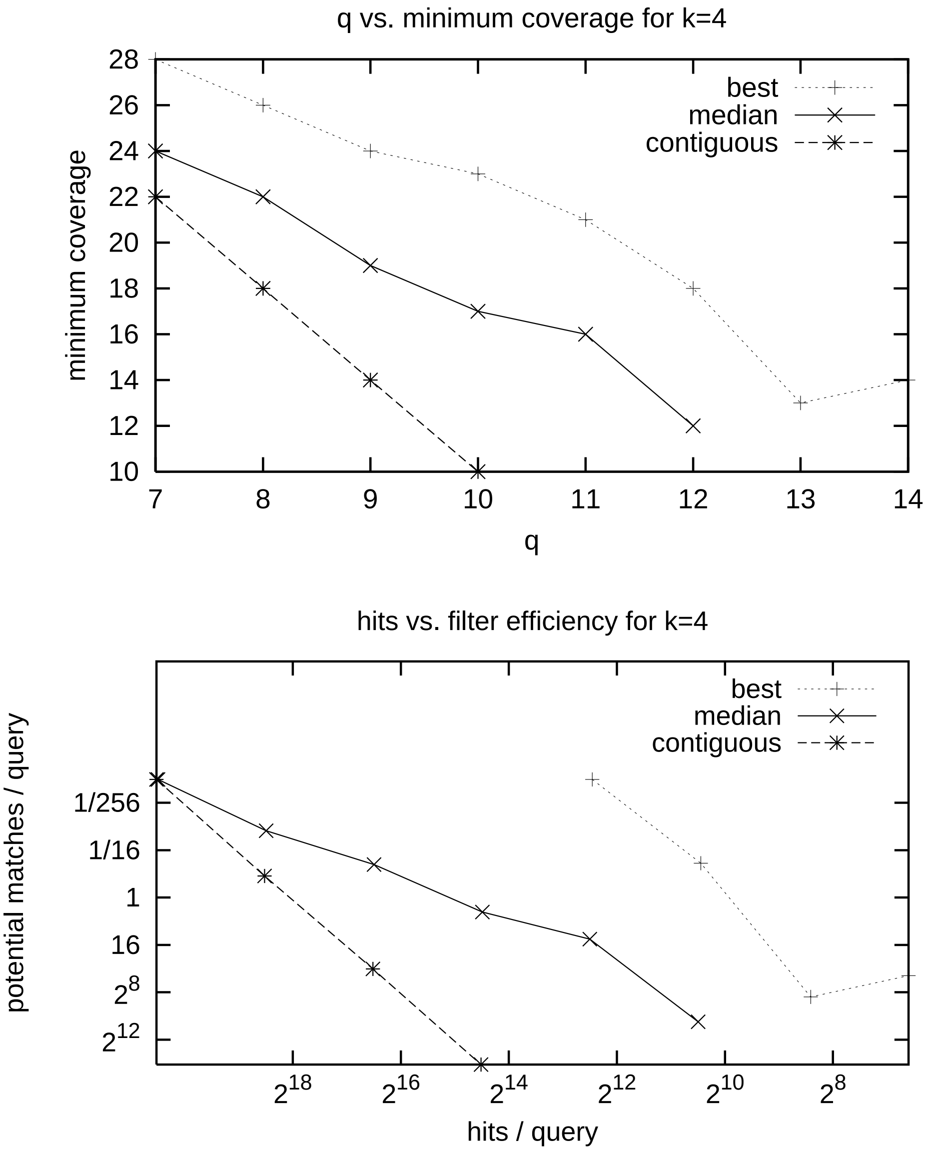


Figure 6.3: Comparison of three classes of shapes, random DNA, $k = 4$, Hamming distance

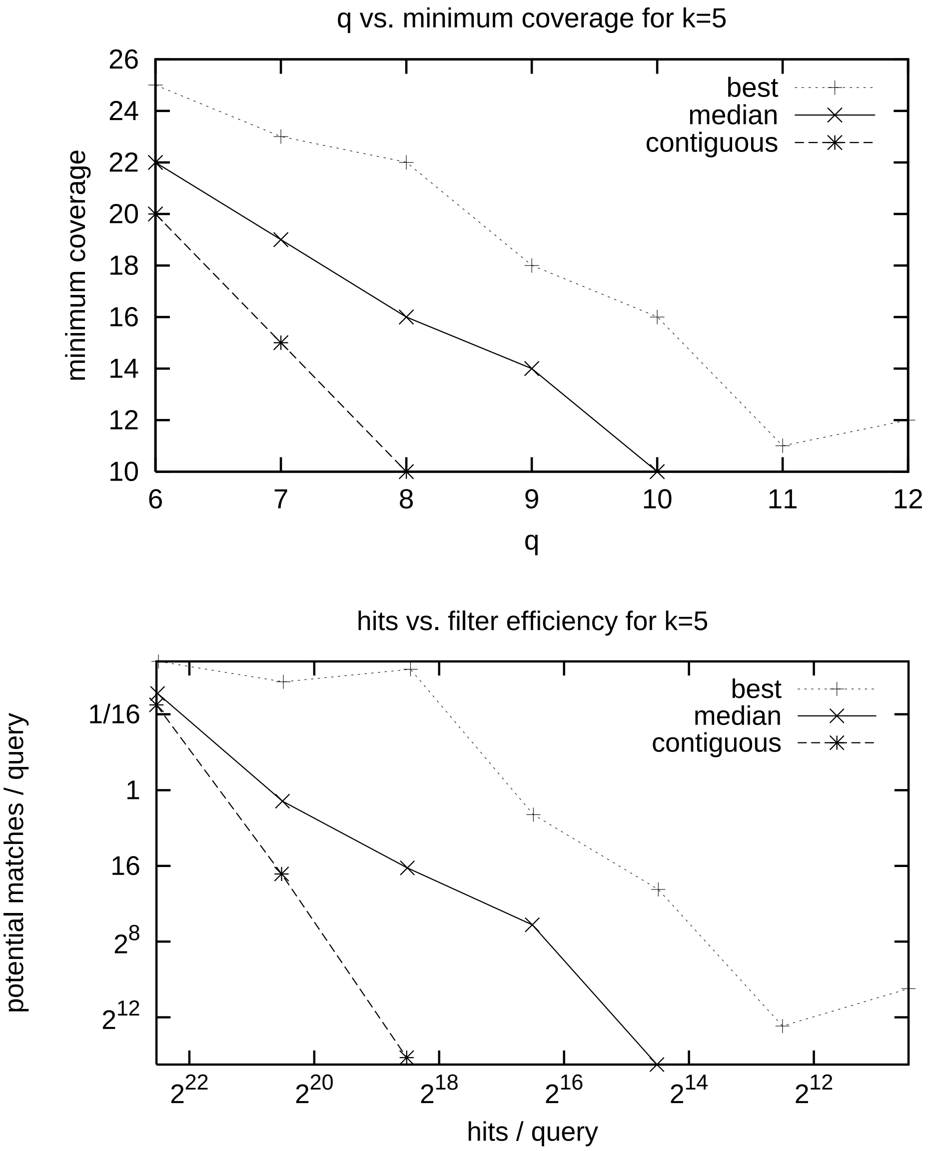
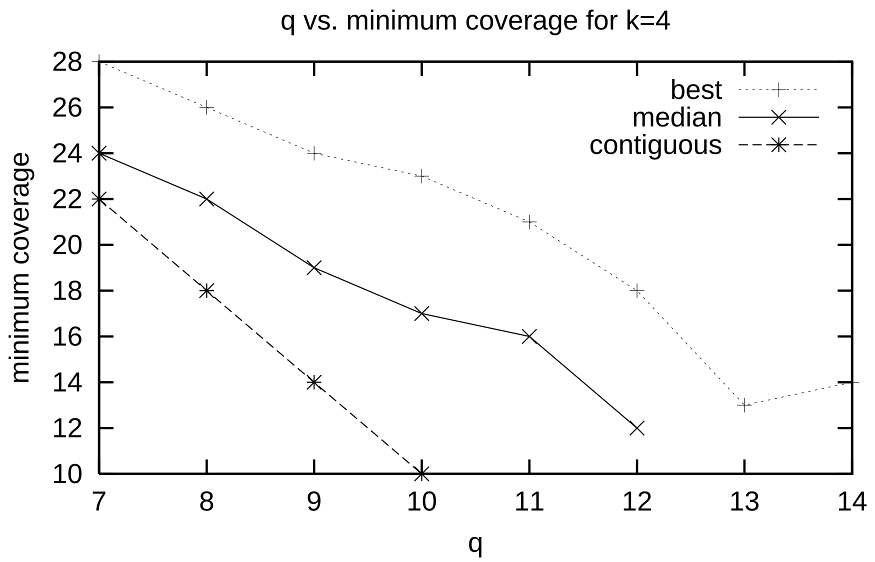


Figure 6.4: Comparison of three classes of shapes, random DNA, $k = 5$, Hamming distance

The plots in Figure 6.5 contain results for $k = 4$ when using the EST database instead of the random database.

The results look very similar to those for the random database. The reason for this is the use of random strings as queries. If using queries with many repetitive elements the performance drops significantly. In practice this problem is usually avoided by repeat screening before searching in a DNA database.



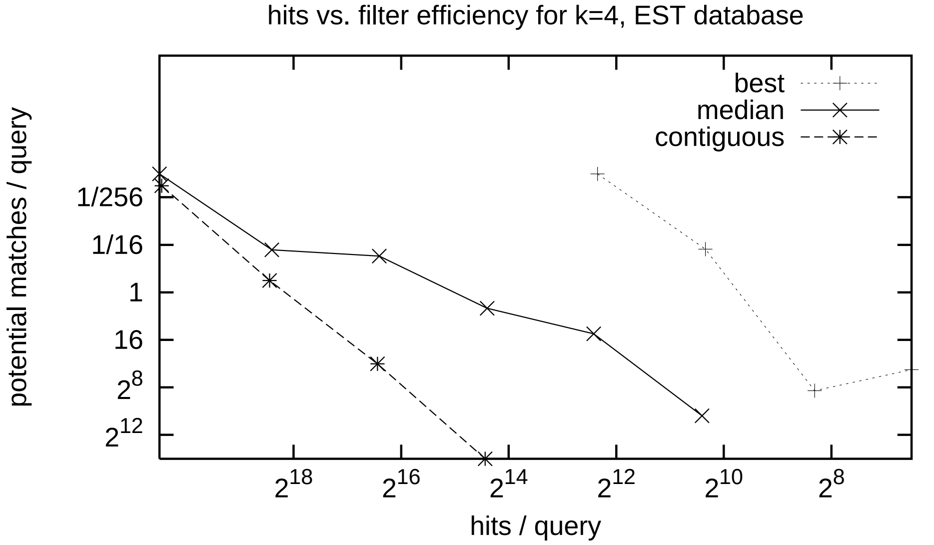


Figure 6.5: Comparison of three classes of shapes, EST DNA, $k = 4$, Hamming distance

For the k-differences problem we ran similar tests. This time we only used the random database and used the previously described one-gap shapes for filtering. We compare the results with those of contiguous, i.e. ungapped q-grams in Figure 6.6. In order to better take the 3 shapes per starting position in the query into account we show the expected number of hits in the first plot instead of the value of q. For the contiguous case the experiments were conducted with $q \in \{7, 8, 9, 10\}$ for $k = 4$ and $q \in \{6, 7, 8\}$ for $k = 5$. For the one-gap shapes we used $q \in \{8, 9, 10, 11, 12, 13\}$ for $k = 4$ and $q \in \{8, 9, 10\}$ for $k = 5$

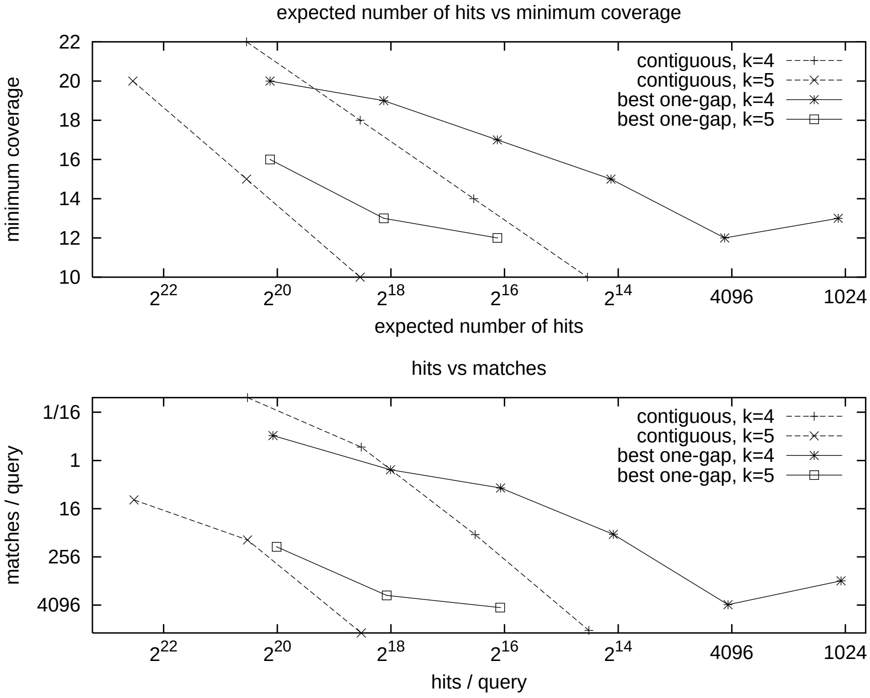


Figure 6.6: Comparison of ungapped with gapped shapes for $k = 4, 5$, edit distance.

