Text Databases

Gonzalo Navarro
Dept. of Computer Science, University of Chile

A *text* is any sequence of symbols (or *characters*) drawn from an *alphabet*. A large portion of the information available worldwide in electronic form is actually in text form (other popular forms are structured and multimedia information). Some examples are: natural language text (for example books, journals, newspapers, jurisprudence databases, corporate information, and the Web), biological sequences (ADN and protein sequences), continuous signals (such as audio, video sequence descriptions and time functions), and so on. Recently, due to the increasing complexity of applications, text and structured data are being merged into so-called semistructured data, which is expressed and manipulated in formats such as XML (out of the scope of this review).

As more text data is available, the challenge to search them for queries of interest becomes more demanding. A *text database* is a system that maintains a (usually large) text collection and provides fast and accurate access to it. These two goals are relatively orthogonal, and both are critical to profit from the text collection.

Traditional database technologies are not well suited to handle text databases. Relational databases structure the data in fixed-length records whose values are atomic and are searched for by equality or ranges. There is no general way to partition a text into atomic records, and treating the whole text as an atomic value is of little interest for the search operations required by applications. The same problem occurs with multimedia data types. Hence the need for specific technology to manage texts.

The simplest query to a text database is just a string pattern: the user enters a string and the system points out all the positions of the collection where the string appears. This simple approach is insufficient for some applications where the user wishes to find a more general *pattern*, not just a plain string. Patterns may contain wild cards (characters that may appear zero or more times), optional characters (that may appear in the text or not), classes of characters (string positions that match a set of characters rather than a single one), gaps (that match any text substring within a range of lengths), etc. In many applications patterns must be *regular expressions*, composed by simple strings and union, concatenation, or repetition of other regular expressions. These extensions are typical of computational biology applications, but also appear in natural language search.

In addition, a text database may provide *approximate searching*, that is, means for recovering from different kinds of *errors* that the text collection (or the user query) may contain. We may have, for example, typing, spelling or optical character recognition errors in natural language texts;

experimental measurement errors or evolutionary differences in biological sequences; and noise or irrelevant differences in continuous signals. Depending on the application, the text database should provide a reasonable error model that permits users to specify some tolerance in their searches. A simple error model is the edit distance, which gives an upper limit to the total number of character insertions, deletions, and substitutions needed to match the user query with a text substring. More complex models give lower costs to more probable changes in the sequences.

The above searches are collectively called *syntactic search*, because they are expressed in terms of sequences of characters present in the text. On natural language texts, *semantic search* is of great value. In this case the user expresses an information need and the system is in charge of retrieving portions of the text collection (documents) that are relevant to that need, even if the query words do not directly appear in the answer. A model that determines how relevant is a document with respect to a query is necessary. This model *ranks* the documents and offers the highest ranked ones to the user. Unlike in the syntactic search, there are no right or wrong answers, just better and worse ones. However, since the time spent by the user in browsing the results is the most valuable resource, good ranking is essential in the success of the search.

1 Syntactic Search

In syntactic searching, the desired outcome of a search pattern is clear, so the main concern of a text database is efficiency. Two approaches are possible to pattern matching: sequential and indexed search (these can be combined, as we will see). Sequential searching assumes that it is not possible to preprocess the text, so only the pattern is preprocessed and then the whole text database is sequentially scanned. Indexed searching, on the other hand, preprocesses the text so as to build a data structure over it (an "index"), which can be used later to speed up searches. Building an index is usually a time and memory demanding task, so indexed searching is possible only when several conditions are met: (*i*) the text must be large enough to justify it against the simpler sequential search, (*ii*) the text must change rarely enough so as to amortize the indexing work over many queries, (*iii*) there must be enough space to hold the index, which can be quite large in some applications.

1.1 Sequential Searching

1.1.1 Simple String Matching

In the simplest search problem, we are given a string pattern P of m characters, a text T of n characters (which may be the concatenation of all texts in the collection), and are required to point out all the occurrences of P in T [5].

