Text Retrieval: Theory and Practice

Ricardo A. Baeza-Yates

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

Abstract

We present the state of the art of the main component of text retrieval systems: the searching engine. We outline the main lines of research and issues involved. We survey recently published results for text searching and we explore the gap between theoretical vs. practical algorithms. The main observation is that simpler ideas are better in practice.

1597 SHAKS. *Lover's Compl.* 2
From off a hill whose concaue wombe reworded
A plaintfull story from a sistring vale.
OED2, **reword**, **sistering** [1]

1 Introduction

Full text retrieval systems are becoming a popular way of providing support for on-line text. Their main advantage is that they avoid the complicated and expensive process of semantic indexing. From the end-user point of view, full text searching of on-line documents is appealing because a valid query is just any word or sentence of the document. However, when the desired answer cannot be obtained with a simple query, the user must perform his/her own semantic processing to guess which strings are likely to locate the desired parts of the text. Thus, full text systems shift the burden of indexing from the database designer to the systems user. Hence, the user is at a disadvantage because he or she is typically not an expert in indexing or morphology, and also may not know the characteristics of the text being searched.

Despite the problem mentioned above, people are often successful and satisfied with their use of full text systems. One obvious reason is that a full text system is better than no system at all. However, a more important observation is that the measures of precision and recall are only appropriate when the task is retrieval of all relevant material, as in a classical library research

*This work was partially supported by grant C-11001 from Fundación Andes and has not been submitted for publication elsewhere.

[1] First recorded reference to semi-infinite string or *sistring*.

situation. On-line searching, on the other hand, often involves looking for just a single fact from a large collection of data. For example, finding how to use a command from the on-line operating system manual does not require finding all relevant information about the command, or finding all information about the command syntax. Thus, it is possible to solve many fact-finding problems using a full text system, even though it may provide inadequate recall, especially when the data itself is redundant [RF90].

The main component of a free-text retrieval system is the text searching engine. Formally, the text searching problem can be defined as follows: Given a *text* string t and a *query* (pattern) q, locate either one occurrence (for example, the first), or all occurrences, of q in t; or return "none present". We are interested in solving this problem for patterns expressed in various query languages, ranging from a simple string to regular expressions; and for either *plain text*, that is, just a piece of text; or *preprocessed text*, that is, a piece of text plus some kind of index.

In section 2 we review the main concepts of text-retrieval systems, text processing and searching. In the subsequent sections we survey recent results for many variations of the problem, for both plain and preprocessed text. For a complete set of references before 1991 and an introduction to the basic algorithms, we refer the reader to [GBY91, Chapter 7].

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. It is not necessarily textual data, is just a sequence of characters. When the text is large, fast searching is provided by building an *index* of the text, which is used in subsequent searches. To improve retrieval capabilities and to simplify several problems addressed in the next section, the index is built over a *normalized* text. Text normalization is achieved by processing the original text using a user-defined set of transformations.

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*. Index-points are usually defined by specifying a letter just before the index-point, and the letter that identifies the beginning of an index-point. Let $\sqcup$ stand for a blank symbol, A for an alphabetical symbol, and X for any symbol. For example, we can define all words of the text as starting points with $\sqcup A$. This is the typical case. However, we may want to index any position of the text using XX. That is the case for a DNA database. The index-points are specified over the normalized text. In addition to the index-points, we may specify pieces of the text that will not be indexed (for example, if a text contains images or non-indexable data). Moreover, we may index different pieces of text in different ways. Figure 1 shows a piece of text, along with a possible set of index-points.

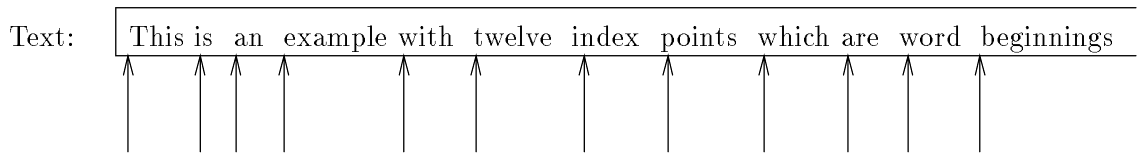

Text:
This is an example with twelve index points which are word beginnings

Figure 1: An example of index-points.

The *query* is a command issued to the retrieval system which returns all occurrences of text matching it which are present in the index (that is, index-points). Each result is called a *hit*. Note that only pieces of text starting in an index-point can be retrieved by a query, and thus only those can be hits. The query is also normalized before using the index to find the *answer*. The answer and the original text are used by the user interface to display the actual matches. All the inter-relationships described so far are shown in Figure 2.