Which value of q and which shape should be chosen in practice depends on the actual implementations of filtration and verification phase and the match region being used. The total amount of hits $|H(P)|$, for example, plays a greater role when using more complex match regions. However, the results and measures we provide are good guidelines for choosing suitable parameters and shapes for a specific implementation.

6.4 The Distribution of Shape Quality

An important part of our work was the development and analysis of methods for predicting filter speed and filtration efficiency of a given filter. To emphasize how important these techniques are, we want to illustrate the great variance in filtration efficiency of the different gapped shapes that can be used for filtering. The large differences between the best and the worst or average shapes and the small number of good shapes justify the efforts we underwent to find good shapes. While the idea of using random gapped shapes, which was introduced in 1993 [CR93], will still deliver better results than the use of ungapped shapes (as can be seen by looking at the median shapes in Figures 6.3 and 6.4), careful choice of the gapped q-grams vastly improves performance and allows taking full advantage of their potential.

In Figure 6.7 we display the distribution of the minimum coverage of all gapped shapes with $q = 8$ and a span of 17 for patterns with $w = 50$ and at most $k = 5$ errors and the k-mismatches problem. We only consider shapes with a positive threshold and treat pairs of shapes where one is a mirror image of the other as one shape.

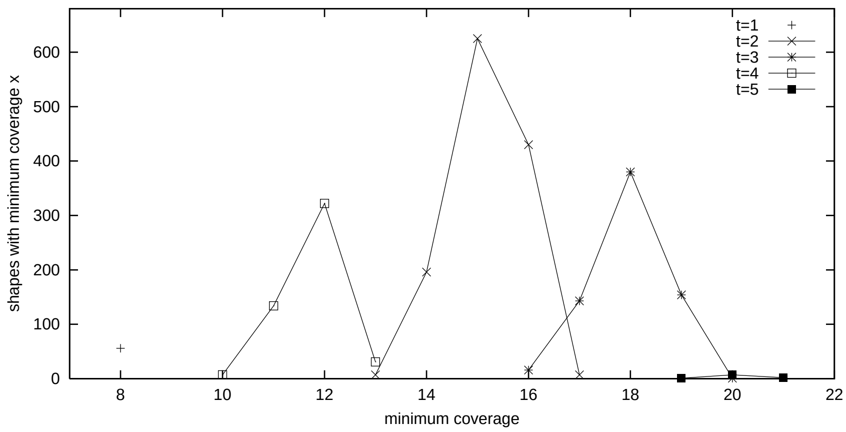


Figure 6.7: Distribution of threshold and minimum coverage of $(8, 17)$-grams for $w = 50$, $k = 5$

It is quite obvious that the difference between the shape with the smallest and the largest minimum coverage is huge in terms of expected filtration efficiency. The shape with the highest minimum coverage has $c_m = 21$, the one with the lowest has $c_m = 8$. This results in a difference of 13 which would reduce the expected number of potential matches by a factor of $|\Sigma|^{13}$. When looking at the set of all gapped shapes with $q = 8$ that can be used to search for patterns of length $w = 50$ with at most $k = 5$ errors, the difference between the best and the worst shape increases even further. The highest minimum coverage for the set of all possible shapes with $q = 8$ is in fact 22 for a span of 29 as can be seen in Table 5.2. Another important fact is also nicely displayed in Figure 6.7: the amount of shapes with maximal minimum coverage is very low and picking a random shape will usually result in a value for c_m significantly below the optimal value. Distribution plots for different parameters look similar.

While the minimum coverage does not exactly predict the filtration efficiency of a given filter, it is the best method we have available as of now to quickly compare different filters and for the case of gapped q-grams to pick those that have optimal or close to optimal filtration efficiency. It provides the important ability to select gapped shapes that are far better than randomly chosen gapped shapes and use them for approximate string matching.

6.5 The Match Zones

We also conducted experiments to evaluate filter behavior for the different match zones. Since it is sometimes sufficient to use a lossy filter, these experiments were conducted to gain insight into how different filters work in the lossy zone. Of course those error levels in the lossy zone are of special interest for which the filter has a high recognition rate, i.e. recognizes most of the approximate matches with that amount of errors as potential matches. See Section 4.7.4 for the definition of the lossy zone. A good example for such an algorithm is BLAST. When using the default parameters, i.e. a word length of 11, BLAST is only guaranteed to find patterns of length 50 with edit distance at most $k = 3$ which corresponds to an identity of 94 %. This means that there exist pairs of strings of length 50 with an edit distance of four which are not reported as potential matches by BLAST. However, the chances of having the four errors distributed in an arrangement that does not produce a single 11-gram match is quite low, i.e. there are only a small number of strings in the 4-neighborhood of a query of length 50 that are not recognized by BLAST. This is equivalent to a high recognition rate. In practice researchers routinely use BLAST with these parameters to search for approximate matches with much more than just three errors. The reason why this still works is the relatively high recognition rate for error values not too far above k.

In order to characterize filter behavior in the lossy zone, we decided to construct an artificial test database. This database is based on a randomly generated query string P of length 50 which was checked to contain no repeats of length q or longer. Using P as a base, we constructed sets of strings with a fixed amount e of errors placed at random locations in P. We ensured that no two errors were located at the same position. For each error level e that was of interest we generated a set of sample strings that had a distance of exactly e from P and stored all these samples in a database S_e. In principle these sets of samples are subets of the e-neighborhoods of P. We then used different filters to compare P with S_e for all relevant values of e. For each value of e we counted how many samples from S_e were recognized as potential matches by the filter. The percentage of samples that were recognized provides an estimate of the probability that a given filter will detect a match with distance e. We are well aware of the fact that the distribution of errors in practical applications will most likely be different from our simple assumption. Also, the use of samples introduces additional errors into these experiments. Nevertheless the results we generated for a set of 1000 samples per value of e show good consistency with the results of our experiments on the EST and random database.

An interesting side-effect of our experiments were approximations of Figure 4.6 on page 50. Even though they are just approximations, they still allow a visual comparison of the filter behavior and the filtration efficiency of different algorithms. The experiments with the constructed database result in relatively smooth curves when plotted with e along the x-axis and the recognition rate along the y-axis. Generating more samples for each value of e could further improve this though. the In the following we will present and interpret results for various filters.

6.5.1 Results for the q-gram Lemma and the Edit Distance

Our initial tests used the classic q-gram lemma to detect potential matches. The Figures 6.8, 6.9 and 6.10 show the results for $|P| = 50$ and t computed with $k \in \{3, 4, 5\}$. For values of e close to but above k, i.e. the lowest values of e in the lossy zones, the recognition rate is still high but then starts to drop rapidly. In Figure 6.8, for example, all filters have a recognition rate above 70% for $e = 4$. A second interesting observation is the effect the choice of q has on the recognition rate. Higher values increase the recognition rate in the lossy zone but also increase the error range covered by the lossy zone, i.e. extend it to the right. In the plots this results in a curve that drops off slower and reaches the bottom further to the right. There are two advantages of higher values for use in lossy filters. One is the higher recognition rates for error values e that are higher but close to k. This extends the useful error range for many applications. The second advantage is of course the size reduction of $H(P)$, the set of hits that need to be processed in the filtration phase. However it also leads to an increase in recognition rate for values of e that are much higher than k. This is a major disadvantage since it leads to substantially more false positive resulting in a sharp decrease of the filtration efficiency.

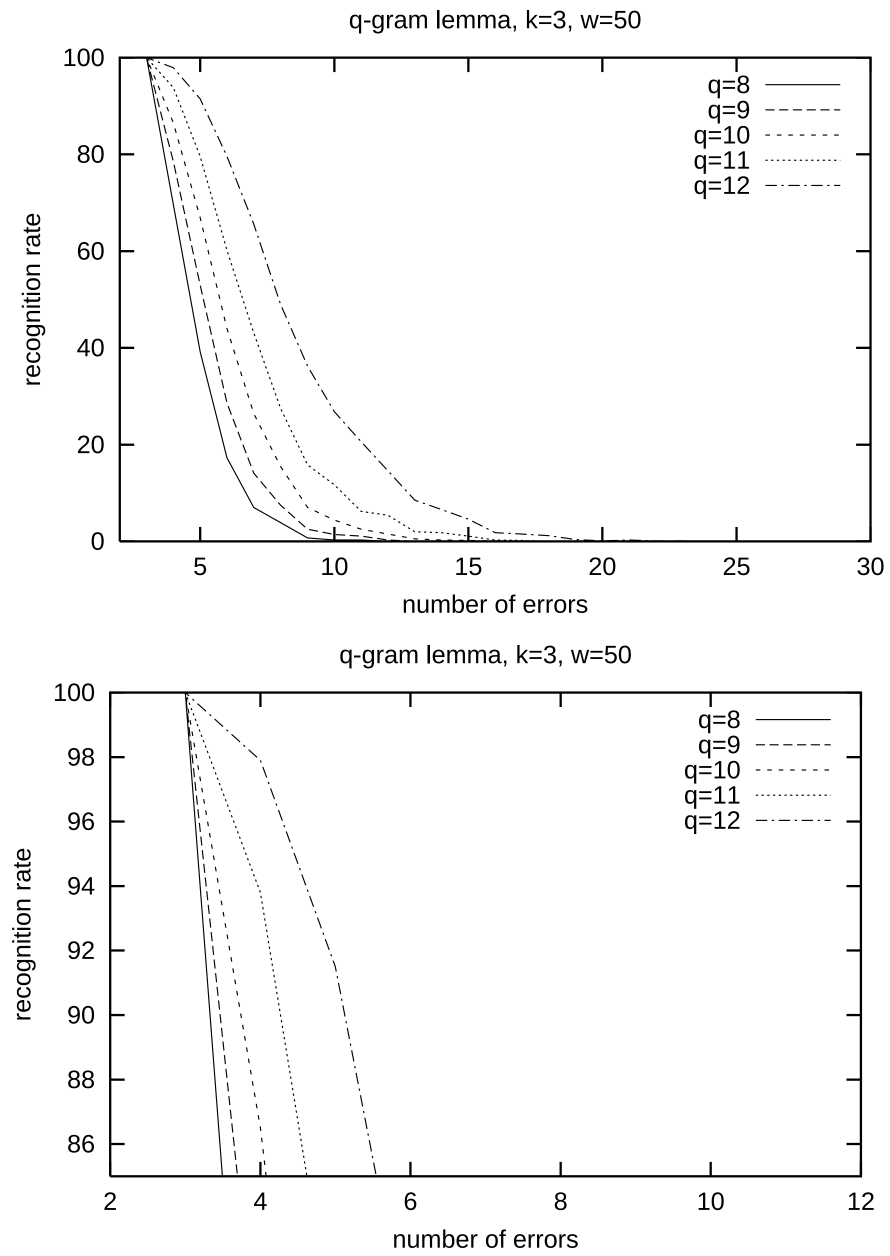


Figure 6.8: Recognition rates for the q-gram lemma, t computed with $k = 3$, $w = 50$ and $q \in \{8, \dots, 12\}$