There exist many string matching algorithms. The most famous ones are Knuth-Morris-Pratt (KMP), for being the first in achieving the optimal $O(n)$ worst case time; and Boyer-Moore (BM), for being the first in achieving so-called "sublinear" search time, meaning that it does not inspect

every text character (actually, its search time improves as m grows). Another famous algorithm is Backward DAWG Matching (BDM), for its average-case optimality, $O(n \log_\sigma(m)/m)$ (where σ is the alphabet size if we assume a uniformly distributed text). Many other algorithms are relevant to theorists for their algorithmic features: optimal in space usage, bounded time to access the next text character, simultaneous worst- and average-case optimality, and so on.

In practice, the most efficient string matching algorithms derive from their theoretically more appealing variants. Shift-Or can be seen as a derivation of KMP, that is twice as fast and $O(n)$ in most practical cases. Horspool is a simplification of BM that is usually the fastest in searching natural language text. Backward Nondeterministic DAWG Matching (BNDM) is a derivation of BDM that is faster to search biological sequences of moderate length. Backward Oracle Matching (BOM) also derives from BDM and is the fastest for longer biological sequences.

The use of finite automata is at the kernel of pattern matching. Many of the above mentioned algorithms rely on automata. Figure 1 shows a nondeterministic finite automaton (NFA) that reaches its final state every time the string it has consumed finishes with a given pattern P. If fed with the text, this automaton will signal all the endpoints of occurrences of P in T. Algorithm KMP consists essentially in making that NFA deterministic (a DFA), and using it to process each text character in $O(1)$ time. Shift-Or, on the other hand, uses the automaton directly in nondeterministic form. The NFA states are mapped to bits in a computer word, and a constant number of logical and arithmetic operations over the computer word simulate the update of all the NFA states simultaneously, as a function of a new text character. The result is twice as fast as KMP in practice, and $O(n)$ provided $m \leq w$, being w the number of bits in the computer word (32 or 64 nowadays). For $m > w$, Shift-Or is still $O(n)$ on average. This technique to map NFA states to bits is called *bit-parallelism*, and it has been extensively used in pattern matching to simulate NFAs directly without converting them to DFAs.

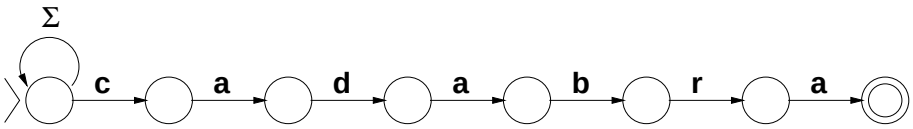

Figure 1: An NFA to recognize strings terminated in pattern `"cadabra"`.

On Figure 2 we see an NFA that recognizes every reversed *prefix* of a pattern P, that is, a prefix of P read backwards (a prefix is a substring that starts at the beginning of the string). Note that the NFA will have active states as long as we have fed it with a reversed substring of P. This automaton is used by BDM and BNDM, the former by converting it to deterministic and the latter with bit-parallelism. The idea is to build such an NFA over the reverse pattern. Given a text *window* (segment of length m) where the pattern could appear, we feed the automaton with the window characters read right to left. If the automaton runs out of active states at some point, it means that the window suffix we have read is not a substring of P, so P cannot appear in any window containing the suffix read. Hence, we can shift the window over T to the position following

the window character that killed the automaton. Actually, we can remember the last time the automaton signaled a prefix of P and shift to align the window to that position. If, on the other hand, we read the whole window and the automaton still has active states, then P is in the window, so we report it and shift to align the window to the last prefix recognized. Figure 3 (left) illustrates. Algorithm BOM is also based on determinizing this NFA, albeit in an imperfect way that gives shorter shifts but runs faster for being simpler.

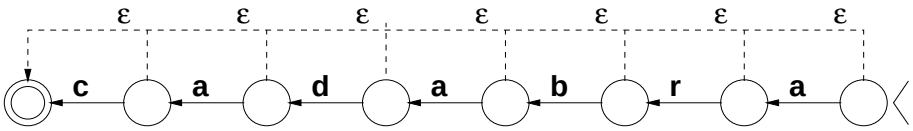

Figure 2: An NFA to recognize the suffixes of `"cadabra"` reversed.