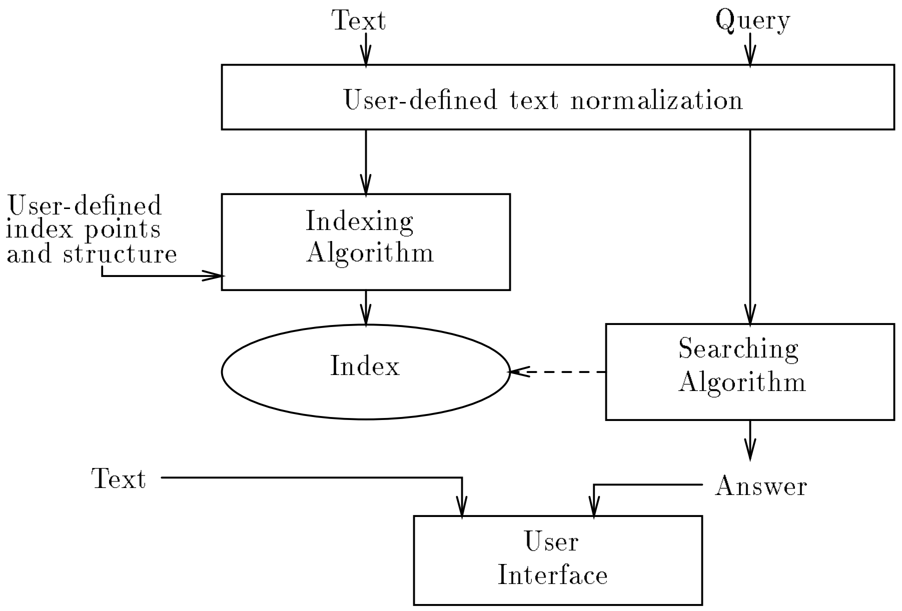


Figure 2: A Full Text Retrieval System.

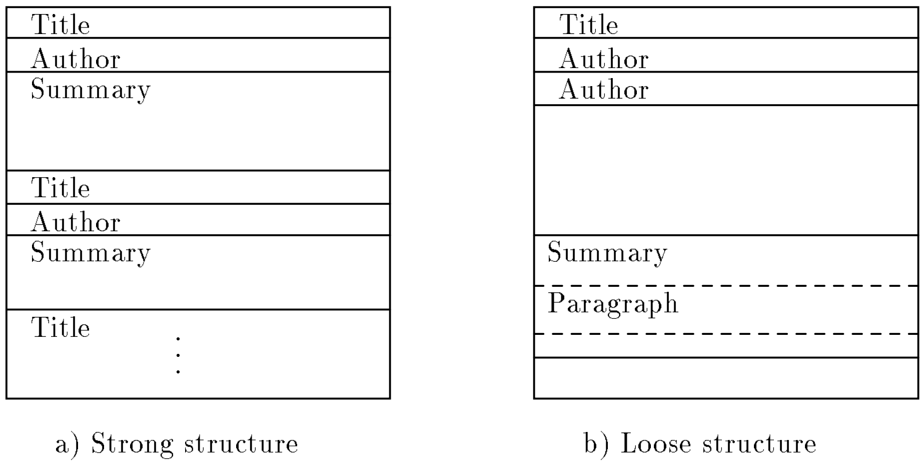


Figure 3: Types of text structure.

Text databases can be classified according to their update frequency as *static* or *dynamic*. Static text does not suffer changes, or its update frequency is very low. Thus, static databases may have new data added and can still be considered fixed. Examples of static text are historical data (no changes) and dictionaries.

For this type of database, it is worthwhile to *preprocess* the text and to *build indices* or other data structures to speed up the query time. The preprocessing time will be amortized during the time the text is searched before the data changes again. For example, this is the case when searching the Oxford English Dictionary (OED). The computerization of this dictionary was carried out between 1985 and 1990 at the University of Waterloo [BGT88].

On the other hand, dynamic text is text that changes too frequently to justify preprocessing, for example, in text-editing applications. In this case, we must use search algorithms that scan the text sequentially, or that have efficient algorithms to update the index.

Text 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. Gonnet [Gon83] used the term *unstructured database* to refer to this type of data. Examples of such collections are: dictionaries, legal cases, articles on wire services, scientific papers, etc.

Text can instead be structured as a sequence of words. Each *word* is a string which does not include any symbol from a special separator set. For example, a "space" is usually an element of such a set.

Optionally, a text may have a structure. The text can be divided into *documents*. The text itself may be considered as one document. Each document may be divided into *fields*. If we force the fields of a document to be fixed in order and number, we say that we have a *strong* structure. On the other hand, if we allow fields to overlap, to nest, or to cover only part of a document, we say that the structure is *loose*. Figure 3 shows an example for each case.

