A Guided Tour to Approximate String Matching

Gonzalo Navarro *

Dept. of Computer Science, University of Chile
Blanco Encalada 2120 - Santiago - Chile
gnavarro@dcc.uchile.cl, http://www.dcc.uchile.cl/~gnavarro

Abstract

We survey the current techniques to cope with the problem of string matching allowing errors. This is becoming a more and more relevant issue for many fast growing areas such as information retrieval and computational biology. We focus on online searching, explaining the problem and its relevance, its statistical behavior, its history and current developments, and the central ideas of the algorithms and their complexities. We present a number of experiments to compare the performance of the different algorithms and show which are the best choices according to each case. We conclude with some future work directions and open problems.

1 Introduction

This work focuses on the problem of *string matching allowing errors*, also called *approximate string matching*. The general goal is to perform string matching of a pattern in a text where one or both of them have suffered some kind of (undesirable) corruption. Some examples are recovering the original signals after their transmission over noisy channels, finding DNA subsequences after possible mutations, and text searching under the presence of typing or spelling errors.

The problem, in its most general form, is to find the positions of a text where a given pattern occurs, allowing a limited number of "errors" in the matches. Each application uses a different error model, which defines how different two strings are. The idea for this "distance" between strings is to make it small when one of the strings is likely to be an erroneous variant of the other under the error model in use.

The goal of this survey is to present an overview of the state of the art in approximate string matching. We focus on online searching, explaining the problem and its relevance, its statistical behavior, its history and current developments, and the central ideas of the algorithms and their complexities. We also consider some variants of the problem which are of interest. We present a number of experiments to compare the performance of the different algorithms and show which are the best choices according to each case. We conclude with some future work directions and open problems.

Unfortunately, the algorithmic nature of the problem strongly depends on the type of "errors" considered, and the solutions range from linear time to NP-complete. The scope of our subject is so broad that we are forced to specialize our focus on a subset of the possible error models. We consider only those defined in terms of replacing some substrings by others at varying costs. Under

*Partially supported by Fondecyt grant 1-990627.

this light, the problem becomes minimizing the total cost to transform the pattern and its text occurrence to make them equal, and reporting the text positions where this cost is low enough.

One of the best studied particular cases of this error model is the so-called *edit distance*, which allows to delete, insert and replace simple characters (by a different one) in both strings. If the different operations have different costs or the costs depend on the characters involved, we speak of *general edit distance*. Otherwise, if all the operations cost 1, we speak of *simple edit distance* or just *edit distance* (*ed*). In this last case we simply seek for the minimum number of insertions, deletions and replacement to make both strings equal. For instance $ed(\texttt{"survey"},\texttt{"surgery"}) = 2$. The edit distance has received a lot of attention because its generalized version is powerful enough for a wide range of applications. Despite that most existing algorithms concentrate on the simple edit distance, many of them can be easily adapted to the generalized edit distance, and we pay attention to this issue throughout this work. Moreover, the few algorithms that exist for the general error model we consider are generalizations of edit distance algorithms.

On the other hand, most of the algorithms designed for the edit distance are easily specialized to other cases of interest. For instance, by allowing only insertions and deletions at cost 1 we can compute the longest common subsequence (LCS) between two strings. Another simplification that has received a lot of attention is the variant that allows only replacements (Hamming distance).

An extension of the edit distance enrichs it with transpositions (i.e. a replacement of the form $ab \to ba$ at cost 1). Transpositions are very important in text searching applications because they are typical typing errors, but few algorithms exist to handle them. However, many algorithms for edit distance can be easily extended to include transpositions, and we keep track of this fact in this work.

Since the edit distance is by far the best studied case, this survey focuses basically on the simple edit distance. However, we also pay attention to extensions such as generalized edit distance, transpositions and general substring replacement, as well as to simplifications such as LCS and Hamming distance. In addition, we also pay attention to some extensions of the type of pattern to search: when the algorithms allow it, we mention the possibility to search some extended patterns and regular expression allowing errors. We point out now what are we *not* covering in this work.

- First, we do not cover other distance functions that do not fit in the model of substring replacement. This is because they are too different from our focus and the paper would loose cohesion. Some of these are: Hamming distance (short survey in [Nav98a]), reversals [KS95] (which allows reversing substrings), block distance [Tic84, EH88, Ukk92, LT97] (which allows rearranging and permuting the substrings), q-gram distance [Ukk92] (based in finding common substrings of fixed length q), allowing swaps [AAL$^+$97, LKPC97], etc. Hamming distance, despite being a simplification of the edit distance, is not covered because specialized algorithms exist for it that go beyond the simplification of an existing algorithm for edit distance.

- Second, we consider pattern matching over sequences of symbols, and at most generalize the pattern to a regular expression. Extensions such as approximate searching in multi-dimensional texts (short survey in [NBY99a]), in graphs [ALL97, Nav98b] or multipattern approximate searching [MM96, BYN97b, Nav97a, BYN98] are not considered. None of these areas are very developed and the algorithms should be easy to grasp once approximate pattern

matching under the simple model is well understood. Many existing algorithms for these problems borrow from those we present here.

- Third, we leave aside non-standard algorithms, such as approximate or parallel algorithms [TU88, LL85, LV89].

- Finally, an important area that we leave aside in this survey is indexed searching, i.e. the process of building a persistent data structure (an index) on the text to speed up the search later. Typical reasons that prevent keeping indices on the text are: extra space requirements (as the indices for approximate searching tend to take many times the text size), volatility of the text (as building the indices is quite costly and needs be amortized over many searches) and simply inadequacy (as the field of indexed approximate string matching is quite immature and the speedup that the indices provide is not always satisfactory).

  Indexed approximate searching is a difficult problem, and the area is quite new and active [JU91, Gon92, Ukk93, Mye94a, HS94, MW94, Cob95, ST96, BYN97a, ANZ97, NBY98b, NBY99b, MNZBY99]. The problem is very important because the texts to handle are so large in some applications that no online algorithm can provide adequate performance. However, virtually all the indexed algorithms are strongly based on online algorithms, and therefore understanding and improving the current online solutions is of interest for indexed approximate searching as well.

These issues have been left aside to keep a reasonable scope in the present work. They certainly deserve separate surveys. Our goal in this survey is to explain the basic tools of approximate string matching, as many of the extensions we are leaving aside are built on the basic algorithms designed for online approximate string matching.

This work is organized as follows. In Section 2 we present in detail some of the most important application areas for approximate string matching. In Section 3 we formally introduce the problem and the basic concepts necessary to follow the rest of the paper. In Section 4 we show some analytical and empirical results about the statistical behavior of the problem.

Sections 5 to 8 cover all the work of interest we could trace on approximate string matching under the edit distance. We divided it in four sections that correspond to different approaches to the problem: dynamic programming, automata, bit-parallelism and filtering algorithms. Each section is presented as a historical tour, so that we do not only explain the work done but also show how it was developed.

Section 9 presents experimental results comparing the most efficient algorithms presented. Finally, we give our conclusions and discuss open questions and future work directions in Section 10.

There exist other surveys on approximate string matching, which are however too old for this fast moving area [HD80, SK83, AG85, GG88, JTU96] (the last one was in its definitive form in 1991). So all previous surveys lack coverage of the latest developments. Our aim is to provide a long awaited update. This work is partially based in [Nav98a], but the coverage of previous work is much more detailed here. The subject is also covered, albeit with less depth, in some textbooks on algorithms [CR94, BYR99].

2 Main Application Areas

The first references to this problem we could trace are from the sixties and seventies, where the problem appeared in a number of different fields. In those times, the main motivation for this kind of search came from computational biology, signal processing, and text retrieval. These are still the largest application areas, and we cover each one here. See also [SK83], which has a lot of information on the birth of this subject.

2.1 Computational Biology

DNA and protein sequences can be seen as long texts over specific alphabets (e.g. {A,C,G,T} in DNA). Those sequences represent the genetic code of living beings. Searching specific sequences over those texts appeared as a fundamental operation for problems such as assembling the DNA chain from the pieces obtained by the experiments, looking for given features in DNA chains, or determining how different two genetic sequences were. This was modeled as searching for given "patterns" in a "text". However, exact searching was of little use for this application, since the patterns rarely matched the text exactly: the experimental measures have errors of different kinds and even the correct chains may have small differences, some of them significant due to mutations and evolutionary alterations and others unimportant. Finding DNA chains very similar to those sought represent significant results as well. Moreover, establishing how different two sequences are is important to reconstruct the tree of the evolution (phylogenetic trees). All these problems required a concept of "similarity", as well as an algorithm to compute it.

This gave a motivation to "search allowing errors". The errors were those operations that biologists knew were common to occur in genetic sequences. The "distance" between two sequences was defined as the minimum (i.e. more likely) sequence of operations to transform one into the other. With regard to likelihood, the operations were assigned a "cost", such that the more likely operations were cheaper. The goal was then to minimize the total cost.

Computational biology has since then evolved and developed a lot, with a special push in recent years due to the "genome" projects that aim at the complete decoding of the DNA and its potential applications. There are other, more exotic problems, such as structure matching or searching for unknown patterns. Even the simple problem where the pattern is known is believed to be NP-complete under some distance functions (e.g. reversals).

Some good references for the applications of approximate pattern matching to computational biology are [Sel74, NW70, SK83, Mye94b, Wat95, YFM96].

2.2 Signal Processing

Another early motivation came from signal processing. One of the largest areas deals with speech recognition, where the general problem is to determine, given an audio signal, a textual message which is being transmitted. Even simplified problems such as discerning a word from a small set of alternatives is complex, since parts of the the signal may be compressed in time, parts of the speech may not be pronounced, etc. A perfect match is practically impossible.

Another problem of this field is error correction. The physical transmission of signals is error-prone. To ensure correct transmission over a physical channel, it is necessary to be able to recover

the correct message after a possible modification (error) introduced during the transmission. The probability of such errors is obtained from the signal processing theory and used to assign a cost to them. In this case we may even not know what we are searching for, we just want a text which is correct (according to the error correcting code used) and closest to the received message. Although this area has not developed too much with respect to approximate searching, it has generated the most important measure of similarity, known as the *Levenshtein distance* [Lev65, Lev66] (also called "edit distance").

Signal processing is a very active area today. The rapidly evolving field of multimedia databases demands the ability to search by content in image, audio and video data, which are potential applications for approximate string matching. We expect in the next years a lot of pressure on non-written human-machine communication, which involves speech recognition. Strong error correcting codes are also sought given the current interest in wireless networks, as the air is a low quality transmission medium.

Good references for the relations of approximate pattern matching with signal processing are [Lev65, Vin68, DM79].

2.3 Text Retrieval

The problem of correcting misspelled words in written text is rather old, perhaps the oldest potential application for approximate string matching. We could find references from the twenties [Mas27], and perhaps there are older ones. Since the sixties, approximate string matching is one of the most popular tools to deal with this problem. For instance, 80% of these errors are corrected allowing just one insertion, deletion, replacement or transposition [Dam64].

There are many areas where this problem appears, and Information Retrieval (IR) is one of the most demanding. IR is about finding the relevant information in a large text collection, and string matching is one of its basic tools.

However, classical string matching is normally not enough, because the text collections are becoming larger (e.g. the Web has largely surpassed the terabyte), more heterogeneous (different languages, for instance) and more error prone. Many are so large and grow so fast that it is impossible to control their quality (e.g. in the Web). A word which is entered incorrectly in the database cannot be retrieved anymore. Moreover, the pattern itself may have errors, for instance in cross-lingual scenarios where a foreign name sought is incorrectly spelled, or in ancient texts that use outdated versions of the language.

For instance, text collections digitalized via optical character recognition (OCR) contain a non-negligible percentage of errors (7% to 16%). The same happens with typing (1% to 3.2%) and spelling (1.5% to 2.5%) errors. Experiments for typing Dutch surname (by Dutchs) reached 38% of spelling errors. All these percentages were obtained from [Kuk92]. Our own experiments with the name "Levenshtein" in Altavista gave more than 30% of errors allowing just one deletion or transposition.

Nowadays, there is virtually no text retrieval product that does not allow some extended search facility to recover from errors in the text of pattern. Other text processing applications are spelling checkers, natural language interfaces, command language interfaces, computer aided tutoring and language learning, to name a few.

A very recent extension which became possible thanks to word-oriented text compression methods is the possibility to perform approximate string matching at the word level [MNZBY99]. That is, the user supplies a phrase to search and the system searches the text positions where the phrase appears with a limited number of *word* insertions, deletions and replacements. It is also possible to disregard the order of the words in the phrases. This allows the query to survive from different wordings of the same idea, which extends the applications of approximate pattern matching well beyond the recovery of syntactic mistakes.

Good references about the relation of approximate string matching and information retrieval are [WF74, LW75, Nes86, OM88, Kuk92, ZD96, FPS97, BYR99].

2.4 Other Areas

The number of applications for approximate string matching grows every day. We have found solutions to the most diverse problems based on approximate string matching, for instance handwriting recognition [LT94], virus and intrusion detection [KS94], image compression [LS97], data mining [DFG+97], pattern recognition [GT78], optical character recognition [EL90], file comparison [Hec78] and screen updating [Gos91], to name a few. Many more applications are mentioned in [SK83, Kuk92].

3 Basic Concepts

We present in this section the important concepts needed to understand all the development that follows. Basic knowledge is assumed on design and analysis of algorithms and data structures, basic text algorithms, and formal languages. If this is not the case we refer the reader to good books in these subjects, such as [AHU74, CLR91, Knu73] (for algorithms), [GBY91, CR94] (for text algorithms) and [HU79] (for formal languages).

We start with some formal definitions related to the problem. Then we cover some data structures not widely known which are relevant for this survey (they are also explained in [GBY91, CR94]). Finally, we make some comments about the tour itself.

3.1 Approximate String Matching

In the discussion that follows, we use s, x, y, z, v, w to represent arbitrary strings, and $a, b, c...$ to represent letters. Writing a sequence of strings and/or letters represents their concatenation. We assume that concepts such as prefix, suffix and substring are known. For any string $s \in \Sigma^*$ we denote its length as $|s|$. We also denote s_i the i-th character of s, for an integer $i \in \{1..|s|\}$. We denote $s_{i..j} = s_i s_{i+1} ... s_j$ (which is the empty string if $i > j$). The empty string is denoted as ε.

In the Introduction we have defined the problem of approximate string matching as that of finding the text positions that match a pattern with up to k errors. We give now a more formal definition.

Let Σ be a finite[1] alphabet of size $|\Sigma| = \sigma$.

[1] However, many algorithms can be adapted to infinite alphabets with an extra $O(\log m)$ factor in their cost. This is because the pattern can have at most m different letters and all the rest can be considered equal for our purposes.

Let $T \in \Sigma^*$ be a *text* of length $n = |T|$.

Let $P \in \Sigma^*$ be a *pattern* of length $m = |P|$.

Let $k \in \mathbb{R}$ be the maximum error allowed.

Let $d : \Sigma^* \times \Sigma^* \rightarrow \mathbb{R}$ be a *distance function.*

The problem is: given T, P, k and $d()$, return the set of all the text positions j such that there exists i such that $d(P, T_{i..j}) \leq k$.

Note that endpoints of occurrences are reported to ensure that the output is of linear size. By reversing all strings we can obtain start points.

In this work we restrict our attention to a subset of the possible distance functions. We consider only those defined in the following form:

> The distance $d(x, y)$ between two strings x and y is the minimal cost of a sequence of *operations* that transform x into y (and ∞ if no such sequence exists). The cost of a sequence of operations is the sum of the costs of the individual operations. The operations are a finite set of rules of the form $\delta(z, w) = c$, where z and w are *different* strings and c is a nonnegative real number. Once the operation has converted a substring z into w, no further operations can be done on w.

Note especially the restriction that forbids acting many times over the same string. Freeing the definition from this condition would allow any rewriting system to be represented, and therefore determining the distance between two strings would not be computable in general.

If for each operation of the form $\delta(z, w)$ there exists the respective operation $\delta(w, z)$ at the same cost, then the distance is symmetric (i.e. $d(x, y) = d(y, x)$). Note also that $d(x, y) \geq 0$ for all strings x and y, that $d(x, x) = 0$, and that it always holds $d(x, z) \leq d(x, y) + d(y, z)$. Hence, if the distance is symmetric, the space of strings forms a *metric space.*

General substring replacement has been used to correct phonetic errors [ZD96]. In most applications, however, the set of possible operations is restricted to:

- *Insertion*: $\delta(\varepsilon, a)$, i.e. inserting the letter a.
- *Deletion*: $\delta(a, \varepsilon)$, i.e. deleting the letter a.
- *Replacement or Substitution*: $\delta(a, b)$ for $a \neq b$, i.e. replacing a by b.
- *Transposition*: $\delta(ab, ba)$ for $a \neq b$, i.e. swap the adjacent letters a and b.

We are now in position to define the most commonly used distance functions (although there are many others).

> – *Levenshtein or Edit distance* [Lev65]: allows insertions, deletions and replacements. In the simplified definition, all the operations cost 1. This can be rephrased as "the minimal number of insertions, deletions and replacements to make two strings equal". In the literature the search problem is in many cases called "string matching with k differences". The distance is symmetric, and it holds $0 \leq d(x, y) \leq \max(|x|, |y|)$.

A table of size σ could be replaced by a search structure over at most $m + 1$ different letters.

– *Hamming distance* [SK83]: allows only replacements, which cost 1 in the simplified definition. In the literature the search problem is in many cases called "string matching with k mismatches". The distance is symmetric, and it is finite whenever $|x| = |y|$. In this case it holds $0 \leq d(x, y) \leq |x|$.

– *Episode distance* [DFG+97]: allows only insertions, which cost 1. In the literature the search problem is in many cases called "episode matching", since it models the case where a sequence of events is sought, where all them must occur within a short period. This distance is not symmetric, and it may not be possible to convert x into y in this case. Hence, $d(x, y)$ is either $|y| - |x|$ or ∞.

– *Longest Common Subsequence distance* [NW70, AG87]: allows only insertions and deletions, all costing 1. The name of this distance refers to the fact that it measures the length of the longest pairing of characters that can be made between both strings, so that the pairings respect the order of the letters. The distance is the number of unpaired characters. The distance is symmetric, and it holds $0 \leq d(x, y) \leq |x| + |y|$.

In all cases, except the episode distance, one can think that the changes can be made over x or y. Insertions on x are the same as deletions in y and vice versa, and replacements can be made in any of the two strings to match the other.

This paper is most concerned with the simple edit distance, which we denote $ed()$. Although transpositions are of interest (especially in case of typing errors), there are few algorithms to deal with them. However, we will consider them at some points of this work (note that a transposition can be simulated with an insertion plus a deletion, but the cost is different). We also will point out when the algorithms can be extended to have different costs of the operations (which is of special interest in computational biology), including the extreme case of not allowing some operations. This includes the other distances mentioned.

Note that if the Hamming or edit distance are used, then the problem makes sense for $0 < k < m$, since if we can perform m operations we can make the pattern match at any text position by means of m replacements. The case $k = 0$ corresponds to exact string matching and is therefore excluded from this work. Under these distances, we call $\alpha = k/m$ the *error level*, which given the above conditions satisfies $0 < \alpha < 1$. This value gives an idea of the "error ratio" allowed in the match (i.e. the fraction of the pattern that can be wrong).

We finish this section with some notes about the algorithms we are going to consider. Like string matching, this area is suitable for very theoretical and for very practical contributions. There exist a number of algorithms with important improvements in their theoretical complexity but very slow in practice. Of course, for carefully built scenarios (say, $m = 100,000$ and $k = 2$) these algorithms could be a practical alternative, but these cases do not appear in applications. Therefore, we point out now which are the parameters of the problem that we consider "practical", i.e. likely to be of use in some application, and when we say later "in practice" we mean under the following assumptions.

– The pattern length can be as short as 5 letters (e.g. text retrieval) and as long as a few hundred letters (e.g. computational biology).

– The number of errors allowed k satisfies that k/m is a moderately low value. Reasonable values range from $1/m$ to $1/2$.

– The text length can be as short as a few thousand letters (e.g. computational biology) and as long as megabytes or gigabytes (e.g. text retrieval).

– The alphabet size σ can be as low as four letters (e.g. DNA) and a high as 256 letters (e.g. compression applications). It is also reasonable to think in even larger alphabets (e.g. oriental languages or word oriented text compression). The alphabet may or may not be random.

3.2 Suffix Trees and Suffix Automata

Suffix trees [Wei73, Knu73, AG85] are widely used data structures for text processing [Apo85]. Any position i in a string S defines automatically a *suffix* of S, namely $S_{i...}$. In essence, a suffix tree is a trie data structure built over all the suffixes of S. At the leaf nodes the pointers to the suffixes are stored. Each leaf represents a suffix and each internal node represents a unique substring of S. Every substring of S can be found by traversing a path from the root. Each node representing the substring ax has a *suffix link* that leads to the node representing the substring x.

To improve space utilization, this trie is compacted into a Patricia tree [Mor68]. This involves compressing unary paths. At the nodes which root a compressed path, an indication of how many characters to skip is stored. Once unary paths are not present the tree has $O(|S|)$ nodes instead of the worst-case $O(|S|^2)$ of the trie (see Figure 1). The structure can be built in time $O(|S|)$ [McC76, Ukk95].

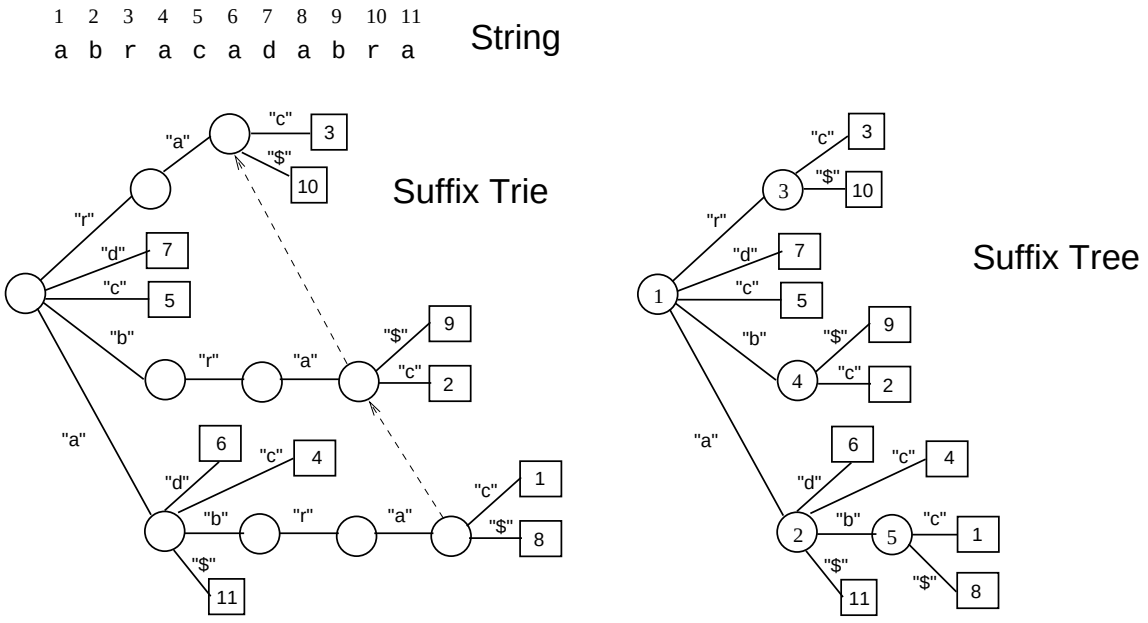

Figure 1: The suffix trie and suffix tree for a sample string. The "$" is a special marker to denote the end of the text. Two suffix links are exemplified in the trie: from `"abra"` to `"bra"` and then to `"ra"`. The internal nodes of the suffix tree show the character position to inspect in the string.

A DAWG (deterministic acyclic word graph) [Cro86, BBH$^+$85] built on a string S is a determin-

istic automaton able to recognize all the substrings of S. As each node in the suffix tree corresponds to a substring, the DAWG is no more than the suffix tree augmented with failure links for the letters not present in the tree. Since final nodes are not distinguished, the DAWG is smaller. DAWGs have similar applications to those of suffix trees, and also need $O(|S|)$ space and construction time. Figure 2 illustrates.