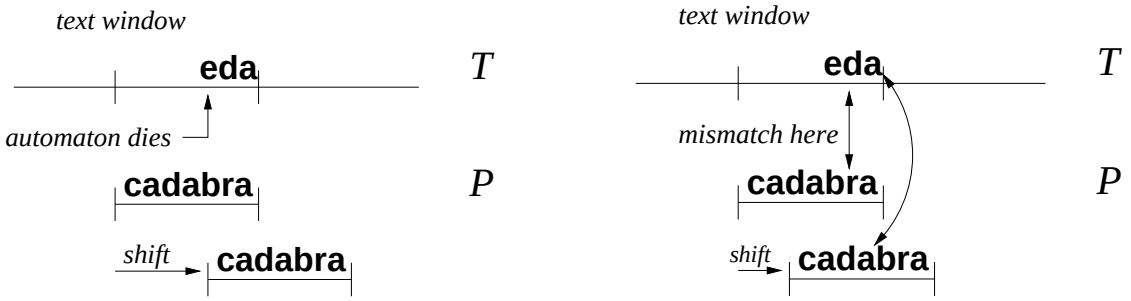

Figure 3: On the left, the BDM search process in a window. On the right, the same window processed by Horspool.

Horspool algorithm is not based on automata. It simply compares the current text window against P, reports an occurrence in case of equality, and then shifts the window so that its last character c aligns with the last occurrence of c in P. This does not lose any possible occurrence and wins because of its simplicity in all but small alphabets. Figure 3 (right) illustrates.

1.1.2 Extended Patterns and Regular Expressions

Patterns with wild cards, optional characters, gaps, and, in general, regular expressions, can be searched for by extending algorithms designed for simple strings [5]. The prevailing approach is, again, based on automata. Figure 4 shows some examples. A regular expression can be converted into an NFA and then a DFA, which can be used for searching in $O(n)$ worst case time. The problem is that this may need space and time exponential in m, which is disturbing in practice even for moderate sized patterns. Another approach is to simulate the NFA directly using bit-parallelism. Very efficient bit-parallel simulations of the restricted extended patterns mentioned above have been

developed. When simulating general regular expressions, however, the same exponential dependency on m appears. It is possible, using bit-parallelism, to search in $O(mn/\log(s))$ time given $O(s)$ space for the preprocessing.

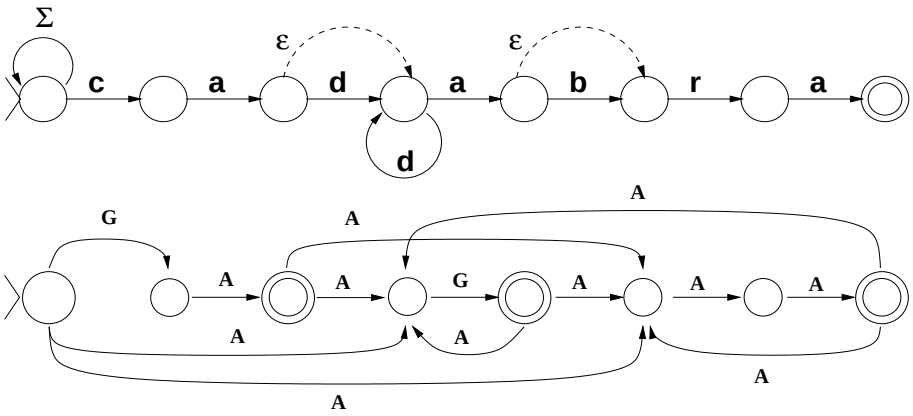

Figure 4: On top, an NFA to recognize an extended pattern with optional and repeatable characters. On the bottom, the NFA of a general regular expression.

1.1.3 Approximate Pattern Matching

Independently of the pattern searched for, it is usually convenient to allow some tolerance in its matching against the text. The usual approach is to define a *distance* between strings that models the desired tolerance, and let the user specify a tolerance threshold k together with P. Then, the system reports text substrings whose distance to P does not exceed k [5, 3].