2.1 The Semi-Infinite String Model

Let us assume that the text to be searched is a single string and padded at its right end with an infinite number of null (or any special) characters. A *semi-infinite string* (*sistring*)[Gon88] is the sequence of characters starting at any position of the text and continuing to the right. For example, if the text is

```
The traditional approach for searching a text is ...
```

the following are some of the possible sistrings:

```
The traditional approach for searching ...
he traditional approach for searching a ...
e traditional approach for searching a ...
onal approach for searching a text is ...
```

Two sistrings starting at different positions are always different. To guarantee that no one semi-infinite string be a prefix of another, it is enough to end the text with a unique end-of-text symbol that appears nowhere else. Thus, sistrings can be unambiguously identified by their starting position. The result of a lexicographic comparison between two sistrings is based on the text of the sistrings, not their positions. Index-points are a subset of the n sistrings of a text.

2.2 Text Normalization

The main goal for text normalization is to standardize its content and to ease searching [Gon87a, FBY92]. Below we give a list and some uses of common operations:

- Mapping of characters, such as upper case letters transformed to lower case letters.
- 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. This is useful to handle synonyms.
- Numbers and dates transformed to a standard format [Gon87b].
- Spelling variants transformed using Soundex-like methods.
- Word stemming (removing suffixes and/or prefixes).

Unfortunately, these filtering operations may also have some disadvantages. Before consulting the database, any query must also be filtered. Furthermore, it is not possible to search for common words, special symbols or upper case letters, nor to distinguish text fragments that have been mapped to the same internal form. However, we will likely find more than what we want, but *never less than* the desired result.

3 Retrieval Algorithms for Plain Text

The basic problem is string searching and its variants. The classical problem is to find a pattern of length m in a text of length n. The main variations are approximate string matching and two dimensional searching. In approximate string matching we allow the pattern to differ from the text up to a maximum number of errors. Depending on the definition of an error, we have different problems. If an error is a character mismatch between the pattern and the text, we have string matching with mismatches. If we also allow insertions and deletions of characters in the pattern or the text, we have string matching with differences.

Instead of presenting the algorithms divided by problems, we present the main ideas used to solve these problems. For a complete bibliographic account and the explanation of every problem, we refer the reader to [GBY91, Chapter 7].

3.1 Comparison based Algorithms

Classical text searching algorithms are based on character comparisons. However, it seems that other computational models are as good or perhaps better than just using character comparisons.

The first important result on string searching is the so called Knuth-Morris-Pratt algorithm [KMP77]. This algorithm, discovered around 1970, was the first to achieve linear worst-case time.

In [KMP77] there is a very nice historical account of this algorithm. The second result is due to Boyer and Moore [BM77] who developed a fast algorithm on average, achieving $O(\log mn/m)$ comparisons. After that, several papers were published on the topic. However, recently it seems that there are an increasing number of people working on the topic and, what is better, improving previous algorithms or proving some previously open problems.

Among the theoretical results, we should mention:

- a $3n$ upper and lower bound for the worst case number of comparisons of the Boyer-Moore algorithm [Col91a] and several papers dealing with the average case analysis of this algorithm [BYGR90, Sch88];
- the $4/3n$ upper bound in the worst case number of comparisons with a $(1 + 1/(2m))n$ lower bound for any comparison-based string matching algorithm [CGG90];
- some results concerning the maximal number of states of a Boyer-Moore type automaton [BYR90, Cho90, Bru91], a problem that is still open;
- improvement of several variations of approximate string matching [WMM91];
- the derivation of classical and improved algorithms through formal proofs of program correctness [Col91b];
- adaptive multiple string searching [AM91].

On the practical side, the main results are:

- improvements to the Boyer-Moore algorithm, through alphabet transformations [BY89b, BY89a], sensitivity to the distribution of the text [BY89b, BY89c, Sun90], word theory [CP91], adaptivity [Smi91], and a taxonomy of string searching algorithms [HS91];
- fast algorithms for long patterns based on n-grams [KST91];
- several new algorithms for string matching with mismatches, based on the Boyer-Moore approach, which are faster and more practical [BY89b, BYG92, GL89, TU90];
- algorithms based on partitioning the pattern [WM91, BYP91];
- faster practical algorithms for string matching with errors, with $O(kn)$ worst case searching time or better on average, where k is the maximum number of errors allowed [GP90, UW89, TU90, CL90, Ukk91]. Empirical and theoretical comparisons of these algorithms can be found in [CL91b, JTU91]

One generalization of string searching is to increase the number of dimensions of the text and the pattern. There have been several results for the two dimensional case, which has a direct application to problems on bit-mapped screens and digitized images. Among them we have faster exact matching algorithms [ZT89, BYR90] and some algorithms for approximate matching [ALV90, AF91, AL91, LV92]. Logarithmic search time can be achieved using special indices [Gon88].

