Text Searching: Theory and Practice

Ricardo A. Baeza-Yates and Gonzalo Navarro

Depto. de Ciencias de la Computación, Universidad de Chile, Casilla 2777, Santiago, Chile.
E-mail: {rbaeza,gnavarro}@dcc.uchile.cl.

Abstract

We present the state of the art of the main component of text retrieval systems: the search engine. We outline the main lines of research and issues involved. We survey the relevant techniques in use today for text searching and explore the gap between theoretical and practical algorithms. The main observation is that simpler ideas are better in practice.

In theory,
there is no difference between theory and practice.
In practice, there is.
Jan L.A. van de Snepscheut

The best theory is inspired by practice.
The best practice is inspired by theory.
Donald E. Knuth

1 Introduction

Full text retrieval systems have become a popular way of providing support for text databases. The full text model permits locating the occurrences of any word, sentence, or simply substring in any document of the collection. Its main advantages, as compared to alternative text retrieval models, are simplicity and efficiency. From the end-user point of view, full text searching of on-line documents is appealing because valid query patterns are just any word or sentence of the documents. In the past, alternatives to the full-text model, such as semantic indexing, had received some attention. Nowadays, those alternatives are confined to very specific domains while most text retrieval systems used in practice rely, in one way or another, on the full text model.

The main component of a full text retrieval system is the text search engine. The task of this engine is to retrieve the occurrences of query patterns in the text collection. Patterns can range from simple words, phrases or strings to more sophisticated forms such as regular expressions. The efficiency of the text search engine is usually crucial for the overall success of the database system.

It is possible to build and maintain an *index* over the text, which is a data structure designed to speed up searches. However, (*i*) indices take space, which may or may not be available; (*ii*) indices take time to build, so construction time must be amortized over many searches, which outrules indexing for texts that will disappear soon such as on-line news; (*iii*) indices are relatively costly to maintain upon text updates, so they may not be useful for frequently changing text such as in text editors; and (*iv*) the search without an index may be fast enough for texts of several megabytes, depending on the application. For these reasons, we may find text search systems that rely on sequential search (without index), indexed search, or a combination of both.

In this chapter we review the main algorithmic techniques to implement a text search engine, with or without indices. We will focus on the most recent and relevant techniques in practice, although we will cover the main theoretical contributions to the problem and highlight the tension between theoretically and practically appealing ideas. We will show how simpler ideas work better in practice. As we do not attempt to fully cover all the existing developments, we refer the interested reader to several books that cover the area in depth [31, 22, 58, 13, 71, 25, 34].

2 Basic Concepts

We start by defining some concepts. Let *text* be the data to be searched. This data may be stored in one or more files, a fact that we disregard for simplicity. Text is not necessarily natural language, but just a sequence (or *string*) of characters. In addition, some pieces of the text might not be retrievable (for example, if a text contains images or non-searchable data), or we may want to search different pieces of text in different ways. For example, in GenBank registers one finds DNA streams mixed with English descriptions and metadata, and retrieval over those parts of the database are very different processes.

2.1 The Full Text Model

The text database can be viewed as a very long string of data. Often text has little or no structure, and in many applications we wish to process the text without concern for the structure. This is typically the case of biological or multimedia sequences. As oriental languages such as Chinese, Japanese, and Korean Kanji are difficult to segment into words, texts written in those languages are usually regarded as a very long string too. For this view of the text, we usually want to retrieve any text substring.

Natural language texts written in Western languages can be regarded as a sequence of *words*. Each word is a maximal string not including any symbol from a special separator set (such as a blank space). Examples of such collections are dictionaries, legal cases, articles on wire services, scientific papers, etc. In this case we might want to retrieve only words or phrases, or also any text substring.

Optionally, a text may have a structure. The text can be logically divided into documents, and each document into fields. Alternatively, the text may have a complex hierarchical or graph structure. This is the case of text databases structured using XML, SGML or HTML, where we might also query the structure of the text. In this chapter we will ignore text structure and regard it as a sequence of characters or words.

2.2 Notation

We introduce some notation now. A string S of length $|S| = s$ will be written as $S_{1\ldots s}$. Its i-th character will be S_i and a string formed by characters i to j of S will be written $S_{i\ldots j}$. Characters are drawn from an alphabet Σ of size $\sigma > 1$. String S is a *prefix* of SS', a *suffix* of $S'S$, and a substring of $S'SS''$, where S' and S'' are arbitrary strings.

The most basic problem in text searching is to find all the positions where a pattern $P = P_{1\ldots m}$ occurs in a text $T = T_{1\ldots n}$, $m \leq n$. In some cases one is satisfied with any occurrence of P in T, but in this chapter we will focus on the most common case where one wants them all. Formally, the search problem can be written as retrieving $\{|X|,\ T = XPY\}$. More generally, pattern P will denote a language, $L(P) \subseteq \Sigma^*$, and our aim will be to find the text occurrences of any string in that language, formally $\{|X|,\ T = XP'Y,\ P' \in L(P)\}$.

We will be interested both in worst-case and average-case time and space complexities. We remind that the worst case is the maximum time/space the algorithm may need over every possible text and pattern, while the average case is the mean time over all possible texts and patterns. For average case results we assume that pattern and text characters are uniformly and independently distributed over the σ characters of the alphabet. This is usually not true, but it is a reasonable model in practice.

We make heavy use of bit manipulations inside computer words, so we need some further notation for these. Computer words have w bits (typically $w = 32$, 64 or 128), and sequence of bits in a computer word are written right to left. The operations to manipulate them are inherited from C language: "|" is the bitwise "or", "&" is the bitwise "and", "<<" shifts all the bits to the left (that is, the bit at position i moves to position $i + 1$) and enters a zero at the rightmost position, ">>" shifts to the right and enters a zero at the leftmost position (unsigned semantics), "$\wedge$" is the bitwise exclusive or (xor), and "$\sim$" complements all the bits. We can also perform arithmetic operations on the computer words such as "+" and "−".

2.3 Text Suffixes

Observe that any occurrence of P in T is a prefix of a suffix of T (this suffix is PY or $P'Y$ in our formal definition). The concept of prefix and suffix plays a central role in text searching. In particular, a model that has proven extremely useful, and that marries very well with the full text model, is to consider the text as the set of its suffixes.

We assume that the text to be searched is a single string padded at its right end with a special terminator character, denoted "$" and smaller than any other. A text suffix is simply a suffix of T, that is, the sequence of characters starting at any position of the text and continuing to the right. Given the terminator, all the text suffixes are different, and no one is a prefix of any other. Moreover, there is a one-to-one correspondence between text positions i and text suffixes $T_{i...n}$.

Under this view, the search problem is to find all the text suffixes starting with P (or $P' \in L(P)$).

2.4 Tries

Finally, let us introduce an extremely useful data structure for text searching. It is called a *trie*, and is a data structure that stores a set of strings and permits determining in time $O(|S|)$ whether string S is in the set, no matter how many strings are stored in the trie. A trie on a set of strings is a tree with one leaf per string and one internal node per different proper prefix of a string. There is an edge labeled c from every node representing prefix S to the node representing prefix Sc. The root represents the empty string. Figure 1 (left) illustrates.

In order to search for S in the trie, we start at the root and follow edge S_1, if it exists. If we succeed, we follow edge S_2 from the node we arrived at, and so on until either (i) we cannot find the proper edge to follow, in which case S is not in the set, or (ii) we use all the characters in S, and if the node arrived at stores a string, we have found S. Note that it is also easy to determine whether S is a prefix of some string stored in the trie: We search for S and all the subtree of the node arrived at contains the strings whose prefix is S.

We might add a special string terminator "$" to all the strings so as to ensure that no string is a prefix of another in the set. In this case there is a one-to-one correspondence between trie leaves and stored strings. To save some space, we usually put the leaves as soon as the string prefix is unique. When the search arrives at such a leaf, the search process continues comparing the search string against the string stored at the leaf. Figure 1 (right) illustrates.

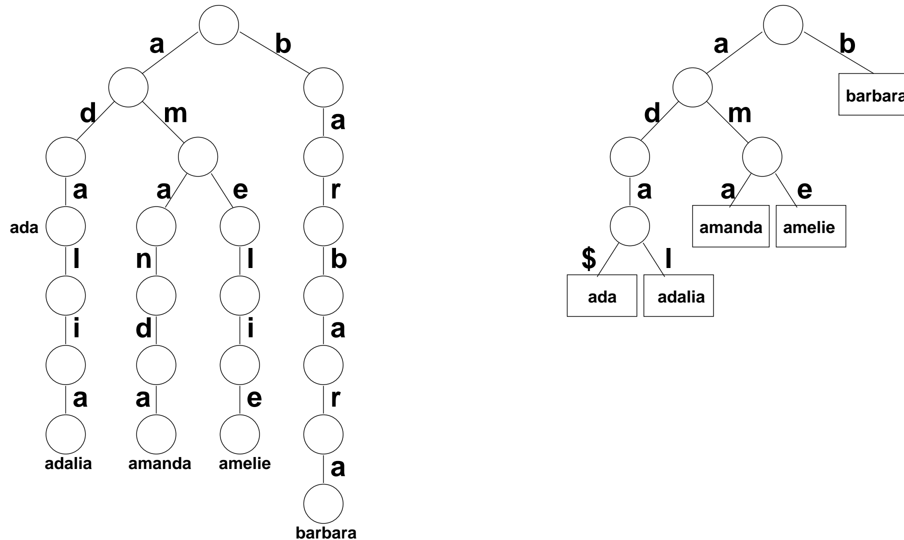

Figure 1: A trie over the strings "ada", "amanda", "amelie", "barbara" and "adalia". On the left, basic formulation. On the right, the version used in practice.

3 Sequential Text Search

In this section we assume that no index on the text is available, so we have to scan all the text in order to report the occurrences of P. We start with simple string patterns and later consider more sophisticated searches.