The most general technique for approximate pattern matching relies on dynamic programming. A virtual matrix is filled such that at cell (i, j) contains the minimum distance between the length-i prefix of P and a substring of T finishing at position j. Each cell can be filled in constant time, giving $O(mn)$ worst case search time.

For some simple (and popular) distance functions, such as the edit distance, much faster algorithms exist. As before, is it possible to express the search as the outcome of an automaton, depicted in Figure 5. This NFA can be made deterministic so as to search in $O(n)$ worst case time. As for regular expressions, the problem is that the preprocessing time and space is exponential in m or k, which makes the DFA approach unpopular. Bit-parallelism, however, can be used to simulate the NFA directly so as to obtain very practical $O(kmn/w)$ worst-case time algorithms. Moreover, the dynamic programming matrix can be computed in bit-parallel so as to obtain $O(mn/w)$ time.

Another useful approach for the case of low k/m values (which are the most interesting in practice) is to *filter* the text to discard most of it with a necessary condition that can be checked fast. For example, if P is cut into $k + 1$ nonoverlapping pieces, then any occurrence within edit

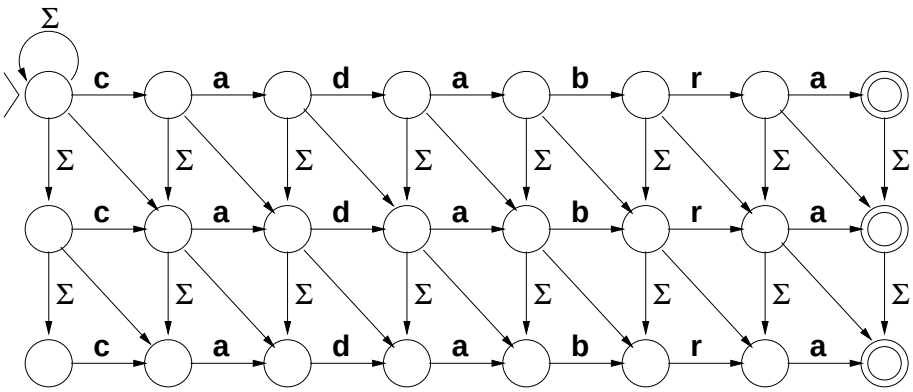

Figure 5: An NFA to find `"cadabra"` within edit distance at most 2. Unlabeled arrows are traversed by Σ and ε.

distance k must contain one of the pieces unaltered. A multipattern search for the pieces followed by classical verification of areas around the occurrences is very simple and one of the fastest algorithms, with average complexity $O(nk \log_\sigma(m)/m)$. For small alphabets, the best algorithms in practice are also optimal on average: $O(n(k + \log_\sigma(m))/m)$.

Several of these techniques can be adapted to approximate searching of regular expressions and extended patterns as well, with good results.

1.2 Indexed Searching

When the text can be preprocessed to build an index, the most popular general-purpose choices are suffix trees and arrays [2]. The best choice for the specific case of natural language is the inverted index, which we review soon.

A *digital tree* or *trie* is a tree where each branch is labeled by a character, and the children of a node are labeled by different characters. The set of strings represented by a trie are obtained by reading all its root-to-leaf paths and concatenating the characters labeling each path. A *suffix trie* is a trie representing all the suffixes of a given text. At the leaves, we store the starting position of the suffixes. Note that, since every substring is the prefix of some suffix, every text substring can be found by following a path from the suffix trie root. Hence, in order to search for all the occurrences of P in T, it is enough to traverse the suffix tree path of T corresponding to the characters of P. All the occurrences of P are then found at the leaves of the subtree we end up. This requires optimal time $O(m + occ)$ to retrieve the occ occurrences of P. Figure 6 illustrates.