3.2 Bit-Parallelism

Consider the following simple parallel algorithm for string searching using m processors. Processor j knows, at any time, whether or not the last j characters of the text that it has seen of the text are equal to the first j characters of the pattern. Consequently, each processor compares the current character that it sees in the text to the $(j+1)$-th character of the pattern, and decides whether the last $j+1$ characters are equal. Processor $m-1$ then, outputs a signal if the last m characters were equal. The process continues by advancing the reading head, and processor j ($j < m-1$) sends its state to processor $j+1$ (after this, processor 0 resets its state), and repeats the process. This scheme is shown in Figure 4, and the search time is n character inspections. The state of every processor can be just a bit if we want to find exact matches or the actual count of matched (or mismatched) characters if we want to solve the string searching with mismatches problem.

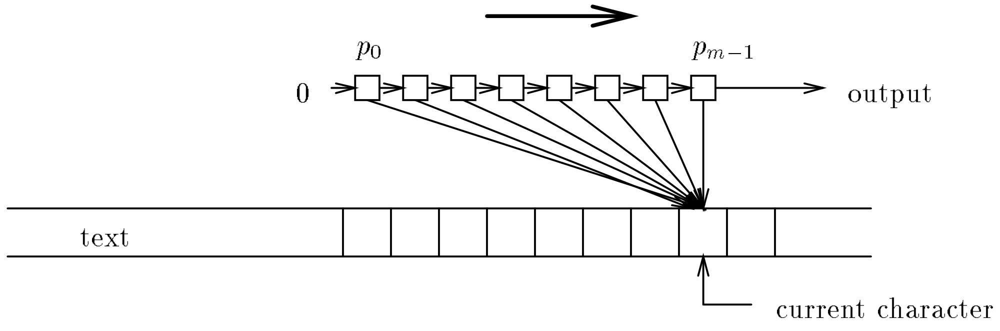


Figure 4: A simple parallel algorithm

Although this parallel algorithm is not optimal, the crucial point is that the processors can be simulated by a bit sequence of size m for exact matching, updating all the states by or-ing (or and-ing) the sequence with a precomputed value for each symbol in the alphabet. (These values depend only on the pattern.) After that, the final step is simulated by shifting the state by one bit. In pseudo-language we have

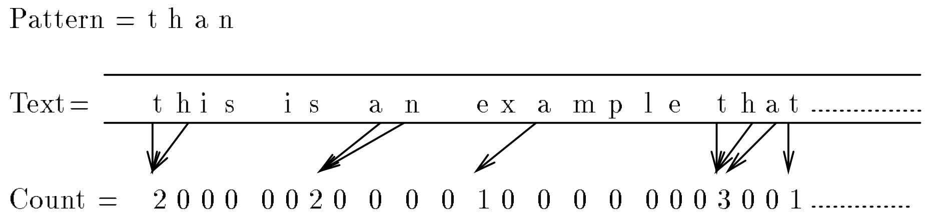
```
newstate = shift left (state or Table[current character])
```

This is extremely fast if m is less or equal than the word size of the machine used.

This idea was introduced by Baeza-Yates and Gonnet [BY89b, BYG89b] and can be easily extended to string matching with classes (every element of the pattern is a set of symbols, rather than one symbol), string matching with mismatches and multiple string searching [BYG89b]. Recently [WM91, WM92] the idea has been extended to string matching with errors, and implemented as "agrep", the fastest tool to searching through files, even when we allow errors.

3.3 Counting Algorithms

Imagine that there is a counter, c_i, for every position of the text. The counters are initialized to zero, and as each character of the text is read and examined, its position in the pattern (if it is a character in the pattern) is used to find a counter corresponding to the beginning of a potential occurrence in the text, and this counter is incremented. If the character is repeated in the pattern, we increment that many counters. If there is an instance of the pattern with no mismatches, the counter at the position of this instance of the pattern will be incremented m times. The number of mismatches is equal to m minus the value of the counter, so we can also use this idea for string matching with mismatches. Only m counters are necessary at any one time, so the counters can be implemented with an array of size m that is traversed in a circular manner. This idea is shown in Figure 5. In practice, because patterns do not have many repeated symbols, the running time is $O(n)$ with $O(m + |\Sigma|)$ extra space, where Σ denotes the underlying alphabet.



Pattern = t h a n
Text= t h i s i s a n e x a m p l e t h a t ....
Count = 2 0 0 0 0 0 2 0 0 0 0 1 0 0 0 0 0 0 0 3 0 0 1 ....

Figure 5: Searching example for the counting approach.

This idea is implicit in [MW89], where is used for string matching with insertions and deletions only. It was independently discovered in [BYP91], which presents the algorithm mentioned above for mismatches. This technique is also very fast and does not use character comparisons.

4 Retrieval Algorithms for Indexed Text

In this section we structure the algorithms according to the data structure used as index for the text. An index is additional information about the text to achieve faster algorithms than for plain text.

If the index allows us to reconstruct the text, we say that it is *complete*. In fact, in that case, we do not have to store the text after the index is built because the index is just an alternate representation for the text. One simple example of an index would be a compressed version of a text. To retrieve data we may use algorithms for plain text while the index is scanned and decompressed. However, in most cases, the text must be stored to achieve fast answer time.

