Sequential Multiple String Matching

(1999; Crochemore, Czumaj, Gąsieniec, Lecroq, Plandowski, Rytter)

Maxime Crochemore, Université de Marne-la-Vallée and
King's College London, monge.univ-mlv.fr/∼mac

Thierry Lecroq, Université de Rouen, monge.univ-mlv.fr/∼lecroq

Entry editor: Gonzalo Navarro

INDEX TERMS: string matching, pattern matching, shift function, failure function, trie, DAWG
SYNONYMS: Dictionary matching

PROBLEM DEFINITION

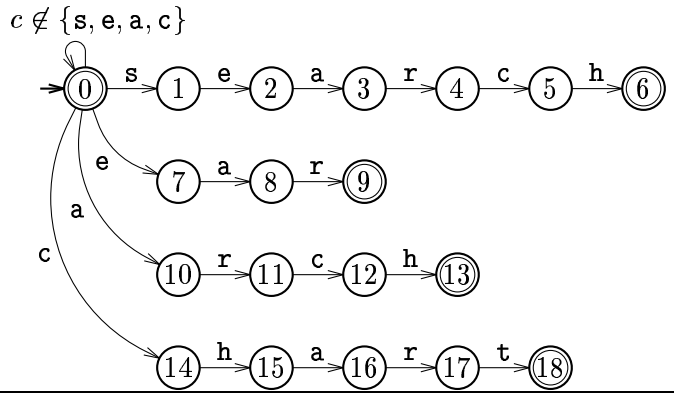
Given a finite set of k *pattern strings* $\mathcal{P} = \{P^1, P^2, \ldots, P^k\}$ and a *text string* $T = t_1 t_2 \ldots t_n$, T and the P^is being sequences over an alphabet Σ of size σ, the *multiple string matching (MSM)* problem is to find one or, more generally, all the text positions where a P^i occurs in T. More precisely the problem is to compute the set $\{j \mid \exists i, P^i = t_j t_{j+1} \ldots t_{j+|P^i|-1}\}$, or equivalently the set $\{j \mid \exists i, P^i = t_{j-|P^i|+1} t_{j-|P^i|+2} \ldots t_j\}$. Note that reporting all the occurrences of the patterns may lead to a quadratic output (for example, when P^is and T are drawn from a one-letter alphabet). The length of the shortest pattern in $\mathcal{P}$ is denoted by ℓmin. The patterns are assumed to be given first and are then to be searched in several texts. This problem is an extension of the exact string matching problem.

Both worst- and average-case complexities are considered. For the latter one assumes that pattern and text are randomly generated by choosing each character uniformly and independently from Σ. For simplicity and practicality the assumption $|P^i| = o(n)$ is set, for $1 \leq i \leq k$, in this entry.

KEY RESULTS

A first solution to the multiple string matching problem consists in applying an exact string matching algorithm for locating each pattern in $\mathcal{P}$. This solution has an $O(kn)$ worst case time complexity. There are more efficient solutions along two main approaches. The first one, due to Aho and Corasick [1], is an extension of the automaton-based solution for matching a single string. The second approach, initiated by Commentz-Walter [3], extends the Boyer-Moore algorithm to several patterns.

The Aho-Corasick algorithm first builds a trie $T(\mathcal{P})$, a digital tree recognizing the patterns of $\mathcal{P}$. The trie $T(\mathcal{P})$ is a tree whose edges are labeled by letters and whose branches



state	0	1	2	3	4	5	6
prefix	ε	s	se	sea	sear	searc	search
fail	0	0	7	8	9	12	13
out					{ear}		{search, arch}

state	7	8	9	10	11	12	13
prefix	e	ea	ear	a	ar	arc	arch
fail	0	10	11	0	0	14	15
out		{ear}				{arch}	

state	14	15	16	17	18	19
prefix	c	ch	cha	char	chart	
fail	0	0	10	11	0	
out					{chart}	

Figure 1: The Pattern Matching Machine or Aho-Corasick automaton for the set of strings {search, ear, arch, chart}.

spell the patterns of $\mathcal{P}$. A node p in the trie $T(\mathcal{P})$ is associated with the unique word w spelled by the path of $T(\mathcal{P})$ from its root to p. The root itself is identified with the empty word ε. Notice that if w is a node in $T(\mathcal{P})$ then w is a prefix of some $P^i \in \mathcal{P}$. If in addition $a \in \Sigma$ then $child(w, a)$ is equal to wa if wa is a node in $T(\mathcal{P})$; it is equal to NIL otherwise.