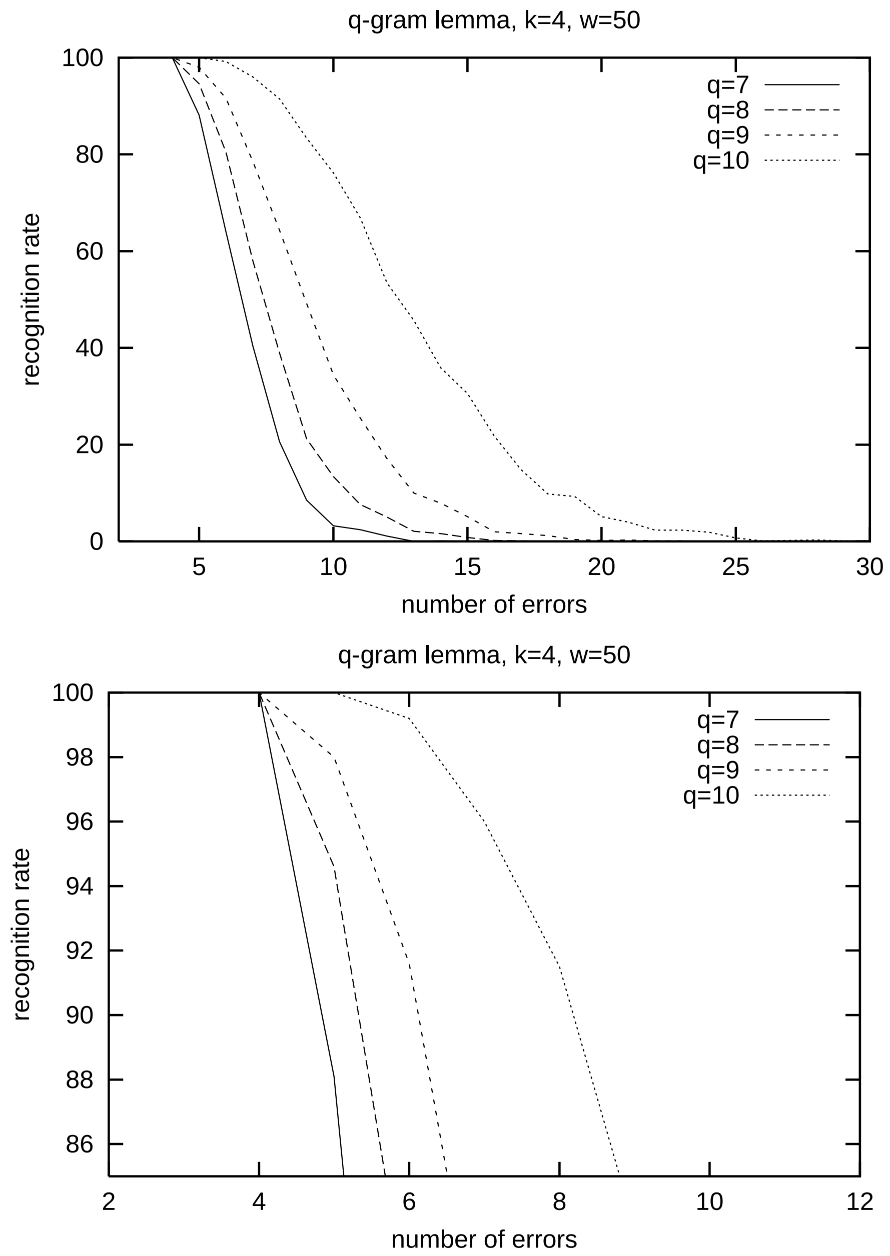


Figure 6.9: Recognition rates for the q-gram lemma, t computed with $k = 4$, $w = 50$ and $q \in \{7, \ldots, 10\}$

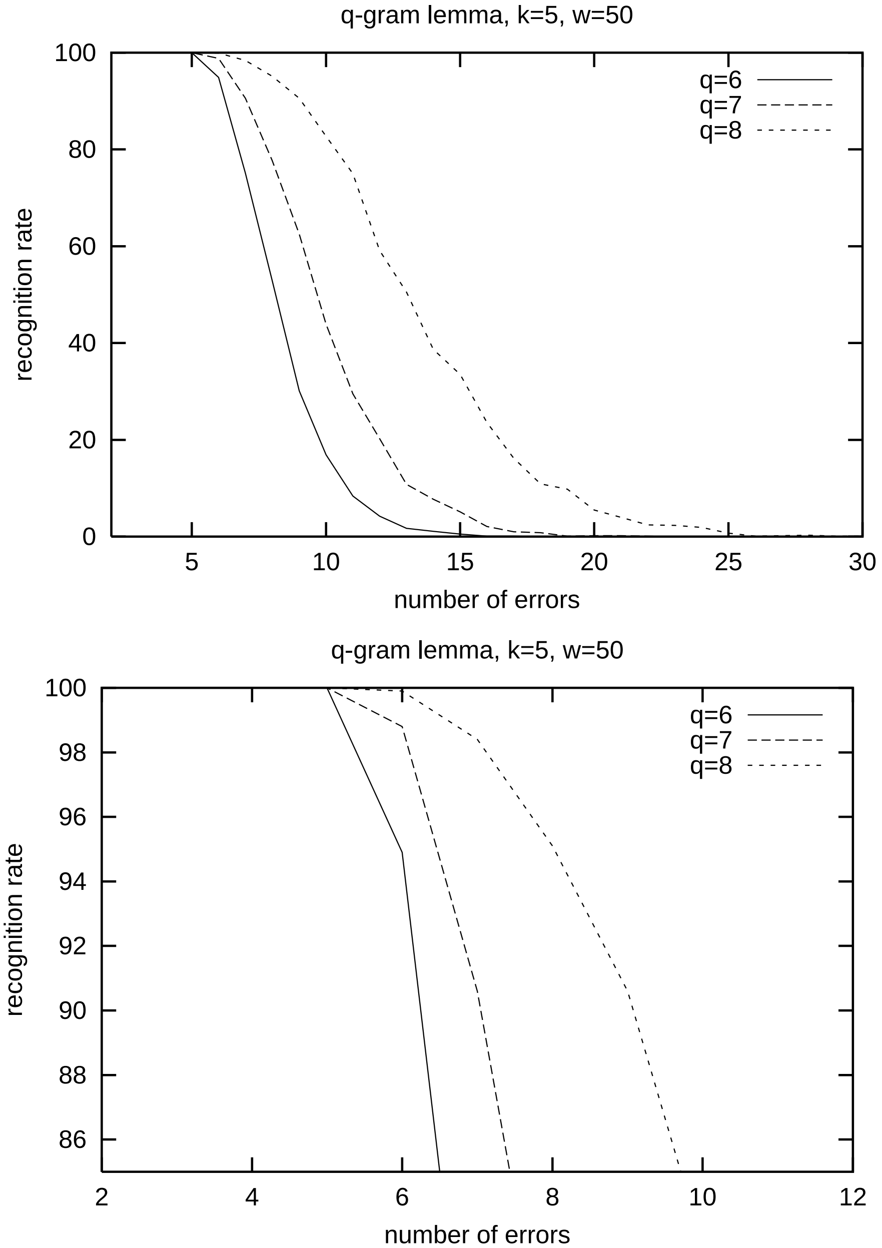
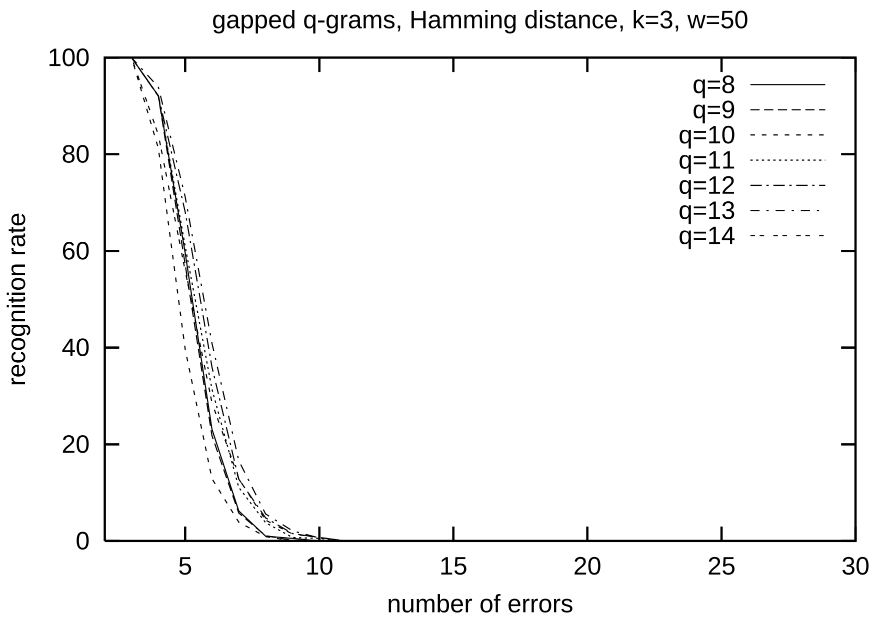


Figure 6.10: Recognition rates for the q-gram lemma, t computed with $k = 5$, $w = 50$ and $q \in \{6, \ldots, 8\}$

6.5.2 Gapped q-grams for the Hamming Distance

The next set of filters we tested were filters using gapped q-grams to solve the k-mismatches problem. The Figures 6.11, 6.12 and 6.13 show the results for $|P| = 50$ and t computed with $k \in \{3, 4, 5\}$. For the lowest values of e in the lossy zones recognition rate is still high but again it starts to drop rapidly for higher values of e. While the choice of q has the same general effect on the recognition rate as for the case of the ungapped q-grams, there is a deviation for $k \in 3, 4$ and $q = 8$. Here the filter with $q = 8$ has a higher recognition rate for error values just above k than those with $q \in 9, 10, 11$. However, for higher error values the expected behavior can be observed and the filter with $q = 8$ has the lowest recognition rates.

A general observation is the fact that the drop in recognition rate is much sharper than for the classic q-gram lemma. This visualizes the dramatically reduced amount of false positives for the gapped q-grams. It is of course caused by the higher minimum coverage which narrows the error range for the lossy zone. An interesting pair of two filters with roughly the same amount of potential matches are the classic filter for $k = 4$ with $q = 8$ (see Figure 6.9) and the gapped filter for $k = 4$ with $q = 12$ (see Figure 6.12). The amount of false positives these filters produce is roughly comparable (in fact, the classic filter produces about four times more, see also Figure 6.3) but the amount of hits they have to process differs by approximately a factor of $|\Sigma|^4$. This means that the verification phase requires far less time while the verification phase will be comparable. In addition to that the gapped q-gram achieves significantly higher recognition rates for $e \in \{5, 6, 7\}$ (99%, 93% and 82% compared to 95%, 81% and 48%). This makes the gapped filter better suited for use as a lossy filter since it's recognition rate remains relatively high for $e \leq 7$. It is an important new result that we only arrived at while conducting these experiments.



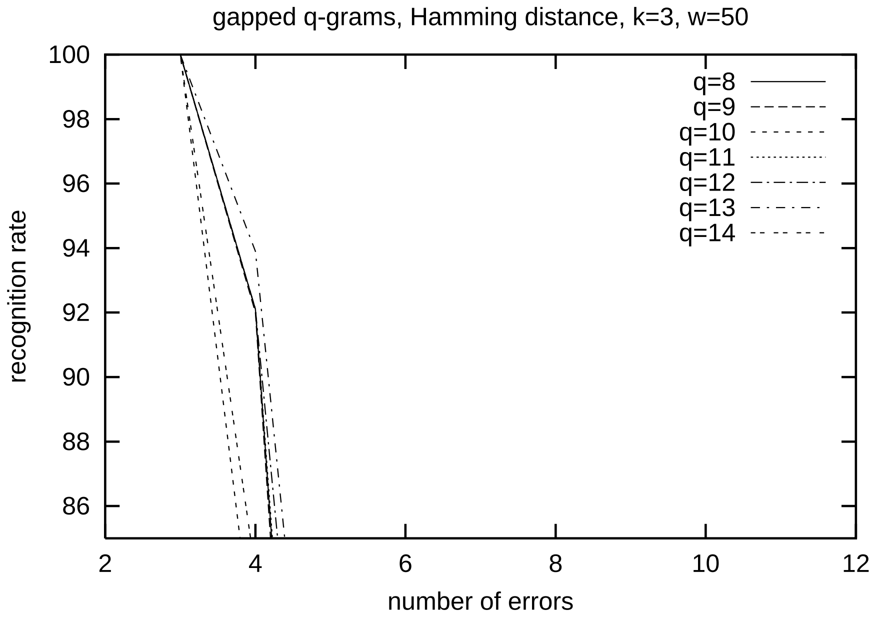
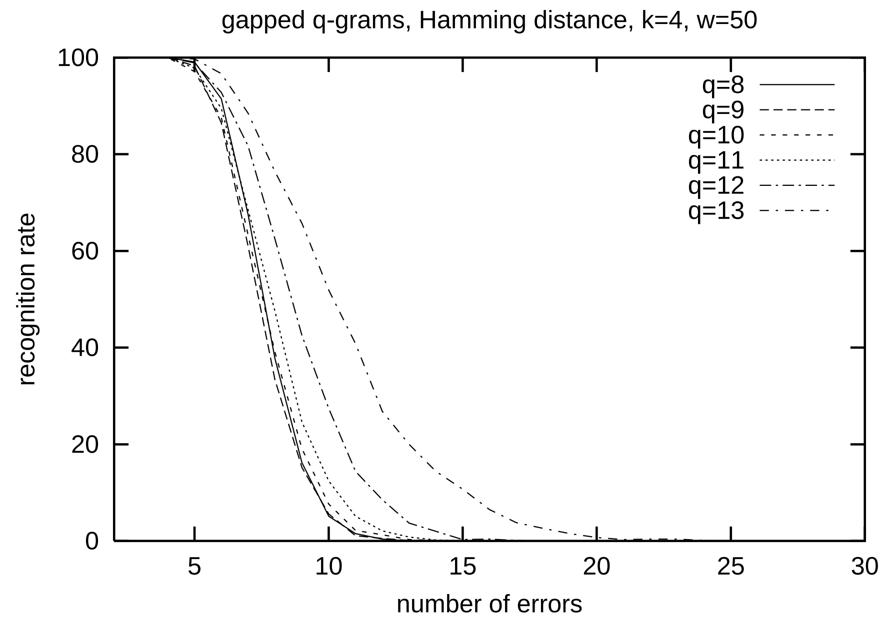


Figure 6.11: Recognition rates for gapped q-grams and the Hamming distance, t computed with $k = 3$, $w = 50$ and $q \in \{8, \ldots, 12\}$