To better understand the problem, let us consider which would be its naive solution. Consider every possible initial position of an occurrence of P in T, that is, $1 \dots n-m+1$. For each such initial position i, compare P with $T_{i \dots i+m-1}$. Report an occurrence whenever the two strings match. The worst case complexity of this algorithm is $O(mn)$. Its average case complexity, however, is $O(n)$, since on average we have to compare $\sigma/(\sigma-1)$ characters before two strings mismatch. Our aim is to do better.

From a theoretical viewpoint, this problem is basically solved, except for very focused questions that still remain.

- The worst-case complexity is clearly $\Omega(n)$ character inspections (for the exact constant see [18]). This has been achieved by Knuth-Morris-Pratt (KMP) algorithm [41] using $O(m)$ space.
- The average-case complexity is $\Omega(n \log_\sigma(m)/m)$ [76]. This has been achieved by Backward DAWG Matching (BDM) algorithm [21] using $O(m)$ space. The algorithm can be made worst-case optimal at the same time (e.g., TurboBDM and TurboRF variants [21]).
- Optimal worst-case algorithms with $O(1)$ extra space exist (the first was [28]), while the same problem is open for the average case.

If one turns attention to practical matters, however, the above algorithms are in several aspects unsatisfactory. For example, in practice KMP is twice as slow as the naive search algorithm, which can be programmed with three lines of code or so. BDM, on the other hand, is rather complicated to implement and not so fast for many typical text searching scenarios. For example, BDM is a good choice to search for patterns of length $m = 100$ on DNA text, but when it comes to search for words and phrases (of length typically less than 30) in natural

language text, it is outperformed by far by a simple variant of Boyer-Moore due to Horspool [36], which also can be coded in a few lines.

In this section we will consider two successful approaches to text searching. These are responsible for most of the relevant techniques in use today. In both approaches we will explore the relation between theoretically and practically appealing algorithms, and we will show that usually the best practical algorithms are simplified versions of their complex theoretical counterparts. The first approach is the use of automata, which theoretical algorithms convert to deterministic form while practical algorithms simulate in their simple and regular nondeterministic form. The second approach is the use of filtering, where theoretical algorithms minimize the number of character inspections at a considerable extra processing cost, while practical algorithms use simpler criteria that inspect more characters but are faster in practice.

3.1 Automata

Consider pattern $P =$ `"abracadabra"`. Figure 2 shows a Nondeterministic Finite Automaton (NFA) that recognizes the language Σ^*P, that is, strings finishing with P. As it can be seen, the automaton has a very regular structure.

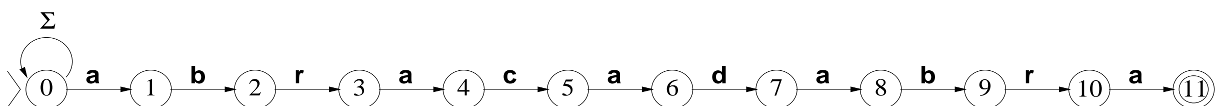

Figure 2: An NFA to search for `"abracadabra"`. The initial state is marked with "⟩" and the final state is double-circled.

This NFA can be used for text searching as follows: feed it with the characters of T. Each time it recognizes a word, it means that we have read a string in the set Σ^*P, or which is the same, we have found the pattern in the text. Then we can report every occurrence of P in T.

Since the NFA has $m + 1$ states, its simulation over n text characters takes $O(mn)$ time, in the worst and average case. The reason is that, in principle, each NFA state can be active or inactive and we have to update them all. This complexity is not appealing if we compare it against the naive algorithm.

A clear option to reduce search time is to make the automaton deterministic (DFA). In this case, there will be only one active state at the time, and processing the automaton over the text will take $O(n)$ search time. Figure 3 shows the DFA for the same pattern. As it can be seen, its regularity has been lost. To the previous "forward" arrows, a number of "backward" arrows have been added, which handle the case where the next text character is not the one we expect to match P.

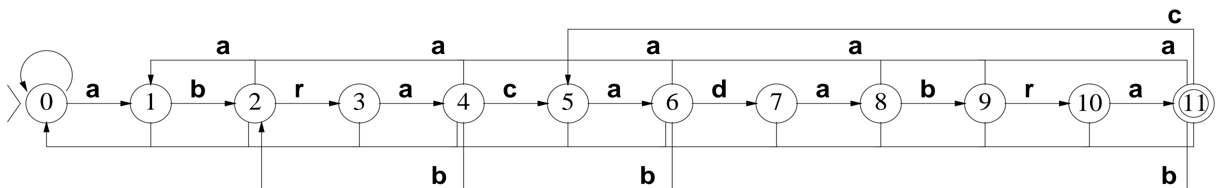

Figure 3: A DFA to search for `"abracadabra"`. The initial state is marked with "⟩" and the final state is double-circled. Unlabeled arrows stand for "any other character".

A problem with deterministic automata is that they could, in principle, need exponential time and space to be built, with respect to the original string. However, this is not the case

of regular expressions of the form $\Sigma^* P$, whose deterministic automata need only $O(m\sigma)$ space because there is a one-to-one correspondence between search states and prefixes of P. Hence using a DFA gives us an $O(m\sigma + n)$ worst-case and average-case time algorithm.

As a matter of fact, the classical algorithm by Knuth, Morris and Pratt (KMP) [41] relies, basically, on the deterministic version of the automaton. In order to reduce preprocessing time and space usage to $O(m)$ (and hence ensure worst-case optimality for the unlikely case $m\sigma > n$), KMP does not store the arrows of the automaton. Rather, for each state, it stores its "failure" state: the largest numbered state reachable by backward arrows, minus one. Upon a failure (that is, when the next text character is not the one we need to move forward in the DFA), KMP goes to the failure state and tries again with the current text letter. If this fails, it moves to the failure state of the failure state, and so on. At worst, this process finishes at the initial state. The correctness of this procedure is clear if we consider that being at state i is equivalent of having matched $P_{1...i}$ against the current text suffix, and that the failure state for i tells which is the longest prefix of $P_{1...i}$ which is also a suffix of $P_{1...i}$. Although a given text position can be compared several times, overall we cannot do more than $2n$ text inspections, because each time we re-compare a text character we move backward in the automaton, and we cannot move forward more than n times (one per text character). Construction of the failure function can be done in $O(m)$ time with a tricky algorithm, which gives the optimal worst-case complexity $O(m + n) = O(n)$ time.

As we have mentioned, despite the algorithmic beauty of KMP algorithm and its worst-case guarantee, it turns out that in practice it is twice as slow as the naive algorithm. There is, however, a much better alternative. We can use *bit parallelism* [9] to directly simulate the NFA, taking advantage of its regular structure.

Let us assume we represent the NFA of Figure 2 using the bits of a computer word. The initial state 0 is always active, so we will not represent it. Hence we need m bits, and let us assume by now that our computer word size, w bits, is large enough to hold the m bits, that is, $m \leq w$. The sequence of m bits will be called a *bit mask*.

Let us preprocess P so as to build a table B of bit masks, indexed by the alphabet Σ. The i-th bit of $B[c]$ will be 1 if and only if $P_i = c$. Let us assume bit mask D stores the active and inactive states of the NFA, so the i-th bit of D tells whether state i is active or not. Initially D will have all its bits in zero.

There are two types of arrows in the NFA. Forward arrows move from state i to $i+1$, as long as the current text character matches P_i. The self-loop at the initial state matches any character and keeps state 0 always active. Hence, using C language notation to operate bit masks, it turns out that D can be updated upon reading text character c with the following simple formula

$$D \quad \longleftarrow \quad ((D << 1) \mid 1) \ \& \ B[c] \ .$$

The shift-left operation simulates the forward arrows. The "1" that is or-ed stands for the initial state, which is always sending active bits to state 1. The "and" operation permits state activations to move from one state to the next only if the current text character matches the corresponding pattern position. After each update of D, we must check whether state m is active, that is,

$$D \ \& \ (1 << (m-1)) \ \ != \ \ 0 \ \ ,$$

and in this case we report an occurrence of P.

This algorithm, by Baeza-Yates and Gonnet [9], is called Shift-And. By using all the bits in complemented form we get the faster Shift-Or [9]. The update formula of Shift-Or is

$$D \ \longleftarrow (D << 1) \mid B[c] \ \ .$$

```
ShiftOr(T_{1...n}, P_{1...m})

    for c ∈ Σ do B[c] ← (1 << m) − 1
    for j ∈ 1...m do B[p_j] ← B[p_j] & ∼(1 << (j − 1))
    D ← (1 << m) − 1
    for i ∈ 1...n do
        D ← (D << 1) | B[t_i]
        if D & (1 << (m − 1)) = 0
          then P occurs at T_{i−m+1...i}
```

Figure 4: Shift-Or algorithm.

Preprocessing of P needs $O(m+\sigma)$ time and $O(\sigma)$ space. The text scanning phase is $O(n)$. Hence we get a very simple algorithm that is $O(\sigma+m+n)$ time and $O(\sigma)$ space in the worst and average case. More important, Shift-Or is faster than the naive algorithm and can be coded in a few lines. Figure 4 gives its pseudocode.

For patterns exceeding the computer word size ($m > w$), a multi-word implementation can be considered. Hence we will have to update $\lceil m/w \rceil$ computer words per text character and the worst case search time will be $O(m + (\sigma+n)\lceil m/w \rceil)$. In practice, since the probability of activating a state larger than w is minimal ($1/\sigma^w$), it is advisable to just search for $P_{1...w}$ using the basic algorithm and compare $P_{w+1...m}$ directly against the text when $P_{1...w}$ matches. Verifications add $n/\sigma^w \times O(1)$ extra time, so the average time remains $O(\sigma+m+n)$.

3.2 Filtering

The previous algorithms try to do their best under the assumption that one has to inspect every text character. In the worst case one cannot do better. However, on average, one can discard large text areas without actually inspecting all their characters. This concept is called *filtering* and was invented by Boyer and Moore [15], only a few years later than KMP (but they were published the same year).

