Pattern-Matching and Text-Compression Algorithms

MAXIME CROCHEMORE

Gaspard Monge Institute, University of Marne-la-Vallée, France

THIERRY LECROQ

Laboratoire d'Informatique de Rouen, University of Rouen, France

Pattern matching is the problem of locating a specific pattern inside raw data. The pattern is usually a collection of strings described in some formal language. Applications require two kinds of solution depending upon which string, the pattern, or the text, is given first. Solutions based on the use of automata or combinatorial properties of strings are commonly implemented to preprocess the pattern. The notion of indices realized by trees or automata is used in the second kind of solutions.

The aim of data compression is to provide representation of data in a reduced form in order to save both storage place and transmission time. There is no loss of information, the compression processes are reversible.

Pattern-matching and text-compression algorithms are two important subjects in the wider domain of text processing. They apply to the manipulation of texts (word editors), to the storage of textual data (text compression), and to data retrieval systems (full text search). They are basic components used in implementations of practical softwares existing under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science (system or software design). Finally, they also play an important role in theoretical computer science by providing challenging problems.

Although data are recorded in various ways, text remains the main way to exchange information. This is particularly evident in literature or linguistics where data are composed of huge corpora and dictionaries, but applies as well to computer science where a large amount of data is stored in linear files. And it is also the case, for instance, in molecular biology because biological molecules can often be approximated as sequences of nucleotides or amino acids. Furthermore, the quantity of available data in these fields tend to double every 18 months. This is the reason that algorithms must be efficient even if the speed and storage capacity of computers increase continuously.

PATTERN MATCHING

When the pattern is a single string the problem is known as string matching: locate all occurrences of a string x of length m in a text y of length n. The string and the text are built over the same alphabet Σ of size σ. The naive algorithm locates all occurrences in time $O(nm)$. But hashing provides a simple method that avoids the quadratic number of symbol comparisons in most practical situations and runs in linear time under reasonable probabilistic assumptions [Harrison 1971; Karp and Rabin 1987].

The first linear-time string-matching algorithm was discovered by Morris and Pratt [1970]. It has been improved by Knuth et al. [1976]. The search behaves like a recognition process by automation, and a character of the text is compared to a character of the pattern no more than $\log_\Phi(m + 1)$ (Φ is the golden

ratio $(1 + \sqrt{5})/2$). Hancart [1993] proves that the delay of a related algorithm discovered by Simon [1994] makes no more than $1 + \log_2 m$ comparisons per text symbol.

Boyer and Moore's [1977] algorithm is considered the most efficient string-matching algorithm in usual applications. A simplified version of it (or the entire algorithm) is often implemented in text editors for the "search" and "substitute" commands. Cole [1995] proves that the maximum number of symbol comparisons is tightly bounded by $3n$ after the preprocessing.

Several variants of Boyer and Moore's algorithm avoid the quadratic behavior when searching for all occurrences of the pattern. The most efficient solutions in term of number of symbol comparisons have been designed by Apostolico and Giancarlo [1986], Crochemore et al. [Turbo-BM 1994], and Colussi [1994]. Empirical results show that the variations of Boyer and Moore's algorithm designed by Sunday [Quick Search 1990] and an algorithm based on the suffix automaton by Crochemore et al. [1994] are the most efficient in practice.

Searching for k patterns by repetitive runs of previous algorithms on the text y gives an overall $O(kn)$ running time. In 1975, Aho and Corasick designed an $O(n \log \sigma)$ algorithm to solve this problem, with a running time independent of the number of patterns. It is implemented by the fgrep command under the UNIX operating system.

The notion of a longest common subsequence (LCS) of two strings is widely used to compare files. The diff command of UNIX implements an algorithm based on this notion where lines of the files are considered as symbols. Informally, the result of a comparison gives the minimum number of operations (insert a symbol or delete a symbol) to transform one string into the other. The comparison of molecular sequences is basically done with a related concept, alignment of strings, which consists of aligning their symbols on vertical lines. This is related to an edit distance, called the Levenshtein distance, with the additional operation of substitution, and with weights associated to operations. Hirchsberg [1975] presents the computation of the LCS in linear space. This is an important result because the algorithm is used on large sequences.

Approximate string matching consists of finding all approximate occurrences of pattern x in text y. Approximate occurrences of x are segments of y that are close to x according to a specific distance: their distance to x must be not greater than a given integer k. Two common distances are the Hamming distance and the Levenshtein distance.