Although optimal at searching, the suffix trie can be of size $O(n^2)$ in the worst case. The *suffix tree* is a variant that guarantees $O(n)$ space and construction time, making the overall scheme optimal in theory. In practice, even the suffix tree may be too large (10 to 20 times the text size), so more compact structures may be preferred. The most popular one is the *suffix array*, which contains

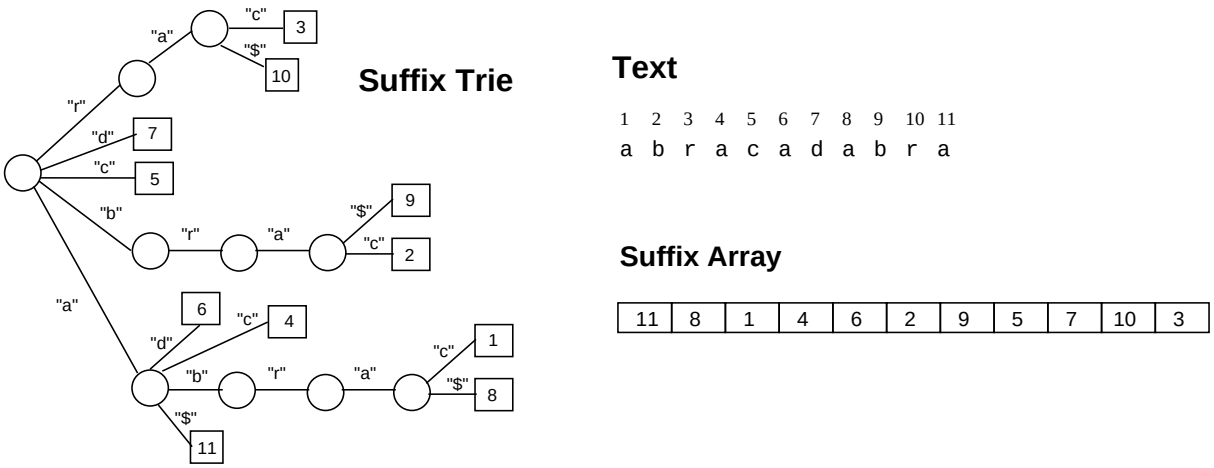

Figure 6: The suffix trie and suffix array for the text `"abracadabra"`.

just the leaves of the suffix tree, or which is the same, the sequence of pointers to all text suffixes in lexicographic order. The suffix array needs 4 times the text size, which is more manageable although still large. Just like all the occurrences of P appear in a single subtree of the suffix tree, they appear in a single interval of the suffix array, which can be binary searched at $O(m \log n)$ worst case time. Very recently, compression techniques for suffix arrays have obtained great success, but the techniques are still immature.

Suffix trees and arrays can be searched for complex patterns too [4]. The general idea is to define an automaton that recognizes the desired pattern, and backtrack in the suffix tree (or simulate a backtrack on the array), going all the branches. The traversal stops either when the automaton runs out of active states (in which case the subtree is abandoned) or when it reaches a final state (in which case the subtree leaves are reported). On most regular expressions and approximate searching, $O(n^\lambda)$ worst case time, for some $0 < \lambda < 1$, is achieved.

Natural language texts, at least in most Western languages, have some remarkable features that make them easier to search [1, 6]. The text can be seen as a sequence of *words*. The number of different words grows slowly with the text size. The query is usually a word or a sequence of words (a phrase). In this case, *inverted indexes* are a low cost solution that provide very good performance. An inverted index is just a list of all the different words in the text (the *vocabulary*), plus a list of the occurrences of each such word in the text collection. A search for a word reduces to a vocabulary search (with hashing, for example) plus outputting the list. A search for a complex pattern that matches a word is solved by sequential scanning over the vocabulary plus outputting the matching word lists. A phrase search is solved by intersecting the occurrences of the involved words. An inverted index typically needs 15% to 40% of the space of the text, but several compression techniques to the index and the text can be successfully applied [6].

2 Semantic Search

In semantic searching, the text collection is seen as a set of *documents*, and the goal is to establish the *relevance* of each document with respect to a user query, so as to present the most relevant ones to the user. The query can be a set of words or phrases, or even another document. As accuracy in the answers is a central concern, the choice of a *model* to compute how relevant is a document with respect to a query is fundamental [1].

2.1 Relevance Models

The most popular relevance model is called the *vector model*, where each document is seen as a vector in a very high dimensional space. Each coordinate represents a word of the collection vocabulary. The value of the document vector along a coordinate is given by the relevance of the corresponding word in distinguishing that document from others. The query is also seen as a vector, and the cosine between two vectors is their similarity measure.