A key concept in filtering is that of a *text window*, which is any range of text positions of the form $i \ldots i+m-1$. The search problem is that of determining which text windows match P. Filtering algorithms *slide* the window left to right on the text. At each position, they inspect some text characters in the window and collect enough information so as to determine whether there is a match or not in the current window. Then, they *shift* the window to a new position. Note that the naive algorithm can be seen as a filtering algorithm that compares P and the text window left to right, and always shifts by one position.

The original idea of Boyer-Moore (BM) algorithms is to compare P and the text window right to left. Upon a match or mismatch, they make use of the information collected in the comparison of the current window so as to shift it by at least one character. The original Boyer-Moore algorithms used the following rules to shift the window as much as possible:

- If we have read suffix s from the window and this has matched the same suffix of P, then we can shift the window until s (in the text) aligns with the last occurrence of s in $P_{1...m-1}$. If there is no such occurrence, then we must align the longest suffix of s that is also a prefix of P.

- If we have read characters from the window and the mismatch has occurred at a character c, then we can shift the window until that position is aligned to a c in P.

In fact, the original BM algorithm precomputes the necessary data to implement only the last heuristic (the first was introduce in the KMP paper), and at each moment it uses the one giving the largest shift. Although in the worst case the algorithm is still $O(mn)$, on average it is $O(n/\min(m,\sigma))$. This was the first algorithm that did not compare all the text characters. The algorithm was later extended to make it $O(n)$ in the worst case [17].

In practice, the algorithm was rather complicated to implement, not to mention if one has to ensure linear worst-case complexity. A practical improvement was obtained by noting that, for large enough alphabets, the second heuristic usually gave good enough shifts (the original idea). Hence computing two shifts to pick the largest was not worth the cost. The resulting algorithm, called Simplified BM, is simpler than BM and competitive.

However, even simpler and faster is Horspool algorithm [36]. Pattern and text window are compared in any order, even left to right (this can be done using a special-purpose machine instruction that compares chunks of memory). If they are equal, the occurrence is reported. In any case, the *last* window character is used to determine the shift. For this sake, a table d is built so that $d[c]$ is the distance from the last occurrence of c in $P_{1...m-1}$ to m. Hence, the window starting at text position i can be shifted by $d[T_{i+m-1}]$. Figure 5 illustrates.

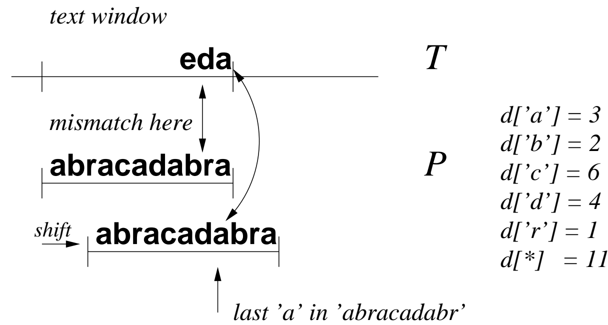

Figure 5: Example of Horspool algorithm over a window terminating in `"eda"`. In the example we assume that P and the window are compared right to left.

Preprocessing requires $O(m + \sigma)$ time and $O(\sigma)$ space, while the average search time is still $O(n/\min(m,\sigma))$. The algorithm fits in less than ten lines of code in most programming languages. A close variant of this algorithm is due to Sunday [62], which uses the text position following the window instead of the last window position. This makes the search time closer to $n/(m+1)$ instead of Horspool's n/m, and is in practice faster than Horspool.

However, if one takes into account that with probability $(\sigma-1)/\sigma$ even the first comparison in the window fails, it is advisable to implement a so-called *skip loop*. This is a pre-filtering phase where the last window character, c, is examined, and if $P_m = c$ then we enter the usual window processing code. Otherwise, the window is shifted by $d[c]$. This not only makes Horspool algorithm much faster (because its processing over most windows is minimal), but also makes it superior to Sunday algorithm, since Horspool examines just one text character from most windows, while Sunday examines at least two. *GNU Grep* (`www.gnu.org`) and Wu and Manber's *Agrep* [72] use Horspool algorithm in most cases, as it is usually the fastest choice in practice. Figure 6 gives its pseudocode.

BM algorithms are not average-optimal (that is, $O(n \log_\sigma(m)/m)$). This is especially noticeable when σ is small compared to m. For example, on DNA, BM algorithms do not shift

Horspool($T_{1...n}$, $P_{1...m}$)

> **for** $c \in \Sigma$ **do** $d[c] \leftarrow m$
> **for** $j \in 1 \dots m-1$ **do** $d[p_j] \leftarrow m - j$
> $i \leftarrow 0$
> **while** $i + m \leq n$ **do**
>   **while** $i + m \leq n \wedge t_{i+m} \neq p_m$ **do** $i \leftarrow i + d[t_{i+m}]$
>   **if** $P_{1...m-1} = T_{i+1...i+m-1}$
>   **then** P occurs at $T_{i+1...i+m}$
> $i \leftarrow i + d[t_{i+m}]$

Figure 6: Horspool algorithm with skip loop included.

windows by more than 4 positions on average, no matter how large is m.

One way to circumvent this weakness is to artificially enlarge the alphabet. This folklore idea has been reinvented several times, see for example [7, 40, 64]. Say that, instead of considering just the last window character to determine the shift, we read the last q characters (that is, the last window q-gram). We preprocess P so that, for each possible q-gram, we record its smallest distance to the end of the pattern. Then, we use Horspool as usual. Note that we pay $O(q)$ character inspections to read a q-gram.

The search time of this algorithm is $O(\sigma^q + m + qn / \min(m, \sigma^q))$, so the optimum is $q = \log_\sigma m$ and this yields the average-optimal $O(m + n \log_\sigma(m)/m)$. This is the technique used in *Agrep* for long patterns. In practice, depending on the architecture, one may take advantage from the fact that the computer word usually can hold a q-gram, and read q-grams in a single machine instruction, so as to get $O(n/m)$ search time.

3.3 Filtering Using Automata

Another way to see why BM is not average-optimal is that it shifts the window *too soon*, that is, as soon as it knows that it can be shifted. Since this occurs after $O(1)$ comparisons, the pattern is shifted so that it aligns correctly with $O(1)$ window characters, and this occurs for a shift of length $O(\sigma)$. Perhaps a bit counterintuitively, one should wait a bit more.

A very elegant algorithm, which was the first in achieving average optimality (and, with some modifications, worst-case optimality at the same time) is Backward DAWG Matching (BDM) [21], by Crochemore et al. This algorithm combines the use of automata with filtering techniques.

BDM preprocesses the *reversed* pattern P^r (that is, P read backwards) so as to build an automaton that recognizes every suffix of P^r. This is called the *suffix automaton* of P^r. The nondeterministic version of this automaton has a very regular structure (see Figure 7).

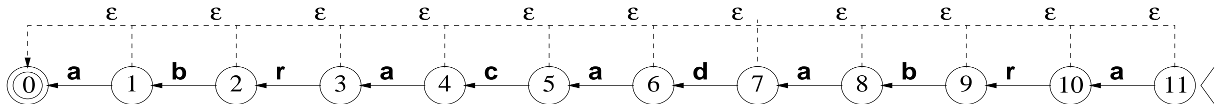

Figure 7: An NFA to recognize the suffixes of `"abracadabra"` reversed. The initial state is marked with "⟨" and the final state is double-circled. Dashed lines correspond to ε-transitions.

The deterministic version of this automaton has $m+1$ states and m to $2m$ edges, and can be built in $O(m)$ time [21]. To achieve this, the automaton must be *incomplete*, that is, only the arrows in the path from the initial to a final state are represented. However, the DFA does not have a regular structure anymore. Figure 8 shows the deterministic suffix automaton for our running example.

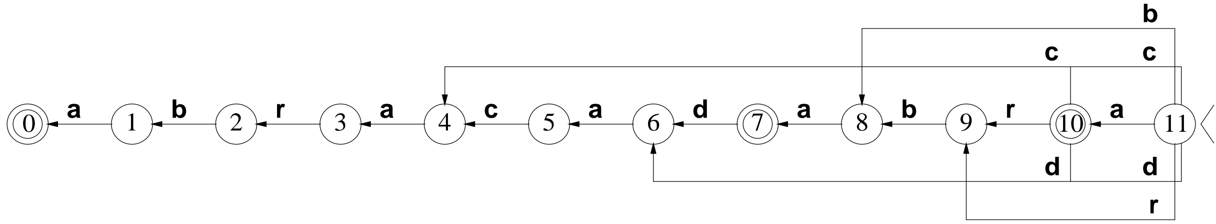

Figure 8: A DFA to recognize the suffixes of `"abracadabra"` reversed. The initial state is marked with "⟨" and the final states are double-circled. Missing transitions go to a (non drawn) sink state.

Since the automaton is deterministic, there is only one active state at the time. Moreover, since it is incomplete, it might be that at some point there is no active state anymore. We say in this case that the automaton is *dead*. In the nondeterministic version, this corresponds to having all states inactive. Note that, if we feed the suffix automaton with a string, at the end it will be dead if and only if the string is *not* a substring of P^r.

The suffix automaton is used as follows. We scan the characters of the text window from right to left and feed the automaton with the text characters, until either the automaton dies or we read all the window. In the first case, the suffix S^r we have read in the window (in reverse order, that is, right to left) is not a substring of P^r, hence S is not a substring of P. This means that no occurrence of P can contain S, and therefore we can shift the window right after the first position of S. In the second case, we have read all the window, of length m, and this is still a substring of P, so we have actually read P and can report it and shift the window by one position.