During a second phase, when patterns are added to the trie, the algorithm initializes an output function *out*. It associates the singleton $\{P^i\}$ with the nodes P^i $(1 \leq i \leq k)$, and associates the empty set with all other nodes of $T(\mathcal{P})$.

Finally, the last phase of the preprocessing consists in building a failure link for each node of the trie, and simultaneously completing the output function. The failure function *fail* is defined on nodes as follows (w is a node): $fail(w) = u$ where u is the longest proper suffix of w that belongs to $T(\mathcal{P})$. Computation of failure links is done during a breadth-first traversal of $T(\mathcal{P})$. Completion of the output function is done while computing the failure function *fail* using the following rule: if $fail(w) = u$ then $out(w) = out(w) \cup out(u)$.

To stop going back with failure links during the computation of the failure links, and also to overpass text characters for which no transition is defined from the root during the searching phase, a loop is added on the root of the trie for these symbols. This finally produces what is called a Pattern Matching Machine or an Aho-Corasick automaton (see Figure 1).

After the preprocessing phase is completed, the searching phase consists in parsing the text T with $T(\mathcal{P})$. This starts at the root of $T(\mathcal{P})$ and uses failure links whenever a character

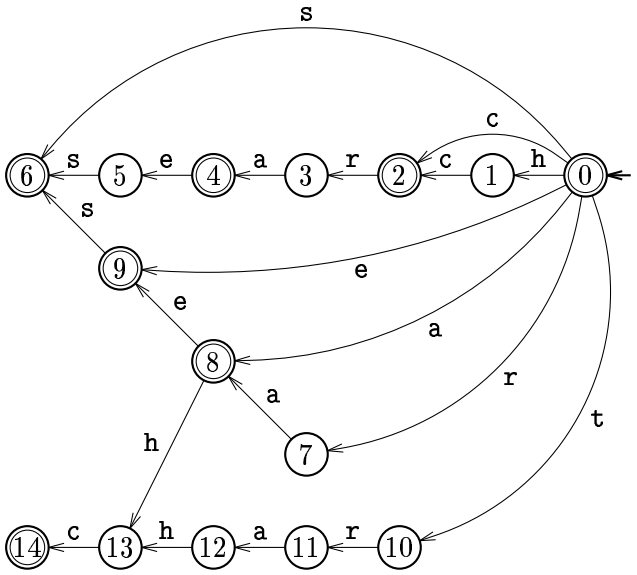


Figure 2: An example of DAWG, index structure used for matching the set of strings {`search`, `ear`, `arch`, `chart`}. The automaton accepts the reverse prefixes of the strings.

in T does not match any label of outgoing edges of the current node. Each time a node with a nonempty output is encountered, this means that the patterns of the output have been discovered in the text, ending at the current position. Then, the position is output.

Theorem 1 (Aho and Corasick 1975 [1]). *After preprocessing $\mathcal{P}$, searching for the occurrences of the strings of $\mathcal{P}$ in a text T can be done in time $O(n \times \log \sigma)$. The running time of the associated preprocessing phase is $O(|\mathcal{P}| \times \log \sigma)$. The extra memory space required for both operations is $O(|\mathcal{P}|)$.*

The Aho-Corasick algorithm is actually a generalization to a finite set of strings of the Morris-Pratt exact string matching algorithm.

Commentz-Walter [3] generalized the Boyer-Moore exact string matching algorithm to Multiple String Matching. Her algorithm builds a trie for the reverse patterns in $\mathcal{P}$ together with two shift tables, and applies a right to left scan strategy. However it is intricate to implement and has a quadratic worst-case time complexity.

The DAWG-match algorithm [4] is a generalization of the BDM exact string matching algorithms. It consists in building an exact indexing structure for the reverse strings of $\mathcal{P}$ such as a factor automaton or a generalized suffix tree, instead as just a trie as in the previous solution (see Figure 2). The overall algorithm can be made optimal by using both an indexing structure for the reverse patterns and an Aho-Corasick automaton for the patterns. Then, searching involves scanning some portions of the text from left to right and some other portions from right to left. This enables to skip large portions of the text T.