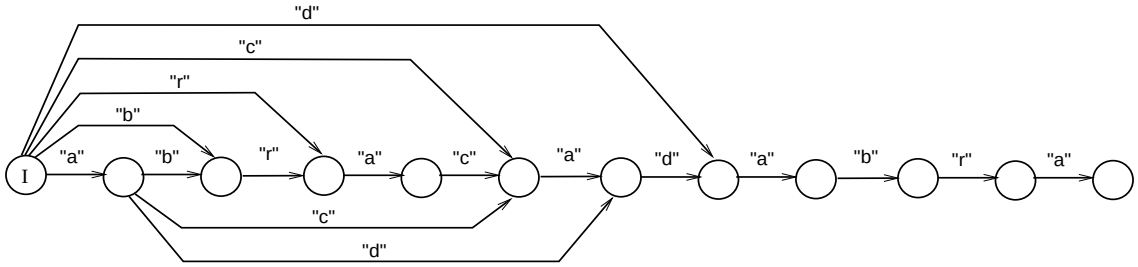

Figure 2: The DAWG or the suffix automaton for the sample string. If all the states are final, it is a DAWG. If only the rightmost state is final then it is a suffix automaton.

A *suffix automaton* on S is an automaton that recognizes all the suffixes of S. The non-deterministic version of this automaton has a very regular structure and is shown in Figure 3 (the deterministic version can be seen in Figure 2).

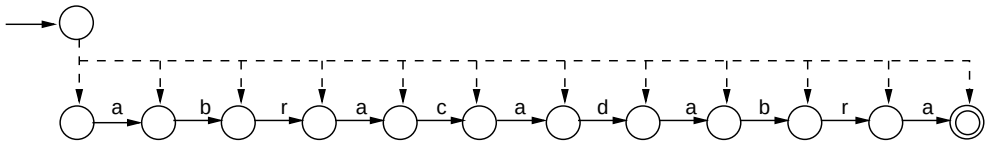

Figure 3: A non-deterministic suffix automaton to recognize any suffix of `"abracadabra"`. Dashed lines represent ε-transitions (i.e. they occur without consuming any input).

3.3 The Tour

Sections 5 to 8 present a historical tour across the four main approaches to online approximate string matching (see Figure 4). In those historical discussions, keep in mind that there may be a long gap between the time when a result is discovered and when it gets finally published in its definitive form. Some apparent inconsistencies can be explained in this way (e.g. algorithms which are "finally" analyzed before they appear). We did our best in the bibliography to trace the earliest version of the works, although the full reference corresponds generally to the final version.

At the beginning of each of these sections we give a taxonomy to help guide the tour. The taxonomy is an acyclic graph where the nodes are the algorithms and the edges mean that the lower work can be seen as an evolution of the upper work (although sometimes the developments are in fact independent).

Finally, we specify some notation regarding time and space complexity. When we say that an algorithm is $O(x)$ time we refer to its worst case (although sometimes we say that explicitly). If the cost is on average, we say so explicitly. We also say sometimes that the algorithm is $O(x)$ cost,

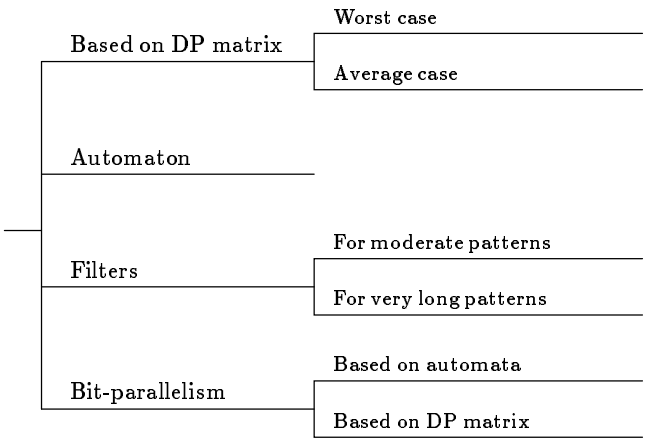

Figure 4: Taxonomy of the types of solutions for online searching.

meaning time. When we refer to space complexity we say so explicitly. The average case analysis normally assumes a random text, where each character is selected uniformly and independently from the alphabet. The pattern is not normally assumed to be random.

4 The Statistics of the Problem

A natural question about approximate searching is: which is the probability of a match? This question is not only interesting by itself, but also essential for the average case analysis of many search algorithms, as seen later. We present now the existing results and an empirical validation. In this section we consider the edit distance only. Some variants can be adapted to these results.

The effort towards analyzing the probabilistic behavior of the edit distance has not given good results in general [KM97]. An exact analysis of the probability of the occurrence of a pattern allowing errors can be found in [RS97]. However, the final expression is extremely complex, has P and k built-in, and does not allow deriving any general result. It can be used for, given a *fixed* P and k, computing the probability (using a computer program) for that particular pattern and number of errors. The results that we present now are simpler and of general use, despite not being exact.

Although the problem of the average edit distance between two strings is closely related to the better studied LCS, the well known results of [CS75, Dek79] can hardly be applied to this case. It can be seen that the average edit distance between two random strings of length m tends to a constant fraction of m as m grows, but the fraction is not known. It holds that for any two strings of length m, $m - lcs \leq ed \leq 2(m - lcs)$, where ed is their edit distance and lcs is the length of their longest common subsequence. As proved in [CS75], the average LCS is between $m/\sqrt{\sigma}$ and $me/\sqrt{\sigma}$ for large σ, and therefore the average edit distance is between $m(1 - e/\sqrt{\sigma})$ and $2m(1 - 1/\sqrt{\sigma})$. For large σ it is conjectured that the true value is $m(1 - 1/\sqrt{\sigma})$ in [SM83].

For our purposes, bounding the probability of a match allowing errors is more important than

the average edit distance. Let $f(m,k)$ be the probability of a random pattern of length m matching a given text position with k errors or less under the edit distance (i.e. that the text position is reported as the end of a match). In [BYN99, Nav98a, NBY99b] upper and lower bounds on the maximum error level α^* for which $f(m,k)$ is exponentially decreasing on m are found. This is important because many algorithms search for potential matches that have to be verified later, and the cost of such verifications is polynomial in m, typically $O(m^2)$. Therefore, if that event occurs with probability $O(\gamma^m)$ for some $\gamma < 1$ then the total cost of verifications is $O(m^2\gamma^m) = o(1)$, which makes the verifications cost negligible.

We first show the analytical bounds for $f(m,k)$, then give a new result on average edit distance, and finally present an experimental verification.

4.1 An Upper Bound

The upper bound for α^* comes from the proof that the matching probability is $f(m,k) = O(\gamma^m)$ for

$$\gamma \;=\; \left(\frac{1}{\sigma\alpha^{\frac{2\alpha}{1-\alpha}}(1-\alpha)^2}\right)^{1-\alpha} \;\leq\; \left(\frac{e^2}{\sigma(1-\alpha)^2}\right)^{1-\alpha} \tag{1}$$

where we note that γ is $1/\sigma$ for $\alpha = 0$ and grows to 1 as α grows. This matching probability is exponentially decreasing on m as long as $\gamma < 1$, which is equivalent to

$$\alpha \;<\; 1 - \frac{e}{\sqrt{\sigma}} - O(1/\sigma) \;\leq\; 1 - \frac{e}{\sqrt{\sigma}} \tag{2}$$

Therefore, $\alpha < 1 - e/\sqrt{\sigma}$ is a conservative condition on the error level that ensures "few" matches. Therefore, the maximum level α^* satisfies $\alpha^* > 1 - e/\sqrt{\sigma}$.

The proof is obtained using a combinatorial model. Based on the observation that $m-k$ common characters must appear in the same order in two strings that match with k errors, all the possible alternatives to select the matching characters from both strings are enumerated. This model, however, does not take full advantage of the properties of the edit distance: even if $m-k$ characters match, the distance can be larger than k. For example, in $ed(abc, bcd) = 2$, i.e. although two characters match, the distance is not 1.

4.2 A Lower Bound

On the other hand, the only optimistic bound we know of is based on considering that only replacements are allowed (i.e. Hamming distance). This distance is simpler to analyze but its matching probability is much lower. Using again a combinatorial model it is shown that the matching probability is $f(m,k) \geq \delta^m\ m^{-1/2}$, where

$$\delta = \left(\frac{1}{(1-\alpha)\sigma}\right)^{1-\alpha}$$

Therefore an upper bound for the maximum α^* value is $\alpha^* \leq 1 - 1/\sigma$, since otherwise it can be proved that $f(m,k)$ is not exponentially decreasing on m (i.e. it is $\Omega(m^{-1/2})$).

4.3 A New Result on Average Edit Distance

We can now prove that the average edit distance is larger than $m(1-e/\sqrt{\sigma})$ for any σ. We define $p(m,k)$ as the probability that the edit distance between two strings of length m is at most k. Note that $p(m,k) \leq f(m,k)$ because in the latter case we can match with any text suffix of length from $m-k$ to $m+k$. Then the average edit distance is

$$\sum_{k=0}^{m} k\ Pr(ed=k) \quad = \quad \sum_{k=0}^{m} Pr(ed>k) \quad = \quad \sum_{k=0}^{m} 1-p(m,k) \quad = \quad m - \sum_{k=0}^{m} p(m,k)$$

which, since $p(m,k)$ increases with k, is larger than

$$m - (Kp(m,K) + (m-K)) \quad = \quad K(1-p(m,K))$$

for any K of our choice. In particular, for $K/m < 1-e/\sqrt{\sigma}$ we have that $p(m,K) \leq f(m,K) = O(\gamma^m)$ for $\gamma < 1$. Therefore choosing $K = m(1-e/\sqrt{\sigma}) - 1$ yields that the edit distance is at least $m(1-e/\sqrt{\sigma}) + O(1)$, for any σ. As we see later, this proofs converts a conjecture about the average running time of an algorithm [CL92] into a fact.

4.4 Empirical Verification

We verify the analysis experimentally in this section (this is also taken from [BYN99, Nav98a]). The experiment consists of generating a large random text ($n = 10$ Mb) and running the search of a random pattern on that text, allowing $k = m$ errors. At each text character, we record the minimum allowed error k for which that text position would match the pattern. We repeat the experiment with 1,000 random patterns.

Finally, we build the cumulative histogram, finding how many text positions have matched with up to k errors, for each k value. We consider that k is "low enough" up to where the histogram values become significant, that is, as long as few text positions have matched. The threshold is set to n/m^2, since m^2 is the normal cost of verifying a match. However, the selection of this threshold is not very important, since the histogram is extremely concentrated. For example, for m in the hundreds, it moves from almost zero to almost n in just five or six increments of k.

Figure 5 shows the results for $\sigma = 32$. On the left we show the histogram we have built, where the matching probability undergoes a sharp increase at α^*. On the right we show the α^* value as m grows. It is clear that α^* is essentially independent on m, although it is a bit lower for short patterns. The increase in the left plot at α^* is so sharp that the right plot would be the same if we plotted the value of the average edit distance divided by m.

Figure 6 uses a stable $m = 300$ to show the α^* value as a function of σ. The curve $\alpha = 1-1/\sqrt{\sigma}$ is included to show its closeness to the experimental data. Least squares give the approximation $\alpha^* = 1-1.09/\sqrt{\sigma}$, with a relative error smaller than 1%. This shows that the upper bound analysis (Eq. (2)) matches reality better, provided we replace e by 1.09 in the formulas.

Therefore, we have shown that the matching probability has a sharp behavior: for low α it is very low, not as low as $1/\sigma^m$ like exact string matching, but still exponentially decreasing in m, with an exponent base larger than $1/\sigma$. At some α value (that we called α^*) it sharply increases and quickly becomes almost 1. This point is close to $\alpha^* = 1-1/\sqrt{\sigma}$ in practice.

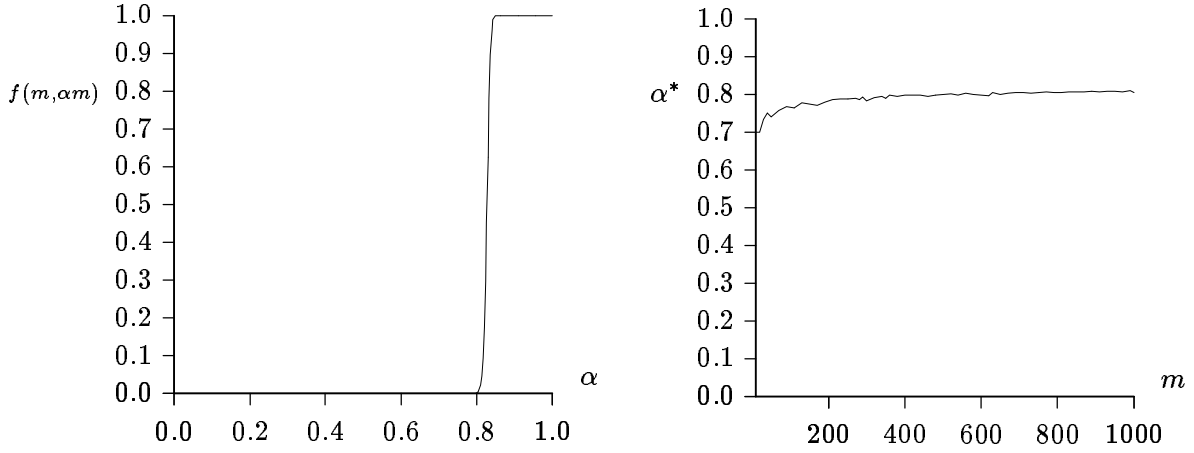

Figure 5: On the left, probability of an approximate match as a function of the error level ($m = 300$). On the right, the observed α^* error level as a function of the pattern length. Both cases correspond to random text with $\sigma = 32$.

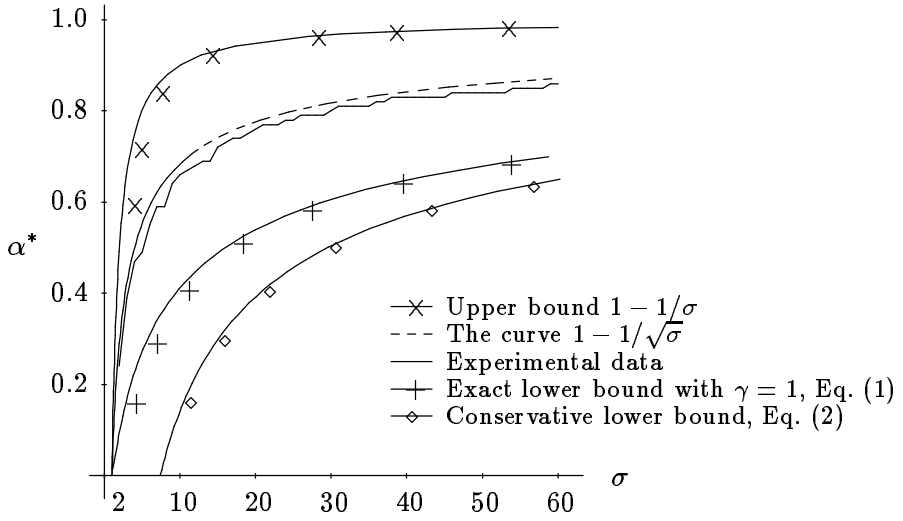

Figure 6: Theoretical and practical values for α^*, for $m = 300$ and different σ values.

This is why the problem has interest only up to a given error level, since for higher errors almost all text positions match. This is also the reason that makes some algorithms to have good average behavior only for low enough error levels. The point $\alpha^* = 1 - 1/\sqrt{\sigma}$ matches the conjecture of [SM83].

5 Dynamic Programming Algorithms

We start our tour with the oldest among the four areas, which inherits directly from the earliest work. Most of the theoretical breakthroughs in the worst case algorithms belong to this category, although only a few of them are really competitive in practice. The latest practical work in this area dates back to 1992, although there are recent theoretical improvements. The major achievements are $O(kn)$ worst-case algorithms and $O(kn/\sqrt{\sigma})$ average-case algorithms, as well as other recent theoretical improvements on the worst-case.

We start by presenting the first algorithm that solved the problem and then give a historical tour on the improvements over the initial solution. Figure 7 helps guide the tour.

5.1 The First Algorithm

We present now the first algorithm to solve the problem. It has been rediscovered many times in the past, in different areas, e.g. [Vin68, NW70, San72, Sel74, WF74, LW75] (there are more in [Ull77, SK83, Kuk92]). However, this algorithm computed the edit distance, and it was converted into a search algorithm only in 1980 by Sellers [Sel80]. Although the algorithm is not very efficient, it is among the most flexible ones to adapt to different distance functions.

We first show how to compute the edit distance between two strings x and y. Later, we extend that algorithm to search a pattern in a text allowing errors. Finally, we show how to handle more general distance functions.

5.1.1 Computing Edit Distance

The algorithm is based on dynamic programming. Imagine that we need to compute $ed(x, y)$. A matrix $C_{0..|x|,0..|y|}$ is filled, where $C_{i,j}$ represents the minimum number of operations needed to match $x_{1..i}$ to $y_{1..j}$. This is computed as follows

$$\begin{aligned} C_{i,0} &= i \\ C_{0,j} &= j \\ C_{i,j} &= \text{if } (x_i = y_j) \text{ then } C_{i-1,j-1} \\ &\quad \text{else } 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}) \end{aligned}$$

where at the end $C_{|x|,|y|} = ed(x, y)$. The rationale of the above formula is as follows. First, $C_{i,0}$ and $C_{0,j}$ represent the edit distance between a string of length i or j and the empty string. Clearly i (respectively j) deletions are needed on the nonempty string. For two non-empty strings of length i and j, we assume inductively that all the edit distances between shorter strings have already been computed, and try to convert $x_{1..i}$ into $y_{1..j}$.

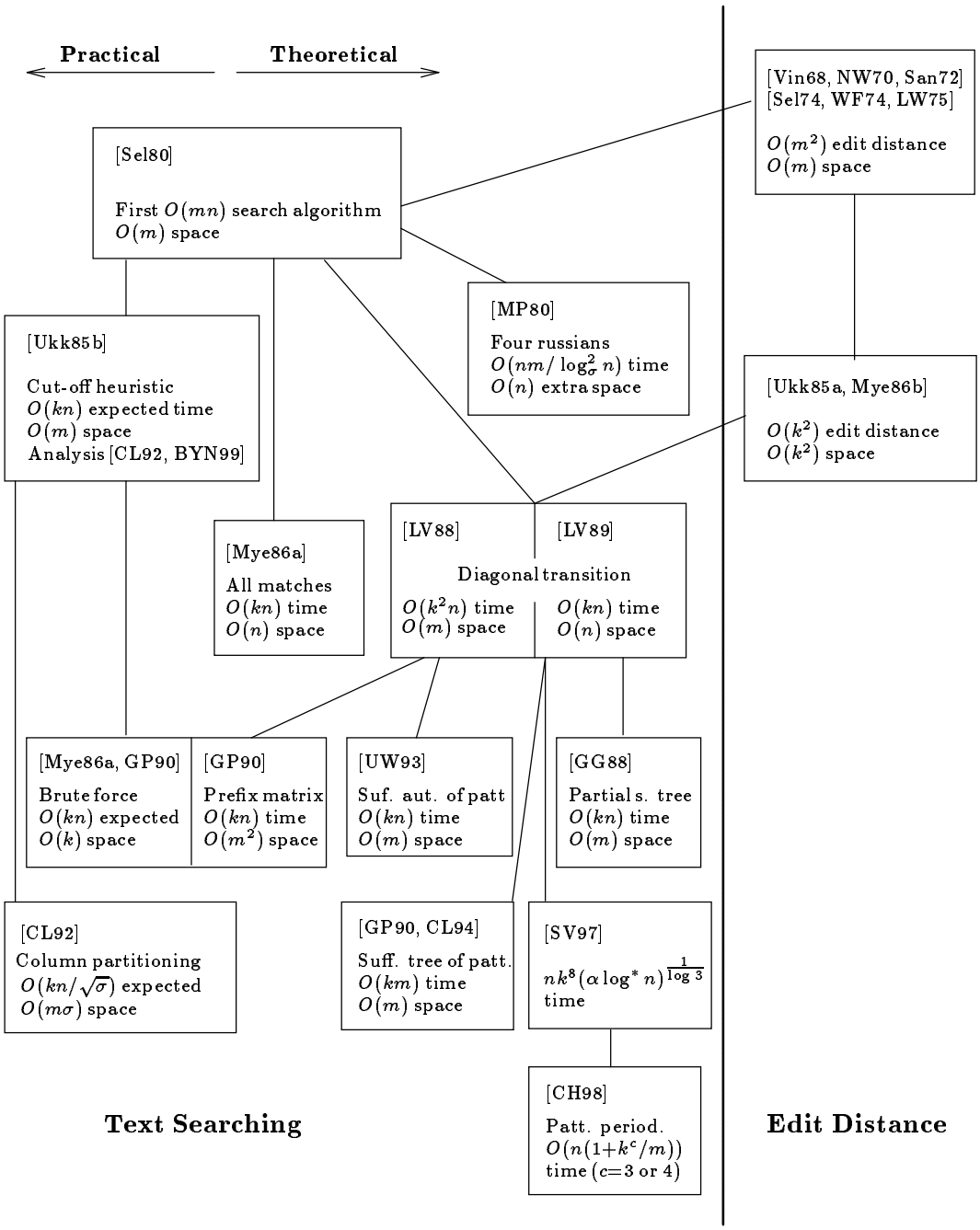

Figure 7: Taxonomy of algorithms based on the dynamic programming matrix.

Consider the last characters x_i and y_j. If they are equal, then we do not need to consider them and we proceed in the best possible way to convert $x_{1..i-1}$ into $y_{1..j-1}$. On the other hand, if they are not equal, we must deal with them in some way. Following the three allowed operations, we can delete x_i and convert in the best way $x_{1..i-1}$ into $y_{1..j}$, insert y_j at the end of $x_{1..i}$ and convert in the best way $x_{1..i}$ into $y_{1..j-1}$, or replace x_i by y_j and convert in the best way $x_{1..i-1}$ into $y_{1..j-1}$. In all cases, the cost is 1 plus the cost for the rest of the process (already computed). Notice that the insertions in one string are equivalent to deletions in the other.

The dynamic programming algorithm must fill the matrix in such a way that the upper, left, and upper-left neighbors of a cell are computed prior to computing that cell. This is easily achieved by either a row-wise left-to-right traversal or a column-wise top-to-bottom traversal. Figure 8 illustrates this algorithm to compute ed("survey", "surgery").

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	**0**	1	2	3	4	5	6
u	2	1	**0**	1	2	3	4	5
r	3	2	1	**0**	1	2	3	4
v	4	3	2	1	**1**	2	3	4
e	5	4	3	2	2	**1**	**2**	3
y	6	5	4	3	3	2	2	**2**

Figure 8: The dynamic programming algorithm to compute the edit distance between "survey" and "surgery". The bold entries show the path to the final result.

Therefore, the algorithm is $O(|x||y|)$ time in the worst and average case. However, the space required is only $O(\min(|x|, |y|))$. This is because, in the case of a column-wise processing, only the previous column must be stored in order to compute the new one, and therefore we just keep one column and update it. We can process the matrix row-wise or column-wise so that the space requirement is minimized.

On the other hand, the sequences of operations performed to transform x into y can be easily recovered from the matrix, simply by proceeding from the cell $C_{|x|,|y|}$ to the cell $C_{0,0}$ following the path (i.e. sequence of operations) that matches the update formula (multiple paths may exist). In this case, however, we need to store the complete matrix or at least an area around the main diagonal.

This matrix has some properties which can be easily proved by induction (see, e.g. [Ukk85a]) and which make it possible to design better algorithms. Some of the most used are that the values of neighboring cells differ in at most one, and that upper-left to lower-right diagonals are nondecreasing.

5.1.2 Text Searching

We show now how to adapt this algorithm to search a short pattern P in a long text T. The algorithm is basically the same, with $x = P$ and $y = T$ (proceeding column-wise so that $O(m)$

space is required). The only difference is that we must allow that any text position is the potential start of a match. This is achieved by setting $C_{0,j} = 0$ for all $j \in 0..n$. That is, the empty pattern matches with zero errors at any text position (because it matches with a text substring of length zero).

The algorithm then initializes its column $C_{0..m}$ with the values $C_i = i$, and processes the text character by character. At each new text character T_j, its column vector is updated to $C'_{0..m}$. The update formula is

$$\begin{aligned} C'_i &= \text{if } (P_i = T_j) \text{ then } C_{i-1} \\ &\quad \text{else } 1 + \min(C'_{i-1}, C_i, C_{i-1}) \end{aligned}$$

and the text positions where $C_m \leq k$ are reported.