Inverted files is the traditional technology of standard text retrieval systems. There have been no significant improvements or new breakthroughs in this technology. We refer the reader to [FBY92] for a detailed treatment of standard information retrieval algorithms.

4.1 Signature Files

Signature files is an application of hashing to sequential text searching. Such compression techniques are not enough to shrink the text to sizes where a sequential search achieves reasonable answer time, hashing is used to build a compact version of the text (between 10% and 20% of the original size). For this task, the text is divided in words and block of words [Fal88], transforming the text into a sequence of bits. To search in this index, first the query is transformed to a bit string, and that string is sequentially searched in the index (or signature file). Because more than one word can be mapped to the same bit string, we may obtain undesired answers, so that every potential answer must be checked again with the original text.

By an adequate selection of parameters, reasonable performance can be achieved in medium size files yielding a low probability of undesired answers (or false drops) [FC84, FC87, SDRK87]. However, this method is not suitable for large files or for applications where it is not possible to divide the text into words, as in protein sequence databases. A complete survey on the different variations of signature files can be found in [FBY92].

4.2 PAT Arrays

Another traditional index is a digital tree or trie. Trie search is based on the digital decomposition of the sistrings stored in it. If the alphabet is ordered, we have a lexicographically ordered tree. The root of the trie uses the first character, the children of the root use the second character, and so on. If the remaining subtrie contains only one string, that string's identity its stored in an external node.

Figure 6 shows a binary trie (binary coded alphabet) for the string "01100100010111..." after inserting the sistrings that start from positions 1 through 8. (In this case, the sistring's identity is represented by its starting position in the text.)

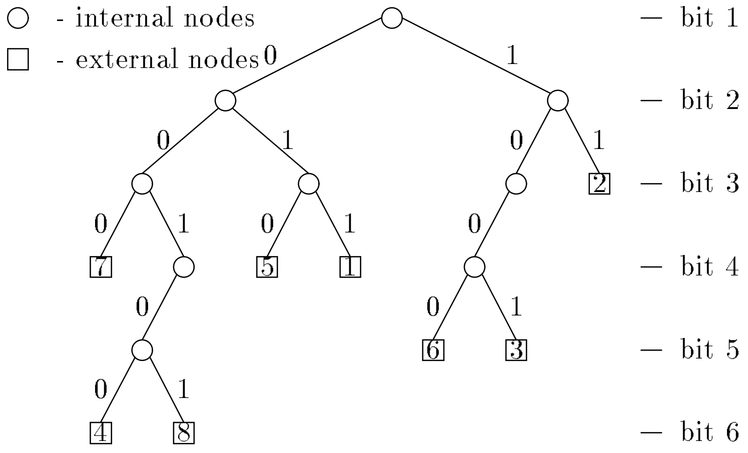


Figure 6: Binary trie (external node label indicates position in the text) for the first eight sistrings in "01100100010111...".

A binary trie may have unary nodes. To avoid unary nodes, we collapse them by indicating in every binary node which bit of the string should be examined. In this was, we make sure that we only need $n-1$ internal nodes and n external nodes, n being the number of index-points or sistrings. This variant is called a Patricia tree.

This data structure easily supports prefix and range searching, as well as proximity and regular expression searching [GBY91].

Although a Patricia tree needs $O(n)$ space, the actual constant is important in practical applications. We need at least two pointers for every internal node, and one pointer for every external node. This means $12n$ bytes for n index points. For the OED this is more than a gigabyte.

A simple and efficient solution to this problem is to store only the external nodes. Thus, the whole index degenerates into a single array (PAT array) of external nodes ordered lexicographically by sistrings. With some additions this idea was independently discovered by Manber and Myers [MM90] where it is called suffix arrays. The size of the index is now $4n$ bytes, where n is the number of index points. For the OED we have approximately 475Mb, if we only index word beginnings.

There is a straightforward argument that shows that these arrays contain most of the information we had in the Patricia tree at the cost of a factor of $\log_2 n$. The argument simply says that for any interval in the array which contains all the external nodes that would be in a subtree, in $\log_2 n$ comparisons in the worst case, we can divide the interval according to the next bit which is different. The sizes of the subtrees are trivially obtained from the limits of any portion of the array, so the only information that is missing is the longest-repetition bit, which is not possible to represent without an additional structure. Any operation on a Patricia tree can be simulated in $O(\log n)$ accesses. Figure 7 shows this structure.