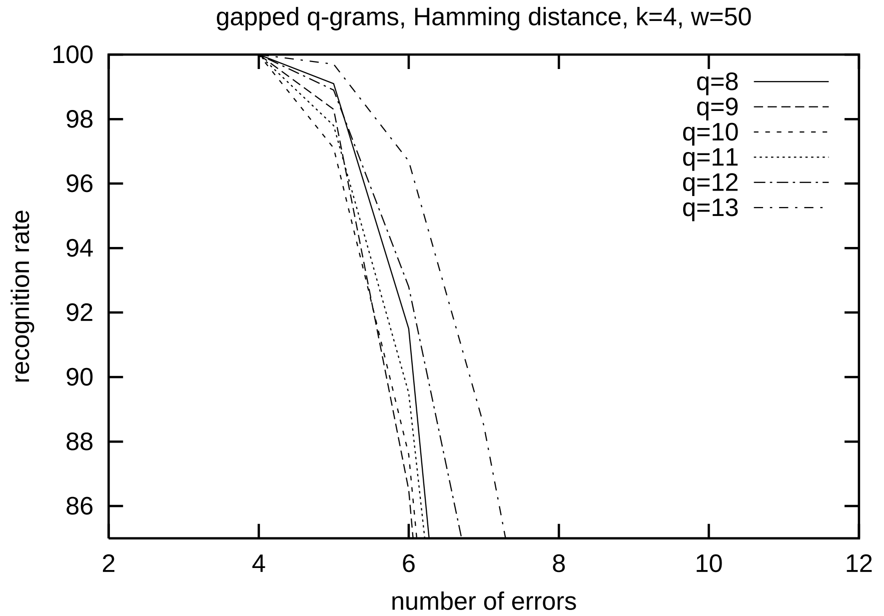


Figure 6.12: Recognition rates for gapped q-grams and the Hamming distance, t computed with $k = 4$, $w = 50$ and $q \in \{7, \ldots, 10\}$

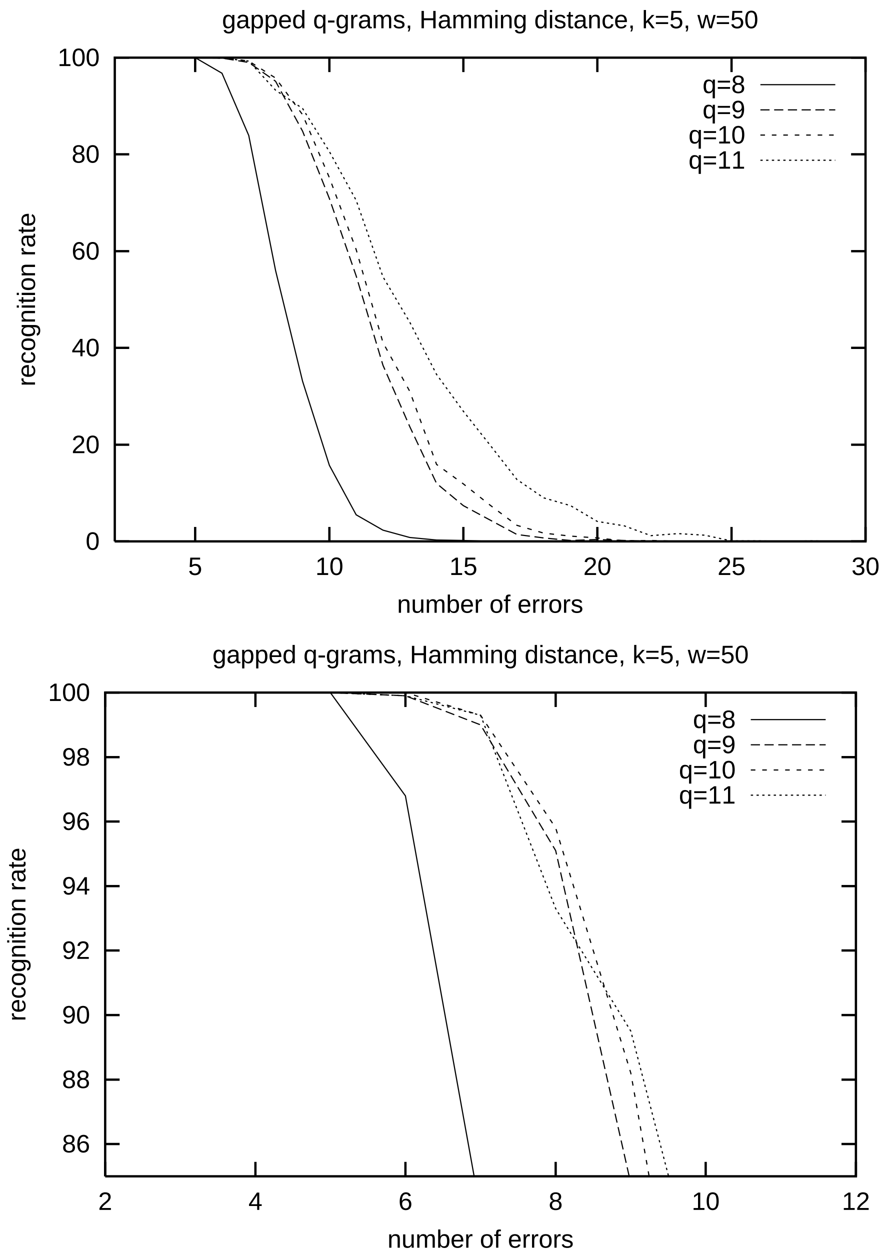
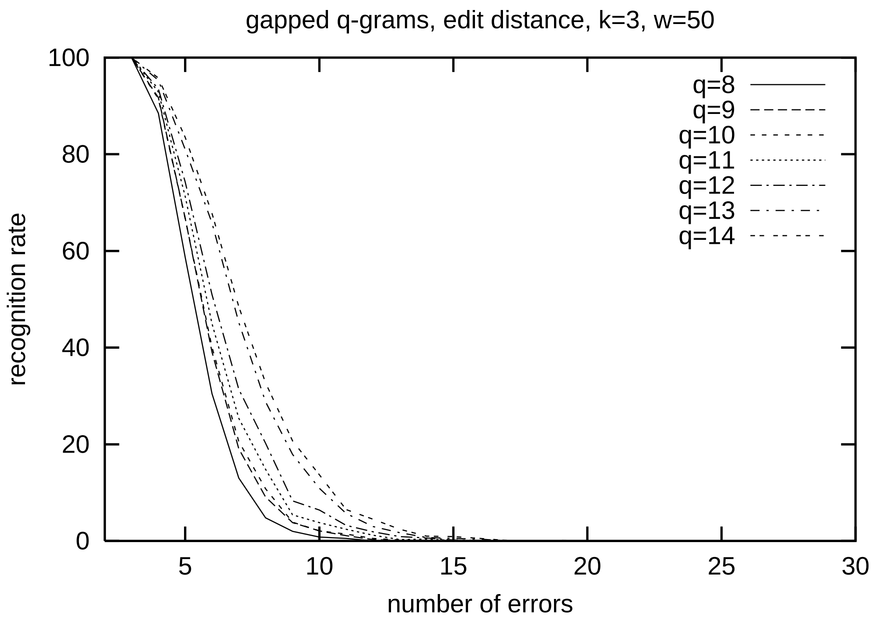


Figure 6.13: Recognition rates for gapped q-grams and the Hamming distance, t computed with $k = 5$, $w = 50$ and $q \in \{6, \ldots, 8\}$

6.5.3 Gapped q-grams for the Edit Distance

We also tested filters using one-gap q-grams to solve the k-differences problem. The Figures 6.14, 6.15 and 6.16 show the results for $|P| = 50$ and t computed with $k \in \{3, 4, 5\}$. For the lowest values of e in the lossy zones recognition rate is still high but again it starts to drop rapidly for higher values of e. For the one-gap q-grams the value of q has the same general influence on the recognition rate as for the case of the filter based on the q-gram lemma.

A comparison with the results for classic q-grams yields results along the lines of those presented in Figure 6.6. Gapped filters with higher values of q have a comparable behavior in the lossy zone for higher values of e (e.g. for $k = 4$ gapped $q = 11$ and classic $q = 9$). However, for values of e close to but above k they have higher recognition rates (same filters for $e \in \{5, 6, 7\}$: 99.7%, 95.6% and 82% compared to 98%, 91.8% and 81%). This further supports the suitability of gapped filters for lossy filtering. Again we can observe that for the k-differences problem gapped q-grams are superior to contiguous q-grams. However, the advantages are not as pronounced as for the k-mismatches problem.



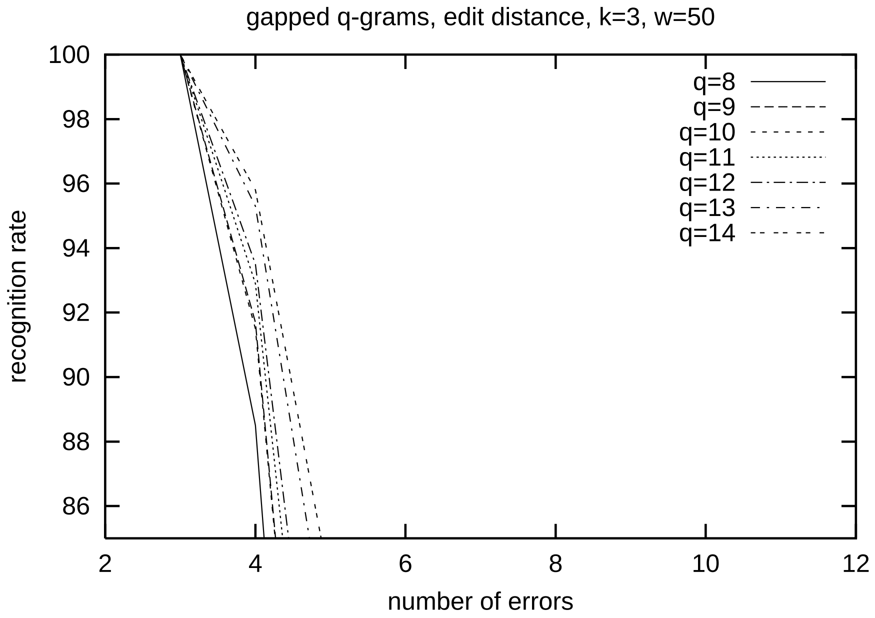


Figure 6.14: Recognition rates for gapped q-grams and the edit distance, t computed with $k = 3$, $w = 50$ and $q \in \{8, \ldots, 12\}$