The search time of this algorithm is $O(mn)$ and its space requirement is $O(m)$. This is a sort of worst case in the analysis of all the algorithms that we consider later. Figure 9 exemplifies this algorithm applied to search the pattern "survey" in the text "surgery" (a very short text indeed) with at most $k = 2$ errors. In this case there are 3 occurrences.

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

Figure 9: The dynamic programming algorithm to search "survey" in the text "surgery" with two errors. Bold entries indicate matching text positions.

Finally, we point out that although we have presented a column-wise algorithm to fill the matrix, many other works are based in alternative filling styles. There are applications for row-wise filling [Nav98b] or even by (upper-left to lower-right) diagonals or "secondary" (upper-right to lower-left) diagonals. Some filling styles, as diagonals, need to set up a different recurrence to compute the cells. We cover them later.

5.1.3 Other Distance Functions

It is easy to adapt this algorithm for the other distance functions mentioned. If the operations have different costs, we add the cost instead of adding 1 when computing $C_{i,j}$, i.e.

$$\begin{aligned} C_{0,0} &= 0 \\ C_{i,j} &= \min(C_{i-1,j-1} + \delta(x_i, y_j), C_{i-1,j} + \delta(x_i, \varepsilon), C_{i,j-1} + \delta(\varepsilon, y_j)) \end{aligned}$$

where we assume $\delta(a, a) = 0$ for any $a \in \Sigma$ and that $C_{-1,j} = C_{i,-1} = \infty$ for all i, j.

For distances that do not allow some operations, we just take them out of the minimization formula, or which is the same, we assign ∞ to their δ cost. For transpositions, we allow a fourth rule that says that $C_{i,j}$ can be $C_{i-2,j-2}+1$ if $x_{i-1}x_i = y_j y_{j-1}$ [LW75].

The most complex case is to allow general substring replacements, in the form of a finite set R of rules. The formula is given in [Ukk85a].

$$\begin{aligned} C_{0,0} &= 0 \\ C_{i,j} &= \min(C_{i-1,j-1} \text{ if } x_i = y_j, \\ &\qquad C_{i-|s_1|,j-|s_2|} + \delta(s_1,s_2) \quad \text{for each } \delta(s_1,s_2) \in R,\ x_{1..i} = x's_1,\ y_{1..j} = y's_2) \end{aligned}$$

An interesting problem is how to compute this recurrence efficiently. A naive approach takes $O(|R|mn)$, where $|R|$ is the sum of all the lengths of the strings in R. A better solution is to build two Aho-Corasick automata [AC75] with the left and right hand sides of the rules, respectively. The automata are run as we advance in both strings (left hand sides in x and right hand sides in y). For each pair of states (i_1, i_2) of the automata we precompute the set of replacements that can be tried (i.e. those δ's whose left and right hand match the suffixes of x and y, respectively, represented by the automata states). Hence, we know in constant time (per cell) the set of possible replacements. The complexity is now much lower, in the worst case it is $O(cmn)$ where c is the maximum number of rules applicable to a single text position.

As said, the dynamic programming approach is unbeaten in flexibility, but its time requirements are indeed high. A number of improved solutions have been proposed along the years. Some of them work only for the edit distance, while others can still be adapted to other distance functions. Before considering the improvements, we mention that there exists a way to see the problem as a shortest path problem on a graph built on the pattern and the text [Ukk85a]. This reformulation has been conceptually useful for more complex variants of the problem.

5.2 Improving the Worst Case

Masek & Paterson 1980 It is interesting that one important worst-case theoretical result on this area is as old as the Sellers [Sel80] algorithm itself. In 1980, Masek and Paterson [MP80] found an algorithm whose worst case cost is $O(mn/\log_\sigma^2 n)$ and requires $O(n)$ extra space. This is an improvement over the $O(mn)$ classical complexity.

The algorithm is based on the Four-Russians technique [ADKF75]. Basically, it replaces the alphabet Σ by r-tuples (i.e. Σ^r) for a small r. Considered algorithmically, it first builds a table of solutions of all the possible problems (i.e. portions of the matrix) of size $r \times r$, and then uses the table to solve the original problem in blocks of size r. Figure 10 illustrates.

The values inside the $r \times r$ size cells depend on the corresponding letters in the pattern and the text, which gives σ^{2r} possibilities. They also depend on the values in the last column and row of the upper and left cells, as well as the bottom-right state of the upper left cell (see Figure 10). Since neighboring cells differ in at most one, there are only three choices for adjacent cells once the current cell is known. Therefore, this adds only $m(3^{2r})$ possibilities. In total, there are $m(3\sigma)^{2r}$ different cells to precompute. Using $O(n)$ memory we have enough space for $r = \log_{3\sigma} n$, and since we finally compute mn/r^2 cells, the final complexity follows.

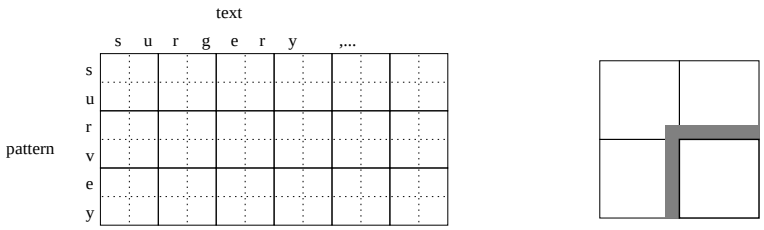

Figure 10: The Masek and Paterson algorithm partitions the dynamic programming matrix in cells ($r = 2$ in this example). On the right, we shaded the entries of adjacent cells that influence the current one.

The algorithm is only of theoretical interest, since as the same authors estimate, it will not beat the classical algorithm for texts below 40 Gb size (and it would need that extra space!). Adapting it to other distance functions seems not difficult, but the dependencies among different cells may become more complex.

Ukkonen 1983 In 1983, Ukkonen [Ukk85a] presented an algorithm able to compute the edit distance between two strings x and y in $O(ed(x,y)^2)$ time, or to check in time $O(k^2)$ whether that distance was $\leq k$ or not. This is the first member of what has been called "diagonal transition algorithms", since it is based in the fact that the diagonals of the dynamic programming matrix (running from the upper-left to the lower-right cells) are monotonically increasing (more than that, $C_{i+1,j+1} \in \{C_{i,j}, C_{i,j}+1\}$). The algorithm is based on computing in constant time the positions where the values along the diagonals are incremented. Only $O(k^2)$ such positions are computed to reach the lower-right decisive cell.

Figure 11 illustrates the idea. Each diagonal *stroke* represents a number of errors, and is a sequence where both strings match. When a stroke of e errors starts, it continues until the adjacent $e-1$ strokes continue or until it keeps matching the text. To compute each stroke in constant time we need to know until where it matches the text. The way to do this in constant time is explained shortly.

Figure 11: On the left, the $O(k^2)$ algorithm to compute the edit distance. On the right, the way to compute the strokes in diagonal transition algorithms. The solid bold line is guaranteed to be part of the new stroke of e errors, while the dashed part continues as long as both strings match.

Landau and Vishkin 1985 In 1985 and 1986, Landau and Vishkin found the first worst-case time improvements for the search problem. All of them and the thread that followed were diagonal transition algorithms. In 1985 [LV88] they show an algorithm which is $O(k^2n)$ time and $O(m)$ space, and in 1986 [LV89] they obtain $O(kn)$ time and $O(n)$ space.

The main idea of Landau and Vishkin was to adapt to text searching the Ukkonen's diagonal transition algorithm for edit distance [Ukk85a]. Basically, the dynamic programming matrix was computed diagonal-wise (i.e. stroke by stroke) instead of column-wise. They wanted to compute in constant time the length of each stroke (i.e. the point where the values along a diagonal were to be incremented). Since a text position was to be reported when matrix row m was reached before incrementing more than k times the values along the diagonal, this gave immediately the $O(kn)$ algorithm. Another way to see it is that each diagonal is abandoned as soon as the k-th stroke ends, there are n diagonals and hence nk strokes, each of them computed in constant time (recall Figure 11).

A recurrence on diagonals (d) and number of errors (e), instead of rows (i) and columns (j), is set up in the following way

$$\begin{aligned}
L_{d,-1} &= L_{n+1,e} = -1, \text{ for all } e, d \\
L_{d,|d|-2} &= |d| - 2, \text{ for } -(k+1) \leq d \leq -1 \\
L_{d,|d|-1} &= |d| - 1, \text{ for } -(k+1) \leq d \leq -1 \\
L_{d,e} &= i + \max_{\ell}(P_{i+1..i+\ell} = T_{d+i+1..d+i+\ell}) \\
&\quad \text{where } i = \max(L_{d,e-1} + 1, L_{d-1,e-1}, L_{d+1,e-1})
\end{aligned}$$

where the external loop updates e from 0 to k and the internal one updates d from $-e$ to n. Negative numbered diagonals are those virtually starting before the first text position. Figure 12 shows our search example using this recurrence.

	-3	-2	-1	0	1	2	3	4	5	6	7
0		0	3	0	0	0	0	0	0	0	0
1	1	1	4	5	3	1	1	1	1	1	1
2	2	5	**6**	**6**	**6**	3	2	3	2	2	2

Figure 12: The diagonal transition matrix to search "survey" in the text "surgery" with two errors. Bold entries indicate matching diagonals. The rows are e values and the columns are the d values.

The difficult part is how to compute the strokes in constant time (i.e. the $\max_\ell(\cdots)$). The problem is equivalent to knowing which is the longest prefix of $P_{i\ldots}$ that matches $T_{j\ldots}$. This data has ben called thereafter "matching statistics". The algorithms of this section differ basically in how they manage to compute the matching statistics fast.

We defer for later the explanation of [LV88]. In [LV89], this is done by building the suffix tree (see Section 3.2) of $T;P$ (text concatenated with pattern), where the huge $O(n)$ extra space comes from. The longest prefix common to both suffixes $P_{i\ldots}$ and $T_{j\ldots}$ can be visualized in the suffix tree

as follows: imagine the root to leaf paths that end in each of the two suffixes. Both parts share the beginning of the path (at least they share the root). The last suffix tree node common to both paths represents a substring which is precisely the longest common prefix. In the literature, this last common node is called *lowest common ancestor* (LCA) of two nodes.

Despite being conceptually clear, it is not easy to find this node in constant time. In 1986, the only existing LCA algorithm was [HT84], which had constant amortized time, i.e. it answered $n' > n$ LCA queries in $O(n')$ time. In our case we have kn queries, so each one finally costed $O(1)$. The resulting algorithm, however, is quite slow in practice.

Myers 1986 In 1986, Myers found also an algorithm with $O(kn)$ worst-case behavior [Mye86a]. It needed $O(n)$ extra space, and shared the idea of computing the k new strokes using the previous ones, as well as the use of a suffix tree on the text for the LCA algorithm. Unlike other algorithms, this one is able to report the $O(kn)$ matching substrings of the text (not only the endpoints) in $O(kn)$ time. This makes the algorithm suitable for more complex applications, for instance in computational biology. The original reference is a technical report and never went to press, but it has been recently included in a larger work [LMS98].

Galil and Giancarlo 1988 In 1988, Galil and Giancarlo [GG88] obtained the same time complexity of Landau and Vishkin using $O(m)$ space. Basically, the suffix tree of the text is built by overlapping pieces of size $O(m)$. The algorithm scans the text four times, being even slower than [LV89]. Therefore, the result was of theoretical interest.

Galil and Park 1989 One year later, in 1989, Galil and Park [GP90] obtained $O(kn)$ worst-case time and $O(m^2)$ space, worse in theory than [GG88] but much better in practice. This time the idea was to build the matching statistics of the pattern against itself (longest match between $P_{i\ldots}$ and $P_{j\ldots}$, hence the $O(m^2)$ complexity), resembling in some sense the basic ideas of [KMP77]. This algorithm is still slow in practice anyway.

The algorithm goes diagonal by diagonal, computing its k strokes, and using the k strokes of the previous diagonal to save some character comparisons in the current one. Note that if there is a stroke in the previous diagonal d spanning rows i_1 to i_2, this means that $T_{d+i_1..d+i_2} = P_{i_1..i_2}$, and therefore we can compare the pattern against itself instead of against the text.

As we have precomputed all the longest matches of the pattern against itself, we can use that information to know the longest match between pattern and text. So we consider the longest match of the pattern against itself, starting at the two positions corresponding to the current stroke and the stroke below it (in the previous diagonal, which we call the "stroke-below"). Depending on which is shorter or longer between this longest match and the stroke-below itself, there are three alternatives (recall Figure 11). First, if the stroke-below ends before the longest match, then the current stroke finishes at the same row of the stroke-below (since it differed from the text and the current stroke still was equal to it). Second, if the stroke ends after the longest match, the current stroke ends where the longest match does (since the stroke-below was still equal to the text when it differed from the current stroke). Finally, if both end together, we cannot know the result and we need to consider the next stroke of the previous diagonal.

As shown in [GP90], the total cost to compute the k strokes[2] of the new diagonal is $O(k)$, since a linear pass is performed over the previous diagonal overall. Despite that they can work $O(k)$ for a single stroke of the new diagonal, they cannot work more than $O(k)$ for the whole diagonal (there are also up to k direct comparisons performed between text and pattern, when the stroke-below is of length zero). A very similar algorithm had been presented before by Landau and Vishkin [LV88], but they have $O(k^2)$ cost to compute the new diagonal.

Finally, Galil and Park show that the $O(m^2)$ extra space can be reduced to $O(m)$ by using a suffix tree of the pattern (not the text as in previous work) and LCA algorithms, so we add different entries in Figure 7. They also show how to add transpositions to the edit operations at the same complexity. This technique can be extended to all these diagonal transition algorithms. We believe that allowing different integral costs for the operations or forbidding some of them can be achieved with simple modifications of the algorithms.

Ukkonen and Wood 1990 An idea similar to that of using the suffix tree of the pattern (and similarly slow in practice) was independently discovered by Ukkonen and Wood in 1990 [UW93]. They use a suffix automaton (described in Section 3.2) on the pattern to find the matching statistics, instead of the table. As the algorithm progresses over the text, the suffix automaton keeps count of the pattern substrings that match the text at any moment. Despite that they report $O(m^2)$ space for the suffix automaton, it cant take $O(m)$ space.

Chang and Lawler 1990 Also in 1990, Chang and Lawler [CL94] repeated the idea that was briefly mentioned in [GP90]: the matching statistics can be computed using the suffix tree of the pattern and LCA algorithms. However, they used a newer and faster LCA algorithm [SV88], truly $O(1)$, and reported the best time among algorithms with guaranteed $O(kn)$ performance. However, the algorithm is still not competitive in practice.

Cole and Hariharan 1998 In 1998, Cole and Hariharan [CH98] presented an algorithm with worst case $O(n(1+k^c/m))$, where $c=3$ if the pattern is "mostly aperiodic" and $c=4$ otherwise[3]. The idea is that, unless a pattern has a lot of self-repetition, only a few diagonals of a diagonal transition algorithm need to be computed.

This algorithm can be thought of as a filter (see next sections) with worst case guarantee useful for very small k. It resembles some ideas of the filters developed in [CL94]. Probably other filters can be proved to have good worst cases under some periodicity assumptions on the pattern, but this thread has not been explored up to now. This algorithm is an evolution over a previous one [SV97], which is more complex and has a worse complexity, namely $O(nk^8(\alpha \log^* n)^{1/\log 3})$. In any case, the interest of this work is theoretical too.

5.3 Improving the Average Case

Ukkonen 1985 The first improvement to the average case is due to Ukkonen in 1985. The algorithm, a short note at the end of [Ukk85b], improved the dynamic programming algorithm

[2] Called "reference triples" in there.

[3] The definition of "mostly aperiodic" is rather technical and related to the number of auto-repetition that occurs in the pattern. Most patterns are "mostly aperiodic".

to $O(kn)$ average time and and $O(m)$ space. This algorithm has been called later the "cut-off heuristic". The main idea is that, since a pattern does not normally match in the text, the values at each column (from top to bottom) quickly reach $k+1$ (i.e. mismatch), and that if a cell has a value larger than $k+1$, the result of the search does not depend on its exact value. A cell is called *active* if its value is at most k. The algorithm simply keeps count of which is the last active cell and avoids working on the rest of the cells.

To keep the last active cell, we must be able to recompute it for each new column. At each new column, the last active cell can be incremented in at most one, so we check if we have activated the next cell at $O(1)$ cost. However, it is also possible that which was the last active cell becomes inactive now. In this case we have to search upwards which is the new last active cell. Despite that we can work $O(m)$ in a given column, we cannot work more than $O(n)$ overall, because there are at most n increments of this value in the whole process, and hence there are no more than n decrements. Hence, the last active cell is maintained at $O(1)$ amortized cost per column.

Ukkonen conjectured that this algorithm was $O(kn)$ on average, but this was proven only in 1992 by Chang and Lampe [CL92]. The proof was refined in 1996 by Baeza-Yates and Navarro [BYN99]. The result can probably be extended to more complex distance functions, although with substrings the last active cell must exceed k by enough to ensure that it can never return to a value smaller than k. In particular, it must have the value $k+2$ if transpositions are allowed.

Myers 1986 An algorithm in [Mye86a] is based on diagonal transitions as those of the previous sections, but the strokes are simply computed by brute force. Myers showed that the resulting algorithm was $O(kn)$ on average. This is clear because the length of the strokes is $\sigma/(\sigma-1) = O(1)$ on average. The same algorithm was proposed again in 1989 by Galil and Park [GP90]. Since only the k strokes need to be stored, the space is $O(k)$.

Chang and Lampe 1992 In 1992, Chang and Lampe [CL92] gave a new algorithm called "column partitioning", based on exploiting a different property of the dynamic programming matrix. They consider again the fact that, along each column, the numbers are normally increasing. They work on "runs" of consecutive increasing cells (a run ends when $C_{i+1} \neq C_i + 1$). They manage to work $O(1)$ per run in the column actualization process.

To update each run in constant time, they precompute $\mathrm{loc}(j,x) = \min_{j' \geq j} P_{j'} = x$ for all pattern positions j and all characters x (hence it needs $O(m\sigma)$ space). At each column of the matrix, they consider the current text character x and the current row j, and know in constant time where the run is going to end (i.e. next character match). The run can end before, namely where the parallel run of the previous column ends.

Based on empirical observations, they conjecture that the average length of the runs is $O(\sqrt{\sigma})$. Notice that this matches our result that the average edit distance is $m(1-e/\sqrt{\sigma})$, since this is the number of increments along columns, and therefore there are $O(m/\sqrt{\sigma})$ non-increments (i.e. runs). From there it is clear that each run has average length $O(\sqrt{\sigma})$. Therefore, we have just proved Chang and Lampe's conjecture.

Since the paper uses the cut-off heuristic of Ukkonen, their average search time is $O(kn/\sqrt{\sigma})$. This is, in practice, the fastest algorithm of this class.

Unlike the other algorithms of this section, it seems difficult to adapt [CL92] to other distance

functions, since the idea strongly relies on the unitary costs. It is mentioned that the algorithm could run in average time $O(kn \log\log(m)/\sigma)$ but it would be unpractical.

6 Algorithms Based on Automata

This area is also rather old. It is interesting because it gives the best worst-case time algorithm ($O(n)$, which matches the lower bound of the problem). However, there is a time and space exponential dependence on m and k that limits its practicality.

We first present the basic solution and then discuss the improvements. Figure 13 shows the historical map of this area.

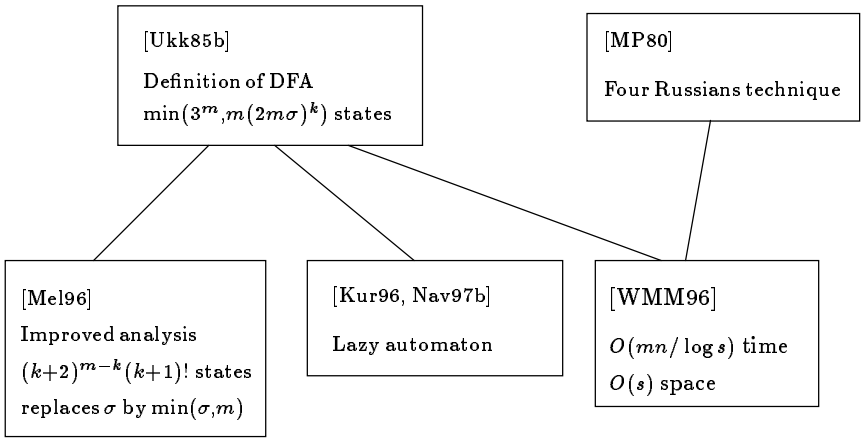

Figure 13: Taxonomy of algorithms based on deterministic automata.

6.1 An Automaton for Approximate Search

An alternative and very useful way to consider the problem is to model the search with a non-deterministic automaton (NFA). This automaton (in its deterministic form) was firstly proposed in [Ukk85b], and firstly used in non-deterministic form (although implicitly) in [WM92a]. It is shown explicitly in [BY91, BY96, BYN99].

Consider the NFA for $k = 2$ errors under edit distance shown in Figure 14. Every row denotes the number of errors seen (the first row zero, the second row one, etc.). Every column represents matching a pattern prefix. Horizontal arrows represent matching a character (i.e. if the pattern and text characters match, we advance in the pattern and in the text). All the others increment the number of errors (move to the next row): vertical arrows insert a character in the pattern (we advance in the text but not in the pattern), solid diagonal arrows replace a character (we advance in the text and pattern), and dashed diagonal arrows delete a character of the pattern (they are ε-transitions, since we advance in the pattern without advancing in the text). The initial self-loop allows a match to start anywhere in the text. The automaton signals (the end of) a match whenever a rightmost state is active. If we do not care about the number of errors of the occurrences, we can consider final states those of the last full diagonal.

It is not hard to see that once a state in the automaton is active, all the states of the same column and higher rows are active too. Moreover, at a given text character, if we collect the smallest active rows at each column, we obtain the vertical vector of the dynamic programming algorithm (in this case $[0, 1, 2, 3, 3, 3, 2]$, compare to Figure 9).

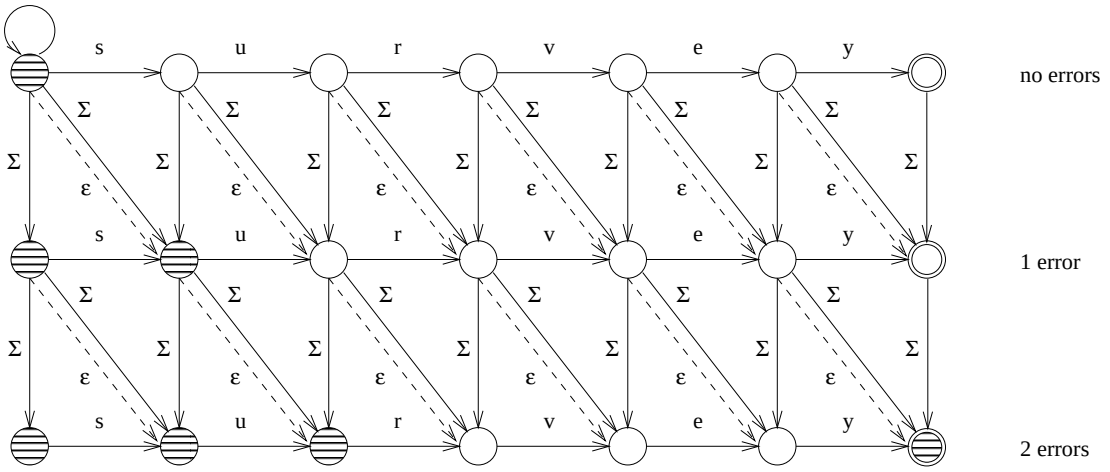

Figure 14: An NFA for approximate string matching of the pattern "survey" with two errors. The shaded states are those active after reading the text "surgery".

Other types of distances (Hamming, LCS and Episode) are obtaining by deleting some arrows of the automaton. Different integer costs for the operations can also be modeled by changing the arrows. For instance, if insertions cost 2 instead of 1, we make the vertical arrows to move from rows i to rows $i + 2$. Transpositions are modeled by adding an extra state $S_{i,j}$ between each pair of states at position (i, j) and $(i + 1, j + 2)$, and arrows labeled P_{i+2} from state (i, j) to S_{ij} and P_{i+1} between $S_{i,j}$ and $(i+1, j+2)$ [Mel96]. Adapting to general substring replacement needs more complex setups but it is always possible.

This automaton can be simply made deterministic to obtain $O(n)$ worst case search time. However, as we see next, the main problem becomes the construction of the DFA (deterministic finite automaton). An alternative solution is based on simulating the NFA instead of making it deterministic.