Actually, the algorithm does better. Each time the suffix automaton reaches a final state we record the window position we are reading. Since the automaton reached a final state, this means that the suffix S^r read is a suffix of P^r, hence S is a prefix of P. Once we have to shift the window (in either case) we can shift it so as to align its beginning with the longest (that is, last) prefix of P recognized in the window. Figure 9 illustrates.

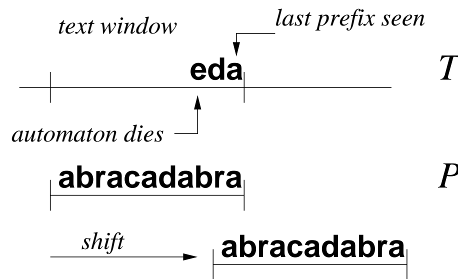

Figure 9: Example of BDM algorithm over a window terminating in `"eda"`.

It is not hard to see that the probability of recognizing a substring of P becomes sufficiently small as soon as we read $O(\log_\sigma m)$ characters, so we shift the window by $m - O(\log_\sigma m)$ after

inspecting $O(\log_\sigma m)$ characters. Hence the algorithm is $O(n \log_\sigma(m)/m)$ average time. In order to make it $O(n)$ worst case time, it is combined with a KMP machine that ensures that no characters are retraversed. This, however, makes the algorithm slower in practice.

The reason why BDM has not gained much popularity in applications is that, on one hand, its optimality becomes significant in practice only for long patterns and small alphabets, and on the other hand, the construction of the DFA is rather complicated. However, there are two practical developments that inherit from BDM ideas and are among the fastest practical algorithms.

The first is Backward Nondeterministic DAWG Matching (BNDM) [57], by Navarro and Raffinot. The idea is to use the same BDM algorithm, but this time simulating the NFA with bit-parallelism instead of building the DFA. As can be seen in Figure 7, the structure of the NFA is rather regular and easy to simulate.

We work directly on P instead of reversing it. Hence the i-th bit of our bit mask D will be active whenever state $i-1$ in the NFA is active. We do not represent the initial state because it is active only in the beginning, to recognize the uninteresting empty substring of P. We build table B as before and initialize $D = 111\ldots1$ (all 1's). Hence, upon reading a new window character c, we update D as

$$D \quad \longleftarrow \quad D \ \&\ B[c]$$

and at this point we can check a couple of conditions: (1) if $D = 0$ then the automaton has died; (2) if $D \ \&\ 1 = 1$ then we have recognized a prefix of P. After this we prepare D for the next window character by right-shifting:

$$D \quad \longleftarrow \quad D >> 1 \ \ .$$

BNDM is faster than BDM and actually the fastest algorithm for small alphabets and not very short patterns (e.g., $m \geq 10$ on DNA and $m \geq 30$ on natural language). It is used in Navarro's *Nrgrep* [52]. Its weakness, however, is that for long patterns we have to update many computer words to simulate it. This time the trick of updating only active computer words does not work because the active states distribute uniformly over the NFA. Hence the only practical choice is to search for $P_{1\ldots w}$ and directly check potential occurrences. Figure 10 gives BNDM pseudocode.

Another practical alternative is algorithm Backward Oracle Matching (BOM) [4], by Allauzen et al., which replaces the suffix automaton by an *oracle*. The oracle is an automaton that recognizes *more* than the suffixes of P^r. Therefore the algorithm makes shorter shifts. However, the fact that the oracle is simpler to build and smaller makes BOM faster in practice than BDM, and faster than BNDM for $m \geq 2w$ or so.

Construction of the oracle for a string S is rather simple compared to the deterministic suffix automaton. It reminds the construction of the failure transitions for KMP. S is processed left to right and for each new character c the path of failures is followed. At each step of the path, a new transition from the current node to the new final node, labeled by c, is inserted. In our running example, the oracle is equal to the suffix automaton.

3.4 Searching for Multiple Patterns

Let us now assume that we want to search for a set of patterns $P_1 \ldots P_r$, all of length m for simplicity of presentation. Several of the approaches designed to search for a single pattern can be extended. In particular, automata and filtering approaches can be gracefully extended. Bit-parallelism, on the other hand, is hardly useful because the resulting automata are rather large, and hence even dividing the NFA update time by w is not enough.

```
BNDM(T_{1...n}, P_{1...m})

    for c ∈ Σ do B[c] ← 0
    for j ∈ 1...m do B[p_j] ← B[p_j] | (1 << (j − 1))
    i ← 0
    while i + m ≤ n do
          D ← (1 << m) − 1
          j ← m; p ← m
          while D ≠ 0 do
                D ← D & B[t_{i+j}]
                j ← j − 1
                if D & 1 ≠ 0
                   then if j > 0
                           then p ← j
                           else P occurs at T_{i+1...i+m}
                D ← D >> 1
          i ← i + p
```

Figure 10: BNDM algorithm.

On the theoretical side, the worst case lower bound to the problem is still $\Omega(n)$ and it can be achieved with an automaton due to Aho and Corasick (AC) [2], using $O(rm)$ preprocessing time and space. The average case lower bound is $\Omega(n \log_\sigma(rm)/m)$ [26] and it is achieved by Set Backward DAWG Matching (SBDM) algorithm [5] based on the ideas of Blumer et al. [14], which is simultaneously $O(n)$ worst case time. Its preprocessing time and space is also $O(rm)$.

Since bit-parallelism is outruled, theory and practice are not so far away in this case. Algorithm AC, which is an extension of KMP, is not only worst-case optimal but also the best choice when m is small or r is very large. The set of patterns is preprocessed so as to build a trie data structure (Section 2.4) on it.

Next, the trie is augmented with *failure links*, which point from every prefix S to the node representing the longest suffix of S that is also a prefix of some string in the set. The whole data structure takes $O(rm)$ space and can be easily built in $O(rm)$ worst-case time.

Then the algorithm works by starting at the root of the trie and scanning text characters one by one. If there is an edge labeled by the character read, we move down in the trie by that edge. Otherwise we move by the failure link and try again until we can follow an edge or we are at the trie root again. Each time we arrive at a leaf we report an occurrence.

Just like KMP can be seen as a space-efficient version of using a DFA that recognizes $\Sigma^* P$, AC can be seen as a space-efficient version of a DFA, of size $O(rm\sigma)$, that recognizes $\Sigma^*(P_1|\ldots|P_r)$.

With respect to filtering algorithms, algorithm BM was extended by Commentz-Walter [19] so as to build a trie on the set of reversed patterns, scan windows right to left while entering the trie at the same time, and precomputing shifts for each node of the trie. Although this is an elegant extension of the BM idea, the resulting algorithm is not competitive nowadays, neither in theory nor in practice.

In practice, an idea that works extremely well is to extend Horspool algorithm. This time using an alphabet of q-grams is mandatory. The idea is that all the q-grams of all the patterns are considered, so that one records the smallest distance between any occurrence of the q-gram

in any of the patterns, and the end of that pattern. The resulting algorithm (WM) [74], by Wu and Manber, obtains average optimality by choosing $q = \log_\sigma(rm)$, and is a very attractive choice, because of its simplicity and efficiency. It is implemented in *Agrep*. Several practical considerations are made in order to efficiently handling large alphabets. WM is in practice the fastest algorithm except when r is very large or σ is small.

With respect to merging automata and filtering, the average-optimal SBDM algorithm is an acceptable choice. It is based on the same idea of BDM, except that this time the suffix automaton recognizes any suffix of any pattern. This automaton can be seen as the trie of the patterns augmented with some extra edges that permit skipping pattern prefixes so as to go down directly to any trie node.

However, algorithm Set Backward Oracle Matching (SBOM) [5] by Allauzen and Raffinot, is simpler and faster. The idea is, again, to use an automaton that recognizes more than the suffixes of the patterns. This benefits from simplicity and becomes faster than SBDM. The construction of the oracle is very similar to that of the AC automaton and can be done in $O(rm)$ space and time. SBOM is the best algorithm for small σ or very large r.

3.5 Searching for Complex Patterns

A more sophisticated form of searching permits patterns contain not only simple characters, but also "wild cards" of different types, such as (*i*) classes of characters like `"[abc]"` to match `"a"`, `"b"` or `"c"`; (*ii*) optional characters or classes like `"a?"`, where `"a"` may appear or not; and (*iii*) repeatable characters or classes like `"a*"`, where `"a"` can appear zero or more times. These can be used to express other well known wild cards such as `"."` (which matches any character), `"*"` (which stands for `".*"`), bounded length gaps (such as any string of length 2 to 4, which can be written as `"...?.?"`), etc. This can go as far as permitting the pattern to be any regular expression, that is, simple strings and any union, repetition and concatenation of regular subexpressions.

As long as complex patterns are regular expressions, they can be converted into DFAs and searched for in $O(n)$ time [3]. This is theoretically appealing and, for sufficiently complex patterns, the most practical choice. However, the preprocessing time and space requirements can be exponential in m. Another alternative is to simulate the automata in nondeterministic form, obtaining $O(mn)$ search time [65].

There are, however, particular cases where specialized solutions exist. On the theoretical side, there exist solutions to search for patterns with classes of characters and simple wild cards [24, 60, 1]. Although theoretically interesting, these are not relevant in practice.

Bit-parallelism, on the other hand, yields extremely simple and efficient solutions to this kind of problems. For example, classes of characters are easily handled in the Shift-And[1] and BNDM algorithms. We only need to set the i-th bit $B[c]$ for any $c \in P_i$.

Optional and repeatable characters are a bit more complicated. Let us assume we want to search for `"ab*ra?cad*ab?ra"`. Figure 11 shows a NFA that searches for it.

We show how to simulate this NFA using Shift-And. Optional characters are marked in a new bit mask O, whose i-th bit is 1 whenever P_i is optional. Repeatable characters are marked as optional in O and also in a table $R[c]$, whose i-th bit is 1 whenever character c can be repeated at pattern position i. In our example, $O = 00101001010$, $R['b'] = 00000000010$ and $R['d'] = 00001000000$; all the others are zero. We identify in O contiguous blocks of 1's and mark their initial and next-to-final positions in masks I and F. In our example, $I = 00101001010$ and $F = 01010010100$. This complication is necessary because there could be contiguous optional

[1]Shift-Or is not faster this time, so we stick to the simpler to explain Shift-And.