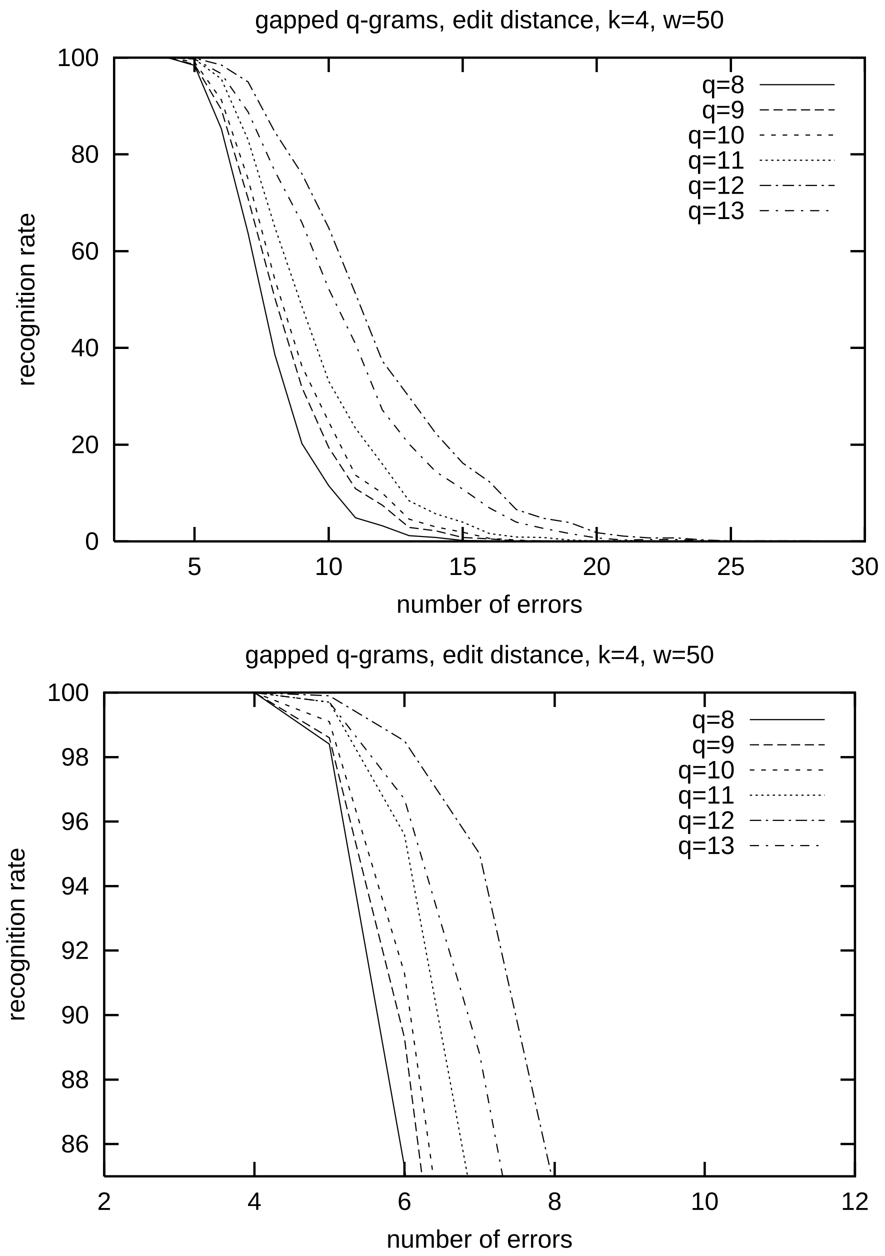
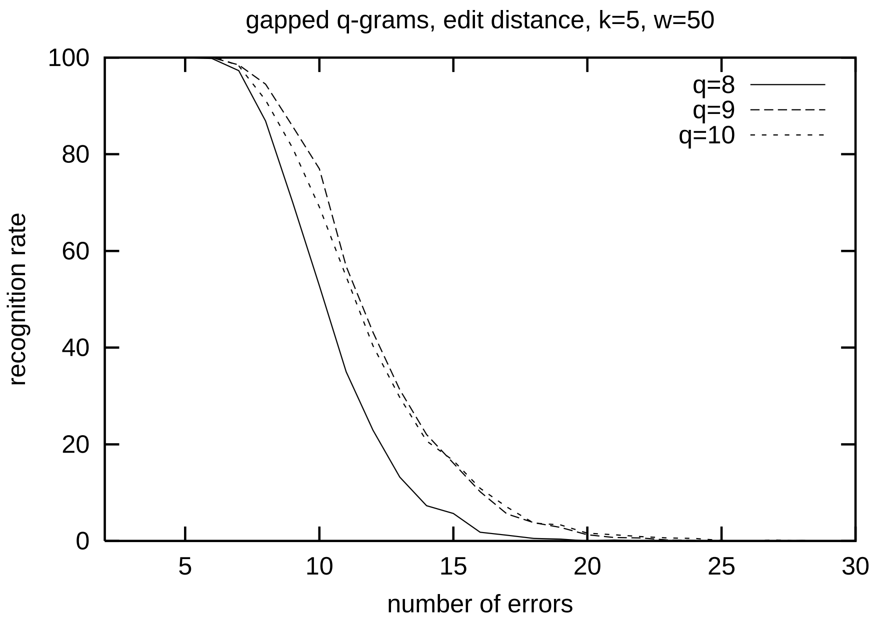


Figure 6.15: Recognition rates for gapped q-grams and the edit distance, t computed with $k = 4$, $w = 50$ and $q \in \{7, \dots, 10\}$



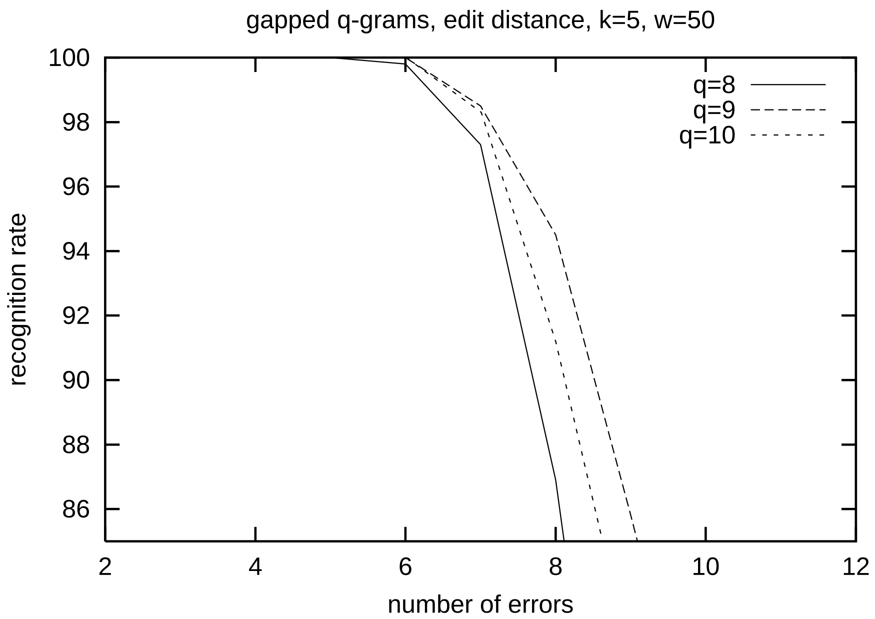
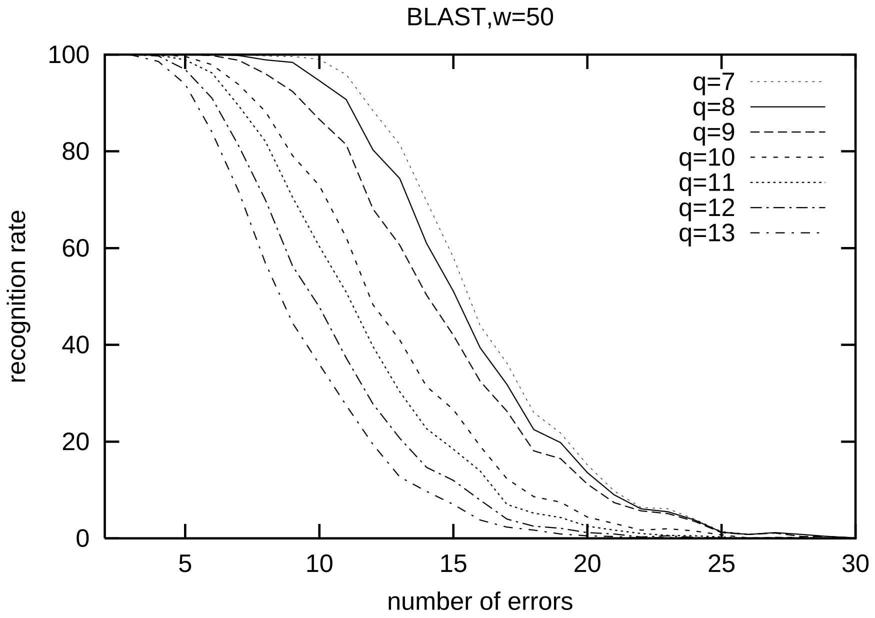


Figure 6.16: Recognition rates for gapped q-grams and the edit distance, t computed with $k = 5$, $w = 50$ and $q \in \{6, \ldots, 8\}$

6.5.4 BLAST

Finally we also ran BLAST on the constructed databases to evaluate its behavior and compare it with other filter algorithms. Figure 6.17 contains the results for $|P| = 50$ and BLAST word lengths $q \in \{7, \ldots, 14\}$. Here the general behavior with respect to the value of q is the same as for all other filters. A notable difference however is the much slower drop-off of the recognition rates for higher values of e. The reason for this is of course the low minimum coverage especially for lower values of q. It depends on the practical problems whether the results stemming from error levels with a low recognition rate are still worth the effort. For applications where any match with a high distance is valuable, the BLAST filter criterion is more powerful than the other methods discussed above. Assume, for example, that a search for very distant approximate matches (i.e. a high error e) has to be conducted in a limited amount of time but relatively small fraction of the true matches is sufficient. For such an application dynamic programming may be too slow and an algorithm like BLAST would be a good solution. If one requires a high or very high recognition rate however, the BLAST filter criterion falls behind filters based on gapped q-grams for moderate values of k.

As a side note, we would like to point out BLAST's relatively poor performance in regions of medium similarity. With default word length our experiments show a recognition rate of only 81% at an error rate of 16% (i.e. $e = 8$) that drops to just below 20% for $e = 15$ (still 70 percent identity!). Of course things look better for matches longer than just 50 characters since error distributions that knock out all matching substrings of length q become less likely. Nevertheless we were somewhat surprised by these figures.



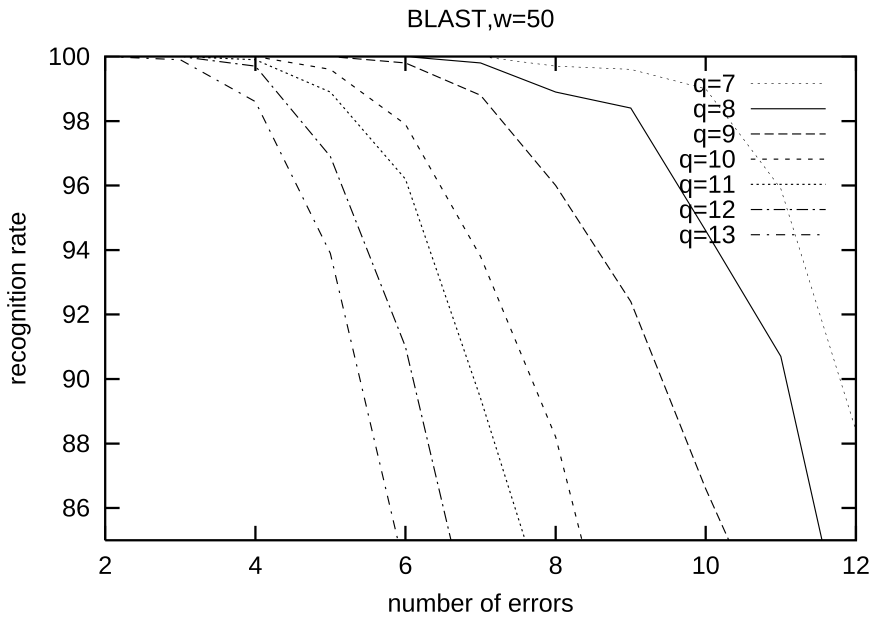


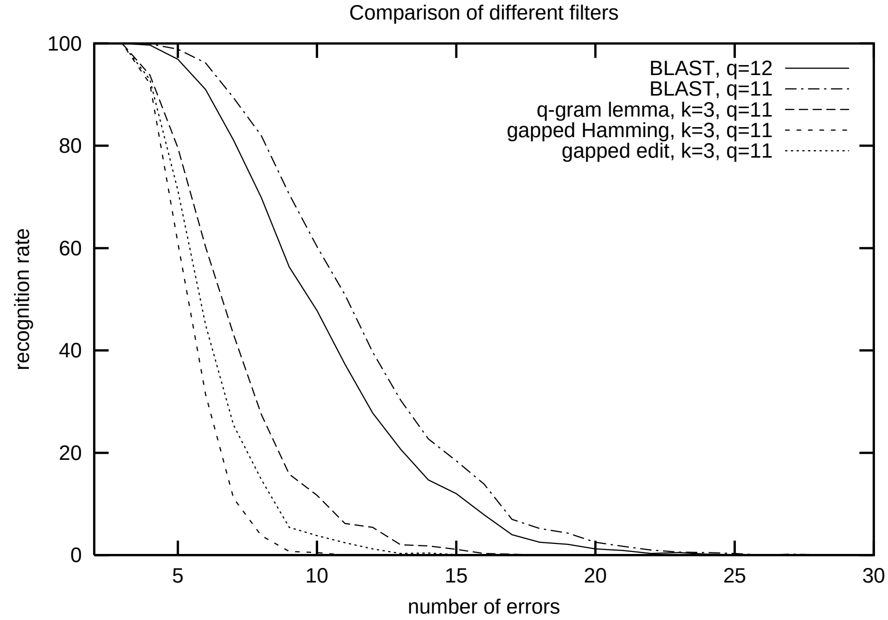
Figure 6.17: Recognition rates for BLAST with word length $q \in \{7, \ldots, 13\}$

6.5.5 Interesting Comparisons

After analysing different filter algorithms we wanted to compare their performance using the results of the constructed databases.

Figure 6.18 shows a comparison of filters that are guaranteed to locate all approximate matches with at most $k = 3$ errors for $w = 50$. We include contiguous q-grams, gapped q-grams and BLAST in this comparison and fixed q at a value of 11 in order to keep the amount of expected hits roughly equal. For BLAST we also allowed a value of $q = 12$, which is the highest value of q for which BLAST is guaranteed to detect all approximate matches to a pattern P of length 50 with edit distance of at most $k = 3$. For use as a lossless filter, the amount of false positives generated is lowest for gapped q-grams and the case of the Hamming distance, somewhat higher for gapped q-grams and the edit distance, higher again when using the classic q-gram Lemma and highest for BLAST.

For $k = 4$ and $k = 5$ we show the comparisons in Figures 6.19 and 6.20 (dropping to $q = 10$ and $q = 8$ respectively). For $k = 4$ the q-gram lemma has a threshold of 1 and therefore the same behavior as BLAST. In both cases we can observe relative performance similar to the case of $k = 3$ for all filter algorithms.