6.2 Implementing the Automaton

Ukkonen 1985 In 1985, Ukkonen proposed the idea of a deterministic automaton for this problem [Ukk85b]. However, an automaton as that of Figure 14 was not explicitly considered. Rather, each possible set of values for the columns of the dynamic programming matrix was a state of the automaton. Once the set of all possible columns and the transitions among them were built, the text was scanned with the resulting automaton, performing exactly one transition per character read.

The big problem with this scheme was that the automaton had a potentially huge number of states, which had to be built and stored. To improve space usage, Ukkonen proved that all the

elements in the columns that were larger than $k+1$ could be replaced by $k+1$ without affecting the output of the search (the lemma was used in the same paper to design the cut-off heuristic described in Section 5.3). This reduced the potential number of different columns. He also showed that adjacent cells in a column differed in at most one. Hence, the column states could be defined as a vector of m incremental values in the set $\{-1, 0, 1\}$.

All this made possible to obtain in [Ukk85b] a nontrivial bound to the number of states of the automaton, namely $O(\min(3^m, m(2m\sigma)^k))$. This size, although much better than the obvious $O((k+1)^m)$, is still very large except for short patterns or very low error levels. The resulting space complexity of the algorithm is m times the above value. This exponential space complexity has to be added to the $O(n)$ time complexity, as the preprocessing time to build the automaton.

As a final comment, Ukkonen suggested that the columns could be only partially computed up to, say, $3k/2$ entries. Since he conjectured (and later was proved in [CL92]) that the columns of interest were $O(k)$ on average, this would normally not affect the algorithm, though it will reduce the number of possible states. If at some point the states not computed were really needed, the algorithm would compute them by dynamic programming.

Notice that to incorporate transpositions and substring replacements into this conception we need to consider that each state is the set of the j last columns of the dynamic programming matrix, where j is the longest left hand side of a rule. In this case it is better to build the automaton of Figure 14 explicitly and make it deterministic.

Wu, Manber and Myers 1992 It was not until 1992 that Wu, Manber and Myers looked again into this problem [WMM96]. The idea was to trade time for space using a Four Russians technique [ADKF75]. Since the cells could be expressed using only values in $\{-1, 0, 1\}$, the columns were partitioned into blocks of r cells (called "regions") which took $2r$ bits each. Instead of precomputing the transitions from a whole column to the next, the transitions from a region to the next region in the column were precomputed, although the current region could now depend on three previous regions (see Figure 15). Since the regions were smaller than the columns, much less space was necessary. The total amount of work was $O(m/r)$ per column in the worst case, and $O(k/r)$ on average. The space requirement was exponential in r. By using $O(n)$ extra space, the algorithm was $O(kn/\log n)$ on average and $O(mn/\log n)$ in the worst case. Notice that this shares with [MP80] the Four Russians approach, but there is an important difference: the states in this case do not depend on the letters of the pattern and text. The states of the "automaton" of [MP80], on the other hand, depend on the text and pattern.

This Four Russians approach is so flexible that this work was extended to handle regular expressions allowing errors [WMM95]. The technique for exact regular expression searching is to pack portions of the deterministic automaton in bits and compute transition tables for each portion. The few transitions among portions are left nondeterministic and simulated one by one. To allow errors, each state is not anymore active or inactive, but they keep count of the minimum number of errors that makes it active, in $O(\log k)$ bits.

Melichar 1995 In 1995, Melichar [Mel96] studied again the size of the deterministic automaton. By considering the properties of the NFA of Figure 14, he refined the bound of [Ukk85b] to $O(\min(3^m, m(2mt)^k, (k+2)^{m-k}(k+1)!)$, where $t = \min(m+1, \sigma)$. The space complexity and

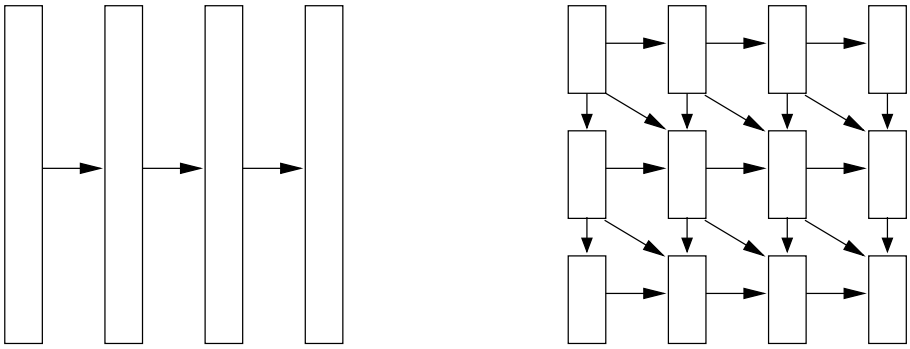

Figure 15: On the left, the automaton of [Ukk85b] where each column is a state. On the right, the automaton of [WMM96] where each region is a state. Both compute the columns of the dynamic programming matrix.

preprocessing time of the automaton is t times the number of states. Melichar also conjectured that this automaton is bigger when there are periodicities in the pattern, which matches with the results of [CH98] (Section 5.2), in the sense that periodic patterns are more problematic. This is in fact a property shared with other problems in string matching.

Kurtz 1996 In 1996, Kurtz [Kur96] proposed another way to reduce the space requirements to at most $O(mn)$. It is an adaptation of [BYG94], which first proposed it for the Hamming distance. The idea was to build the automaton in lazy form, i.e. build only the states and transitions actually reached in the processing of the text. The automaton starts as just one initial state and the states and transitions are built as needed. By doing this, all those transitions that Ukkonen [Ukk85b] considered that were not necessary were not built in fact, without need to guess. The price was the extra overhead of a lazy construction versus a direct construction, but the idea pays off. Kurtz also proposed to have built only the initial part of the automaton (which should be the most commonly traversed states) to save space.

Navarro studied in [Nav97b, Nav98a] the growth of the complete and lazy automata as a function of m, k and n (this last value for the lazy automaton only). The empirical results show that the lazy automaton grows with the text at a rate of $O(n^\beta)$, for $0 < \beta < 1$ dependent on σ, m and k. Some replacement policies designed to work with bounded memory are proposed in [Nav98a].

7 Bit-Parallelism

These algorithms are based on exploiting the parallelism of the computer when it works on bits. This is also a new (after 1990) and very active area. The basic idea is to "parallelize" another algorithm using bits. The results are interesting from the practical point of view, and are especially significative when short patterns are involved (typical in text retrieval). They may work effectively for any error level.

In this section we find elements which strictly could belong to other sections, since we parallelize other algorithms. There are two main trends: parallelize the work of the non-deterministic

automaton that solves the problem (Figure 14), and parallelize the work of the dynamic programming matrix.

We first explain the technique and then the results achieved using it. Figure 16 shows the historical development of this area.

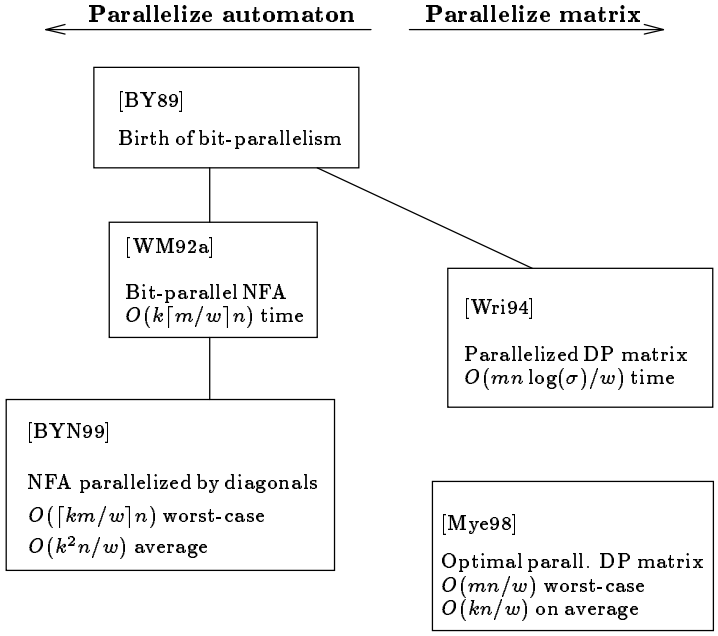

Figure 16: Taxonomy of bit-parallel algorithms.

7.1 The Technique of Bit-Parallelism

This technique, of common use in string matching [BY91, BY92], was born in the PhD. Thesis of Baeza-Yates [BY89]. It consists in taking advantage of the intrinsic parallelism of the bit operations inside a computer word. By using cleverly this fact, the number of operations that an algorithm performs can be cut down by a factor of at most w, where w is the number of bits in the computer word. Since in current architectures w is 32 or 64, the speedup is very significative in practice and improves with technological progress. In order to relate the behavior of bit-parallel algorithms to other works, it is normally assumed that $w = \Theta(\log n)$, as dictated by the RAM model of computation. We prefer, however, to keep w as an independent value. We introduce now some notation we use for bit-parallel algorithms.

- The length of the computer word (in bits) is w.
- We denote as $b_\ell ... b_1$ the bits of a mask of length ℓ. This mask is stored somewhere inside the computer word. Since the length w of the computer word is fixed, we are hiding the details on where we store the ℓ bits inside it.
- We use exponentiation to denote bit repetition (e.g. $0^3 1 = 0001$).

– We use C-like syntax for operations on the bits of computer words: "|" is the bitwise-or, "&" is the bitwise-and, "$\hat{}$" is the bitwise-xor and "$\sim$" complements all the bits. The shift-left operation, "<<", moves the bits to the left and enters zeros from the right, i.e. $b_m b_{m-1}...b_2 b_1 << r = b_{m-r}...b_2 b_1 0^r$. The shift-right, ">>" moves the bits in the other direction. Finally, we can perform arithmetic operations on the bits, such as addition and subtraction, which operates the bits as if they formed a number. For instance, $b_\ell...b_x 10000 - 1 = b_\ell...b_x 01111$.

We explain now the first bit-parallel algorithm, Shift-Or [BYG92], since it is the basis of much of which follows. The algorithm searches a pattern in a text (without errors) by parallelizing the operation of a non-deterministic finite automaton that looks for the pattern. Figure 17 illustrates this automaton.

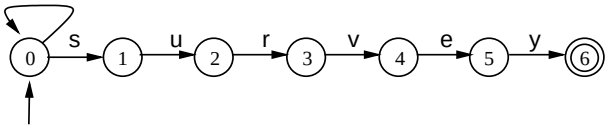

Figure 17: Nondeterministic automaton that searches "survey" exactly.

This automaton has $m+1$ states, and can be simulated in its non-deterministic form in $O(mn)$ time. The Shift-Or algorithm achieves $O(mn/w)$ worst-case time (i.e. optimal speedup). Notice that if we convert the non-deterministic automaton to a deterministic one to have $O(n)$ search time, we get an improved version of the KMP algorithm [KMP77]. However KMP is twice as slow for $m \leq w$.

The algorithm first builds a table B which for each character c stores a bit mask $B[c] = b_m...b_1$. The mask in $B[c]$ has the bit b_i in one if and only if $P_i = c$. The state of the search is kept in a machine word $D = d_m...d_1$, where d_i is one whenever $P_{1..i}$ matches the end of the text read up to now (i.e. the state numbered i in Figure 17 is active). Therefore, a match is reported whenever $d_m = 1$.

D is set to 1^m originally, and for each new text character T_j, D is updated using the formula[4]

$$D' \leftarrow ((D >> 1) \mid 10^{m-1}) \ \& \ B[T_j]$$

The formula is correct because the i-th bit is set if and only if the $(i-1)$-th bit was set for the previous text character and the new text character matches the pattern at position i. In other words, $T_{j-i+1..j} = P_{1..i}$ if and only if $T_{j-i+1..j-1} = P_{1..i-1}$ and $T_j = P_i$. It is possible to relate this formula to the movement that occurs in the non-deterministic automaton for each new text character: each state gets the value of the previous state, but this happens only if the text character matches the corresponding arrow.

For patterns longer than the computer word (i.e. $m > w$), the algorithm uses $\lceil m/w \rceil$ computer words for the simulation (not all them are active all the time). The algorithm is $O(n)$ on average.

[4] The real algorithm uses the bits with the inverse meaning and therefore the operation "$\mid 10^{m-1}$" is not necessary. It also shifts in the other direction to ensure that a fresh zero fills the hole left by the shift, which is more machine dependent for the right shift. We have preferred to explain this more didactic version.

It is easy to extend Shift-Or to handle *classes of characters*. In this extension, each position in the pattern matches with a set of characters rather than with a single character. The classical string matching algorithms are not so easily extended. In Shift-Or, it is enough to set the i-th bit of $B[c]$ for every $c \in P_i$ (P_i is a set now). For instance, to search for "survey" in case-insensitive form, we just set to 1 the first bit of B["s"] and of B["S"], and the same with the rest. Shift-Or can also search for multiple patterns (where the complexity is $O(mn/w)$ if we consider that m is the total length of all the patterns), and it was later enhanced [WM92a] to support a larger set of extended patterns and even regular expressions.

Many online text algorithms can be seen as implementations of an automaton (classically, in its deterministic form). Bit-parallelism has since its invention became a general way to simulate simple non-deterministic automata instead of converting them to deterministic. It has the advantage of being much simpler, in many cases faster (since it makes better usage of the registers of the computer word), and easier to extend to handle complex patterns than its classical counterparts. Its main disadvantage is the limitation it imposes with regard to the size of the computer word. In many cases its adaptations to cope with longer patterns are not so efficient.

7.2 Parallelizing Non-deterministic Automata

Wu and Manber 1992 In 1992, Wu and Manber [WM92a] published a number of ideas that had a great impact in the future of practical text searching. They first extended the Shift-Or algorithm to handle *wild cards* (i.e. allow an arbitrary number of characters between two given positions in the pattern), and regular expressions (the most flexible pattern that can be efficiently searched). What is of more interest to us is that they presented a simple scheme to combine any of the preceding extensions with approximate string matching.

The idea is to simulate the NFA of Figure 14 using bit-parallelism, so that each row i of the automaton fits in a computer word R_i (each state is represented by a bit). For each new text character, all the transitions of the automaton are simulated using bit operations among the $k+1$ computer words. Notice that all the $k+1$ computer words have the same structure (i.e. the same bit is aligned to the same text position). The update formula to obtain the new R'_i values at text position j from the current R_i values is

$$\begin{aligned} R'_0 &= ((R_0 >> 1) \mid 10^{m-1}) \;\&\; B[T_j] \\ R'_{i+1} &= ((R_{i+1} >> 1) \;\&\; B[T_j]) \;\mid\; R_i \;\mid\; (R_i >> 1) \;\mid\; (R'_i >> 1) \end{aligned}$$

and we start the search with $R_i = 1^i 0^{m-i}$. As expected, R_0 undergoes a simple Shift-Or process, while the other rows receive ones (i.e. active states) from previous rows as well. In the formula for R'_{i+1} are expressed, in that order, horizontal, vertical, diagonal and dashed diagonal arrows.

The cost of this simulation is $O(k\lceil m/w \rceil n)$ in the worst and average case, which is $O(kn)$ for patterns typical in text searching (i.e. $m \leq w$). This is a perfect speedup over the serial simulation of the automaton, which would cost $O(mkn)$ time. Notice that for short patterns, this is competitive to the best worst-case algorithms.

Thanks to the simplicity of the construction, the rows of the pattern can be changed by a different automaton. As long as one is able to solve a problem for exact string matching, one makes $k+1$ copies of the resulting computer word, performs the same operations in the $k+1$ words (plus

the arrows that connect the words) and one has an algorithm to find the same pattern allowing errors. Hence, they are able to perform approximate string matching with sets of characters, wild cards, and regular expressions. They also allow some extensions unique of approximate searching: a part of the pattern can be searched with errors and another may be forced to match exactly, and different integer costs of the edit operations can be accommodated (including not allowing some of them). Finally, they are able to search a set of patterns at the same time, but this capability is very limited (since all the patterns must fit in a computer word).

The great flexibility obtained encouraged the authors to build a software called *Agrep* [WM92b][5], where all these capabilities are implemented (although some particular cases are solved in a different manner). This software has been taken as a reference in all the subsequent research.

Baeza-Yates and Navarro 1996 In 1996, Baeza-Yates and Navarro presented a new bit-parallel algorithm able to parallelize the computation of the automaton even more [BYN99]. The classical dynamic programming algorithm can be thought of as a column-wise "parallelization" of the automaton [BY96], and Wu and Manber [WM92a] proposed a row-wise parallelization. Neither algorithm was able to increase the parallelism (even if all the NFA states fit in a computer word) because of the ε-transitions of the automaton, which caused what we call *zero-time* dependencies. That is, the current values of two rows or two columns depend on each other, and hence cannot be computed in parallel.

In [BYN99] the bit-parallel formula for a *diagonal* parallelization was found. They packed the states of the automaton along diagonals instead of rows or columns, which run in the same direction of the diagonal arrows (notice that this is totally different from the diagonals of the dynamic programming matrix). This idea had been mentioned much earlier by Baeza-Yates [BY91] but no bit-parallel formula was found. There are $m-k+1$ complete diagonals (the others are not really necessary) which are numbered from 0 to $m-k$. The number D_i is the row of the first active state in diagonal i (all the subsequent states in the diagonal are active because of the ε-transitions). The new D'_i values after reading text position j are computed as

$$D'_i \;=\; \min(\; D_i+1,\;\; D_{i+1}+1,\;\; g(D_{i-1},T_j)\;)$$

where the first term represents the insertions, the second term the replacements and the last term the matches (deletions are implicit since we represent only the lowest active state of each diagonal). The main problem is how to compute the function g, defined as

$$g(D_i,T_j) \;=\; \min(\;\{k+1\}\;\cup\;\{\;r\;/\;r\geq D_i\;\wedge\;P_{i+r}=T_j\;\}\;)$$

Notice that an active state that crosses a horizontal edge has to propagate all the way down by the diagonal. This was finally solved in 1996 [BYN99, Nav98a] by representing the D_i values in unary and using arithmetic operations on the bits, which have the desired propagation effects. The formula can be understood either numerically (operating the D_i's) or logically (simulating the arrows of the automaton).

The resulting algorithm is $O(n)$ worst case time and very fast in practice if all the bits of the automaton fit in the computer word (while [WM92a] keeps $O(kn)$). In general, it is $O(\lceil k(m-$

[5] Available at `ftp.cs.arizona.edu`.

$k)/w\rceil n)$ worst case time, and $O(\lceil k^2/w\rceil n)$ on average since the Ukkonen cut-off heuristic is used (see Section 5.3). The scheme can handle classes of characters, wild cards and different integral costs in the edit operations.

7.3 Parallelizing the Dynamic Programming Matrix

Wright 1994 In 1994, Wright [Wri94] presented the first work using bit-parallelism on the dynamic programming matrix. The idea was to consider *secondary diagonals* (i.e. those that run from the upper-right to the bottom-left) of the matrix. The main observation is that the elements of the matrix follow the recurrence[6]

$$\begin{array}{rcl} C_{i,j} & = & C_{i-1,j-1} \text{ if } P_i = T_j \text{ or } C_{i-1,j} = C_{i-1,j-1} - 1 \text{ or } C_{i,j-1} = C_{i-1,j-1} - 1 \\ & & C_{i-1,j-1} + 1 \text{ otherwise} \end{array}$$

which shows that the new secondary diagonal can be computed using the two previous ones. The algorithm stores the differences between $C_{i,j}$ and $C_{i-1,j-1}$ and represents the recurrence using modulo 4 arithmetic. The algorithm packs many pattern and text characters in a computer word and performs in parallel a number of pattern versus text comparisons, then using the vector of the results of the comparisons to update many cells of the diagonal in parallel. Since it has to store characters of the alphabet in the bits, the algorithm is $O(nm\log(\sigma)/w)$ in the worst and average case. This was competitive at that time for very small alphabets (e.g. DNA). As the author recognizes, it seems quite difficult to adapt this algorithm for other distance functions.

Myers 1998 In 1998, Myers [Mye98] found a better way to parallelize the computation of the dynamic programming matrix. He represented the differences along columns instead of the columns themselves, so that two bits per cell were enough (in fact this algorithm can be seen as the bit-parallel implementation of the automaton which is made deterministic in [WMM96], see Section 6.2). A new recurrence is found where the cells of the dynamic programming matrix are expressed using horizontal and vertical differences, i.e. $\Delta v_{i,j} = C_{i,j} - C_{i-1,j}$ and $\Delta h_{i,j} = C_{i,j} - C_{i,j-1}$:

$$\begin{array}{rcl} \Delta v_{i,j} & = & \min(-Eq_{i,j}, \Delta v_{i,j-1}, \Delta h_{i-1,j}) \; + \; (1 - \Delta h_{i-1,j}) \\ \Delta h_{i,j} & = & \min(-Eq_{i,j}, \Delta v_{i,j-1}, \Delta h_{i-1,j}) \; + \; (1 - \Delta v_{i,j-1}) \end{array}$$

where $Eq_{i,j}$ is 1 if $P_i = T_j$ and zero otherwise. The idea is to keep packed binary vectors representing the current (i.e. j-th) values of the differences, and finding the way to update the vectors in a single operation. Each cell $C_{i,j}$ is seen as a small processor that receives inputs $\Delta v_{i,j-1}$, $\Delta h_{i-1,j}$ and $Eq_{i,j}$ and produces outputs $\Delta v_{i,j}$ and $\Delta h_{i,j}$. There are $3 \times 3 \times 2 = 18$ possible inputs and a simple formula is found to express the cell logic (unlike [Wri94], the approach is logical rather than arithmetical). The hard part is to parallelize the work along the column, because of the zero-time dependency problem. The author finds a solution which, despite that a very different model is used, is very similar to that of [BYN99].

[6] The original one in [Wri94] has errors.

The result is an algorithm that uses better the bits of the computer word, with a worst case of $O(\lceil m/w \rceil n)$ and an average case of $O(\lceil k/w \rceil n)$ since it uses the Ukkonen cut-off (Section 5.3). The update formula is a little more complex than that of [BYN99] and hence the algorithm is a bit slower, but it adapts better to longer patterns because less computer words are needed.

As it is difficult to improve over $O(kn)$ algorithms, this algorithm may be the last word with respect to asymptotic efficiency of parallelization, except for the possibility to parallelize an $O(kn)$ *worst case* algorithm. As it is now common to expect in bit-parallel algorithms, this scheme is able to search some extended patterns as well, but it seems difficult to adapt it to other distance functions.

8 Filtering Algorithms

Our last category is quite young, starting in 1990 and still very active. It is formed by algorithms that filter the text, quickly discarding text areas which cannot match. Filtering algorithms address only the average case, and their major interest is the potential for algorithms that do not inspect all text characters. The major theoretical achievement is an algorithm with average cost $O(n(k + \log_\sigma m)/m)$, which has been proven optimal. In practice, filtering algorithms are the fastest too. All of them, however, are limited in their applicability by the error level α. Moreover, they need a non-filter algorithm to check the potential matches.

We first explain the general concept and then consider the developments that have occurred in this area. See Figure 18.

8.1 The Concept of Filtering

Filtering is based on the fact that it may be much easier to tell that a text position does not match than to tell that it matches. For instance, if neither `"sur"` nor `"vey"` appear in a text area, then `"survey"` cannot be found there with one error under the edit distance. This is because a single edit operation cannot alter both halves of the pattern.

Most filtering algorithms take advantage of this fact by searching pieces of the pattern without errors. Since the exact searching algorithms can be much faster than approximate searching ones, filtering algorithms can be very competitive (in fact, they dominate on a large range of parameters).

It is important to notice that a filtering algorithm is normally unable to discover the matching text positions by itself. Rather, it is used to discard (hopefully large) areas of the text which cannot contain a match. For instance, in our example, it is necessary that either `"sur"` or `"vey"` appears in an approximate occurrence, but it is not sufficient. Any filtering algorithm must be coupled with a process that verifies all those text positions that could not be discarded by the filter.

Virtually any non-filtering algorithm can be used for this verification, and in many cases the developers of a filtering algorithm do not care in looking for the best verification algorithm, but they just use the dynamic programming algorithm. That selection is normally independent, but the verification algorithm must behave well on short texts because it can be started at many different text positions to work on small text areas. By careful programming it is almost always possible to keep the worst-case behavior of the verifying algorithm (i.e. avoid verifying overlapping areas).