A popular measure of the the relevance of a word to distinguish a document from others is $tf \times idf$, where $tf = f/mf$, f is the number of times the word appears in the document, mf the number of times the most frequent word of the document appears in it, $idf = \log(N/n)$, N is the total number of documents, and n the number of documents where the word appears.

For Web searching, this model is usually enriched with the information given by the links between Web pages, so that pages pointed to from "relevant" pages are also considered relevant.

Another less popular model, which however is usually mixed with the vector model, is the boolean model. In this case documents are relevant or not relevant. The query specifies which words must appear in relevant documents, which must not appear, and indicates intersections, unions and complementation of resulting document sets. There are several other less popular models of relevance.

2.2 Evaluation

An issue related to relevance computation is how to evaluate the accurateness of the results. The most popular measures are called *precision* and *recall*. Precision indicates which fraction of the retrieved documents are actually relevant. Recall indicates which fraction of the relevant documents were actually retrieved. A human judgement is necessary to establish which are the documents really relevant to the query. Note that it is easy to have higher precision by retrieving less documents and higher recall by retrieving more documents, since the ranking just orders the document but does not indicate how many to retrieve. Hence the correct measure is a "precision-recall plot", where the precision and recall obtained when retrieving more and more documents are drawn in a two-dimensional plot (one coordinate for precision and another for recall). For example, one can plot the precision obtained for recall values of 0%, 10%, 20%, ..., 100%.

2.3 Indexes

Variants of the inverted index are useful for semantic searches as well. These indexes indicate only the documents where each word appears, as well as its number of occurrences in there. This is useful to compute term f in the vector ranking. Moreover, since only the few highest ranked documents will be output, documents with highest f value will be the most interesting ones, so the lists of documents where each term appears are sorted by decreasing f value. Several heuristics are used to avoid processing long lists of probably irrelevant documents. Although these may alter the correct outcome of the query, the relevance model is already a guess of what the user wants, so it is admissible to distort it a bit in exchange for large savings in efficiency. This tradeoff is not permitted in syntactic searching.

Future Trends

Textual databases face several challenges for the upcoming years.

- The increasing size of text collections, which in cases like the Web reach several terabytes. Maintaining efficiency and accuracy of results under that scenario is much more complex than for medium sized collections.
- The increasing complexity of search operations, typical of applications in computational biology and other areas. This poses the need to find more complex patterns, which are usually much more difficult to search for than simple strings.
- The low quality of the data, with different reasons depending on the application: no quality control in the Web, experimental errors in DNA sequences, etc. This poses the need for approximate matching methods well suited to each application.
- The size of the indexes, which especially for general texts is more relevant than the search speed. Over a large text, such an index requires frequent access to secondary memory, which is much slower than main memory.
- The need to deal with secondary memory, which is a rather undeveloped feature in some indexes, especially those for general texts. Some indexes still perform bad in secondary memory.
- The need to cope with changes in the text collection, as text indexes usually require expensive rebuilding upon text changes.
- The need to provide, for text databases, the same level of data management achieved for atomic values in traditional databases, with respect to powerful query languages (for example permitting joins), concurrency, recovery, security, etc.

References

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

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

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

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

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

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

Relevant Terms

Approximate searching: searching permitting some differences between the pattern specification and its text occurrences.

Bit-parallelism: a technique to store several values in a single computer word so as to process them all in one shot.

Edit distance: a measure of string similarity counting the number of character insertions, deletions and substitutions needed to convert one string into the other.

Indexed searching: searching with the help of an index, a data structure previously built on the text.

Precision and recall: measures of retrieval quality, relating the documents retrieved with those actually relevant to a query.

Text database: a database system managing a collection of texts.

Sequential searching: searching without preprocessing the text.

Suffix trees and arrays: text indexes that permit fast access to any text substring.

Semantic search: a search on natural language text based on the meaning rather than the syntax of the text.

Syntactic search: a text search based on string patterns that appear in the text, without reference to meaning.

Vector model: a model to measure the relevance between a document and a query, based on sharing distinctive words.