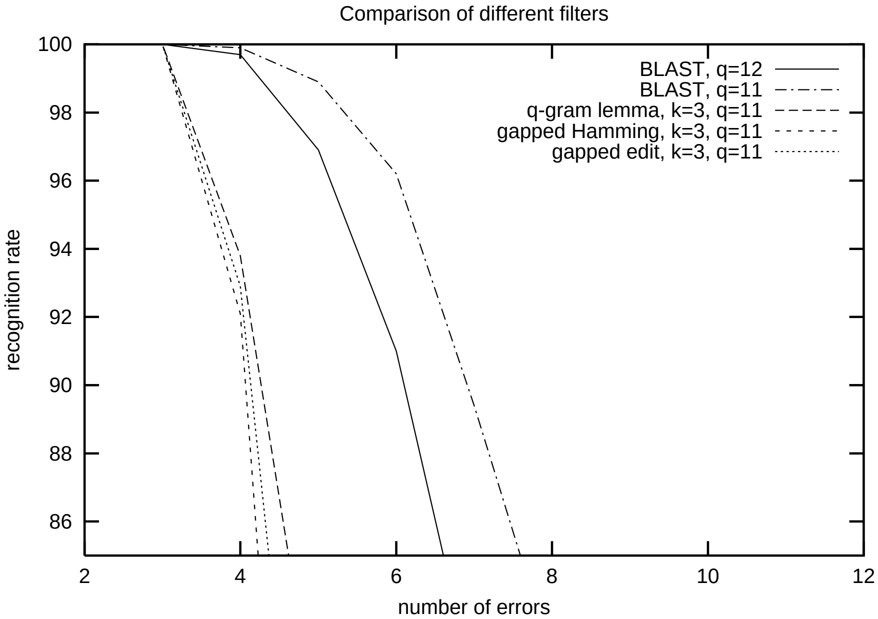


Figure 6.18: Recognition rates for several different filters with the lossless zone extending up to $k = 3$

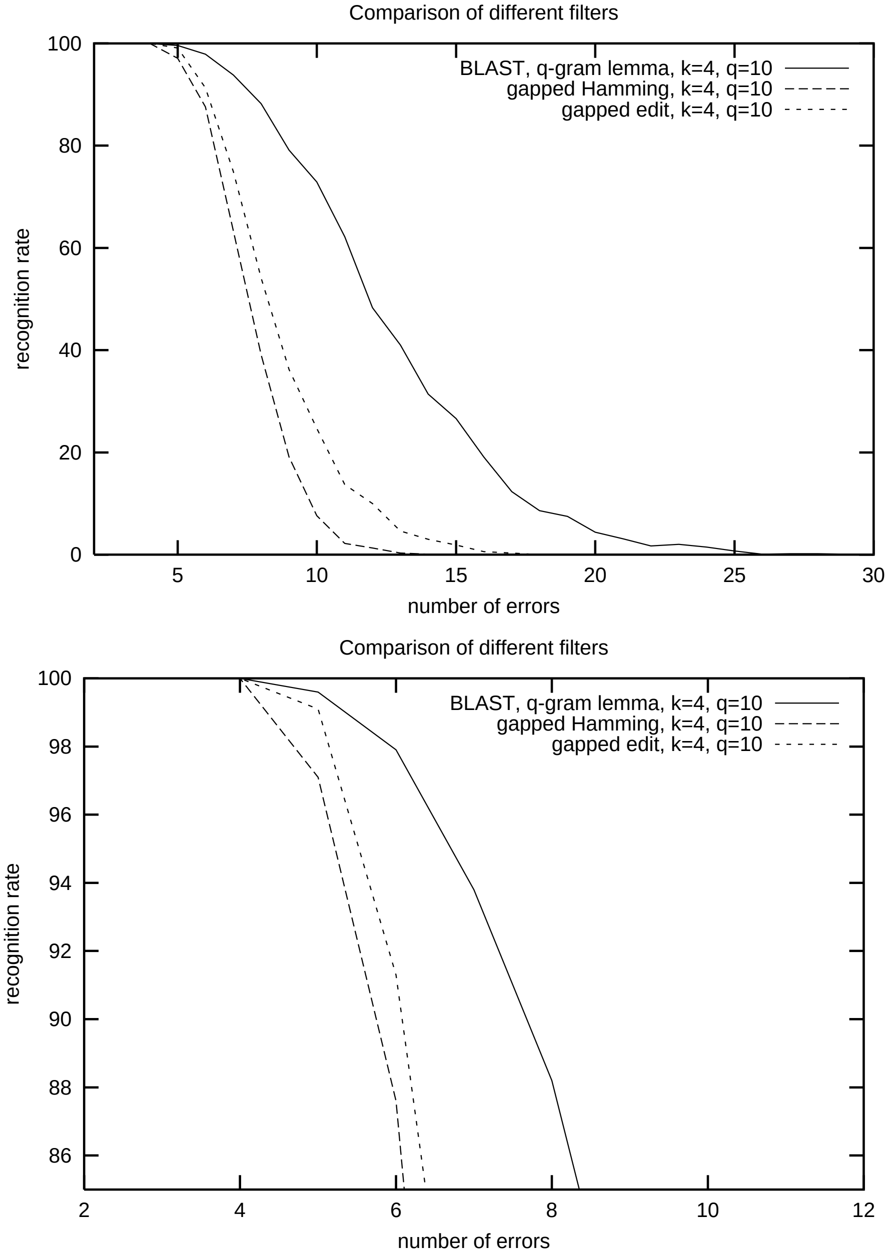


Figure 6.19: Recognition rates for several different filters with the lossless zone extending up to $k = 4$

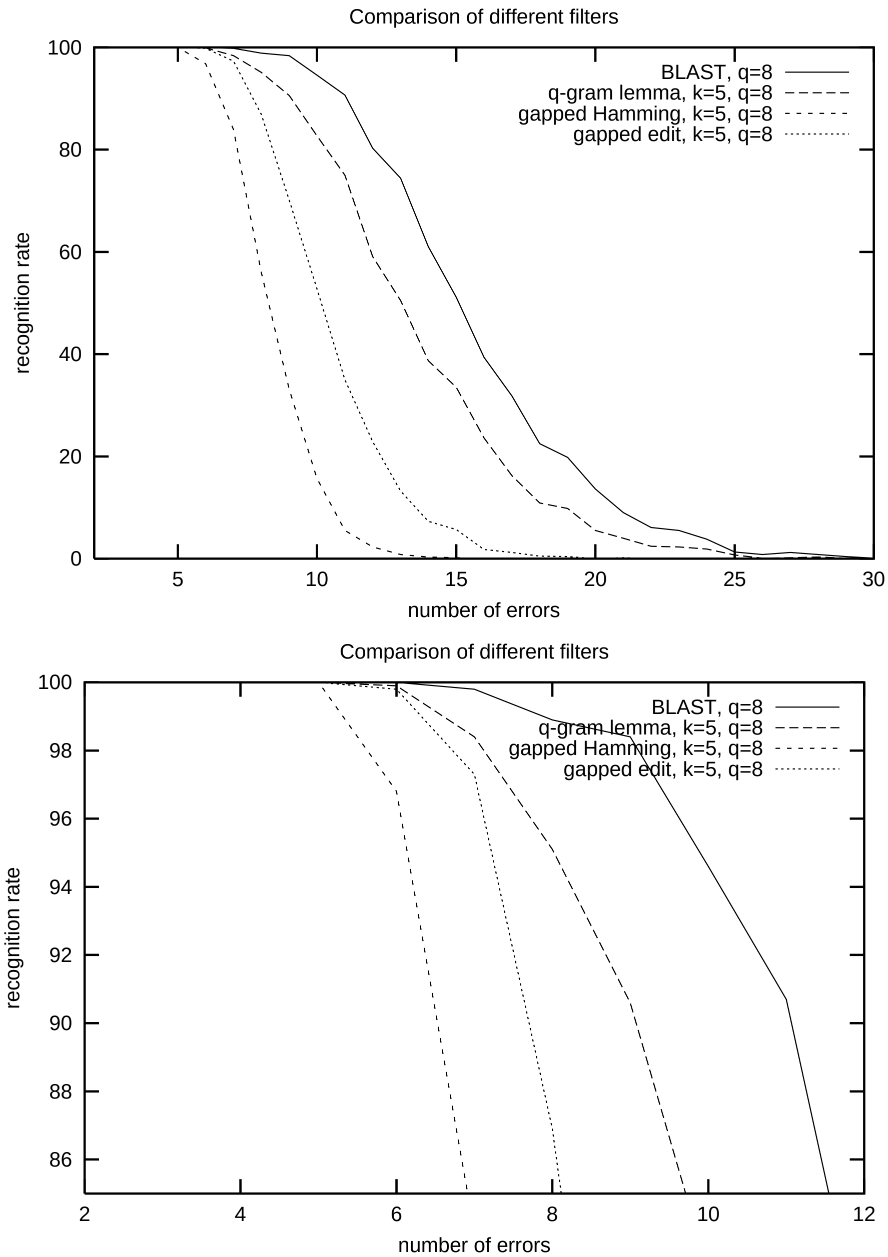


Figure 6.20: Recognition rates for several different filters with the lossless zone extending up to $k = 5$

Things are different if using these filters as lossy filters. It is difficult to define a good measure of quality for such filters since the goals can vary from application to application. The reason for this are the different motivations for using such a filter. Sometimes one is simply interested in a very fast filter that detects nearly every true match up to an error bound e_{max} and for which a small amount of false negatives is acceptable. Often one only considers matches with a similarity above a certain cutoff for further evaluation in a different algorithm. An example of this is again EST clustering where the selection of similar sequences is based on such a cutoff. The second scenario is a highly efficient search for relatively distant approximate matches in which even a large number of false negatives can be accepted. For the former case gapped q-grams are extremely useful. Their recognition rate shows a property that is quite desirable for solving this problem. Initially it drops slower but then much more rapidly than for example that of BLAST or of the classic q-gram lemma. This means that while the recognition rate is high for the region that is just above the error bound k but below or equal to e_{max}, the number of false positives reported is still low, i.e. there are fewer false negatives *and* false positives. If using the q-gram Lemma or BLAST to achieve the same recognition rate up to e_{max}, one has to allow for much more false positives, i.e. matches with more than e_{max} errors. Figure 6.21 illustrates this for an error bound $e_{max} = 7$ and a recognition rate bound of 98%. This is equivalent to a search in which one wants to find at least 98% of all true matches with up to 7 errors. For the second problem where approximate matches with very high error levels are desired even if the recognition rate for that error level is low, BLAST is the best solution since using BLAST with a low value of q results in a very low minimum coverage and therefore the most matches with very low similarity.

The above conclusions were only possible due to the additional information about filter behavior which is provided by the experiments with the constructed databases. Of course they have to be taken with a grain of salt since the placement of the errors and the sequences themselves are artificial. Nevertheless, since we are mainly interested in a *relative* comparison of different algorithms they still provide us with important insights.

6.5.6 Concluding Remarks about the Constructed Databases

Even though the experiments based on constructed databases have several limitations, they still provide a good general impression of the behavior of a given filter. They contain more information than the simple error limit k and can be quite useful for choosing the right filter for a given application. A large database of filters and their corresponding curves could therefore be a valuable tool for decisions about algorithm and parameter choice for approximate string matching problems. A second benefit of