Finally, the performance of filtering algorithms is very sensitive to the error level α. Most filters work very well on low error levels and very bad otherwise. This is related to the amount of text

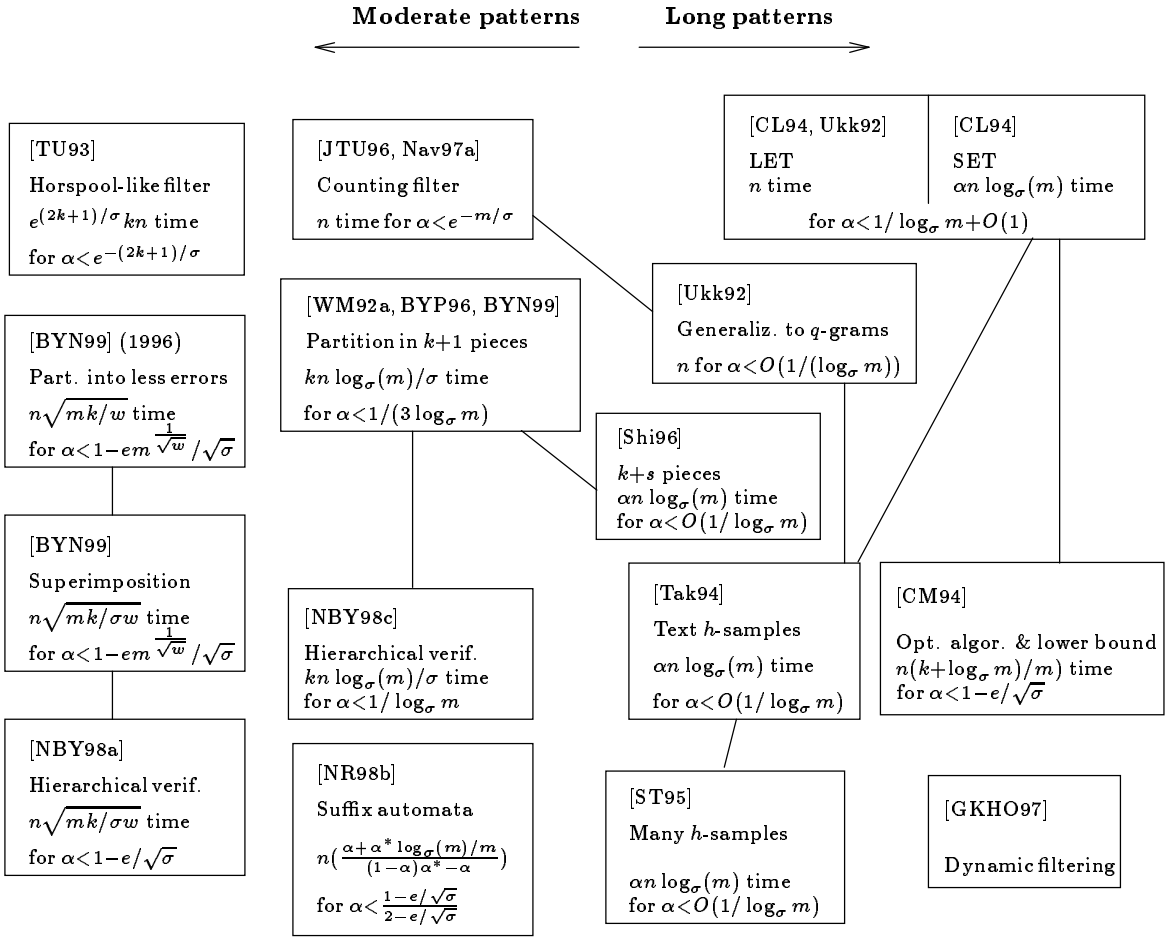

Figure 18: Taxonomy of filtering algorithms. Complexities are all on average.

that the filter is able to discard. When evaluating filtering algorithms, it is important not only to consider their time efficiency but also their tolerance to errors.

A term normally used when referring to filters is "sublinearity". It is said that a filter is sublinear when it does not inspect all the characters of the text (like the Boyer-Moore [BM77] algorithms for exact searching, which can be at best $O(n/m)$). However, no online algorithm can be truly sublinear, i.e. $o(n)$, if m is independent of n. This is only achievable with indexing algorithms.

We divide this area in two parts: moderate and very long patterns. The algorithms for the two areas are normally different, since more complex filters are worthwhile only for longer patterns.

8.2 Moderate Patterns

Tarhio and Ukkonen 1990 Tarhio and Ukkonen [TU93][7] launched this area in 1990, publishing an algorithm that used Boyer-Moore-Horspool techniques [BM77, Hor80] to filter the text. The idea is to align the pattern with a text window and scan the text backwards. The scanning ends where more than k "bad" text characters are found. A "bad" character is one that not only does not match the pattern position it is aligned with, but it also does not match any pattern character at a distance of k characters or less. More formally, assume that the window starts at text position $j+1$, and therefore T_{j+i} is aligned with P_i. Then T_{j+i} is bad when $Bad(i, T_{j+i})$, where $Bad(i, c)$ has been precomputed as $c \notin \{P_{i-k}, P_{i-k+1}, ..., P_i, ...P_{i+k}\}$.

The idea of the bad characters is that we know for sure that we have to make an error to match them, i.e. they will not match as a byproduct of inserting or deleting other characters. When more than k characters that are errors for sure are found, the current text window can be abandoned and shifted forward. If, on the other hand, the beginning of the window is reached, the area $T_{j+1-k..j+m}$ must be checked with a classical algorithm.

To know how much can we shift the window, the authors show that there is no point in shifting P to a new position j' where none of the $k+1$ text characters that are at the end of the current window $(T_{j+m-k}, ...T_{j+m})$ match the corresponding character of P, i.e. where $T_{j+m-r} \neq P_{m-r-(j'-j)}$. If those differences are fixed with replacements we make $k+1$ errors, and if they can be fixed with less than $k+1$ operations, then it is because we aligned some of the involved pattern and text characters using insertions and deletions. In this case, we would have obtained the same effect aligning the matching characters from start.

So for each pattern position $i \in \{m-k..m\}$ and each text character a that could be aligned to position i (i.e. for all $a \in \Sigma$) the shift to align a in the pattern is precomputed, i.e. $Shift(i, a) = \min_{s>0}\{P_{i-s} = a\}$ (or m if no such s exists). Later, the shift for the window us computed as $\min_{i \in m-k..m} Shift(i, T_{j+i})$. This last minimum is computed together with the backward window traversal.

The analysis in [TU93] shows that the search time is $O(kn(k/\sigma + 1/(m-k))$, without considering verifications. In the Appendix we show that the amount of verification is negligible for $\alpha < e^{-(2k+1)/\sigma}$. The analysis is valid for $m >> \sigma > k$, so we can simplify the search time to $O(k^2 n/\sigma)$. The algorithm is competitive in practice for low error levels. Interestingly, the version $k = 0$ corresponds exactly to Horspool algorithm [Hor80]. Like Horspool, it does not take proper advantage of very long patterns. The algorithm can probably be adapted to other simple distance functions if we define k as the minimum number of errors needed to reject a string.

Jokinen, Tarhio and Ukkonen 1991 In 1991, Jokinen, Tarhio and Ukkonen [JTU96] adapted a previous filter for the k-mismatches problem [GL89]. The filter is based on the simple fact that inside any match with at most k errors there must be at least $m-k$ letters belonging to the pattern. The filter does not care about the order of those letters. This is a simple version of [CL94] (see Section 8.3), with less filtering efficiency but simpler implementation.

The search algorithm slides a window of length m over the text[8] and keeps count of the number of window characters that belong to the pattern. This is easily done with a table that for each

[7] See also [JTU96], which has a correction to the algorithm.

[8] The original version used a variable size window. This simplification is from [Nav97a].

character a stores a counter of a's in the pattern that have not yet been seen in the text window. The counter is incremented when an a enters the window and decremented when it leaves the window. Each time a positive counter is decremented, the window character is considered as belonging to the pattern. When there are $m-k$ such characters, the area is verified with a classical algorithm.

The algorithm was analyzed by Navarro in 1997 [Nav97a] using a model of urns and balls. He shows that the algorithm is $O(n)$ time for $\alpha < e^{-m/\sigma}$. Some possible extensions are studied in [Nav98a].

The resulting algorithm is competitive in practice for short patterns, but it worsens for long ones. It is simple to adapt to other distance functions, just by determining how many characters must match in an approximate occurrence.

Wu and Manber 1992 In 1992, a very simple filter was proposed by Wu and Manber [WM92a] (among many other ideas of that work). The basic idea is in fact very old [Riv76]: if a pattern is cut in $k+1$ pieces, then at least *one* of the pieces must appear unchanged in an approximate occurrence. This is evident, since k errors cannot alter the $k+1$ pieces. The proposal was then to split the pattern in $k+1$ approximately equal length pieces, search the pieces in the text, and check the neighborhood of their matches (of length $m+2k$). They used an extension of Shift-Or [BYG92] to search all the pieces simultaneously in $O(mn/w)$ time. In the same 1992, Baeza-Yates and Perleberg [BYP96] suggested better algorithms for the multipattern search: an Aho-Corasick machine [AC75] to guarantee $O(n)$ search time (excluding verifications), or Commentz-Walter [CW79].

Only in 1996 the improvement was really implemented [BYN99], by adapting the Boyer-Moore-Sunday algorithm [Sun90] to multipattern search (using a trie of patterns and a pessimistic shift table). The resulting algorithm is surprisingly fast in practice for low error levels.

There is no closed expression for the average case cost of this algorithm [BYR90], but we show in the Appendix that a gross approximation is $O(kn\log_\sigma(m)/\sigma)$. Two independent proofs in [BYN99, BYP96] show that the cost of the search dominates for $\alpha < 1/(3\log_\sigma m)$. A simple way to see it is to consider that checking a text area costs $O(m^2)$ and is done when any of the $k+1$ pieces of length $m/(k+1)$ matches, which happens with probability near $k/\sigma^{1/\alpha}$. The result follows from requiring the average verification cost to be $O(1)$.

This filter can be adapted, with some care, to other distance functions. The main issue is to determine how many pieces can an edit operation destroy and how many edit operations can be made before surpassing the error threshold. For example, a transposition can destroy two pieces in one operation, so we would need to split the pattern in $2k+1$ pieces to ensure that one is unaltered. A more clever solution for this case is to leave a hole of one character between each pair of pieces, so the transposition cannot alter both.

Baeza-Yates and Navarro 1996 The bit-parallel algorithms presented in Section 7 [BYN99] were also the basis for novel filtering techniques. As the basic algorithm is limited to short patterns, they split longer patterns in j parts, making them short enough to be searchable with the basic bit-parallel automaton (using one computer word).

The method is based on a more general version of the partition into $k+1$ pieces [Mye94a, BYN99]. For any j, if we cut the pattern in j pieces, then at least one of them appears with $\lfloor k/j \rfloor$ errors in any occurrence of the pattern. This is clear because if each piece needs more than k/j

errors to match, then the complete match needs more than k errors.

Hence, the pattern was split in j pieces (of length m/j) which were searched with k/j errors using the basic algorithm. Each time a piece was found, the neighborhood was verified to check for the complete pattern. Notice that the error level α for the pieces is kept unchanged.

The resulting algorithm is $O(n\sqrt{mk/w})$ on average. Its maximum α value is $1-em^{O(1/\sqrt{w})}/\sqrt{\sigma}$, smaller than $1-e/\sqrt{\sigma}$ and worsening as m grows. This may be surprising since the error level α is the same for the subproblems. The reason is that the verification cost keeps $O(m^2)$ but the matching probability is $O(\gamma^{m/j})$, larger than $O(\gamma^m)$ (see Section 4).

In 1997, the technique was enriched with "superimposition" [BYN99]. The idea is to avoid performing one separate search for each piece of the pattern. A multipattern approximate searching is designed using the ability of bit-parallelism to search for classes of characters. Assume that we want to search `"survey"` and `"secret"`. We search the pattern `"s[ue][rc][vr]e[yt]"`, where $[ab]$ means $\{a,b\}$. In the NFA of Figure 14, the horizontal arrows are traversable by more than one letter. Clearly any match of each of the two patterns is also a match of the superimposed pattern, but not vice-versa (e.g. `"servet"` matches with zero errors). So the filter is weakened but the search is made faster. Superimposition allowed to lower the average search time to $O(n)$ for $\alpha < 1-em^{O(1/\sqrt{w})}\sqrt{m/\sigma\sqrt{w}}$ and to $O(n\sqrt{mk/(\sigma w)})$ for the maximum α of the 1996 version. By using a j value smaller than the necessary to put the automata in single machine words, an intermediate scheme was obtained that softly adapted to higher error levels. The algorithm was $O(kn\log(m)/w)$ for $\alpha < 1-e/\sqrt{\sigma}$.

Navarro and Baeza-Yates 1998 The final twist to the previous scheme was the introduction of "hierarchical verification" in 1998 [NBY98a]. For simplicity assume that the pattern is partitioned in $j=2^r$ pieces, although the technique is general. The pattern is split in two halves, each one to be searched with $\lfloor k/2 \rfloor$ errors. Each half is recursively split in two and so on, until the pattern is short enough to make its NFA fit in a computer word (see Figure 19). The leaves of this tree are the pieces actually searched. When a leaf finds a match, instead of checking the whole pattern as in the previous technique, its parent is checked (in a small area around the piece that matched). If the parent is not found, the verification stops, otherwise it continues with the grandparent until the root (i.e. the whole pattern) is found. This is correct because the partitioning scheme applies to each level of the tree: the grandparent cannot appear if none of its children appear, even if a grandchild appeared.

Figure 19 shows an example. If one searches the pattern `"aaabbbcccddd"` with four errors in the text `"xxxbbxxxxxxx"`, and split the pattern in four pieces to be searched with one error, the piece `"bbb"` will be found in the text. In the original approach, one would verify the complete pattern in the text area, while with the new approach one verifies only its parent `"aaabbb"` and immediately determine that there cannot be a complete match.

An orthogonal hierarchical verification technique is also presented in [NBY98a] to include superimposition in this scheme. If the superimposition of 4 patterns matches, the set is split in two sets of two patterns each, and it is checked whether some of them match instead of verifying all the 4 patterns one by one.

The analysis in [Nav98a, NBY98a] shows that the average verification cost drops to $O((m/j)^2)$. Only now the problem scales well (i.e. $O(\gamma^{m/j})$ verification probability and $O((m/j)^2)$ verification

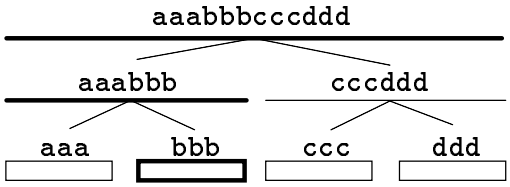

Figure 19: The hierarchical verification method for a pattern split in 4 parts. The boxes (leaves) are the elements which are really searched, and the root represents the whole pattern. At least one pattern at each level must match in any occurrence of the complete pattern. If the bold box is found, all the bold lines may be verified.

cost). With hierarchical verification, the verification cost keeps negligible for $\alpha < 1 - e/\sqrt{\sigma}$. All the simple extensions of bit-parallel algorithms apply, although the partition into j pieces may need some redesign for other distances. Notice that it is very difficult to break the barrier of $\alpha^* = 1 - e/\sqrt{\sigma}$ for any filter because, as shown in Section 4, there are too many *real* matches, and even the best filters must check real matches.

In the same 1998, the same authors [NBY98c, Nav98a] added hierarchical verification to the filter that splits the pattern in $k+1$ pieces and searches them with zero errors. The analysis shows that with this technique the verification cost does not dominate the search time for $\alpha < 1/\log_\sigma m$. The resulting filter is the fastest for most cases of interest.

Navarro and Raffinot 1998 In 1998 Navarro and Raffinot [NR98b, Nav98a] presented a novel approach based on suffix automata (see Section 3.2). They adapted an exact string matching algorithm, BDM, to allow errors.

The idea of the original BDM algorithm is as follows [CCG+94, CR94]. The deterministic suffix automaton of the *reverse* pattern is built, so it recognizes the reverse prefixes of the pattern. Then the pattern is aligned with a text window, and the window is scanned backwards with the automaton (this is why the pattern is reversed). The automaton is active as long as what it has read is a substring of the pattern. Each time the automaton reaches a final state, it has seen a pattern prefix, so we remember the last time it happened. If the automaton arrives with active states to the beginning of the window then the pattern has been found, otherwise what is there is not a substring of the pattern and hence the pattern cannot be in the window. In any case the last window position that matched a pattern prefix gives the next initial window position. The algorithm BNDM [NR98a] is a bit-parallel implementation (using the nondeterministic suffix automaton, see Figure 3) which is much faster in practice and allows searching for classes of characters, etc.

The modification of [NR98b, Nav98a] is to build an NFA to search the reversed pattern allowing errors, modify it to match any pattern suffix, and apply essentially the same algorithm BNDM using this automaton. Figure 20 shows the resulting automaton.

This automaton recognizes any reverse prefix of P allowing k errors. The window will be abandoned when no pattern substring matches with k errors what was read. The window is shifted to the next pattern prefix found with k errors. The matches must start exactly at the initial window position. The window length is $m - k$, not m, to ensure that if there is an occurrence starting at

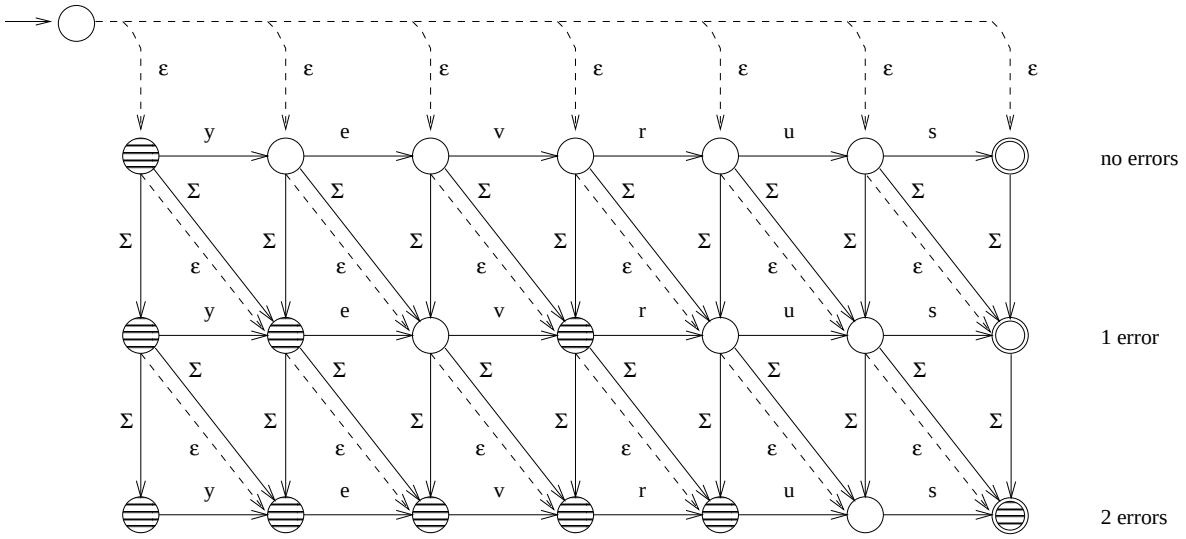

Figure 20: The construction to search any reverse prefix of "survey" allowing 2 errors.

the window position then a substring of the pattern occurs in any suffix of the window (so we do not abandon the window before reaching the occurrence). Reaching the beginning of the window does not guarantee a match, however, so we have to check the area by computing edit distance from the beginning of the window (at most $m + k$ text characters).

In the Appendix it is shown that the average complexity[9] is $O(n(\alpha + \alpha^* \log_\sigma(m)/m)/((1 - \alpha)\alpha^* - \alpha)$ and the filter works well for $\alpha < (1 - e/\sqrt{\sigma})/(2 - e/\sqrt{\sigma})$, which for large alphabets tends to $1/2$. The result is competitive for low error levels, but the pattern cannot be very long because of the bit-parallel implementation. Notice that trying to do this with the deterministic BDM would have generated a very complex construction, while the algorithm is simple with the nondeterministic automaton. Moreover, a deterministic automaton would have too many states, just as in Section 6.2. All the simple extensions of bit-parallelism apply, provided the window length $m - k$ is carefully reconsidered.

8.3 Very Long Patterns

Chang and Lawler 1990 In 1990, Chang and Lawler [CL94] presented two algorithms (better analyzed in [GKHO97]). A first one, called LET (for "linear expected time"), works as follows: the text is traversed linearly, and at each time the longest pattern substring that matches the text is maintained. When the substring cannot be further extended, it starts again from the current text position. Figure 21 illustrates.

The crucial observation is that, if less than $m - k$ text characters have been covered by concatenating k longest substrings, then the text area does not match the pattern. This is evident since a match is formed by $k + 1$ correct strokes (recall Section 5.2) separated by k errors. Moreover, the strokes need to be ordered, which is not required by the filter.

[9] The original analysis of [Nav98a] is inaccurate.

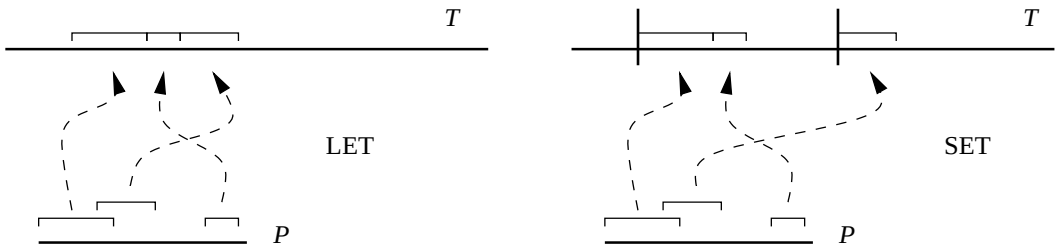

Figure 21: Algorithms LET and SET. LET covers all the text with pattern substrings, while SET works only at block beginnings and stop when it finds k differences.

The algorithm uses a suffix tree on the pattern to determine in a linear pass the longest pattern substring matching the text seen up to now. Notice that the article is from 1990, the same year when Ukkonen and Wood did the same with a suffix automaton [UW93] (see Section 5.2). Therefore, the filtering is $O(n)$ time. The authors use [LV89] as the verifying algorithm and therefore the worst case is $O(kn)$. The authors show that the filtering time dominates for $\alpha < 1/\log_\sigma m + O(1)$. The constants are involved, but practical figures are $\alpha \leq 0.35$ for $\sigma = 64$ or $\alpha \leq 0.15$ for $\sigma = 4$.

The second algorithm presented was called SET (for "sublinear expected time"). The idea is similar to LET, except because the text is split in fixed blocks of size $(m-k)/2$, and the check for k contiguous strokes starts only at block boundaries. Since the shortest match is of length $m-k$, some of these blocks is always contained completely in a match. If one is able to discard the block, no occurrence can contain it. This is also illustrated in Figure 21.

The sublinearity is clear once it is proven that in $O(k \log_\sigma m)$ comparisons a block is discarded on average. Since $2n/(m-k)$ blocks are considered, the average time is $O(\alpha n \log_\sigma(m)/(1-\alpha))$. The maximum α level keeps the same as in LET, so the complexity can be simplified to $O(\alpha n \log_\sigma m)$. Although the proof that limits the comparisons per block is quite involved, it is not hard to see intuitively why it is true: the probability of finding in the pattern a stroke of length ℓ is limited by m/σ^ℓ, and the detailed proof shows that $\ell = \log_\sigma m$ is on average the longest stroke found. This contrast with the result of [Mye86a] (Section 5.3), that shows that k strokes add up $O(k)$ length. The difference is that here we can take the strokes from anywhere in the pattern.

Both LET and SET are effective for very long patterns only, since their overhead does not pay off on short patterns. Different distance functions can be accommodated after re-reasoning the adequate k values.

Ukkonen 1992 In 1992, Ukkonen [Ukk92] independently rediscovered some of the ideas of Chang and Lampe. He presented two filtering algorithms, one of which (based on what he called "maximal matches") is similar to LET of [CL94] (in fact Ukkonen presents it as a new "block distance" computable in linear time and shows that it serves as a filter for the edit distance). The other filter is the first reference to "q-grams" for online searching (there are much older ones in indexed searching [Ull77]).

A q-gram is a substring of length q. A filter was proposed based on counting the number of q-grams shared between the pattern and a text window (this is presented in terms of a new "q-gram distance" which may be of interest by its own). A pattern of length m has $(m-q+1)$ overlapping

q-grams. Each error can alter q q-grams of the pattern, and therefore $(m - q + 1 - kq)$ pattern q-grams must appear in any occurrence. Figure 22 illustrates.