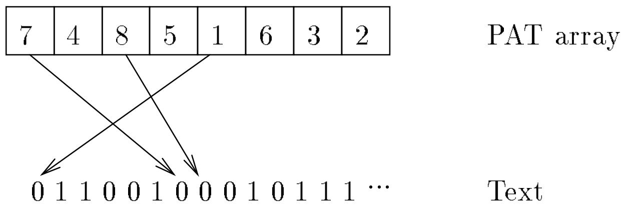


Figure 7: A PAT array

It turns out that it is not necessary to simulate the Patricia tree for prefix and range searching, so we obtain an algorithm which is $O(\log n)$ instead of $O(\log^2 n)$ for these operations. Actually prefix searching and range searching become more uniform. Both can be implemented by doing an indirect binary search over the array with the results of the comparisons being less than, equal to, or greater than. In this way searching takes at most $2\log_2 n - 1$ comparisons and $4\log_2 n$ disk accesses (both in the worst case). In summary, prefix searching and range searching can be done in time $O(\log_2 n)$ with a storage cost which is exactly one word (a sistring pointer) per index point (sistring). This represents a significant economy in space at the cost of a modest deterioration in access time [GBYS92].

For regular expression searching, the time increases by a factor of $O(\log n)$ [BYG89a]. Details about proximity searching (searching for the occurrence of a string near other string) are given by Manber and Baeza-Yates [MBY91]. For this problem, it is possible to know the number of occurrences in logarithmic time, and all the answers in $O(n^{1/4})$ time plus the number of occurrences. In both cases, $O(n)$ extra space is needed.

It is possible to build PAT arrays in $O(n \log n)$ time. However, practical algorithms have to access the text in a sequential way only for large databases. For example, for the OED, we have 119 million index points. Even if we were using an algorithm which used a single random access per entry point, the total disk time would be more than a month. It is currently possible to index the OED (600Mb) during a weekend [GBYS92]. This technology is currently used only in two commercial systems [Faw89, Ars91].

4.3 Approximate Searching and DNA Applications

One of the main applications of character-based index is in protein sequence analysis and related problems. Currently, there are two main approaches: the oldest one is biological research using the computer as a tool utilizing traditional algorithms. The main problem has been the large amount of data to be processed and the complexity of the underlying analysis. The other approach is computer science research in which algorithms are applied to problems like sequence alignment. However, many times the problems considered are either too theoretical or too simple. Nevertheless, the gap between these two worlds is being shortened. Some examples are [JU91, CL91a, Mye].

One remarkable project has been the matching of a complete protein database to itself [GB91, GCB91] at the Swiss Technology Institute. The main result of this process has been to compute better cost alignment matrices and deletion gap functions based on the result of all-against-all matching. This has shown that previous cost matrices and gap functions were based on partial empirical results, lacking theoretical foundations. This has been possible through new algorithms for approximate string matching based on PAT arrays, which achieve expected sublinear time and are practical [BYG90, BYG].

5 Summary

For most text searching problems, there is no algorithm which is the best for every application. The choice of algorithm will depend on the input, such as the type of text used (for example, English vs. DNA data), the size of the text and/or pattern, etc. Also, recent results show that further research should be undertaken to explore

- Non-comparison based algorithms (for example, based on arithmetical and logical operations).
- Hybrid and adaptive algorithms.
- Good average case algorithms (that is, practical).
- New uses for old concepts (for example, n-grams).
- Reductions to simpler problems.
- New indices for text, tailored to special problems (for example, approximate matching).

Traditionally, indices are based on words, fields, and documents. Although it is useful to exploit the structure of the text, recent results suggest that string-based indices are much more flexible and powerful. In particular, the main differences are:

- A given structure (documents, fields, etc.) for the text is not necessary.
- The concept of word is not used, obtaining an index where prefixes or phrases can also be retrieved.
- Emphasis is given to retrieve also text positions (highlight of occurrences), and not only documents.

- If desirable, every possible position of the text can be indexed. This facility is usually not supported in traditional systems. This is very important when the text is just a sequence of symbols, for example, a DNA database [GB91]. Beware that in this case, the answers will be any possible substring, and not just word beginnings.

This approach is used in PAT arrays. The author is currently developing character-based signature files combining hashing and compression techniques.

References

[AF91] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. In *SODA'91*, pages 212–223, San Francisco, CA, Jan 1991.

[AL91] A. Amir and G. Landau. Fast parallel and serial multidimensional approximate array matching. *Theoretical Computer Science*, 81:97–115, 1991. Also as report CS-TR-2288, Dept. of Computer Science, Univ. of Maryland, 1989.

[ALV90] A. Amir, G.M. Landau, and U. Vishkin. Efficient pattern matching with scaling. In *SODA*, pages 344–357, San Francisco, Jan 1990. ACM-SIAM.

[AM91] A. Amir and Farach M. Adaptive dictionary matching. In *32st FOCS*, San Juan, PR, Oct 1991. IEEE.

[Ars91] Ars Innovandi. *Search City*, 1991.

[BGT88] D.L. Berg, G.H. Gonnet, and F.W. Tompa. The New Oxford English Dictionary Project at the University of Waterloo. Technical Report OED-88-01, UW Centre for the New Oxford English Dictionary, 1988.