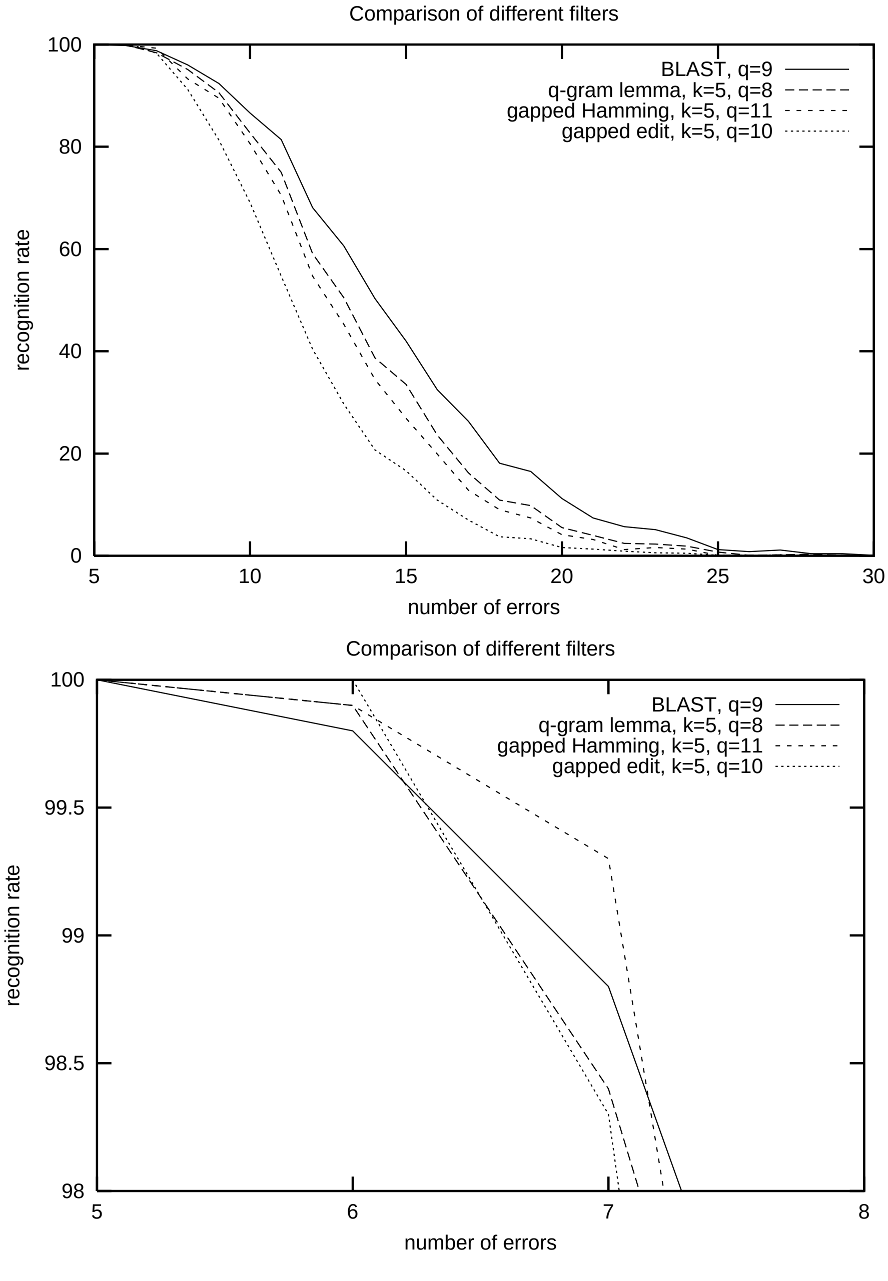


Figure 6.21: Recognition rates for several different lossy filters with an error bound $e_{max} = 7$ for which at least 98% of the actual matches are recognized

these experiments was the additional correctness check of our implementations. It was simple to create the databases S_e for which $e \leq k$ and use those databases to check whether all samples were recognized as potential matches by our filter implementations.

Chapter 7

Conclusion

In this work we discuss filter algorithms for approximate string matching. We review existing algorithms and describe an implementation of an algorithm based on existing work combined with new, own ideas. This algorithm, QUASAR, turns out to be significantly faster than current implementations for approximate string matching with low error levels. We also introduce two new filter algorithms for the k-differences and the k-mismatches problem that are based on using gapped substrings, so-called q-grams, of the query to locate potential matches in a target string or database.

QUASAR is a filter algorithm based on ideas by P. Jokinen and E. Ukkonen which we improved and adapted to our requirements. It uses their q-gram lemma in combination with a match region that is a modified version of one introduced in their initial q-gram paper. The index structure is a combination of two well known data structures, a word lookup table and a suffix array. Furthermore we implemented a new technique for adapting the global algorithm to local approximate string matching. QUASAR was designed for high similarity approximate string matching and was used in EST clustering by a group of researchers at the German Cancer Research Center in Heidelberg.

For the two new filters based on gapped q-grams we had to solve several problems. The first and most important was the computation of the optimal threshold for a given filter and shape. This is a prerequisite for applying such a filter. We developed two different algorithms for computing the threshold of the respective filters, implemented and successfully tested these algorithms for a range of parameters. Our algorithms are efficient and suitable for application on a large number of different gapped q-grams.

The next question that arises is that of q-gram selection. Since there is a large number of available shapes to choose from, we had to devise a method to select good shapes, i.e. those with low runtime and high filtration efficiency. We developed two measures which allow us to predict the approximate amount of hits a filter has to process as well as the expected number of potential matches it will produce. We also provide efficient methods to compute these measures. This allows a full exhaustive search of all potentially interesting shapes for a given set of parameters in a reasonable amount of time with the goal of selecting optimal or near-optimal shapes for these parameter values. We conducted an experimental analysis of the prediction quality of the two measures that yielded encouraging results.

Apart from the techniques required for choosing good gapped q-grams we also analyzed the variation in shape quality across the set of available shapes. In previous work on gapped shapes these shapes were selected randomly. We show that the variation in shape quality is very large and that a careful selection of the gapped q-grams results in significant performance gains. This underlines the importance of the tools we developed for measuring shape quality.

We also discuss the changes to existing filter algorithms necessary to adapt them to gapped q-grams. They are very straightforward and the overhead introduced by gapped shapes is very low for large databases.

Even though we provide two filter algorithms that are ready for practical applications, we would also like to point out that they are just a first step in the research of filter algorithms based on gapped q-grams. There are many interesting open questions that remain and further development in this area could lead to even better filters.

Examples for further directions could be the use of more than one gapped shape at once. Rigoutsos and Califano already provide a first use of sets of shapes in their work on FLASH. However, the careful selection of shapes that work well in combination could vastly improve their approach.

Another idea for approximate string matching is the use of approximate q-grams. This approach could be extended or combined with gapped q-grams to allow higher values of q or k.

Substrings or q-grams are used in many areas outside of approximate string matching. It could be interesting to analyze these problems and evaluate whether gapped q-grams offer potential for improving existing algorithms.

Returning to our work on approximate string matching we would like to point out that there are two interesting open problems left. While we managed to provide efficient tools for an exhaustive analysis of all gapped q-grams suitable for a given set of parameters, we were not able to devise an algorithm for designing or constructing a good gapped q-gram for a certain problem. This would be interesting for applications where the parameters change frequently, making an exhaustive analysis too expensive. Another interesting problem would be an improvement of the measure for predicting filtration efficiency. While our formula based on the minimum coverage works quite well in practice, it would still be interesting to increase its accuracy even more, for example by taking covers slightly larger than the minimum cover into account.

Another goal could be the development of a filter database containing a collection of good filters including the parameter ranges for which these filters make sense. One of the problems of approximate string matching remains the strong dependency on the problem parameters. There exist many different algorithms and filter algorithms for approximate string matching and which of them is actually the best depends heavily on the required value of k and the length of query and database. So far no satisfying database or algorithm for selecting the best or a near-optimal implementation for a given problem exists.

Recently results on compressed index structures have been reported [FM01, GV00]. For large scale application it may be interesting to replace the simple, uncompressed indexes used in many filter algorithms by these space-saving data structures.

Bibliography

[AG86] A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM Journal of Computing*, 15:98–105, 1986.

[AG97] A. Apostolico and Z. Galil, editors. *Pattern matching algorithms.* Oxford University Press, 1997.

[AGM+90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[AMS+97] S.F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[BCF+99] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based database searching using a suffix array. In S. Istrail, P. Pevzner, and M. Waterman, editors, *Proceedings of the 3rd Annual International Conference on Computational Molecular Biology*, pages 77–83, Lyon, France, 1999. ACM Press.

[BGW00] A. I. Buchsbaum, R. Giancarlo, and J. R. Westbrook. On the determinization of weighted finite automata. *SIAM Journal of Computing*, 30(5):1502–1531, 2000.

[BH00] V. Bafna and D. H. Huson. The conserved exon method for gene finding. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology*, pages 3–12, 2000.

[BK01] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 73–85. Springer, 2001.

[BK02] S. Burkhardt and J. Kärkkäinen. One-gapped q-gram filters for Levenshtein distance. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, volume 2373 of *Lecture Notes in Computer Science*, pages 225–234. Springer, 2002.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[CF02] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* The MIT Press, 1990.

[CM94] W. I. Chang and T. G. Marr. Approximate string matching and local similarity. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, volume 807 of *LNCS*, pages 259–273. Springer, 1994.

[CR93] A. Califano and I. Rigoutsos. FLASH: A fast look-up algorithm for string homology. In L. Hunter, D. Searls, and J. Shavlik, editors, *Proceedings of the First International Conference on Intelligent Systems for Molecular Biology*, pages 56–64, Bethesda, MD, 1993. AAAI Press.

[Far97] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 137–143, 1997.

[FG99] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the Association for Computing Machinery*, 46:236–280, 1999.

[FM01] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 269–278, 2001.

[GBYS92] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. *New indices for text: PAT trees and PAT arrays.* Prentice-Hall, 1992.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[GV00] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd ACM Symposium on the Theory of Computing*, pages 397–406, 2000.

[Ham50] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29:147–160, 1950.

[HS94] N. Holsti and E. Sutinen. Approximate string matching using q-gram places. In *Proceedings of the 7th Finnish Symposium on Computer Science*, pages 23–32, 1994.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[JU91] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proceedings of the 16th Symposium on Mathematical Foundations of Computer Science*, volume 520 of *Lecture Notes in Computer Science*, pages 240–248. Springer-Verlag, 1991.

[Kär02] J. Kärkkäinen. Cpmputing the threshold for q-gram filters. In *Algorithm Theory – SWAT 2002*, volume 2368, pages 348–357, 2002.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:322–350, 1977.

[KR87] R. Karp and M. Rabin. Efficient randomized pattern matching algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.

[Kur99] S. Kurtz. Reducing the space requirements of suffix trees. *Software-Practice and Experience*, 29:1149–1171, 1999.

[KV98] A. Krause and M. Vingron. A set-theoretic approach to database searching and clustering. *Bioinformatics*, 14:430–438, 1998.

[Lev66] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 6:707–710, 1966.

[LST96] O. Lehtinen, E. Sutinen, and J. Tarhio. Experiments on block indexing. In R. Baeza-Yates N. Ziviani and K. Guimarães, editors, *Proceedings of* *the 3rd South American Workshop on String Processing*, pages 183–193, Recife, Brazil, 1996. Carleton University Press.

[MAM+02] R. J. Mural, M. D. Adams, E. W. Myers, H. O. Smith, G. L. Miklos, R. Wides, A. Halpern, P. W. Li, G. G. Sutton, J. Nadeau, S. L. Salzberg, R. A. Holt, C. D. Kodira, F. Lu, L. Chen, Z. Deng, C. C. Evangelista, W. Gan, T. J. Heiman, J. Li, Z. Li, G. V. Merkulov, N. V. Milshina, A. K. Naik, R. Qi, B. C. Shue, A. Wang, J. Wang, X. Wang, X. Yan, J. Ye, S. Yooseph, Q. Zhao, L. Zheng, S. C. Zhu, K. Biddick, R. Bolanos, A. L. Delcher, I. M. Dew, D. Fasulo, M. J. Flanigan, D. H. Huson, S. A. Kravitz, J. R. Miller, C. M. Mobarry, K. Reinert, K. A. Remington, Q. Zhang, X. H. Zheng, D. R. Nusskern, Z. Lai, Y. Lei, W. Zhong, A. Yao, P. Guan, R. R. Ji, Z. Gu, Z. Y. Wang, F. Zhong, C. Xiao, C. C. Chiang, M. Yandell, J. R. Wortman, P. G. Amanatides, S. L. Hladun, E. C. Pratts, J. E. Johnson, K. L. Dodson, K. J. Woodford, C. A. Evans, B. Gropman, D. B. Rusch, E. Venter, M. Wang, T. J. Smith, J. T. Houck, D. E. Tompkins, C. Haynes, D. Jacob, S. H. Chin, D. R. Allen, C. E. Dahlke, R. Sanders, K. Li, X. Liu, A. A. Levitsky, W. H. Majoros, Q. Chen, A. C. Xia, J. R. Lopez, M. T. Donnelly, M. H. Newman, A. Glodek, C. L. Kraft, M. Nodell, F. Ali, H. J. An, D. Pitts, K. Y. Beeson, S. Cai, M. Carnes, A. Carver, P. M. Caulk, A. Center, Y. H. Chen, M. L. Cheng, M. D. Coyne, M. Crowder, S. Danaher, L. B. Davenport, R. Desilets, S. M. Dietz, L. Doup, P. Dullaghan, S. Ferriera, C. R. Fosler, H. C. Gire, A. Gluecksmann, J. D. Gocayne, J. Gray, B. Hart, J. Haynes, J. Hoover, T. Howland, C. Ibegwam, M. Jalali, D. Johns, L. Kline, D. S. Ma, S. MacCawley, A. Magoon, F. Mann, D. May, T. C. McIntosh, S. Mehta, L. Moy, M. C. Moy, B. J. Murphy, S. D. Murphy, K. A. Nelson, Z. Nuri, K. A. Parker, A. C. Prudhomme, V. N. Puri, H. Qureshi, J. C. Raley, M. S. Reardon, M. A. Regier, Y. H. Rogers, D. L. Romblad, J. Schutz, J. L. Scott, R. Scott, C. D. Sitter, M. Smallwood, A. C. Sprague, E. Stewart, R. V. Strong, E. Suh, K. Sylvester, R. Thomas, N. N. Tint, C. Tsonis, G. Wang, G. Wang, M. S. Williams, S. M. Williams, S. M. Windsor, K. Wolfe, M. M. Wu, J. Zaveri, K. Chaturvedi, A. E. Gabrielian, Z. Ke, J. Sun, G. Subramanian, and J. C. Venter. A comparison of whole-genome shotgun-derived mouse chromosome 16 and the human genome. *Science*, 296(5573):1661–1671, 2002.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23:262–272, 1976.

[Mea55] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MM93] U. Manber and E. W. Myers. Suffix arrays: a new method for on-line search. *SIAM Journal of Computing*, 22:935–948, 1993.