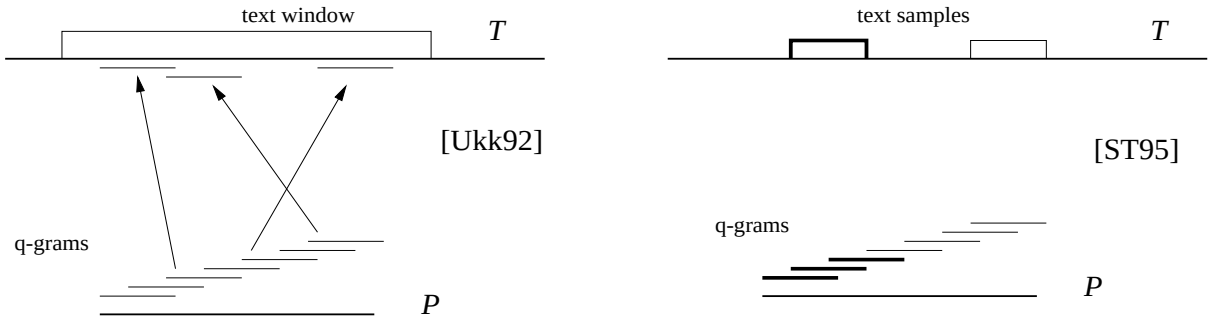

Figure 22: Q-gram algorithm. The left one [Ukk92] counts the number of pattern q-grams in a text window. The right one [ST95] finds sequences of pattern q-grams in approximately the same text positions (we have put in bold a text sample and the possible q-grams to match it).

Notice that this is a generalization of the counting filter of [JTU96] (Section 8.2), which would correspond to $q = 1$. The search algorithm is similar as well, although of course keeping a table with a counter for each of the σ^q q-grams is impractical (especially because only $m - q + 1$ of them are present). Ukkonen uses a suffix tree to keep count in linear time of the last q-gram seen (the relevant information can be attached to the $m - q + 1$ important nodes at depth q in the suffix tree).

The filter takes therefore linear time. There is no analysis to show which is the maximum error level tolerated by the filter, so we attempt a gross analysis in the Appendix, valid for large m. The result is that the filter works well for $\alpha < O(1/\log_\sigma m)$, and that the optimal q to obtain it is $q = \log_\sigma m$. The search algorithm is more complicated than that of [JTU96]. Therefore, using larger q values only pays off for larger patterns. Different distance functions are easily accommodated by recomputing the number of q grams that must be preserved in any occurrence.

Takaoka 1994 In 1994, Takaoka [Tak94] presented a simplification of [CL94]. He considered h-samples of the text (which are non-overlapping q-grams of the text taken each h characters, for $h \geq q$). The idea was that if one h-sample was found in the pattern, then a neighborhood of the area was verified.

By using $h = \lfloor (m - k - q + 1)/(k + 1) \rfloor$ one cannot miss a match. The easiest way to see this is to start with $k = 0$. Clearly we need $h = m - q + 1$ to not loose any matches. For larger k, recall that if the pattern is split in $k + 1$ pieces some of them must appear with no errors. The filter divides h by $k + 1$ to ensure that any occurrence of those pieces will be found (we are assuming $q < m/(k + 1)$).

Using a suffix tree of the pattern the h-sample can be found in $O(q)$ time, and therefore the filtering time is $O(qn/h)$, which is $O(\alpha n \log_\sigma(m)/(1 - \alpha))$ if the optimal $q = \log_\sigma m$ is used. The error level is again $\alpha < O(1/\log_\sigma m)$, which makes the time $O(\alpha n \log_\sigma m)$.

Chang and Marr 1994 It looks like $O(\alpha n \log_\sigma m)$ is the best complexity achievable by using filters, and that it will work only for $\alpha = O(1/\log_\sigma m)$, but in 1994 Chang and Marr obtained an algorithm which was

$$O\left(\frac{k + \log_\sigma m}{m}\, n\right)$$

for $\alpha < \rho_\sigma$, where ρ_σ depends only on σ and it tends to $1 - e/\sqrt{\sigma}$ for very large σ. At the same time, they proved that this was a lower bound for the average complexity of the problem (and therefore their algorithm was optimal on average). This is a major theoretical breakthrough.

The lower bound is obtained by taking the maximum (or sum) of two simple facts: the first one is the $O(n \log_\sigma(m)/m)$ bound of [Yao79] for exact string matching, and the second one is the obvious fact that in order to discard a block of m text characters, at least k characters should be examined to find the k errors (and hence $O(kn/m)$ is a lower bound). Also, the maximum error level is optimal according to Section 4. What is impressive is that an algorithm with such complexity was found.

The algorithm is a variation of SET [CL94]. It is of polynomial space in m, i.e. $O(m^t)$ space for some constant t which depends on σ. It is based on splitting the text in contiguous substrings of length $\ell = t \log_\sigma m$. Instead of finding in the pattern the longest exact matches starting at the beginning of blocks of size $(m - k)/2$, it searches the text substrings of length ℓ in the pattern allowing errors.

The algorithm proceeds as follows. The best matches allowing errors inside P are precomputed for every ℓ-tuple (hence the $O(m^t)$ space). Starting at the beginning of the block, it searches consecutive ℓ-tuples in the pattern (each in $O(\ell)$ time), until the total number of errors made exceeds k. If by that time it has not yet covered $m - k$ text characters, the block can be safely skipped.

The reason why this works is a simple extension of that for SET. We have found an area contained in the possible occurrence which cannot be covered with k errors (even allowing the use of unordered portions of the pattern for the match). The algorithm is only practical for very long patterns, and can be extended for other distances with the same ideas of the other filtration and q-gram methods.

It is interesting to notice that $\alpha \leq 1 - e/\sqrt{\sigma}$ is the limit we have discussed in Section 4, which is a firm barrier for any filtering mechanism. Chang and Lawler proved an asymptotic result, while a general bound is proved in [BYN99]. The filters of [CM94, NBY98a] reduce the problem to *less* errors instead of to *zero* errors. An interesting observation is that it seems that all the filters that partition the problem into exact search can be applied for $\alpha = O(1/\log_\sigma m)$, and that in order to improve this to $1 - e/\sqrt{\sigma}$ we must partition the problem into (smaller) approximate searching subproblems.

Sutinen and Tarhio 1995 Sutinen and Tarhio generalized the Takaoka filter in 1995 [ST95], improving its filtering efficiency. This is the first filter that takes into account the relative positions of the pattern pieces that match in the text (all the previous matched pieces of the pattern in any order). The generalization is to force that s q-grams of the pattern match (not just one). The pieces must conserve their relative ordering in the pattern and must not be more than k characters

away from their correct position (otherwise we need to make more than k errors to use them). This method is also illustrated in Figure 22.

In this case, the sampling step is reduced to $h = \lfloor (m - k - q + 1)/(k + s) \rfloor$. The reason for this reduction is that, to ensure that s pieces of the pattern match, we need to cut the pattern in $k + s$ pieces. To search the pieces forcing that they are not too far away from their correct positions, the pattern is divided in $k + s$ pieces and a hashed set is created for each piece. The set contains the q-grams of the piece and some neighboring ones too (because the sample can be slightly misaligned). At search time, instead of a single h-sample, they consider text windows of contiguous sequences of $k + s$ h-samples. Each of these h-samples are searched in the corresponding set, and if at least s are found the area is verified. This is a sort of Hamming distance, and the authors resort to an efficient algorithm for that distance [BYG92] to process the text.

The resulting algorithm is $O(\alpha n \log_\sigma m)$ on average using optimal $q = \log_\sigma m$, and works well for $\alpha < 1/\log_\sigma m$. The algorithm is better suited for long patterns, although with $s = 2$ it can be reasonably applied to short ones as well. In fact the analysis is done for $s = 2$ only in [ST95].

Shi 1996 In 1996 Shi [Shi96] proposed to extend the idea of the $k + 1$ pieces (explained in Section 8.2) to $k + s$ pieces, so that at least s pieces must match. This idea is implicit in the filter of Sutinen and Tarhio but had not been explicitly written down. Shi compared his filter against the simple one, finding that the filtering efficiency was improved. However, this improvement will be noticeable only for long patterns. Moreover, the online searching efficiency is degraded because the pieces are shorter (which affects any Boyer-Moore-like search), and because the verification logic is more complex. No analysis is presented in the paper, but we conjecture that the optimum s is $O(1)$ and therefore the same complexity and tolerance to errors is maintained.

Giegerich, Kurtz, Hischke and Ohlebusch 1996 Also in 1996, a general method to improve filters was developed [GKHO97]. The idea is to mix the phases of filtering and checking, so that the verification of a text area is abandoned as soon as the combined information from the filter (number of guaranteed differences left) and the verification in progress (number of actual differences seen) shows that a match is not possible. As they show, however, the improvement occurs in a very narrow area of α. This is a consequence of the statistics of this problem that we have discussed in Section 4.

9 Experiments

We perform in this section an empirical comparison among the algorithms described along this work. Our goal is to show which are the best options at hand depending on the case. Nearly 40 algorithms have been surveyed, some of them without existing implementation and many of them already known to be impractical. To avoid an excessively lengthly comparison among algorithms known to be not competitive, we have left aside many of them.

9.1 Included and Excluded Algorithms

A large group of excluded algorithms is from those theoretical ones based on the dynamic programming matrix. We remark that all these algorithm, despite not being competitive in practice, represent (or represented at their time) a valuable contribution to the development of the algorithmic aspect of the problem. The dynamic programming algorithm [Sel80] is excluded because the cut-off heuristic of Ukkonen [Ukk85b] is well known to be faster (e.g. in [CL92] and in our internal tests); the algorithm of [MP80] is argued in the same paper to be worse than dynamic programming (which is quite bad) for $n < 40$ Gb; [LV88] has bad complexity and was improved later by many others in theory and practice; [LV89] is implemented with a better LCA algorithm in [CL92] and found too slow; [Mye86a] is considered slow in practice by the same author in [WMM96]; [GG88] is clearly slower than [LV89]; [GP90], one of the fastest among the $O(kn)$ worst case algorithms, is shown to be extremely slow in [UW93, CL92, Wri94] and in internal tests done by ourselves; [UW93] is shown to be slow in [JTU96]; the $O(kn)$ algorithm implemented in [CL94] is in the same paper argued to be the fast of the group and shown to be not competitive in practice; [SV97, CH98] are clearly theoretical, their complexities show that the patterns have to be very long and the error level too low to be of practical application. To give an idea of how slow is "slow", we found [GP90] 10 times slower than Ukkonen's cut-off heuristic (a similar result is reported by Chang and Lampe [CL92]). Finally, other $O(kn)$ average time algorithms are proposed in [Mye86a, GP90], and they are shown to be very similar to Ukkonen's cut-off [Ukk85b] in [CL92]. Since the cut-off heuristic is already not very competitive we leave aside the other similar algorithms. Therefore, from the group based on dynamic programming we consider only cut-off heuristic (mainly as a reference) and [CL92], which is the only competitive in practice.

From the algorithms based on automata we consider the DFA algorithm [Ukk85b], but prefer its lazy version implemented in [Nav97b], which is equally fast for small automata and much faster for large automata. We also consider the four Russians algorithm of [WMM96]. From the bit-parallel algorithms we consider [WM92a, BYN99, Mye98], leaving aside [Wri94]. As shown in the 1996 version of [BYN99], the algorithm of [Wri94] was competitive only on binary text, and this was shown to not hold anymore in [Mye98].

From the filtering algorithms, we have included [TU93]; the counting filter proposed in [JTU96] (as simplified in [Nav97a]); the algorithm of [NR98b, Nav98a]; and those of [ST95] and [Tak94] (this last one seen as the case $s = 1$ of [ST95], since this implementation worked better). We have also included the filters proposed in [BYN99, NBY98a, Nav98a], preferring to present only the last version which incorporates all the twists of superimposition, hierarchical verification and mixed partitioning. Many previous versions are outperformed by this one. We have also included the best version of the filters that partition the pattern in $k + 1$ pieces, namely the one incorporating hierarchical verification [NBY98c, Nav98a]. In those publications it is shown that this version clearly outperforms the previous ones proposed in [WM92a, BYP96, BYN99]. Finally, we are discarding some filters [CL94, Ukk92, CM94, Shi96] which are applicable only to very long patterns, since this case is excluded from our experiments as explained shortly. Some comparisons among them were carried out by Chang and Lampe [CL92], showing that LET is equivalent to the cut-off algorithm with $k = 20$, and that the time for SET is 2α times that of LET. LET was shown to be the fastest with patterns of hundred letters long and a few errors in [JTU96], but we recall that many modern filters were not included in that comparison.

We list now the algorithms included and the relevant comments on them. All the algorithms implemented by ourselves represent our best coding effort and have been found similar or faster than other implementations found elsewhere. The implementations coming from other authors were checked with the same standards and in some cases their code was improved with better register usage and I/O management.

CTF The cut-off heuristic of [Ukk85b] implemented by ourselves.

CLP The column partitioning algorithm of [CL92], implemented by them. We replaced their I/O by ours, which was faster.

DFA The lazy deterministic automaton of [Nav97b], implemented by ourselves.

RUS The four Russians algorithm of [WMM96], implemented by them. We tried different r values (related to the time/space tradeoff) and found that the best option was always $r = 5$ in our machine.

BPR The NFA bit-parallelized by rows [WM92a], implemented by ourselves and restricted to $m \leq w$. Separate code is used for $k = 1$, 2, 3 and $k > 3$. We could continue writing separate versions but have considered that this was reasonable up to $k = 3$, as at that point the algorithm is not competitive anyway.

BPD The NFA bit-parallelized by diagonals [BYN99], implemented by ourselves. Here we do not include any filtering technique.

BPM The bit-parallel implementation of the dynamic programming matrix [Mye98], implemented by that author.

BMH The adaptation of Horspool to allow errors [TU93], implemented by them. We use their algorithm 2 (which was faster), improve some register usage and replace their I/O by ours, which is faster.

CNT The counting filter of [JTU96], as simplified in [Nav97a] and implemented by ourselves.

EXP Partitioning in $k+1$ pieces plus hierarchical verification [NBY98c, Nav98a], implemented by ourselves.

BPP The bit-parallel algorithms of [BYN99, NBY98a, Nav98a] using pattern partitioning, superimposition and hierarchical verification. The implementation is ours and is a packaged software that can be downloaded from the Web page of the author.

BND The BNDM algorithm adapted to allow errors in [NR98b, Nav98a], implemented by ourselves and restricted to $m \leq w$. Separate code is used for $k = 1$, 2, 3 and $k > 3$. We could continue writing separate versions but have considered that this was reasonable up to $k = 3$.

QG2 The q-gram filter of [ST95], implemented by them and used with $s = 2$ (since $s = 1$ is the algorithm [Tak94], see next item; and $s > 2$ worked well only for very long patterns). The code is restricted to $k \leq w/2 - 3$, and it is also not run when q is found to be 1 since the performance is very poor. We improved register usage and replaced the I/O management by our faster versions.

QG1 The q-gram algorithm of [Tak94], run as the special case $s = 1$ of the previous item. The same restrictions on the code apply.

We made our best effort to uniformize the algorithms. The I/O is the same in all cases: the text is read in chunks of 64 Kb to improve locality (this is the optimum in our machine) and care is taken to not lose or repeat matches in the borders; `open` is used instead of `fopen` because this last one is slower. We also uniformize internal conventions: only a final special character (zero) is used at the end of the buffer to help algorithms recognize it; and only the number of matches found is reported.

We separate in the experiments the filtering and non-filtering algorithms. This is because the filters can in general use any non-filter to check for potential matches, so the best algorithm is formed by a combination of both. All the filtering algorithms in the experiments use the cut-off algorithm [Ukk85b] as their verification engine, except for BPP (whose very essence is to switch smoothly to BPD) and BND (that uses a reverse BPR to search in the window and a forward BPR for the verifications).

9.2 Experimental Setup

Apart from the algorithms to be included and their details, we describe our experimental setup. We measure CPU times and show the results in tenths of seconds per megabyte. Our machine is a Sun UltraSparc-1 of 167 MHz and 64 Mb of main memory, we run Solaris 2.5.1 and the texts are in a local disk of 2 Gb. Our experiments were run on texts of 10 Mb of size and repeated 20 times (with different search patterns). The same patterns are used for all the algorithms.

Considering the applications, we have selected three types of texts.

DNA This file is formed by concatenating the 1.34 Mb DNA chain of `h.influenzae` with itself until obtaining 10 Mb. Lines are cut at 60 characters. The patterns are selected randomly from the text avoiding line breaks if possible. The alphabet size is 4 except for a few exceptions along the file, and the results are similar to a random four-letter text.

Natural language This file is formed by 1.29 Mb of writings of B. Franklin filtered to lower case and separators converted to a space (except line breaks which are respected). This mimics common Information Retrieval scenarios. The text is replicated to obtain 10 Mb and search patterns are randomly selected from the same text at word beginnings. The results on this text are roughly equivalent to a random text over 15 characters.

Speech We obtained speech files from discussions on the U.S. Law from Indiana University, in PCM format with 8 bits per sample. Of course the standard edit distance is of no use here, since it has to take into account the absolute values of the differences between two characters. We simplified the problem to use edit distance: we reduced the range of values to 64 by quantization, therefore considering equal two samples that lie in the same range. We used the first 10 Mb of the resulting file. The results are similar to those on a random text of 50 letters, although the file shows smooth changes from a letter to the next.

We present results using different pattern lengths and error levels, in two flavors: we fix m and show the effect of increasing k, or we fix α and show the effect of increasing m. A given algorithm may not appear at all in a plot when its times are above the y range of interest or its restrictions on m and k do not intersect with the x range of interest. In particular, filters are shown only for $\alpha \leq 1/2$. We remind that in most applications the error levels of interest are low.

9.3 Results

Figure 23 shows the results for short patterns ($m = 10$) and varying k. In non-filtering algorithms BPD is normally the fastest, up to 30% faster than the next one, BPM. The DFA is also quite close in most cases. For $k = 1$, a specialized version of BPR is slightly faster than BPD (recall that for $k > 3$ BPR starts to use a nonspecialized algorithm, hence the jump). An exception to this situation occurs in DNA text, where for $k = 4$ and $k = 5$ BPD shows a nonmonotonic behavior and BPM becomes the fastest. This behavior comes from its $O(k(m-k)n/w)$ complexity[10], which in texts with larger alphabets is not noticeable because the cut-off heuristic keeps the cost unchanged. Indeed, the behavior of BPD would have been totally stable if we chose $m = 9$ instead of $m = 10$, because the problem would fit in a computer word all the time. BPM, on the other hand, handles much longer patterns keeping such stability, although it takes up to 50% mor time than BPD.

With respect to filters, we see that EXP is the fastest for low error levels. The value of "low" increases for larger alphabets. At some point, BPP starts to dominate. BPP adapts smoothly to higher error levels by slowly switching to BPD, so BPP is a good alternative for intermediate error levels, from where EXP ceases to work until it should switch to BPD. However, this range is void on DNA and English text for $m = 10$. Other filters competitive with EXP are BND and BMH. In fact, BND is the fastest for $k = 1$ on DNA, although really no filter works very well in that case. Finally, QG2 does not appear because it only worked for $k = 1$ and it was worse than QG1.

The best choice for short patterns seems to be using EXP while it works and switching to the best bit-parallel algorithm for higher errors. Moreover, the verification algorithm for EXP should be BPR or BPD (which are the fastest where EXP dominates).

Figure 24 shows the case of longer patterns ($m = 30$). Many of the observations are still valid in this case. However, the algorithm BPM shows in this case its advantage over BPD, since still all the problem fits in a computer word for BPM and it does not for BPD. Hence in the left plots the best algorithm is BPM except for low k, where BPR or BPD are better. With respect to filters, EXP or BND are the fastest depending on the alphabet, until a certain error level is reached. At that point BPP becomes the fastest, in some cases still faster than BPM. Notice that for DNA a specialized version of BND for $k = 4$ and even 5 could be the fastest choice.

In Figure 25 we consider the case of fixed $\alpha = 0.1$ and growing m. The results repeat somewhat with regard to non-filtering algorithms: BPR is the best for $k = 1$ (i.e. $m = 10$), then BPD is the best until certain pattern length (which varies from 30 on DNA to 80 on speech) and finally BPM becomes the fastest. Note that for so low error level the number of active columns is quite small, which permits algorithms like BPD and BPM keeping their good behavior for patterns much longer than what they can handle in a single machine word. The DFA is also quite competitive until its memory requirements become unreasonable.

The real change, however, is in the filters. In this case PEX becomes the star filter in English and speech texts, by far unbeaten. The situation on DNA, on the other hand, is quite complex. For $m \leq 30$ BND is the fastest, and indeed an extended implementation of it allowing longer patterns could keep being the fastest for a few more points. However, that case would have to handle four errors, and only a specialized implementation for fixed $k = 4$, 5, etc. could keep a competitive performance. We have determined that such specialized code was worthwhile up to $k = 3$ only. When BND ceases to be applicable, PEX becomes the fastest algorithm and finally QG2 beats it

[10] Another reason for this behavior is that there are integer round-off effects that produce nonmonotonic results.

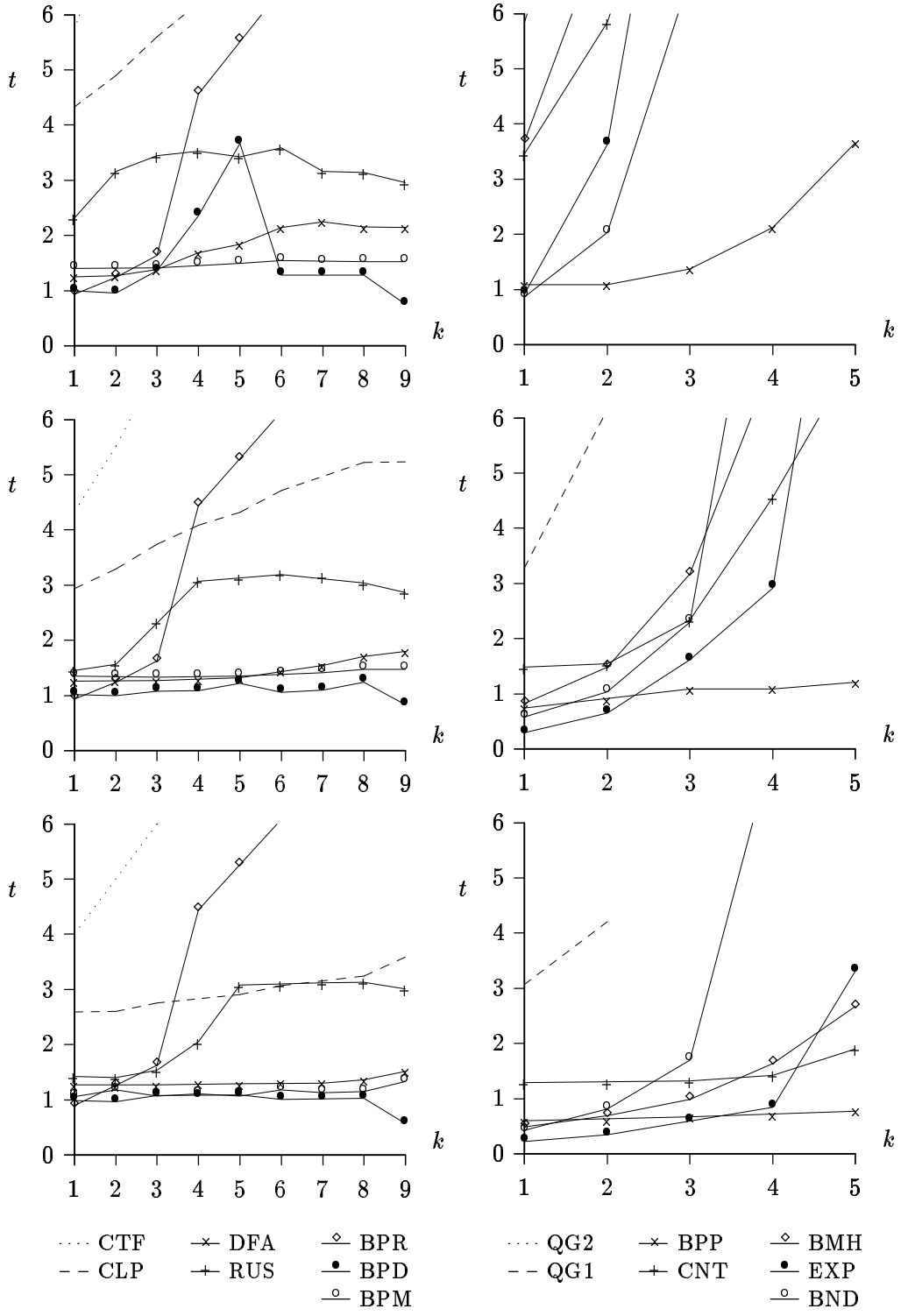

Figure 23: Results for $m = 10$ and varying k. The left plots show non-filtering and the right plots show filtering algorithms. Rows 1 to 3 show DNA, English and speech files, respectively.