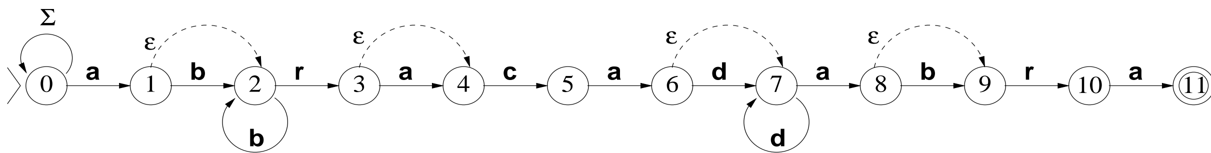

Figure 11: NFA that searches for `"ab*ra?cad*ab?ra"`. The initial state is marked with "〉" and the final state is double-circled.

characters. The update formula for mask D after reading text character c, when we permit optional and repeatable characters, is as follows

$$\begin{aligned} D &\leftarrow ((D << 1) \mid 1) \ \& \ B[c]) \ \mid \ (D \ \& \ R[c]) \\ E &\leftarrow D \mid F \\ D &\leftarrow D \mid (O \ \& \ ((\sim (E - I)) \ \wedge \ E)) \ . \end{aligned}$$

We do not attempt to explain all the details of this formula, but just to show that it is a rather simple solution to a complex problem. It can be used for a Shift-And-like scanning of the text or for a BNDM-like search. In this latter case, the window length should be that of the shortest possible occurrence of P, and when we arrive at the beginning of the window, the presence of an occurrence is not guaranteed. Rather, if we have found a pattern prefix at the beginning of the window, we should verify whether an occurrence starts at that position. This is done by simulating an automaton like that of Figure 11 with the initial self-loop removed.

In general, worst-case search time is $O(n)$ for short patterns and $O(nm/w)$ for longer ones using bit-parallelism. A BNDM-like solution can be much faster depending on the sizes of the classes and number of optional and repeatable characters. In practice, these solutions are simple and practical, and they have been implemented in *nrgrep* [52].

Full regular expressions are more resistant to simple approaches. Several lines of attack have been tried, both classical and bit-parallel, to obtain a tradeoff between search time and space. Both have converged to $O(mn/\log s)$ time using $O(s)$ space [50, 56].

Finally, efficient searching for multiple complex patterns is an open problem.

3.6 Approximate Pattern Matching

An extremely useful search option in text retrieval systems is to allow a limited number k, of differences between pattern P and its occurrences in T. The search problem varies widely depending on what is considered a "difference".

A simple choice is to assume that a difference is a substitution of one character by another. Therefore, we are permitted to change up to k letters in P in order to match it in a text position. This problem can be nicely solved using bit-parallelism, and it was in fact one of the first extensions to Shift-Or, by Baeza-Yates and Gonnet, showing the potential of the approach [9].

Let us reconsider the NFA of Figure 2. After having read text position j, state i is active whenever $P_{1\ldots i}$ matches $T_{j-i+1\ldots j}$. This time, let us consider that each state i is not simply active or inactive, but that it remembers how many characters differ between $P_{1\ldots i}$ and $T_{j-i+1\ldots j}$. Every time the counter at state m does not exceed k, we have an occurrence with at most k mismatches.

Our bit mask D holds m counters, $c_m \ldots c_1$, for each of which we have to reserve $\ell = \lceil \log_2(m+1) \rceil$ bits. We precompute a bit mask table of counters $B[c]$, whose i-th counter will be 0 if $P_i = c$ and 1 otherwise. Hence, after reading text character c we can update D simply

by doing

$$D \quad \leftarrow \quad (D << \ell) + B[c]$$

which moves each counter to the next state and adds a new mismatch if appropriate. After each new text character is processed, we report an occurrence whenever $D >> (m-1)\ell \ \leq \ k$.

The resulting algorithm is called Shift-Add, and it takes $O(n)$ worst-case time on short patterns and $O(mn \log m)$ in general. Actually, with slightly more complex code, it is possible to use only $\lceil \log_2(k+2) \rceil$ bits for the counters, to obtain a worst-case complexity of $O(mn \log k)$. This technique [9] was later combined with BNDM by Navarro and Raffinot [57], and the resulting algorithm is the fastest in practice.

A more interesting case, however, arises when differences can be not only character replacements, but also insertions and deletions. This problem has received a lot of attention since the sixties because of its multiple motivations in signal processing, computational biology, and text retrieval. A full recent survey on the subject is [51].

The classical solution to this problem, by Sellers [61], is to fill a matrix C of $m+1$ rows and $n+1$ columns, such that $C_{i,j}$ is the minimum number of differences necessary to convert $P_{1 \ldots i}$ into a suffix of $T_{1 \ldots j}$. It turns out that $C_{i,0} = i$, $C_{0,j} = 0$, and

$$C_{i+1,j+1} \quad = \quad \min(1 + C_{i,j+1}, 1 + C_{i+1,j}, \delta(P_i, T_j) + C_{i,j})$$

where $\delta(x,y)$ is 0 if $x = y$ and 1 otherwise. This gives an $O(mn)$ worst-case search algorithm, which needs only $O(m)$ space because only the previous column of C needs to be stored when we move to the next text position. A simple twist by Ukkonen [66] obtains $O(kn)$ average time. These form the so-called "classical" algorithms for the problem, and they are still unparalleled in flexibility when it comes to to solve more complex problems.

Let us quickly review the current state of the art regarding this problem from the theoretical side:

- The worst-case complexity is $\Omega(n)$, which can be achieved by using a DFA due to Ukkonen [66] that searches for P allowing k differences. However, this requires exponential space on m, as shown in the same paper.

- The worst-case complexity using space polynomial in m is unknown. The best algorithms are $O(kn)$ time, for example by Galil and Park [27].

- The average-case complexity is $O((k + \log_\sigma m)n/m)$, as proved by Chang and Marr [16]. In the same article an algorithm with that complexity is given, for $k/m < 1/3 - O(\sigma^{-1/2})$.

In practice, just like for simple string matching, the best current solutions are based either on bit-parallelism or on filtering.

3.6.1 Automata

The NFA for approximate searching of our pattern `"abracadabra"` permitting $k = 2$ differences can be seen in Figure 12. There are $k+1$ rows, zero to k. Row zero is exactly as in Figure 2, for exact searching. Row r represents having matched a prefix of P with r differences. Vertical arrows insert a character in P (or skip a character in T). Diagonal arrows move either by Σ, which replaces a character of P by that of T, or by ε, which does not consume any text character, so this deletes (skips) a character in P.

Although this automaton can be made deterministic, the DFA needs exponential space and time to build, so this option is not practical. Simulation in nondeterministic form requires, in principle, $O(mkn)$ time, which is not attractive compared to the classical $O(mn)$ solution.

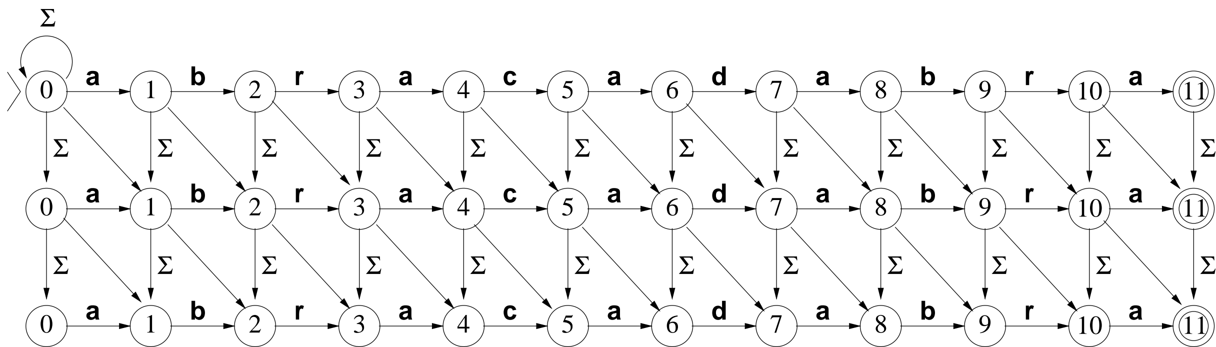

Figure 12: NFA that searches for "`abracadabra`" permitting up to 2 character insertions, deletions and substitutions. The initial state is marked "⟩". Unlabeled transitions move by $\Sigma \cup \{\varepsilon\}$.

Bit-parallelism, however, can be applied to solve this problem in various ways. The simplest way, by Wu and Manber [73], is to extend Shift-And so as to simulate each automaton row separately. Let us assume that bit mask R_i simulates the i-th row of the NFA. Hence, upon reading a new text character c, the R_i values can be updated to R'_i using the following formula:

$$\begin{aligned} R'_0 &\leftarrow ((R_0 << 1) \mid 1) \ \& \ B[c] \\ R'_{i+1} &\leftarrow ((R_{i+1} << 1) \ \& \ B[c]) \mid R_i \mid (R_i << 1) \mid (R'_i << 1) \mid 1 \end{aligned}$$

which simulates all the different arrows in the NFA, as it can be easily verified. This solution is simple and flexible and is implemented in *Agrep* and *Nrgrep*. On short patterns it is $O(kn)$ worst-case time, and $O(k\lceil m/w \rceil n)$ in general.

A faster algorithm, by Baeza-Yates and Navarro [11], cuts the NFA by diagonals rather than by rows. The interesting property of this partitioning is that different diagonals do not depend on the current value of each other (like row R'_{i+1} depends on row R'_i, which prevents computing them in parallel). Hence all the diagonals can be packed in a single computer word. This yields a worst-case time of $O(\lceil km/w \rceil n)$, which on short patterns is much faster than the simulation by rows, and currently is the fastest bit-parallel solution for short patterns.