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

[Bru91] Véronique Bruyère. Thèse annexe, automates de boyer-moore. Technical report, Institut de Mathématique et d'Informatique, Université de Mons-hainaut, 1991.

[BY89a] R. Baeza-Yates. Improved string searching. *Software-Practice and Experience*, 19(3):257–271, 1989.

[BY89b] R.A. Baeza-Yates. *Efficient Text Searching*. PhD thesis, Dept. of Computer Science, University of Waterloo, May 1989. Also as Research Report CS-89-17.

[BY89c] R.A. Baeza-Yates. String searching algorithms revisited. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Workshop in Algorithms and Data Structures*, pages 75–96, Ottawa, Canada, August 1989. Springer Verlag Lecture Notes on Computer Science 382.

[BYG] R. Baeza-Yates and G.H. Gonnet. Approximate string matching in sublinear expected time. in preparation.

[BYG89a] R. Baeza-Yates and G.H. Gonnet. Efficient text searching of regular expressions. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *ICALP'89, Lecture Notes in Computer Science 372*, pages 46–62, Stresa, Italy, July 1989. Springer-Verlag. Also as UW Centre for the New OED Report, OED-89-01, Univ. of Waterloo, April, 1989.

[BYG89b] R. Baeza-Yates and G.H. Gonnet. A new approach to text searching. In *Proc. of 12th ACM SIGIR*, pages 168–175, Cambridge, Mass., June 1989. (Addendum in ACM SIGIR Forum, V. 23, Numbers 3, 4, 1989, page 7.). To appear in *Communications of CACM*.

[BYG90] R.A. Baeza-Yates and G.H. Gonnet. All-against-all sequence matching. Dept. of Computer Science, Universidad de Chile, 1990.

[BYG92] R. Baeza-Yates and G.H. Gonnet. Fast string matching with mismatches. *Information and Computation*, 1992. (to appear). Also as Tech. Report CS-88-36, Dept. of Computer Science, University of Waterloo, 1988.

[BYGR90] R. Baeza-Yates, G. Gonnet, and M. Regnier. Analysis of Boyer-Moore-type string searching algorithms. In *1st ACM-SIAM Symposium on Discrete Algorithms*, pages 328–343, San Francisco, January 1990.

[BYP91] R.A. Baeza-Yates and C.H. Perleberg. Fast and practical approximate pattern matching. Technical report, Dept. of Computer Science, Univ. of Chile, 1991.

[BYR90] R. Baeza-Yates and M. Regnier. Fast algorithms for two dimensional and multiple pattern matching. In R. Karlsson and J. Gilbert, editors, *2nd Scandinavian Workshop in Algorithmic Theory, SWAT'90*, Lecture Notes in Computer Science 447, pages 332–347, Bergen, Norway, July 1990. Springer-Verlag.

[CGG90] L. Colussi, Z. Galil, and R. Giancarlo. The exact complexity of string matching. In *31st FOCS*, volume 1, pages 135–143, St. Louis, MO, Oct 1990. IEEE.

[Cho90] C. Choffrut. An optimal algorithm for building the Boyer-Moore automaton. *Bull. of EATCS*, 40:217–224, Jan 1990.

[CL90] W. Chang and E. Lawler. Approximated string matching in sublinear expected time. In *Proc. 31st FOCS*, pages 116–124, St. Louis, MO, Oct 1990. IEEE.

[CL91a] W. Chang and E. Lawler. Sublinear expected time approximate matching and biological applications. 1991.

[CL91b] W. Chang and E. Lawler. Theoretical and empirical comparisons of approximate string matching algorithms. 1991.

[Col91a] R. Cole. Tight bounds on the complexity of the Boyer-Moore string matching algorithm. In *SODA'91*, pages 224–233, San Francisco, CA, Jan 1991.

[Col91b] L. Colussi. Correctness and efficiency of pattern matching algorithms. *Information and Computation*, 95:225–251, 1991.

[CP91] M. Crochemore and D. Perrin. Two-way string-matching. *J.ACM*, 38(3):651–675, 1991.

[Fal88] C. Faloutsos. Signature files : an integrated access method for text and attributes suitable for optical disk storage. *BIT*, 28(4):736–754, 1988.

[Faw89] H. Fawcett. *A Text Searching System: PAT 3.3, User's Guide.* Centre for the New Oxford English Dictionary, University of Waterloo, 1989.

[FBY92] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Algorithms and Data Structures.* Prentice-Hall, 1992. (to appear).

[FC84] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM TOOIS*, 2(4):267–288, Oct 1984.

[FC87] C. Faloutsos and S. Christodoulakis. Description and performance analysis of signature file methods. *ACM TOOIS*, 5(3):237–257, 1987.

[GB91] G.H. Gonnet and S.A. Benner. Computational biochemistry research at ETH. Informatik/Organic Chemistry, ETH, Zurich, 1991.