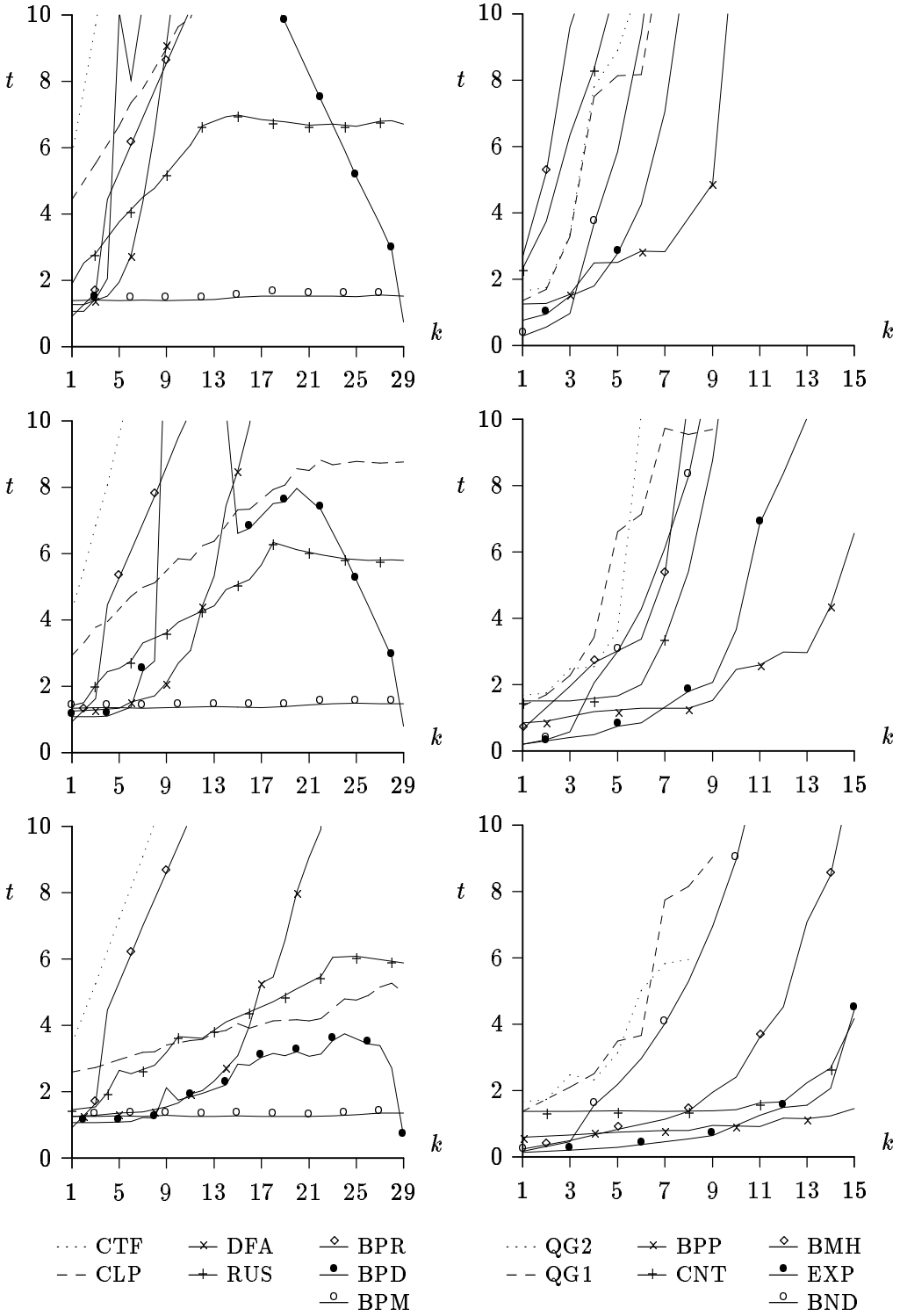

Figure 24: Results for $m = 30$ and varying k. The left plots show non-filtering and the right plots show filtering algorithms. Rows 1 to 3 show DNA, English and speech files, respectively.

(for $m \geq 60$). However, notice that for $m > 30$ all the filters are beaten by BPM and therefore make little sense (on DNA).

There is a final phenomenon that deserves mention with respect to filters. The algorithms QG1 and QG2 improve as m grows. These algorithms are the most practical and the only ones we tested among the family of algorithms suitable for very long patterns. This shows that, despite that all these algorithms would not be competitive in our tests (where $m \leq 100$), they should be considered in scenarios where the patterns are much longer and the error level keeps very low. In such a scenario, those algorithms would finally beat all the algorithms we are considering here.

The situation becomes worse for the filters when we consider $\alpha = 0.3$ and varying m (Figure 26). On DNA, no filter can beat the non-filtering algorithms, and among these the tricks to keep few active columns do not work well. This favors the algorithms that pack more information per bit, which makes BPM the best in all cases except for $m = 10$ (where BPD is better). The situation is almost the same on English text, except because BPP works reasonably well and becomes quite similar to BPM (the periods where each one dominates are interleaved). On speech, on the other hand, the scenario is similar for non-filtering algorithms, but the PEX filter still beats all them, as 30% of errors is low enough on the speech files. Note in passing that the error level is too high for QG1 and QG2, which can only be applied in a short range and yield bad results.

To give an idea of which are the areas where each algorithm dominates, Figure 27 shows the case of English text. There is more information in Figure 27 that what can be inferred from previous plots, such as the area where RUS is better than BPM. We have shown the non-filtering algorithms and superimposed in gray the area where the filters dominate. Therefore, in the grayed area the best choice is to use the corresponding filter using the dominating non-filter as its verification engine. In the non grayed area it is better to use the dominating non-filtering algorithm directly, with no filter.

A code implementing such a heuristic (including only EXP, BPD and BPP) is publicly available from the Web page of the author[11]. This combined code is faster than each isolated algorithm, although of course it is not really a single algorithm but the combination of the best choices.

10 Conclusions

We reach the end of this tour on approximate string matching. Our goal has been to present and explain the main ideas that exist behind the existing algorithms, to classify them according to the type of approach proposed, and to show how they perform in practice in a subset of the possible practical scenarios. We have shown that the oldest approaches, based on the dynamic programming matrix, yielded the most important theoretical developments, but in general the algorithms have been improved by modern developments based on filtering and bit-parallelism. In particular, the fastest algorithms combine a fast filter to discard most of the text with a fastest non-filter algorithm to check the potential matches.

We show some plots summarizing the contents of the survey. Figure 28 shows the historical order in which the algorithms appeared in the different areas. Figure 29 shows a worst case time/space complexity plot for the non-filtering algorithms. Figure 30 considers filtration algorithms, showing

[11] `http://www.dcc.uchile.cl/~gnavarro/pubcode`. To apply EXP the option `-ep` must be used.

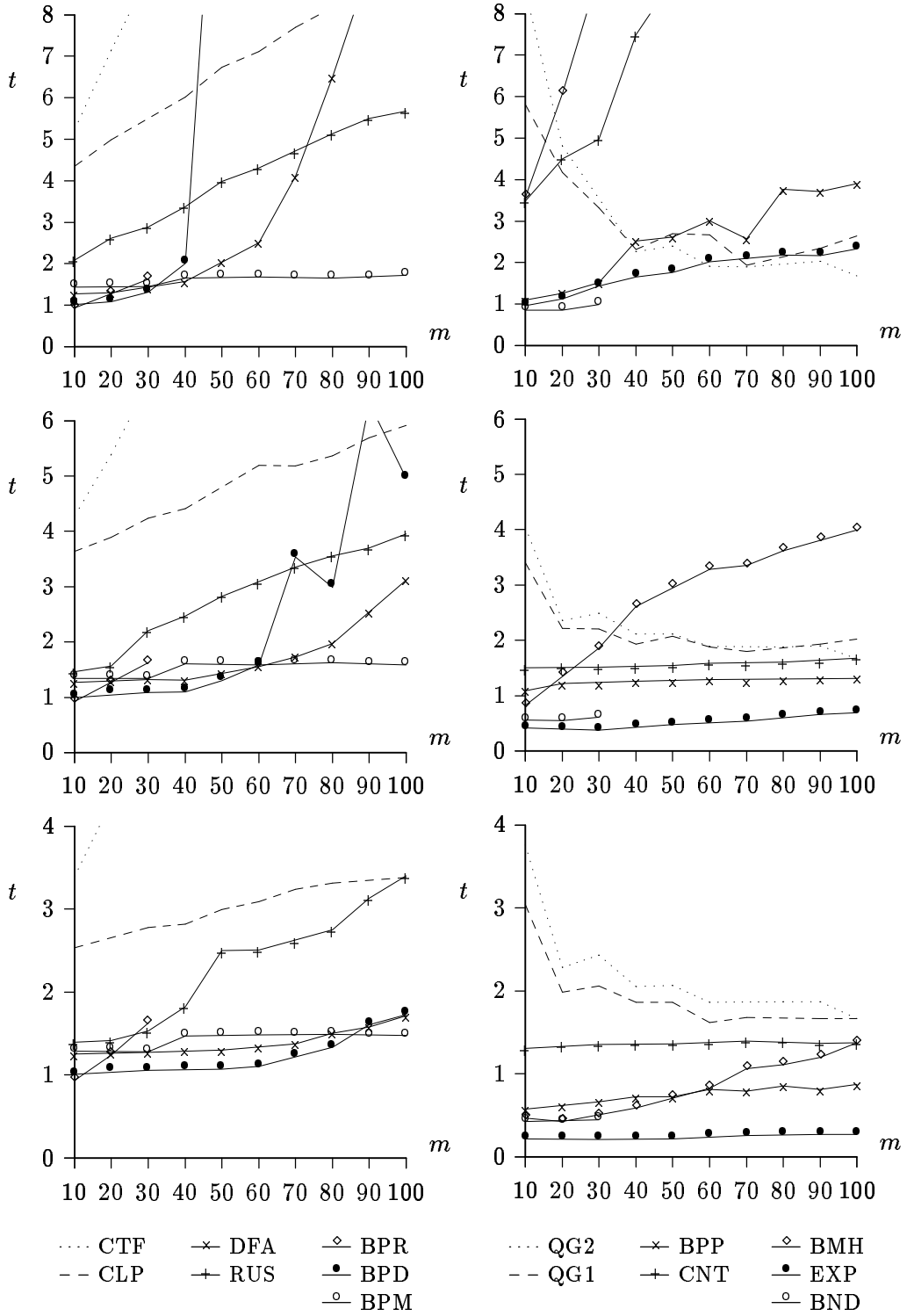

Figure 25: Results for $\alpha = 0.1$ and varying m. The left plots show non-filtering and the right plots show filtering algorithms. Rows 1 to 3 show DNA, English and speech files, respectively.

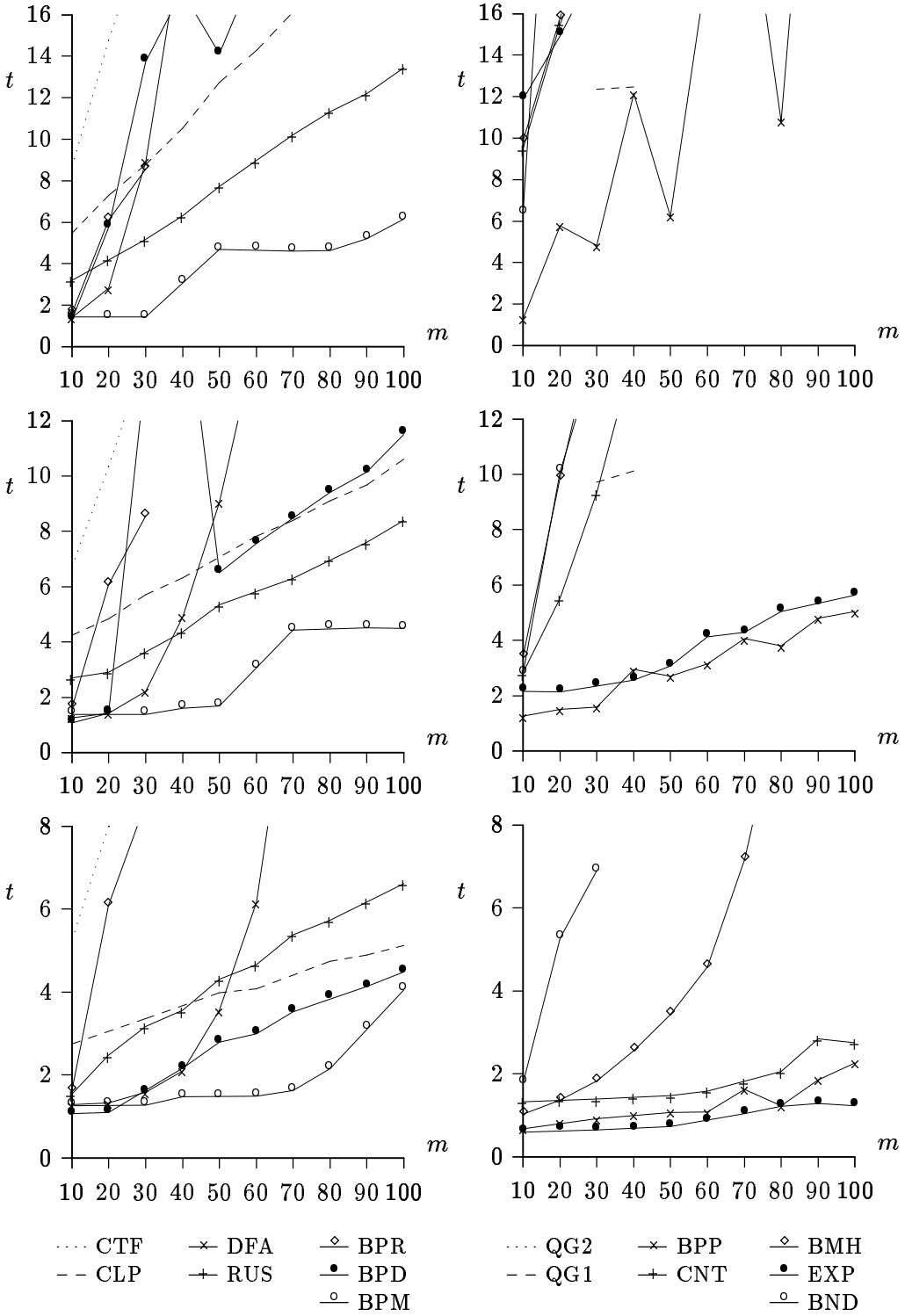

Figure 26: Results for $\alpha = 0.3$ and varying m. The left plots show non-filtering and the right plots show filtering algorithms. Rows 1 to 3 show DNA, English and speech files, respectively.

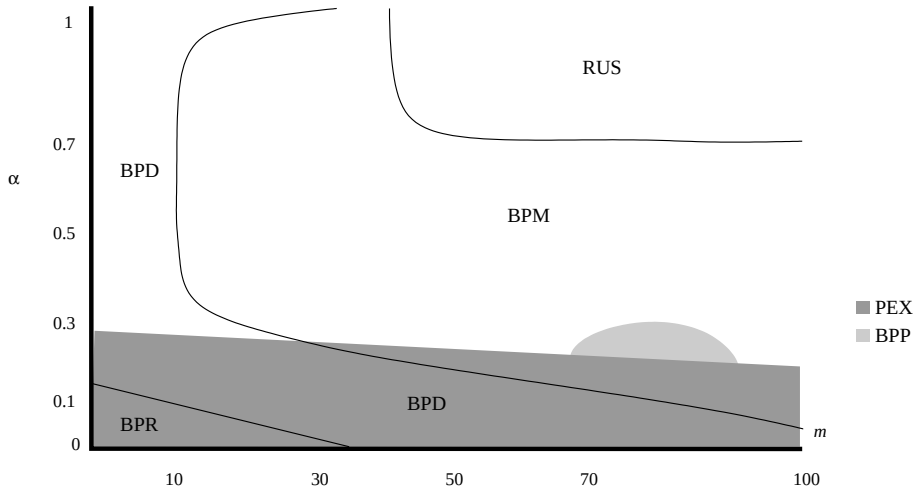

Figure 27: The areas where each algorithm is the best, graying that of filtering algorithms.

their average case complexity and the maximum error level α for which they work. Some practical assumptions have been made to order the different functions of k, m, σ, w and n.

Approximate string matching is a very active research area and it should continue in that status in the foreseeable future: strong genome projects in computational biology, the pressure for oral human-machine communication and the heterogeneity and spelling errors present in textual databases are just a sample of the reasons that are driving researchers to look for faster and more flexible algorithms for approximate pattern matching.

It is interesting to point out which theoretical and practical questions are still open in the area.

- A difficult open question is about the exact matching probability and average edit distance between two random strings. We found a new bound in this survey, but the problem is still open.
- A worst case lower bound of the problem is clearly $O(n)$, but the only algorithms achieving it have space and preprocessing cost exponential in m or k. The only improvements to the worst case with polynomial space complexity are the $O(kn)$ algorithms and, for very small k, $O(n(1+k^4/m))$. Is it possible to improve the algorithms or to find a better lower bound for this case?
- The previous question has also a practical side: is it possible to find an algorithm which is $O(kn)$ in the worst case and efficient in practice? Using bit-parallelism there are good practical algorithms that achieve $O(kn/w)$ on average and $O(mn/w)$ in the worst case.
- The lower bound of the problem for the average case is known to be $O(n(k+\log_\sigma m)/m)$ and there exists an algorithm achieving it, so from the theoretical point of view that problem is closed. However, from the practical side we have that the algorithms approaching those limits work well only for very long patterns, while a much simpler algorithm (EXP) is the best for moderate and short patterns. Is is possible to find a unified approach, good in practice and with that theoretical complexity?

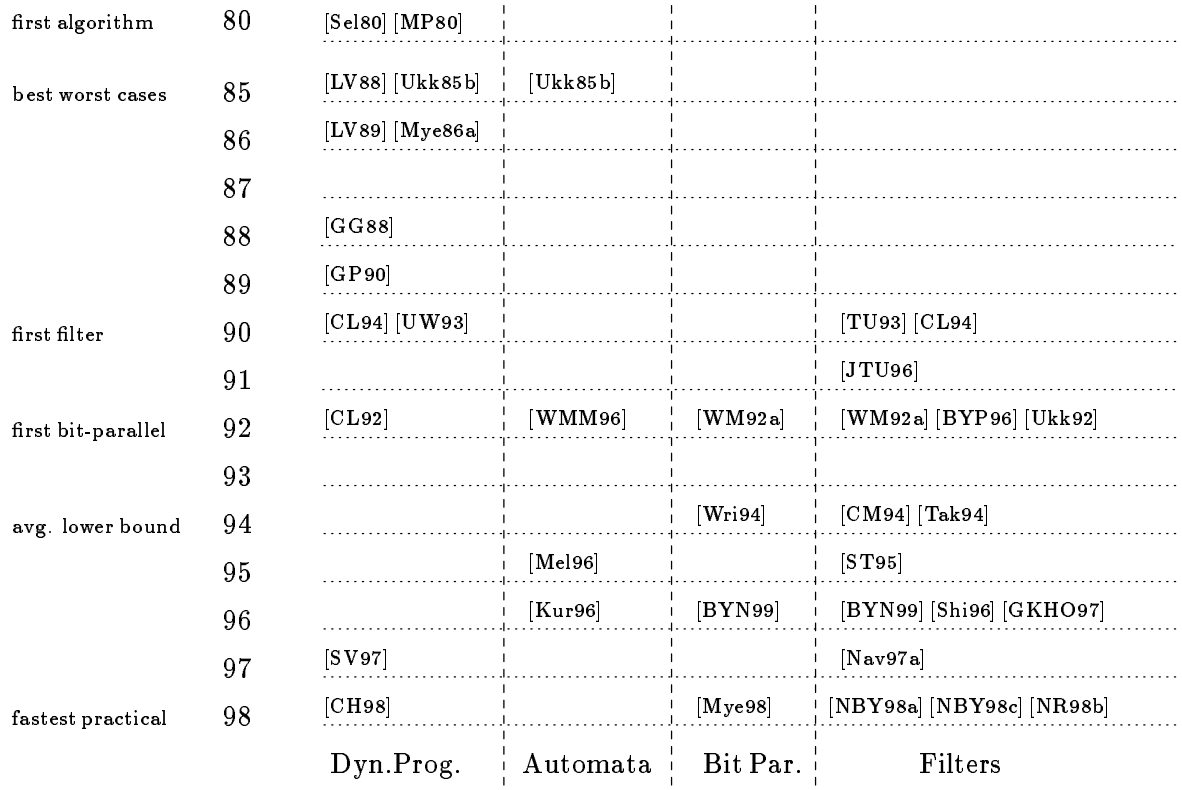

Figure 28: Historical development of the different areas.

- Another practical question on filtering algorithms is: is it possible in practice to improve over the current best existing algorithms?
- Finally, there are many other open questions related to offline approximate searching, which is a much less mature area needing more research.

Acknowledgements

The author wishes to thank the many top researchers in this area for their willingness to exchange ideas and/or share their implementations: Amihood Amir, Ricardo Baeza-Yates, William Chang, Udi Manber, Gene Myers, Erkki Sutinen, Tadao Takaoka, Jorma Tarhio, Esko Ukkonen and Alden Wright.

References

[AAL+97] A. Amir, Y. Aumann, G. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. In *Proc. FOCS'97*, pages 144–153, 1997.

[AC75] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Comm. of the ACM*, 18(6):333–340, 1975.

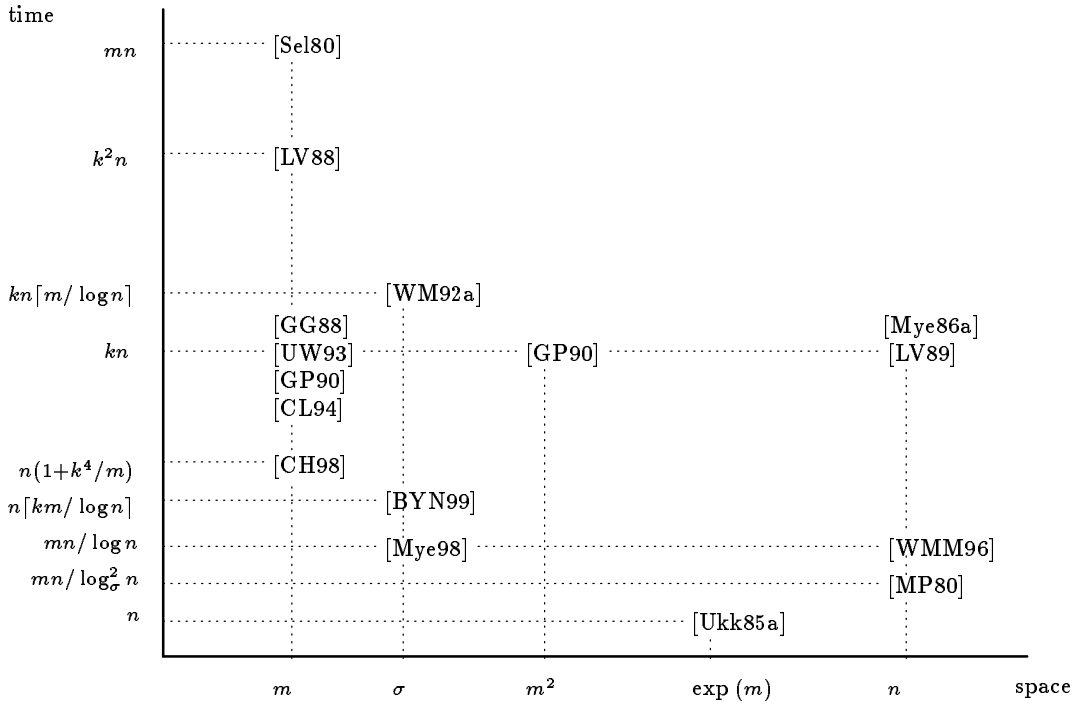

Figure 29: Worst case time and space complexity of non-filtering algorithms. We replaced w by $\Theta(\log n)$.