Theorem 2 (Crochemore et al. 1999 [4]). *The DAWG-match algorithm performs at most $2n$ symbol comparisons. Assuming that the sum of the length of the patterns in $\mathcal{P}$ is less than ℓmin^k, the DAWG-match algorithm makes on average $O((n \log \ell min)/\ell min)$ inspections of text characters.*

The bottleneck of the DAWG-match algorithm is the construction time and space consumption of the exact indexing structure. This can be avoided by replacing the exact indexing

structure by a factor oracle for a set of strings. When the factor oracle is used alone, it gives the Set Backward Oracle Matching (SBOM) algorithm [2]. It is an exact algorithm that behaves almost as well as the DAWG-match algorithm.

The bit-parallelism technique can be used to simulate the DAWG-match algorithm. It gives the MultiBNDM algorithm of Navarro and Raffinot [7]. This strategy is efficient when $k \times \ell min$ bits fit in a few computer words. The prefixes of strings of $\mathcal{P}$ of length ℓmin are packed together in a bit vector. Then, the search is similar to the BNDM exact string matching and is performed for all the prefixes at the same time.

The use of the generalization of the bad-character shift alone as done in the Horspool exact string matching algorithm gives poor performances for the MSM problem due to the high probability of finding each character of the alphabet in one of the strings of $\mathcal{P}$.

The algorithm of Wu and Manber [11] considers blocks of length ℓ. Blocks of such a length are hashed using a function h into values less than *maxvalue*. Then $shift[h(B)]$ is defined as the minimum between $|P^i| - j$ and $\ell min - \ell + 1$ with $B = p^i_{j-\ell+1} \dots p^i_j$ for $1 \leq i \leq k$ and $1 \leq j \leq |P^i|$. The value of ℓ varies with the minimum length of the strings in $\mathcal{P}$ and the size of the alphabet. The value of *maxvalue* varies with the memory space available.

The searching phase of the algorithm consists in reading blocks B of length ℓ. If $shift[h(B)] > 0$ then a shift of length $shift[h(B)]$ is applied. Otherwise, when $shift[h(B)] = 0$ the patterns ending with block B are examined one by one in the text. The first block to be scanned is $t_{\ell min-\ell+1} \dots t_{\ell min}$. This method is incorporated in the *agrep* command [10].

APPLICATIONS

MSM algorithms serve as basis for: multidimensional pattern matching and approximate pattern matching with wildcards. The problem has many applications in computational biology, database search, bibliographic search, virus detection in data flows, and several others.

EXPERIMENTAL RESULTS

The book of G. Navarro and M. Raffinot [8] is a good introduction to the domain. It presents experimental graphics that report experimental evaluation of multiple string matching algorithms for different alphabet sizes, pattern lengths, and sizes of pattern set.

URL to CODE

Well-known packages offering efficient MSM are *agrep* (`http://webglimpse.net/download.html`, top-level subdirectory `agrep/`) and *grep* with the -F option (`http://www.gnu.org/software/grep/grep.html`).

CROSS REFERENCES

Sequential exact string matching is the version where a single pattern is searched for in a text; *Indexed string matching* refers to the case where the text can be preprocessed; *Regular*

expression matching is the more complex case where the pattern can be a regular expression; *Multidimensional String Matching* is the case where the text dimension is greater than one.

RECOMMENDED READING

Further information can be found in the three following books: [5], [6] and [9].

References

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

[2] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: a new structure for pattern matching. In *SOFSEM'99*, LNCS 1725, pages 291–306. Springer-Verlag, 1999.

[3] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of ICALP'79*, LNCS 71, pages 118–132. Springer-Verlag, 1979.

[4] M. Crochemore, A. Czumaj, L. Gąsieniec, T. Lecroq, W. Plandowski, and W. Rytter. Fast practical multi-pattern matching. *Inf. Process. Lett.*, 71((3–4)):107–113, 1999.

[5] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings.* Cambridge University Press, 2007.

[6] D. Gusfield. *Algorithms on strings, trees and sequences.* Cambridge University Press, 1997.

[7] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM J. Exp. Algorithms*, 5:4, 2000.

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

[9] W. F. Smyth. *Computing Patterns in Strings.* Addison Wesley Longman, 2002.

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

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