With the Hamming distance related to the number of mismatches between the pattern and its approximate occurrences, the problem is also called approximate string matching with k mismatches. With the Levenshtein distance (or edit distance) the problem is known as the approximate string matching with k differences. Approximate string searching is a lively domain of research. It includes, for instance, the notion of regular expressions to represent sets of strings. Algorithms based on regular expression are commonly found in books related to compiling techniques. The Shift-Or algorithm by Baeza-Yates and Gonnet [1992], and by Wu and Manber [1992] is a method that is both very fast in practice and very easy to implement. It adapts to the two preceding problems.

In applications where the text is to be searched for several patterns, the text needs to be preprocessed. Even if no further information is known on their syntactic structure, it is possible and indeed extremely efficient to build an index that supports searches. Data structures to represent indices on text files are: suffix trees [Weiner 1973; McCreight 1976; Ukkonen 1994], direct acyclic word graph [Blumer et al. 1985], suffix automata [Crochemore 1986], and suffix arrays [Manber and Myers 1993]. All algorithms (except for suffix arrays) build the index in time $O(n \log \sigma)$.

TEXT COMPRESSION

The following methods yield two basic data compression algorithms that produce good compression ratios and run in linear time.

The first strategy is a statistical encoding that takes into account the frequencies of symbols to build a uniquely decipherable code optimal with respect to the compression criterion. The Huffman [1951] method provides such an optimal statistical coding. It admits a dynamic version in which symbol counting is done at coding time. The compact command of UNIX implements this version.

Ziv and Lempel [1977] designed a compression method using encoding segments. These segments are stored in a dictionary that is built during the compression process. When a segment of the dictionary is encountered later while scanning the original text it is substituted by its index in the dictionary. In the model where portions of the text are replaced by pointers on previous occurrences, the Ziv and Lempel compression scheme can be proved to be asymptotically optimal (on large enough texts satisfying good conditions on the probability distribution of symbols). The dictionary is the central point of the algorithm. Furthermore, a hashing technique makes its implementation efficient. This technique, improved by Welch [1984], is implemented by the compress command of the UNIX operating system.

The problems and algorithms discussed give a sample of text-processing methods. Several other algorithms improve on their performance when the memory space or the number of processors of a parallel machine are considered, for example. Methods also extend to other discrete objects such as trees and images.

REFERENCES

Listed in the following are either books entirely devoted to pattern-matching or text-compression algorithms or books on the design of general algorithms that contain a whole chapter on the topic. All references mentioned in the text may be found in these books.

AHO, A. V. 1990. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science, Algorithms and Complexity, Vol. A.* J. van Leeuwen Ed., Elsevier, Amsterdam, Ch. 5, 255–330.

BELL, T. C., CLEARY J. G., AND WITTEN, I. H. 1990. *Text Compression.* Prentice Hall, Englewood Cliffs, NJ.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms.* MIT Press, Cambridge, MA, Ch. 34, 853–885.

CROCHEMORE, M. AND RYTTER, W. 1994. *Text Algorithms.* Oxford University Press, New York.

GONNET, G. H. AND BAEZA-YATES, R. A. 1991. *Handbook of Algorithms and Data Structures.* Addison-Wesley, Reading, MA, Ch. 7, 251–288.

NELSON, M. 1992. *The Data Compression Book.* M&T Books, New York.

SEDGEWICK, R. 1990. *Algorithms in C.* Addison-Wesley, Reading, MA, Ch. 19 and 22.

STEPHEN, G. A. 1994. *String Searching Algorithms.* World Scientific Press, River Edge, NJ.

Presentation on theme: "Chapter 9: Text Processing Pattern Matching Data Compression."â€” Presentation transcriptÂ 5 Brute-Force Algorithm The brute-force pattern matching algorithm compares the pattern P with the text T for each possible shift of P relative to T, until either a match is found, or all placements of the pattern have been tried Brute-force pattern matching runs in time O(nm) Example of worst case: T ï€½ aaa â€¦ ah P ï€½ aaah may occur in. images and DNA sequences unlikely in English text Algorithm BruteForceMatch(T, P) Input text T of size n and pattern P of size m Output starting index of a substring of T equal to P or ï€ 1 if no such substring exists for i ï¬ 0 to n ï€ m { test shift i of the patte Pattern matching When the pattern is a single string the problem is known as string matching: locate all occurrences of a string x of length m in a text y of length n. The string and the text are built over the same alphabet Sigma of size oe. The naive algorithm locates all occurrences in time O(nm). But hashing provides a simple method that avoids the quadratic. Pattern-Matching and Text-Compression Algorithms. Article (PDF Available) in ACM Computing Surveys 28(1):39-41 Â· March 1996 with 139 Reads. DOI: 10.1145/234313.234331 Â· Source: DBLP.Â We also present two algorithms for matching patterns in images that are extensions of string-matching algorithms. In several applications, texts need to be structured before being searched. Even if no further information is known about their syntactic structure, it is possible and indeed extremely ecient to build a data structure that supports searches.