Bit-parallelism has been also used to simulate the processing of matrix C instead of the NFA. Consecutive values of C differ only by ± 1, so each matrix column can be represented using $2m$ bits. Myers [48] used this property to design an algorithm whose complexity is $O(\lceil m/w \rceil n)$ in the worst case and $O(km/w)$ on average. As this is an optimal speedup over the best classical algorithms, it is unlikely that this complexity is improved by any other bit-parallel algorithm. However, on short patterns, the diagonal-wise simulation of the NFA is around 15% faster.

3.6.2 Filtration

A simple folklore property of approximate searching is that if P is split into $k+1$ pieces, then at least one piece can appear unaltered inside any approximate occurrence of P, since k differences cannot alter more than k pattern pieces. Then, multipattern search for the $k+1$ pieces of P points out all the potential occurrences of P. All those candidate positions are verified with a classical or bit-parallel algorithm.

The multipattern search takes time $O(kn \log_\sigma(m)/m)$ using an average-optimal algorithm. The time for verifications stays negligible as long as $k < m/(3 \log_\sigma m)$. This idea has been used several times, most prominently in *Agrep* (see [73]). It is simple to implement and yields extremely fast average search time in most practical cases.

However, the above algorithm has a limit of applicability that prevents using it when k/m becomes too large. Moreover, this limit is narrower for smaller σ. This is, for example, the case in several computational biology applications. In this case, the optimal algorithm by Chang and Marr [16] becomes a practical choice. It can be used up to $k/m < 1/3 - O(\sigma^{-1/2})$ and has optimal average complexity (however, it is not as fast as partition into $k+1$ pieces when the latter can be applied). Their idea is an improvement of previous algorithms that use q-grams (for example [69]).

This algorithm works as follows. The text is divided into consecutive blocks of length $(m-k)/2$. Since any occurrence of P is of length at least $m-k$, any occurrence contains at least one complete block. A number q is chosen, in a way that will be made clear soon.

The preprocessing of P consists in computing, for every string S of length q, the minimum number of differences needed to find S inside P. Preprocessing time is $O(m\sigma^q)$ and space is $O(\sigma^q)$.

The search stage proceeds as follows. For each text block Z, its consecutive q-grams $Z_{1\ldots q}$, $Z_{q+1\ldots 2q}$, ... are read one by one, and their minimum differences to match them inside P are accumulated. This is continued until (i) the accumulated differences exceed k, in which case the current block cannot be contained in an occurrence, and we can go on to the next block; (ii) we reach the end of the block, in which case the block is verified.

It is shown that on average one examines $O(k/q+1)$ q-grams per block, so $O(k+q)$ characters are examined. On the other hand, q must be $\Omega(\log_\sigma m)$ so that the probability of verifying a block is low enough. This yields overall time $O((k+\log_\sigma m)n/m)$ for $k/m < 1/3 - O(\sigma^{-1/2})$, with space polynomial in m.

For the same limit on k/m, another practical algorithm is obtained by combining a bit-parallel algorithm with a BNDM scheme, as done by Navarro and Raffinot [57] and Hyyrö and Navarro [37]. For higher k/m ratios, no good filtering algorithm exists, and the only choice is to resort to classical or pure bit-parallel algorithms.

3.6.3 Extensions

Recently, the average-optimal results for approximate string matching were extended to searching for r patterns by Fredriksson and Navarro [26]. They showed that the average complexity of the problem is $O((k+\log_\sigma(rm))n/m)$, and obtained an algorithm achieving that complexity. This is similar to the algorithm for one pattern, except that q-grams are preprocessed so as to find their minimum number of differences across all the patterns in the set. This time $q = \Theta(\log_\sigma(rm))$. They showed that the result is useful in practice.

The other choice to address this problem is to partition each pattern in the set into $k+1$ pieces and search for the $r(k+1)$ pieces in the set, verifying each piece occurrence for a complete occurrence of the patterns containing the piece. The search time is not optimal, $O(kn\log_\sigma(rm)/m)$, but it is the best in practice for low k/m values.

Approximate searching of complex pattern has been addressed by Navarro in *Nrgrep*, by combining Shift-And and BNDM with the row-wise simulation of the NFA that recognizes P. Each row of the NFA now recognizes a complex pattern P.

Finally, approximate searching of regular expressions can be done in $O(mn)$ time and $O(m)$ space, as shown by Myers and Miller [49], and in general in $O(mn/\log s)$ time using $O(s)$ space, as shown by Wu et al. and Navarro [75, 53].

4 Indexed Text Searching

In this section we review the main techniques to maintain an index over the text, to speed up searches. As explained, the text must be sufficiently static (few or no changes over time) to justify indexing, as maintaining an index is costly compared to a single search. The best candidates for indexing are fully static texts such as historical data and dictionaries, but indexing is also applied to rather more dynamic data like the Web due to its large volume.

4.1 Index Points

In sequential searching we assumed that the user wanted to find any substring of the text. This was because the algorithms are not radically different if this is not the case. However, indexing schemes depend much more on the retrieval model, so we pay some attention to the issue now.

Depending on the retrieval needs, the user must define which positions of the text will be indexed. Every position that must be indexed is called an *index-point*. Only text substrings starting at index points will be found by the search. For example, if the user wants to search only for whole words or phrases, then index points will be initial word positions. On the other hand, if the user wants to find any substring (for example in a DNA database), then every text position is an index point. Figure 13 shows a piece of text, with a possible set of index-points.

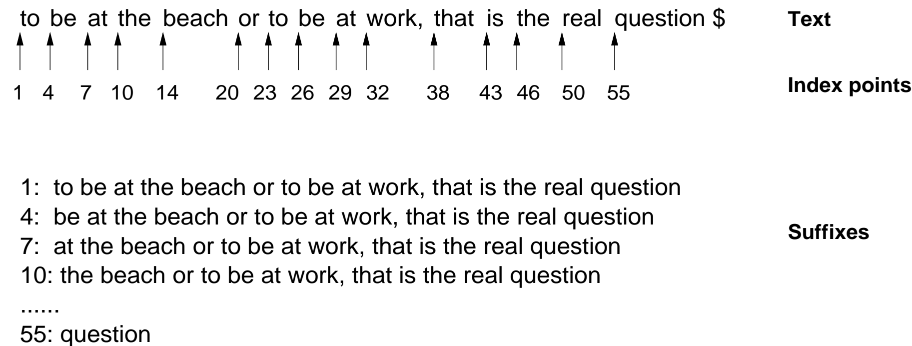

Figure 13: An example of index-points and their suffixes.

Under the view that the text is a set of its suffixes (Section 2.3), we take only the suffixes starting at index points as the representation of the text. If the search is based on finding the pattern as a prefix of any of those suffixes, it is automatically guaranteed that only index points will be searched. As an additional advantage, this reduces the space required by the index.

4.2 Architecture

To improve retrieval capabilities, text content is usually standardized via a set of user-defined transformations called *normalization*. Some common transformations on natural language text are: mapping of characters, such as upper case letters transformed to lower case; special symbols removed and sequences of multiple spaces or punctuation symbols reduced to one space (blank), among others; definition of word separator/continuation letters and how they are handled; common words removed using a list of *stop words*; mapping of strings, for example, to handle synonyms; numbers and dates transformed to a standard format; spelling variants transformed using Soundex-like methods; and word stemming (removing suffixes and/or prefixes).

Normalization is usually done over a logical view of the text, not on the source text files. The normalized text, together with the user-defined rules that determine which are index points, is

used to build the index.

The search pattern must also be normalized before searching the index for it. Once the text search engine retrieves the set of text positions that match the pattern, the original text is used to feed the user interface with the information that must be visualized.

Figure 14 shows the overall architecture.

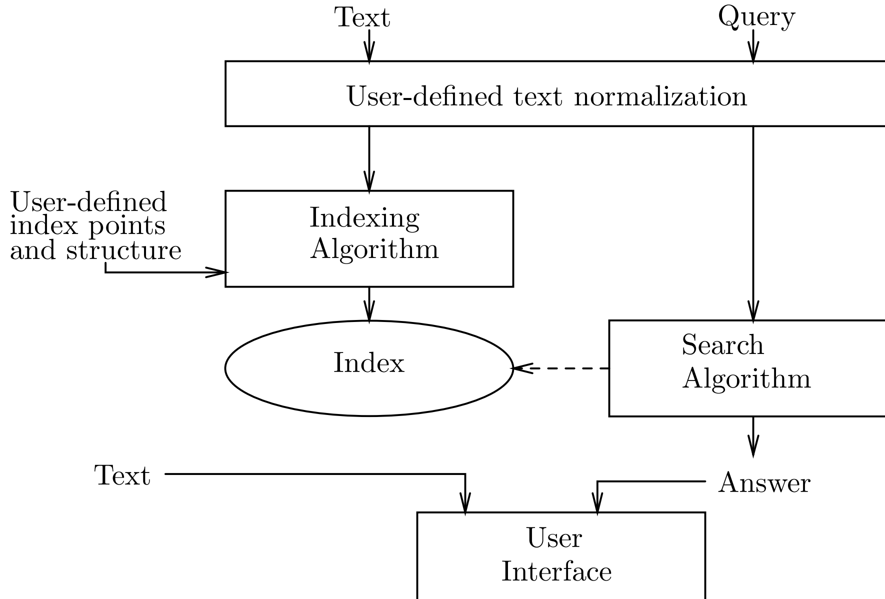

Figure 14: An indexed full text retrieval system.

4.3 Suffix Trees and Suffix Arrays

The cleanest implementation of the full text model uses the fact that the text can be seen as the set of its suffixes (that have been chosen as index points). Those suffixes can be stored into a trie data structure, and the result is called the *suffix trie* of T. The only difference is that, instead of storing the full suffix string $T_{i\ldots n}$ at each leaf of the trie, we store only its initial text position i.