[ADKF75] V. Arlazarov, E. Dinic, M. Konrod, and I. Faradzev. On economic construction of the transitive closure of a directed graph. *Soviet Mathematics Doklady*, 11:1209–1210, 1975. Original in Russian in *Doklady Akademi Nauk SSSR*, v. 194, 1970.

[AG85] A. Apostolico and Z. Galil. *Combinatorial Algorithms on Words*. NATO ISI Series. Springer-Verlag, 1985.

[AG87] A. Apostolico and C. Guerra. The Longest Common Subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[ALL97] A. Amir, M. Lewenstein, and N. Lewenstein. Pattern matching in hypertext. In *Proc. WADS'97*, LNCS 1272, pages 160–173. Springer-Verlag, 1997.

[ANZ97] M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In *Proc. WSP'97*, pages 2–20. Carleton University Press, 1997.

[Apo85] A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, pages 85–96. Springer-Verlag, 1985.

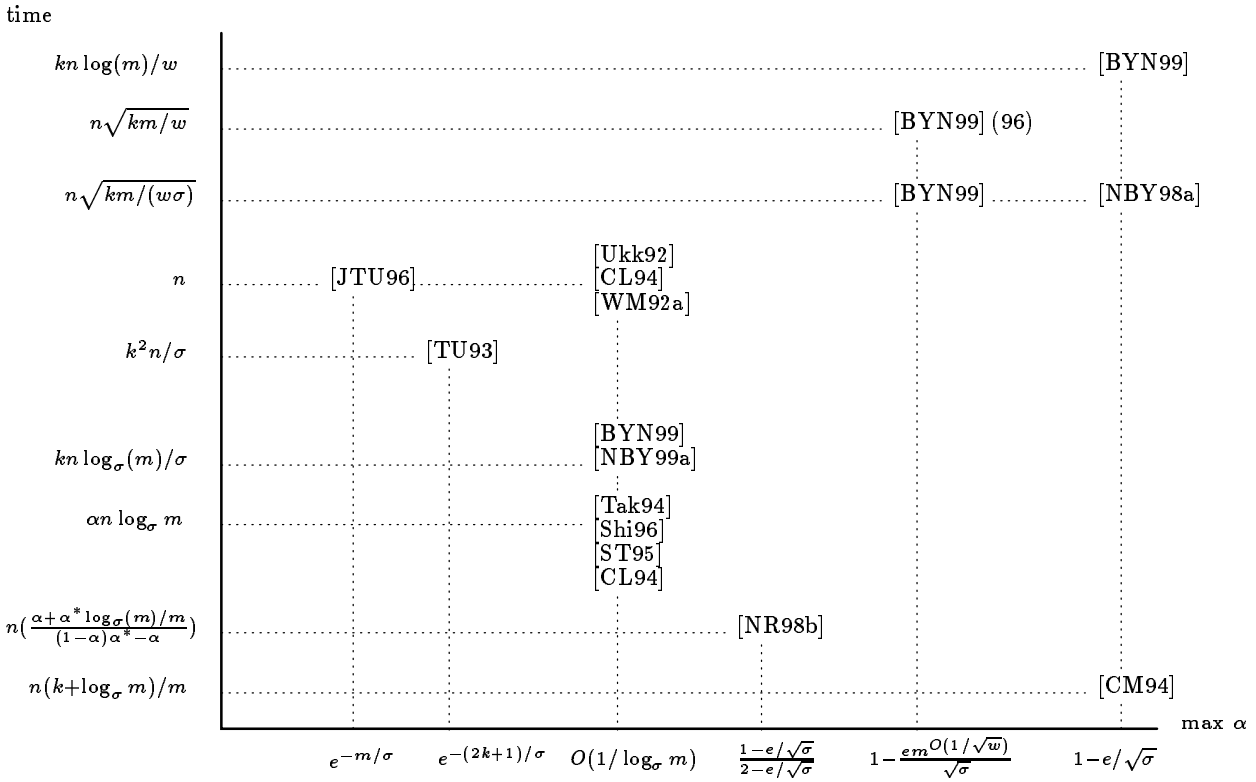

Figure 30: Average time and maximum tolerated error level for the filtration algorithms.

[BBH+85] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen, and J. Seiferas. The samllest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[BM77] R. Boyer and J. Moore. A fast string searching algorithm. *Comm. of the ACM*, 20(10):762–772, 1977.

[BY89] R. Baeza-Yates. *Efficient Text Searching*. PhD thesis, Dept. of Computer Science, Univ. of Waterloo, May 1989. Also as Research Report CS-89-17.

[BY91] R. Baeza-Yates. Some new results on approximate string matching. In *Workshop on Data Structures*, Dagstuhl, Germany, 1991. Abstract.

[BY92] R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, 1992.

[BY96] R. Baeza-Yates. A unified view of string matching algorithms. In *SOFSEM'96: Theory and Practice of Informatics*, LNCS 1175, pages 1–15. Springer-Verlag, 1996.

[BYG92] R. Baeza-Yates and G. Gonnet. A new approach to text searching. *Comm. of the ACM*, 35(10):74–82, 1992. Preliminary version in *ACM SIGIR'89*, 1989.

[BYG94] R. Baeza-Yates and G. Gonnet. Fast string matching with mismatches. *Information and Computation*, 108(2):187–199, 1994. Preliminary version as Tech. Report CS-88-36, Data Structuring Group, Univ. of Waterloo, Sept. 1988.

[BYN97a] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. In *Proc. ACM CIKM'97*, pages 1–8, 1997. Extended version to appear in *JASIS*.

[BYN97b] R. Baeza-Yates and G. Navarro. Multiple approximate string matching. In *Proc. WADS'97*, LNCS 1272, pages 174–184. Springer-Verlag, 1997.

[BYN98] R. Baeza-Yates and G. Navarro. New and faster filters for multiple approximate string matching. Technical Report TR/DCC-98-10, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/-gnavarro/multi.ps.gz`.

[BYN99] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999. Preliminary versions in *Proc. CPM'96*, LNCS 1075, 1996, and in *Proc. WSP'96*, Carleton University Press, 1996.

[BYP96] R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. *Information Processing Letters*, 59:21–27, 1996. Preliminary version in *CPM'92*, LNCS 644, 1992.

[BYR90] R. Baeza-Yates and M. Régnier. Fast algorithms for two dimensional and multiple pattern matching. In *Proc. SWAT'90*, LNCS 447, pages 332–347. Springer-Verlag, 1990.

[BYR99] R. Baeza-Yates and B. Ribeiro, editors. *Modern Information Retrieval.* Addison-Wesley, 1999.

[CCG+94] A. Czumaj, M. Crochemore, L. Gasieniec, S. Jarominek, Thierry Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12:247–267, 1994.

[CH98] R. Cole and R. Hariharan. Approximate string matching: a simpler faster algorithm. In *Proc. ACM-SIAM SODA'98*, pages 463–472, 1998.

[CL92] W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, LNCS 644, pages 172–181. Springer-Verlag, 1992.

[CL94] W. Chang and E. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994. Preliminary version in *FOCS'90*, 1990.

[CLR91] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1991.

[CM94] W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. CPM'94*, LNCS 807, pages 259–273. Springer-Verlag, 1994.

[Cob95] A. Cobbs. Fast approximate matching using suffix trees. In *Proc. CPM'95*, pages 41–54, 1995.

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

[Cro86] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

[CS75] V. Chvátal and D. Sankoff. Longest common subsequences of two random sequences. *Journal of Applied Probability*, 12:306–315, 1975.

[CW79] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. ICALP'79*, LNCS 6, pages 118–132. Springer-Verlag, 1979.

[Dam64] F. Damerau. A technique for computer detection and correction of spelling errors. *Comm. of the ACM*, 7(3):171–176, 1964.

[Dek79] J. Deken. Some limit results for longest common subsequences. *Discrete Mathematics*, 26:17–31, 1979.

[DFG+97] G. Das, R. Fleisher, L. Gasieniek, D. Gunopulos, and J. Kärkäinen. Episode matching. In *Proc. CPM'97*, LNCS 1264, pages 12–27. Springer-Verlag, 1997.

[DM79] R. Dixon and T. Martin, editors. *Automatic speech and speaker recognition.* IEEE Press, 1979.

[EH88] A. Ehrenfeucht and D. Haussler. A new distance metric on strings computable in linear time. *Discrete Applied Mathematics*, 20:191–203, 1988.

[EL90] D. Elliman and I. Lancaster. A review of segmentation and contextual analysis techniques for text recognition. *Pattern Recognition*, 23(3/4):337–346, 1990.

[FPS97] J. French, A. Powell, and E. Schulman. Applications of approximate word matching in information retrieval. In *Proc. ACM CIKM'97*, pages 9–15, 1997.

[GBY91] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 2nd edition, 1991.

[GG88] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4:33–72, 1988.

[GKHO97] R. Giegerich, S. Kurtz, F. Hischke, and E. Ohlebusch. A general technique to improve filter algorithms for approximate string matching. In *Proc. WSP'97*, pages 38–52. Carleton University Press, 1997. Preliminary version as Technical Report 96-01, Universität Bielefeld, Germany, 1996.

[GL89] R. Grossi and F. Luccio. Simple and efficient string matching with k mismatches. *Information Processing Letters*, 33(3):113–120, 1989.

[Gon92] G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zuerich, Switzerland, 1992.

[Gos91] J. Gosling. A redisplay algorithm. In *Proc. ACM SIGPLAN/SIGOA Symp. on Text Manipulation*, pages 123–129, 1991.

[GP90] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM Journal on Computing*, 19(6):989–999, 1990. Preliminary version in *ICALP'89*, LNCS 372, 1989.

[GT78] R. González and M. Thomason. *Syntactic pattern recognition.* Addison-Wesley, 1978.

[HD80] P. Hall and G. Dowling. Approximate string matching. *ACM Computing Surveys*, 12(4):381–402, 1980.

[Hec78] P. Heckel. A technique for isolating differences between files. *Comm. of the ACM*, 21(4):264–268, 1978.

[Hor80] R. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–506, 1980.

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

[HT84] D. Harel and E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

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

[JTU96] P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Software Practice and Experience*, 26(12):1439–1458, 1996. Preliminary version in Technical Report A-1991-7, Dept. of Computer Science, Univ. of Helsinki, 1991.

[JU91] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. MFCS'91*, volume 16, pages 240–248. Springer-Verlag, 1991.

[KM97] S. Kurtz and G. Myers. Estimating the probability of approximate matches. In *Proc. CPM'97*, LNCS 1264, pages 52–64. Springer-Verlag, 1997.

[KMP77] D. Knuth, J. Morris, Jr, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[Knu73] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[KS94] S. Kumar and E. Spafford. A pattern-matching model for intrusion detection. In *Proc. National Computer Security Conference*, pages 11–21, 1994.

[KS95] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for the inversion distance between two permutations. *Algorithmica*, 13:180–210, 1995.

[Kuk92] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.

[Kur96] S. Kurtz. Approximate string searching under weighted edit distance. In *Proc. WSP'96*, pages 156–170. Carleton University Press, 1996.

[Lev65] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[Lev66] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966. Original in Russian in *Doklady Akademii Nauk SSSR, 163(4):845–848, 1965.*

[LKPC97] J. Lee, D. Kim, K. Park, and Y. Cho. Efficient algorithms for approximate string matching with swaps. In *Proc. CPM'97*, LNCS 1264, pages 28–39. Springer-Verlag, 1997.

[LL85] R. Lipton and D. Lopresti. A systolic array for rapid string comparison. In *Proc. Chapel Hill Conference on VLSI*, pages 363–376, 1985.

[LMS98] G. Landau, E. Myers, and J. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998.

[LS97] T. Luczak and W. Szpankowski. A suboptimal lossy data compression based on approximate pattern matching. *IEEE Trans. on Information Theory*, 43:1439–1451, 1997.

[LT94] D. Lopresti and A. Tomkins. On the searchability of electronic ink. In *Proc. 4th International Workshop on Frontiers in Handwriting Recognition*, pages 156–165, 1994.

[LT97] D. Lopresti and A. Tomkins. Block edit models for approximate string matching. *Theoretical Computer Science*, 181(1):159–179, 1997.

[LV88] G. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer and Systems Science*, 37:63–78, 1988. Preliminary version in *FOCS'85*, 1985.

[LV89] G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10:157–169, 1989. Preliminary version in *ACM STOC'86*, 1986.

[LW75] R. Lowrance and R. Wagner. An extension of the string-to-string correction problem. *Journal of the ACM*, 22:177–183, 1975.

[Mas27] H. Masters. A study of spelling errors. *Univ. of Iowa Studies in Education*, 4(4), 1927.

[McC76] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[Mel96] B. Melichar. String matching with k differences by finite automata. In *Proc. ICPR'96*, pages 256–260. IEEE CS Press, 1996. Preliminary version in *Computer Analysis of Images and Patterns*, LNCS 970, 1995.

[MM96] R. Muth and U. Manber. Approximate multiple string search. In *Proc. CPM'96*, LNCS 1075, pages 75–86. Springer-Verlag, 1996.

[MNZBY99] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Block addressing inverted compressed files. Submitted to *Kluwer Information Retrieval Journal*, 1999.

[Mor68] D. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[MP80] W. Masek and M. Paterson. A faster algorithm for computing string edit distances. *Journal of Computer and System Sciences*, 20:18–31, 1980.

[MW94] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Technical Conference*, pages 23–32. USENIX Association, Berkeley, CA, USA, Winter 1994. Preliminary version as Technical Report 93-34, Dept. of Computer Science, Univ. of Arizona, Oct. 1993.

[Mye86a] G. Myers. Incremental alignment algorithms and their applications. Technical Report 86–22, Dept. of Computer Science, Univ. of Arizona, 1986.

[Mye86b] G. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[Mye94a] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, 1994. Perliminary version in Technical Report TR90-25, Computer Science Dept., Univ. of Arizona, Sept. 1991.

[Mye94b] G. Myers. *Algorithmic Advances for Searching Biosequence Databases*, pages 121–135. Plenum Press, 1994.

[Mye98] G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic progamming. In *Proc. CPM'98*, LNCS 1448, pages 1–13. Springer-Verlag, 1998.

[Nav97a] G. Navarro. Multiple approximate string matching by counting. In *Proc. WSP'97*, pages 125–139. Carleton University Press, 1997.

[Nav97b] G. Navarro. A partial deterministic automaton for approximate string matching. In *Proc. WSP'97*, pages 112–124. Carleton University Press, 1997.

[Nav98a] G. Navarro. *Approximate Text Searching.* PhD thesis, Dept. of Computer Science, Univ. of Chile, December 1998. Technical Report TR/DCC-98-14. `ftp://-ftp.dcc.uchile.cl/pub/users/gnavarro/thesis98.ps.gz`.

[Nav98b] G. Navarro. Improved approximate pattern matching on hypertext. In *Proc. LATIN'98*, LNCS 1380, pages 351–357. Springer-Verlag, 1998. Extended version to appear in *Theoretical Computer Science.*

[NBY98a] G. Navarro and R. Baeza-Yates. Improving an algorithm for approximate pattern matching. Technical Report TR/DCC-98-5, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/-dexp.ps.gz`.

[NBY98b] 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`.

[NBY98c] G. Navarro and R. Baeza-Yates. Very fast and simple approximate string matching. Technical Report TR/DCC-98-6, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/hpexact.ps.gz`.

[NBY99a] G. Navarro and R. Baeza-Yates. Fast multi-dimensional approximate pattern matching. In *Proc. CPM'99*, LNCS, Warwick, England, 1999. To appear.

[NBY99b] G. Navarro and R. Baeza-Yates. A new indexing method for approximate string matching. In *Proc. of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, LNCS, 1999. To appear. `ftp//ftp.dcc.uchile.cl/pub/users/-gnavarro/cpm99.2.ps.gz`.

[Nes86] J. Nesbit. The accuracy of approximate string matching algorithms. *Journal of Computer-Based Instruction*, 13(3):80–83, 1986.

[NR98a] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proc. CPM'98*, LNCS 1448, pages 14–33. Springer-Verlag, 1998.

[NR98b] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. Technical Report TR/DCC-98-4, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/-gnavarro/bndm2.ps.gz`.

[NW70] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:444–453, 1970.

[OM88] O. Owolabi and R. McGregor. Fast approximate string matching. *Software Practice and Experience*, 18(4):387–393, 1988.

[Riv76] R. Rivest. Partial-match retrieval algorithms. *SIAM Journal on Computing*, 5(1), 1976.

[RS97] M. Régnier and W. Szpankowski. On the approximate pattern occurrence in a text. In *Proc. Compression and Complexity of SEQUENCES'97*. IEEE Press, 1997.

[San72] D. Sankoff. Matching sequences under deletion/insertion constraints. In *Proc. of the National Academy of Sciences of the USA*, volume 69, pages 4–6, 1972.

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

[Sel80] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.

[Shi96] F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. WSP'96*, pages 257–271. Carleton University Press, 1996.

[SK83] D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, 1983.

[SM83] D. Sankoff and S. Mainville. *Common subsequences and monotone subsequences*, pages 363–365. Addison-Wesley, 1983.

[ST95] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. ESA'95*, LNCS 979, pages 327–340. Springer-Verlag, 1995.

[ST96] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 50–61. Springer-Verlag, 1996.

[Sun90] D. Sunday. A very fast substring search algorithm. *Comm. of the ACM*, 33(8):132–142, 1990.

[SV88] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[SV97] S. Sahinalp and U. Vishkin. Approximate pattern matching using locally consistent parsing. Manuscript, University of Maryland Institute for Advanced Computer Studies (UMIACS), 1997.

[Tak94] T. Takaoka. Approximate pattern matching with samples. In *Proc. ISAAC'94*, LNCS 834, pages 234–242. Springer-Verlag, 1994.

[Tic84] W. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

[TU88] J. Tarhio and E. Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science*, 57:131–145, 1988.

[TU93] J. Tarhio and E. Ukkonen. Approximate Boyer-Moore string matching. *SIAM Journal on Computing*, 22(2):243–260, 1993. Preliminary version in *SWAT'90*, LNCS 447, 1990.

[Ukk85a] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985. Preliminary version in *Proc. Int. Conf. Found. Comp. Theory*, LNCS 158, 1983.

[Ukk85b] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[Ukk92] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science*, 1:191–211, 1992.

[Ukk93] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. CPM'93*, pages 228–242, 1993.

[Ukk95] E. Ukkonen. Constructing suffix trees on-line in linear time. *Algorithmica*, 14(3):249–260, 1995.

[Ull77] J. Ullman. A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *The Computer Journal*, 10:141–147, 1977.

[UW93] E. Ukkonen and D. Wood. Approximate string matching with suffix automata. *Algorithmica*, 10:353–364, 1993. Preliminary version in Report A-1990-4, Dept. of Computer Science, Univ. of Helsinki, April 1990.

[Vin68] T. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4:52–58, 1968.

[Wat95] M. Waterman. *Introduction to Computational Biology.* Chapman and Hall, 1995.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *Proc. IEEE Symp. on Switching and Automata Theory*, pages 1–11, 1973.

[WF74] R. Wagner and M. Fisher. The string to string correction problem. *Jornal of the ACM*, 21:168–178, 1974.

[WM92a] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, 1992.

[WM92b] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX Technical Conference*, pages 153–162, Berkeley, CA, USA, Winter 1992. USENIX Association.

[WMM95] S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate regular expression matching. *Journal of Algorithms*, 19(3):346–360, 1995. Submitted in 1992.

[WMM96] S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996. Preliminary version as Technical Report TR29-36, Computer Science Dept., Univ. of Arizona, 1992.

[Wri94] A. Wright. Approximate string matching using within-word parallelism. *Software Practice and Experience*, 24(4):337–362, 1994.

[Yao79] A. Yao. The complexity of pattern matching for a random string. *SIAM Journal on Computing*, 8:368–387, 1979.

[YFM96] T. Yap, O. Frieder, and R. Martino. *High performance computational methods for biological sequence analysis*. Kluwer Academic Publishers, 1996.

[ZD96] J. Zobel and P. Dart. Phonetic string matching: lessons from information retrieval. In *Proc. SIGIR'96*, pages 166–172, 1996.

Appendix: Some Analyses

Since some of the source papers lack an analysis or they do not analyze exactly what is of our interest, we have provided a simple analysis for them. This is not the purpose of this survey, so we have contented ourselves with rough figures. In particular, our analyses are valid for $\sigma << m$. All refer to filters and are organized according to the original paragraphs, so the reader should first read the algorithm description to understand the terminology of each.

Tarhio and Ukkonen 1990 First, the probability of a text character being "bad" is that of not matching $2k+1$ pattern positions, i.e. $P_{bad} = (1-1/\sigma)^{2k+1} \approx e^{-(2k+1)/\sigma}$, so we try on average $1/P_{bad}$ characters until finding a bad one. Since $k+1$ bad characters have to be found, we work $O(k/P_{bad})$ to abandon the window. On the other hand, the probability of verifying a text window is that of reaching its beginning. We approximate that probability by equating m to the average portion of the traversed window (k/P_{bad}), to obtain $\alpha < e^{-(2k+1)/\sigma}$.

Wu and Manber 1992 The Sunday algorithm can be analyzed as follows. To see how far can we verify in the current window, consider that the $(k+1)$ patterns have to fail. Each one fails on average in $\log_\sigma(m/(k+1))$ character comparisons, but the time for all them to fail is longer. By Yao's bound [Yao79], this cannot be less than $\log_\sigma m$. Otherwise we could split the test of a single pattern in $(k+1)$ tests of subpatterns and all them would fail in less than $\log_\sigma m$ time, breaking the lower bound. To compute the average shift, consider that k characters must be different from the last window character, and therefore the average shift is σ/k. The final complexity is therefore $O(kn\log_\sigma(m)/\sigma)$. This is optimistic but we conjecture that it is the correct complexity. An upper bound is obtained by replacing k by k^2 (i.e. adding the times for all the pieces to fail).

Navarro and Raffinot 1998 The automaton matches with the text window with k errors almost surely until k/α^* characters have been inspected (so that the error level becomes lower than α^*). From there on, it becomes exponentially decreasing on γ, which can be made $1/\sigma$ in $O(k)$ total

steps. From that point on we are in a case of exact string matching and then $\log_\sigma m$ characters are inspected, for a total of $O(k/\alpha^* + \log_\sigma m)$. When the window is shifted to the last prefix that matched with k errors, this is also at k/α^* distance from the end of the window, on average. The window length is $m-k$, and therefore we shift the window in $m-k-k/\alpha^*$ on average. Therefore, the total amount of work is $O(n(\alpha + \alpha^* \log_\sigma(m)/m)/((1-\alpha)\alpha^* - \alpha)$. The filter works well unless the probability of finding a pattern prefix with errors at the beginning of the window is high. This is the same to say that $k/\alpha^* = m-k$, which gives $\alpha < (1 - e/\sqrt{\sigma})/(2 - e/\sqrt{\sigma})$.

Ukkonen 1992 The probability of finding a given q-gram in the text window is $1-(1-1/\sigma^q)^m \approx 1 - e^{-m/\sigma^q}$. So the probability of verifying the text position is that of finding $(m-q+1-kq)$ q-grams of the pattern, i.e. $\binom{m-q+1}{kq}(1-e^{-m/\sigma^q})^{m-q+1-kq}$. This must be $O(1/m^2)$ to not interfere with the search time. Taking logarithms and approximating the combinatorials using Stirling's $n! = (n/e)^n\sqrt{2\pi n}(1+O(1/n))$ we arrive to

$$kq \quad < \quad \frac{2\log_\sigma m + (m-q+1)\log_\sigma(1-e^{-m/\sigma^q})}{\log_\sigma(1-e^{-m/\sigma^q}) + \log_\sigma(kq) - \log_\sigma(m-q+1)}$$

from where by replacing $q = \log_\sigma m$ we obtain

$$\alpha \quad < \quad \frac{1}{\log_\sigma m(\log_\sigma \alpha + \log_\sigma \log_\sigma m)} \quad = \quad O\left(\frac{1}{\log_\sigma m}\right)$$

a quite common result for this type of filters. The choice $q = \log_\sigma m$ is because the result improves as q grows, but it is necessary that $q \leq \log_\sigma m$ holds, since otherwise $\log_\sigma(1-e^{-m/\sigma^q})$ becomes zero and the result worsens.