[GBY91] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures - In Pascal and C*. Addison-Wesley, Wokingham, UK, 1991. (second edition).

[GBYS92] G.H. Gonnet, R. Baeza-Yates, and T. Snider. *PAT arrays: An alternative to Inverted Files*, chapter 5. Prentice-Hall, 1992.

[GCB91] G. Gonnet, M. Cohen, and S. Benner. The all-against-all matching of a major protein sequence data base: Reliable parameters for aligning sequences and scoring gaps. Swiss Federal Institute of Technology, 1991.

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

[Gon83] G.H. Gonnet. Unstructured data bases or very efficient text searching. In *ACM PODS*, volume 2, pages 117–124, Atlanta, GA, Mar 1983.

[Gon87a] G.H. Gonnet. Examples of PAT applied to the Oxford English Dictionary. Technical Report OED-87-02, Centre for the New OED., University of Waterloo, 1987.

[Gon87b] G.H. Gonnet. Extracting information from a text database: An example with dates and numerical data. In *Third Annual Conference of the UW Centre for the New Oxford English Dictionary*, pages 89–85, Waterloo, Canada, 1987.

[Gon88] G.H. Gonnet. Efficient searching of text and pictures (extended abstract). Technical Report OED-88-02, Centre for the New OED., University of Waterloo, 1988.

[GP90] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. on Computing*, 19(6):989–999, 1990.

[HS91] A. Hume and D.M. Sunday. Fast string searching. *Software - Practice and Experience*, 21(11):1221–1248, Nov 1991.

[JTU91] P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. University of Helsinki, 1991.

[JU91] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In A. Tarlecki, editor, *16th MFCS*, LNCS 520, pages 240–248, Kazimierz Dolny, Poland, Sep 1991.

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

[KST91] J.Y. Kim and J. Shawe-Taylor. Fast string matching using an n-gram algorithm. University of London, 1991.

[LV92] G. Landau and U. Vishkin. Pattern matching in a digitized image (extended abstract). In *3rd ACM-SIAM SODA*, Orlando, FL, Jan 1992. SIAM.

[MBY91] U. Manber and R. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Information Processing Letters*, 37:133–136, February 1991.

[MM90] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *1st ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, San Francisco, January 1990.

[MW89] U. Manber and S. Wu. An algorithm for approximate string matching with non uniform costs. Technical Report TR-89-19, Department of Computer Science, University of Arizona, Tucson, Arizona, Sept 1989.

[Mye] E.W. Myers. Incremental alignment algorithms and their applications. *SIAM J. on Computing*. (to appear).

[RF90] D.R. Raymond and H.J. Fawcett. Playing detective with full text searching software. Technical Report OED-90-03, Centre for the New OED., University of Waterloo, 1990.

[Sch88] R. Schaback. On the expected sublinearity of the Boyer-Moore algorithm. *SIAM J on Computing*, 17:548–658, 1988.

[SDRK87] R. Sacks-Davis, K. Ramamohanarao, and A. Kent. Multikey access methods based on superimposed coding techniques. *ACM TODS*, 12(4):655–696, 1987.

[Smi91] P.D. Smith. Experiments with a very fast substring search algorithm. *Software - Practice and Experience*, 21(10):1065–1074, Oct 1991.

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

[TU90] J. Tarhio and E. Ukkonen. Boyer-moore approach to approximate string matching. In J.R. Gilbert and R.G. Karlsson, editors, *2nd Scandinavian Workshop in Algorithmic Theory, SWAT'90*, Lecture Notes in Computer Science 447, pages 348–359, Bergen, Norway, July 1990. Springer-Verlag.

[Ukk91] E. Ukkonen. Approximate string matching with q-grams and maximal matches. Technical report, Institute for Informatik, Universitat Freiburg, Freiburg, Germany, May 1991. to appear in Theoretical Computer Science.

[UW89] E. Ukkonen and D. Wood. A simple on-line algorithm to approximate string matching. Technical report, Dept. of Computer Science, Univ. of Helsinki, Helsinki, Finland, 1989.

[WM91] S. Wu and U. Manber. Fast text searching with errors. Technical Report TR-91-11, Department of Computer Science, University of Arizona, Tucson, Arizona, June 1991.

[WM92] S. Wu and U. Manber. Agrep - a fast approximate pattern-matching tool. In *Proceedings of USENIX Winter 1992 conference*, San Francisco, CA, Jan 1992.

[WMM91] S. Wu, U. Manber, and E. Myers. Improving the running times for some string-matching problems. Technical Report TR-91-20, Department of Computer Science, University of Arizona, Tucson, Arizona, Aug 1991.

[ZT89] R.F. Zhu and T. Takaoka. A technique for two-dimensional pattern matching. *Communications of the ACM*, 32(9):1110–1120, September 1989.