As explained in Section 2.4, we can search for any string S in a trie in worst-case time $O(|S|)$. In particular, we can determine whether P is a prefix of any text suffix using the suffix trie of T in worst-case time $O(|P|)$. Since any occurrence of P in T is a prefix of some suffix of T, it follows that we can solve the search problem in optimal time, $O(m)$. Moreover, if we traverse all the leaves of the subtree arrived at in our search for P, the initial suffix positions stored in those leaves are precisely the initial occurrence positions of P in T.

Suffix tries permit searching for complex patterns as well. In general, the idea is that, since any occurrence of P will be the prefix of a text suffix, and all text suffixes can be found by traversing the suffix trie from the root, then one can backtrack over the suffix trie, following all its paths, using an automaton that recognizes $L(P)$. Every time the automaton dies (that is, goes to the sink state if it is a complete DFA or runs out of active states otherwise), we know that the current trie node is not a prefix of any occurrence of P and we abandon that branch. Every time the automaton recognizes a string, we report all the positions stored at the leaves of the subtree where recognition took place.

This algorithm is general and works in sublinear time (that is, $o(n)$) in most cases. For general regular expressions, the number of nodes traversed is $O(n^\lambda)$, where $0 \leq \lambda \leq 1$ depends on the structure of the regular expression, as shown by Baeza-Yates and Gonnet [10]. For approximate searching the time is exponential in k, as shown by Ukkonen [68], and it can be done $O(n^\lambda)$ again by appropriately partitioning the pattern into pieces, as shown by Navarro and Baeza-Yates [54]. This time λ will be smaller than 1 if $k/m < 1 - e/\sqrt{\sigma}$. Other types of sophisticated searches have been considered [43, 6, 34, 31].

The only problem with this approach is that the suffix trie may be rather large compared to the text. In the worst case, it can be of size $O(n^2)$, although on average it is of size $O(n)$. To ensure $O(n)$ size in the worst case, any unary path in the suffix trie is compacted into a single edge. Edges now will be labeled by a string, not only a character. There are several ways to represent the string labeling those edges in constant space. One is a couple of pointers to the text, since those strings are text substrings. Another is to simply store its first character and length, so the search skips some characters of P, which must be verified at the end against $T_{i...i+m-1}$, where i is the text position at *any* leaf in the answer. Since the resulting tree has degree at least two and it has $O(n)$ leaves (text suffixes) it follows that it is $O(n)$ size in the worst case. The result, due to Weiner [70], is called a *suffix tree* [6, 31].

Suffix trees can be built in optimal $O(n)$ worst-case time, as shown by McCreight and others [46, 67, 29], and can simulate any algorithm over suffix tries with the same complexity. So in practice they are preferred over suffix tries, although it is much easier to think of algorithms over the suffix trie and then implement them over the suffix tree.

Still, suffix trees are unsatisfactory in most practical applications. The constant that multiplies their $O(n)$ space complexity is large. Naively implemented, suffix trees require at least 20 bytes per index point (this can be as bad as 20 times the text size). Even the most space-efficient implementations, by Giegerich et al. [30], takes 9 to 12 bytes per index point. Moreover, its construction accesses memory at random, so it cannot be efficiently built (nor searched) in secondary memory. There exist other related data structures such as Blumer et al.'s DAWG (an automaton that recognizes all the text substrings, which can be obtained by minimizing the suffix trie) [22], and Crochemore and Vérin's Compact DAWG (the result of minimizing the suffix tree) [23].

A much more efficient way to reduce space is to collect the tree leaves, from left to right, into an array. The resulting array holds all the positions of all the text suffixes (that are index points). Moreover, since they have been collected left to right from the suffix tree, suffixes are ordered in lexicographical order in the array. This array was independently discovered by Gonnet [33], who called it *PAT array* [32], and by Manber and Myers, who gave it the more frequently used name *suffix array* [44].

Typically the suffix array takes 4 bytes per index point, which is close to the space usage of uncompressed inverted files when we index only word beginnings, and is 4 times the text size if we index every text character. The suffix array for our running example is given in Figure 15.

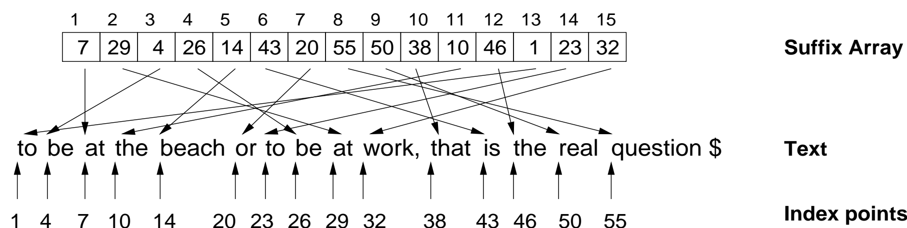

Figure 15: The suffix array for our example text.

Since suffixes are lexicographically sorted, all the suffixes that start with some prefix form an interval in the suffix array. This permits searching for P in T using two binary searches in the suffix array, for the initial and final interval position. This takes $O(m \log n)$ time in the worst case, which can be reduced to $O(\log n)$ by using some extra structures (that take more space) [44]. At the end, all the array entries in the resulting interval are reported. Figure 16 illustrates.

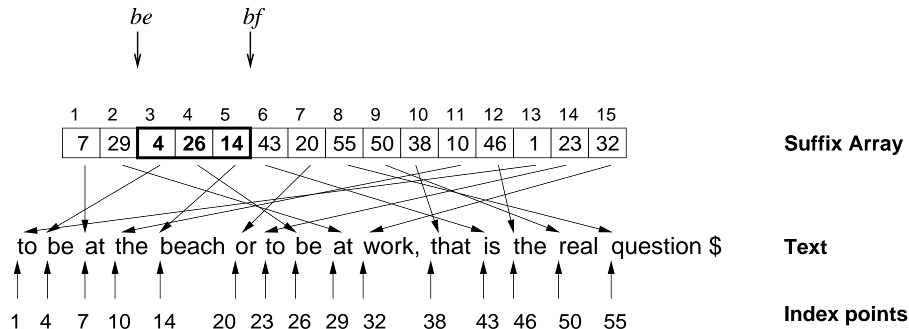

Figure 16: Searching for "be" in our example suffix array.

Indeed, it is not hard to see that suffix trie nodes correspond to suffix array intervals (of length 1 in the case of trie leaves). This mapping can be used to translate any algorithm designed over the suffix trie onto a suffix array. Every time we follow an edge in the suffix trie, we must move to a smaller subinterval of the current interval in the suffix array, at the cost of two binary searches. Hence, all the complexities obtained over suffix trees can be obtained on suffix arrays, at an $O(\log n)$ extra cost factor.

Construction of suffix arrays can be done with $O(n \log n)$ string comparisons by using any sorting algorithm (note that a string comparison may take long, especially if the text has long repeated substrings). A more sophisticated algorithm is given by Manber and Myers [44], which is $O(n \log n)$ worst-case time and $O(n \log \log n)$ average time. Recently, $O(n)$ worst-case time construction of suffix arrays has been achieved, simultaneously by Kim et al., Ko & Aluru, and Karkkainen & Sanders [39, 42, 38].

When the suffix array and/or the text do not fit in main memory, however, the above construction algorithms are impractical. The original algorithm devised by Gonnet et al. [32] is still one of the best to build the suffix array in secondary memory [20]. Given M main memory, they need $O(n^2 \log(M)/M)$ worst-case disk transfer time.

Once the suffix array has been built in secondary memory, some care must be taken in order to search it because random accesses to disk are expensive. A useful technique by Baeza-Yates et al. [8] is to build a *supra-index*, obtained by sampling the suffix array at regular intervals and collecting the first q characters of the sampled suffixes. The supra-index should be small and fit in main memory, so most of the binary search can be carried over the supra-index at no disk transfer cost. Figure 17 illustrates.

Yet another way to reduce space requirement of suffix tries is to prune them at depth q. That is, every different text q-gram (at the beginning of index points) will be collected, and its positions stored in a list associated to it, in increasing text position. The index can take much less space than the suffix array because those increasing text positions can be encoded differentially and compressed. It is still possible to search in this kind of indices by looking for q-grams of P and doing some verification directly against the text. We can index only some of the text q-grams in order to make the index even smaller, and still it is possible to search for long enough patterns. This permits having indices that are smaller than the text, although they

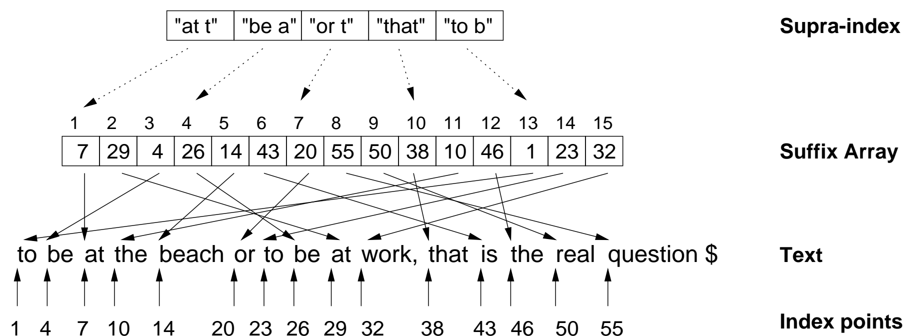

Figure 17: A supra-index for our example suffix array.

have higher search time and do not scale well for very large texts. This line has been pursued by Sutinen, Tarhio, and others [63, 59], and can cope with both exact and approximate searching. This is particularly attractive for computational biology applications.