[MN95] K. Mehlhorn and S. Näher. Leda, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, 1995.

[Moh97] M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23:269–311, 1997.

[Mor68] D. R. Morrison. Patricia: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the Association for Computing Machinery*, 15:514–534, 1968.

[MSD$^+$00] E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, M. J. Flanigan, S. A. Kravitz, C. M. Mobarry, K. H. Reinert, K. A. Remington, E. L. Anson, R. A. Bolanos, H. H. Chou, C. M. Jordan, A. L. Halpern, S. Lonardi, E. M. Beasley, R. C. Brandon, L. Chen, P. J. Dunn, Z. Lai, Y. Liang, D. R. Nusskern, M. Zhan, Q. Zhang, X. Zheng, G. M. Rubin, M. D. Adams, and J. C. Venter. A whole-genome assemble of drosophila. *Science*, 287:2196–2204, 2000.

[MTL02] B. Ma, J. Tromp, and M. Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18:440–445, 2002.

[Mye94] E. W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, 1994.

[Mye99] E. W. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the Association for Computing Machinery*, 46(3):395–415, 1999.

[Nav01] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33:31–88, 2001.

[NBY98] 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`.

[NBY00] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000.

[NBYST01] 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. Special issue on Managing Text Natively and in DBMSs.

[NSTT00] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, volume 1848 of *LNCS*, pages 350–363. Springer-Verlag, 2000.

[NW70] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[Pea90] W. R. Pearson. Rapid and sensitive sequence comparison with fastp and fasta. *Methods in Enzymology*, 183:63–98, 1990.

[PL88] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence analysis. *Proceedings of the National Academy of Science, USA*, 85:2444–2448, 1988.

[PW95] P. A. Pevzner and M. S. Waterman. Multiple filtration and approximate pattern matching. *Algorithmica*, 13(1/2):135–154, 1995.

[Sel74] P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal of Applied Mathematics*, 26:787–793, 1974.

[Shi96] F. Shi. Fast approximate string matching with q-blocks sequences. In *Proceedings of the 3rd South American Workshop on String Processing*, pages 257–271. Carleton University Press, 1996.

[ST95] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In P. G. Spirakis, editor, *Proceedings of the 3rd Annual European Symposium on Algorithms*, volume 979 of *Lecture Notes in Computer Science*, pages 327–340. Springer-Verlag, 1995.

[ST96] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, volume 1075 of *Lecture Notes in Computer Science*, pages 50–63. Springer, 1996.

[SW81] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[Tak94] T. Takaoka. Approximate pattern matching with samples. In Ding-Zhu Du and Xiang sun Zhang, editors, *Proceedings of the 5th International*

Symposium on Algorithms and Computation, volume 834 of *Lecture Notes in Computer Science*, pages 236–242. Springer-Verlag, 1994.

[Ukk92] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science*, 92(1):191–212, 1992.

[Ukk95] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249–260, 1995.

[VAM$^+$01] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, J. D. Gocayne, P. Amanatides, R. M. Ballew, D. H. Huson, J. R. Wortman, Q. Zhang, C. D. Kodira, X. H. Zheng, L. Chen, M. Skupski, G. Subramanian, P. D. Thomas, J. Zhang, G. L. Gabor Miklos, C. Nelson, S. Broder, A. G. Clark, J. Nadeau, V. A. McKusick, N. Zinder, A. J. Levine, R. J. Roberts, M. Simon, C. Slayman, M. Hunkapiller, R. Bolanos, A. Delcher, I. Dew, D. Fasulo, M. Flanigan, L. Florea, A. Halpern, S. Hannenhalli, S. Kravitz, S. Levy, C. Mobarry, K. Reinert, K. Remington, J. Threideh, E. Beasley, K. Biddick, V. Bonazzi, R. Brandon, M. Cargill, I. Chandramouliswaran, R. Charlab, K. Chaturvedi, Z. Deng, V. Di Francesco, P. Dunn, K. Eilbeck, C. Evangelista, A. E. Gabrielian, W. Gan, W. Ge, F. Gong, Z. Gu, P. Guan, T. J. Heiman, M. E. Higgins, R. R. Ji, Z. Ke, K. A. Ketchum, Z. Lai, Y. Lei, Z. Li, J. Li, Y. Liang, X. Lin, F. Lu, G. V. Merkulov, N. Milshina, H. M. Moore, A. K. Naik, V. A. Narayan, B. Neelam, D. Nusskern, D. B. Rusch, S. Salzberg, W. Shao, B. Shue, J. Sun, Z. Wang, A. Wang, X. Wang, J. Wang, M. Wei, R. Wides, C. Xiao, C. Yan, A. Yao, J. Ye, M. Zhan, W. Zhang, H. Zhang, Q. Zhao, L. Zheng, F. Zhong, W. Zhong, S. Zhu, S. Zhao, D. Gilbert, S. Baumhueter, G. Spier, C. Carter, A. Cravchik, T. Woodage, F. Ali, H. An, A. Awe, D. Baldwin, H. Baden, M. Barnstead, I. Barrow, K. Beeson, D. Busam, A. Carver, A. Center, M. L. Cheng, L. Curry, S. Danaher, L. Davenport, R. Desilets, S. Dietz, K. Dodson, L. Doup, S. Ferriera, N. Garg, A. Gluecksmann, B. Hart, J. Haynes, C. Haynes, C. Heiner, S. Hladun, D. Hostin, J. Houck, T. Howland, C. Ibegwam, J. Johnson, F. Kalush, L. Kline, S. Koduru, A. Love, F. Mann, D. May, S. McCawley, T. McIntosh, I. McMullen, M. Moy, L. Moy, B. Murphy, K. Nelson, C. Pfannkoch, E. Pratts, V. Puri, H. Qureshi, M. Reardon, R. Rodriguez, Y. H. Rogers, D. Romblad, B. Ruhfel, R. Scott, C. Sitter, M. Smallwood, E. Stewart, R. Strong, E. Suh, R. Thomas, N. N. Tint, S. Tse, C. Vech, G. Wang, J. Wetter, S. Williams, M. Williams, S. Windsor, E. Deen, K. Wolfe, J. Zaveri, K. Zaveri, J. F. Abril, R. Guigo, M. J. Campbell, K. V. Sjolander, B. Karlak, A. Kejariwal, H. Mi, B. Lazareva, T. Hatton, A. Narechania, K. Diemer, A. Muruganujan, N. Guo, S. Sato, V. Bafna, S. Istrail, R. Lippert, R. Schwartz, B. Walenz, S. Yooseph, D. Allen, A. Basu, J. Baxendale, L. Blick, M. Caminha, J. Stine, P. Caulk, Y. H. Chiang, M. Coyne, C. Dahlke, A. Mays, M. Dombroski, M. Donnelly, D. Ely, S. Esparham, C. Fosler, H. Gire, S. Glanowski, K. Glasser, A. Glodek, M. Gorokhov, K. Graham, B. Gropman, M. Harris, J. Heil, S. Henderson, J. Hoover, D. Jennings, C. Jordan, J. Jordan, J. Kasha, L. Kagan, C. Kraft, A. Levitsky, M. Lewis, X. Liu, J. Lopez, D. Ma, W. Majoros, J. McDaniel, S. Murphy, M. Newman, T. Nguyen, N. Nguyen, M. Nodell, S. Pan, J. Peck, M. Peterson, W. Rowe, R. Sanders, J. Scott, M. Simpson, T. Smith, A. Sprague, T. Stockwell, R. Turner, E. Venter, M. Wang, M. Wen, D. Wu, M. Wu, A. Xia, A. Zandieh, and X. Zhu. The sequence of the human genome. *Science*, 291:1304–1351, 2001.

[VAS+98] J. C. Venter, M. D. Adams, G. G. Sutton, A. R. Kerlavage, H. O. Smith, and M. Hunkapillar. Shotgun sequencing of the human genome. *Science*, 280:1540–1542, 1998.

[Wal95] M. M. Waldrop. On-line archives let biologists interrogate the genome. *Science*, 269:1356–1358, 1995.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.

[WM97] J. Weber and E. W. Myers. Human whole-genome shotgun sequencing. *Genome Research*, 7:401–409, 1997.

[Zie01] M. Ziegelmann. *Constrained Shortest Paths and Related Problems.* PhD thesis, Universität des Saarlandes, Germany, 2001.

Summary

Approximate string matching is one of the classic problems in computer science. In contrast to the simpler exact string matching problem, which consists of locating all exact matches between a query or pattern string and a target string or database, it includes recognizing all approximate matches with respect to a certain measure of similarity or distance.

Early methods for locating approximate matches of a pattern P in a string S were based on dynamic programming and had a runtime of $O(|P||S|)$. While these algorithms can detect even very distant approximate matches, they suffer from their comparatively slow runtime, especially for applications with very large databases or many queries.

This problem led to the development of various faster methods for approximate string matching in general and for computational biology in particular. A common principle shared by many of these algorithms is the idea of filtering. Filter algorithms typically consist of two phases. In the first, the filtration phase, a highly efficient filter criterion is used to detect potential matches between query and subject. In the second phase the potential matches have to be analyzed to detect the true matches with a conventional algorithm. The second phase is called verification phase.

There are two subclasses of approximate string matching. One instance – called the *on-line* approximate string matching problem – does not allow preprocessing of the database S before the actual search is conducted. The time spent for any examination of S during the search is considered part of the search time. In the other – the *off-line* version – one can examine the database S, gathering information about it and saving this information in index data structures in a preprocessing stage. The time required for preprocessing is not considered part of the runtime of a search in this model. The construction time of an index and the amount of space required for storing it are nevertheless quite important to the practical usefulness of such algorithms.

Off-line string matching is useful for databases that change slowly and where a large number of searches on the same database are necessary. Frequent changes of the database can only be handled if the preprocessing step is sufficiently fast. On-line string matching is more useful for rapidly changing databases and for cases where only a small number of searches in a given database have to be performed. In these cases the overhead for preprocessing the database becomes significant and reduces the overall efficiency of index-based approaches.

The first part of our work resulted in the development of QUASAR. We extended

an approach described by Jokinen and Ukkonen and implemented it. They introduced an algorithm called q-gram filtration and a simple index structure resulting in a very efficient filter algorithm. q-grams are short strings of length q, usually substrings of the query and/or the target. We implemented their proposal and improved several important components. The implementation which we called QUASAR resulted in an executable that was able to solve EST-clustering, a problem from computational biology, about 20 times faster than BLAST.

The nature of the q-gram lemma which is the foundation of the q-gram filtration limits the classic q-gram lemma to approximate string matching with relatively high degrees of similarity.

In the second part of our work we therefore introduce two new filters that can benefit from efficient indices but also allow for higher sensitivity. They are based on substrings with gaps, one for the Hamming and one for the edit or Levenshtein distance. Instead of looking for exact matches of a string of length q one can decide to only match certain letters of the string. The other characters – those in the gaps – are basically 'don't-care' characters or wildcards, i.e. one can match them with any possible character.

Developing filters that use gapped q-grams turned out to be fairly complicated. When using the Hamming distance, most problems could be overcome with reasonable effort. However, the case of the Levenshtein distance is much more complex. The two main problems were the selection of good gapped q-grams, i.e. arrangements of gaps, and the computation of the required number of matching q-grams for a region of S to be considered a potential match. In the case of the classic q-gram lemma this is simple but for gapped q-grams it requires several non-trivial algorithms. Another problem which is specific to the case of the Levenshtein distance is the question of handling insertions and deletions in potential approximate matches. We came up with a relatively simple method that does not have a very big impact on the runtime of our solution.

We implemented the algorithms based on gapped q-grams and analyzed their performance in a number of experiments. We use two well defined measures, the number of items that have to be retrieved from the index data structure and the number of potential matches returned by the filter to allow an unbiased comparison of different filter algorithms. These values allow characterizing the runtime of the filtration and verification phase. We then compared the filter approach of BLAST, the classic q-gram lemma and our two algorithms using gapped q-grams for the Hamming and the edit distance with each other. Since filter algorithms in general are only suitable for moderate to high similarity searches we only looked at these cases.

For the Hamming distance we were able to provide a wide range of improved filters. They reduced the amount of data to be processed in the filtration phase by up to three orders of magnitude and the amount of potential matches by up to six orders of magnitude when compared to the classic q-gram lemma. The differences to the BLAST filter criterion were even larger. Filters that offer a tradeoff between speeding up the filtration and the verification phase also exist. If using the edit distance, our filters based on gapped q-grams reduced the amount of data to be processed in the filtration phase and the amount of potential matches by up two oders of magnitude when compared to the classic q-gram lemma. Again we also provided filters that offer a tradeoff between speeding up the filtration and the verification phase.

In a second set of experiments we analysed the potential of filters based on gapped q-grams for use as *lossy* filters, i.e. algorithms that are faster but may miss a small number of actual matches in the filtration phase. The results imply that gapped q-grams are far better suited than conventional ungapped q-grams for use in lossy filters. A comparison with the filter criterion implemented in BLAST also yielded very encouraging results not only for high levels of string similarity but also for moderate levels.

Index