4.4 Inverted Indices

When the text is (Western) natural language and users are satisfied with retrieving just words and phrases (not any substring), inverted indices are a very attractive choice [13]. An inverted index is simply a list of all the words appearing in the text (called the *vocabulary*), where each word has attached a list of all its text positions, in increasing text order. The set of different words is organized to permit fast searching, for example in a hash table. Figure 18 illustrates.

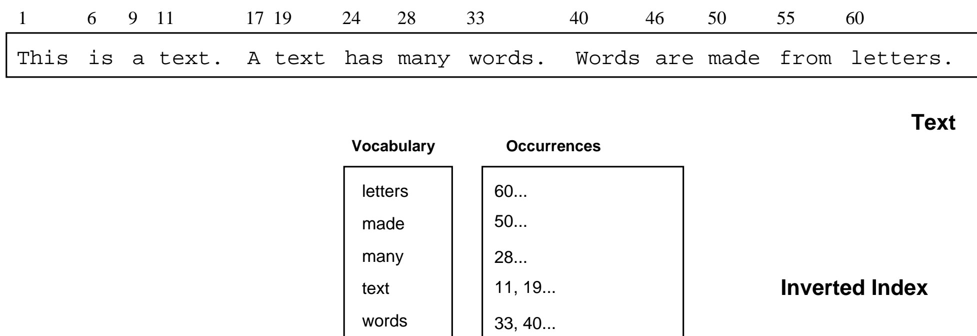

Figure 18: A text and its inverted index.

Searching for a simple word in an inverted index is as easy as looking for it in the vocabulary and returning its list. Searching for a phrase requires fetching the lists of each word and (essentially) intersecting them so as to find the places where they appear consecutively in the text. More complex searching, such as finding all the words that match some regular expression or approximate searching for a word, is possible by scanning the whole vocabulary using a sequential text search algorithm, returning all the words that matched the pattern, and computing the union of their occurrence lists, as first suggested by Manber and Wu [45]. This works well because the vocabulary is small compared to the text. It is empirically known that the vocabulary grows approximately like $O(\sqrt{n})$ [35].

We note, however, that this is just an intra-word search: if a spurious insertion in the text split a word into two, an approximate search for the word allowing $k = 1$ differences will not find the word. Similarly, regular expressions containing separators cannot be searched for with

this technique.

Inverted indices can be easily built in $O(n)$ time in main memory. In secondary memory, one has to build partial indices that fit in main memory and then merge them, which can be done in $O(n \log(n/M))$ disk transfer time. Searching in secondary memory is efficient because the lists are accessed sequentially. The vocabulary usually fits in main memory because it grows slowly compared to the text, as explained. This would not be the case if we had used instead fixed substrings of length q, as done in q-gram indices.

Compared with suffix arrays, inverted indices take in principle the same amount of space: one pointer per index point. However, since they store long lists of increasing positions that are accessed sequentially, differential encoding permits saving 50% of space at least. Moreover, it might also be convenient to deliver text positions in increasing order, rather than at random like the suffix array. On the other hand, suffix arrays permit searching for complex patterns that cannot be searched for using inverted indices. Even phrase searching is simpler (and can be faster) using a suffix array.

Finally, the space of the inverted index can be further reduced by combining it with sequential searching. The text is divided into blocks, and words point to the blocks where they appear, rather than their exact position. A text search must first obtain which blocks contain the word and then sequentially scan the blocks for the desired word or phrase. This produces a space-time tradeoff related to the block size. The idea was first proposed by Manber and Wu, who implemented it in an index called *Glimpse* [45] designed for personal use. Baeza-Yates and Navarro [12] proved that, by choosing appropriately the block size, the index can take $o(n)$ space and have $o(n)$ average search time simultaneously.

Finally, in natural language the text itself can be compressed up to 25%-30% of its original size and searched directly, without decompression, faster than the original text, as shown by Moura et al. [47]. This idea has been combined with inverted indices pointing to blocks by Navarro et al. [55].

References

[1] K. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16:1039–1051, 1987.

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

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] C. Allauzen, M. Crochemore, and M. Raffinot. Efficient experimental string matching by weak factor recognition. In *Proc. 12th Ann. Symp. on Combinatorial Pattern Matching (CPM'01)*, LNCS v. 2089, pages 51–72, 2001.

[5] C. Allauzen and M. Raffinot. Factor oracle of a set of words. Technical report 99–11, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999.

[6] A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 85–96. Springer-Verlag, 1985.

[7] R. Baeza-Yates. Improved string searching. *Software-Practice and Experience*, 19(3):257–271, 1989.

[8] R. Baeza-Yates, E. Barbosa, and N. Ziviani. Hierarchies of indices for text searching. *Information Systems*, 21(6):497–514, 1996.

[9] R. Baeza-Yates and G. Gonnet. A new approach to text searching. In *Proc. 12th Ann. Int. ACM Conf. on Research and Development in Information Retrieval (SIGIR'89)*, pages 168–175, 1989. (Addendum in ACM SIGIR Forum, V. 23, Numbers 3, 4, 1989, page 7.).

[10] R. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM*, 43(6):915–936, 1996.

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

[12] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. *Journal of the American Society for Information Science*, 51(1):69–82, 2000.

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

[14] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnel. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

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

[16] W. Chang and T. Marr. Approximate string matching with local similarity. In *Proc. 5th Ann. Symp. on Combinatorial Pattern Matching (CPM'94)*, LNCS v. 807, pages 259–273, 1994.

[17] R. Cole. Tight bounds on the complexity of the Boyer-Moore string matching algorithm. In *Proc. 2nd ACM-SIAM Ann. Symp. on Discrete Algorithms (SODA'91)*, pages 224–233, 1991.

[18] L. Colussi, Z. Galil, and R. Giancarlo. The exact complexity of string matching. In *Proc. 31st IEEE Ann. Symp. on Foundations of Computer Science*, volume 1, pages 135–143, 1990.

[19] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. 6th Int. Coll. on Automata, Languages and Programming (ICALP'79)*, LNCS v. 71, pages 118–132, 1979.

[20] A. Crauser and P. Ferragina. On constructing suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[21] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

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

[23] M. Crochemore and R. Vérin. Direct construction of compact directed acyclic word graphs. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM'97)*, LNCS v. 1264, pages 116–129, 1997.

[24] M. Fischer and M. Paterson. String matching and other products. In *Proc. 7th SIAM-AMS Complexity of Computation*, pages 113–125. American Mathematical Society, 1974.

[25] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992.

[26] K. Fredriksson and G. Navarro. Average-optimal multiple approximate string matching. In *Proc. 14th Ann. Symp. on Combinatorial Pattern Matching (CPM'03)*, LNCS v. 2676, pages 109–128, 2003.

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

[28] Z. Galil and J. Seiferas. Linear–time string matching using only a fixed number of local storage locations. *Theoretical Computer Science*, 13:331–336, 1981.

[29] R. Giegerich and S. Kurtz. From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[30] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proc. 3rd Workshop on Algorithm Engineering (WAE'99)*, LNCS v. 1668, pages 30–42, 1999.

[31] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures – In Pascal and C*. Addison-Wesley, 2nd edition, 1991.

[32] G. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: Pat trees and pat arrays. In W. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Algorithms and Data Structures*, chapter 5, pages 66–82. Prentice-Hall, 1992.

[33] G.H. Gonnet. PAT 3.1: An efficient text searching system, User's manual. UW Centre for the New OED, University of Waterloo, 1987.

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

[35] H. Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, 1978.

[36] R. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10(6):501–506, 1980.

[37] H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM'02)*, LNCS 2373, pages 203–224, 2002.

[38] J. Karkkainen and P. Sanders. Simple linear work suffix array construction. In *ICALP*, to appear, 2003.

[39] D. Kim, J. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Ann. Symp. on Combinatorial Pattern Matching (CPM'03)*, LNCS v. 2676, pages 186–199, 2003.

[40] J. Kim and J. Shawe-Taylor. Fast string matching using an n-gram algorithm. University of London, 1991.

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

[42] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Ann. Symp. on Combinatorial Pattern Matching (CPM'03)*, LNCS v. 2676, pages 200–210, 2003.

[43] U. Manber and R. A. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Information Processing Letters*, 37(3):133–136, 1991.

[44] U. Manber and E. W. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[45] 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.

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

[47] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

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

[49] E. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

[50] E. W. Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(2):430–448, 1992.

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

[52] G. Navarro. Nr-grep: a fast and flexible pattern matching tool. *Software Practice and Experience*, 31:1265–1312, 2001.

[53] G. Navarro. Approximate regular expression searching with arbitrary integer weights. Technical Report TR/DCC-2002-6, Department of Computer Science, University of Chile, July 2002.

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

[55] G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[56] G. Navarro and M. Raffinot. Fast regular expression search. In *Proc. 3rd Workshop on Algorithm Engineering (WAE'99)*, LNCS v. 1668, pages 199–213, 1999.

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

[58] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

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

[60] R. Pinter. Efficient string matching with don't-care patterns. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 11–29. Springer-Verlag, 1985.

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

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

[63] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96)*, LNCS v. 1075, pages 50–61, 1996.

[64] J. Tarhio and H. Peltola. String matching in the DNA alphabet. *Software Practice and Experience*, 27(7):851–861, 1997.

[65] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.

[66] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1–3):132–137, 1985.

[67] E. Ukkonen. Constructing suffix trees on-line in linear time. In *Proc. 12th IFIP World Computer Congress (IFIP'92)*, pages 484–492. North-Holland, 1992.

[68] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. 4th Annual Symposium on Combinatorial Pattern Matching (CPM'93)*, LNCS v. 520, pages 228–242, 1993.

[69] 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.

[70] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[71] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, 2nd edition, 1999.

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

[73] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35:83–91, 1992.

[74] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Report TR-94-17, Department of Computer Science, University of Arizona, 1994.

[75] S. Wu, U. Manber, and E. Myers. A subquadratic algorithm for approximate regular expression matching. *Journal of Algorithms*, 19(3):346–360, 1995.

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