

# Celatro™

pattern matching library

## Overview of Exact Matching Algorithms



### Abstract

The **exact string matching** problem is one of the classic problems of computer science, along with **sorting** and **searching**.<sup>1</sup> Matching a string, or *pattern*,  $P$ , to a *text*,  $T$ , means finding all occurrences  $P$  in  $T$  (e.g., all occurrences of the character string "the" in this paragraph). This paper describes some of the basic problems of exact matching, and provides a history of solutions.

For purposes of this discussion, the term *exact* will be taken to include case sensitivity: e.g., "the" will *not* match "The".

In addition, problems with extended characters sets will also be ignored as being beyond the scope of this treatment. Generally speaking, matching strings will have the same underlying byte representation.

### The Naïve Algorithm

The logical place to begin a review of exact matching algorithms is at the **naïve algorithm**: all the characters from  $P$  beginning at  $0$  are compared to characters from  $T$  beginning at  $0$ , up to  $P[m - 1]$ , where  $m$  is traditionally the length of  $P$ , which is then "shifted" forward so that  $P[0]$  is lined up with  $T[1]$ , and  $m$  comparisons are again made. The computation complexity of such a method is designated  $\Theta(nm)$ , i.e., the search time is proportional to the length of  $P$  times the length of  $T$ ,  $n$ , since every character in  $P$  will be compared to every character in  $T$ .

A complexity of  $\Theta(n)$ , proportional only to the length of  $T$ , is the obvious ideal.<sup>2</sup> This can be approached, theoretically, if previous comparisons are not repeated, i.e., if information gained from one iteration is not thrown away before the next, as the **naïve algorithm** clearly does. To this end, every exact matching algorithm somehow preprocesses  $P$ .

### Preprocessing P

Two important earlier solutions, called **Main-Lorentz** and **Knuth-Morris-Pratt**, look for repeated substrings in the pattern.<sup>3</sup> When the pattern then matches only partially against the text, the algorithm is able to shift the pattern forward by more than one character to a new "window," and then to use the information obtained from the previous iteration to *avoid comparing substrings that have already matched*, or, conversely, to *compare substrings that can be expected to mismatch*. The ideal such algorithm would have to compare matching substrings within the pattern and the text only once, approaching a complexity of  $\Theta(n)$ . Such a rule, applied to both preprocessing and searching, is called a **match rule**.

**Boyer and Moore's algorithm** ingeniously begins comparing characters at the end of the pattern, in order to implement the **occurrence rule**: if character  $T[i]$  does not match  $P[j + i]$ , shift  $P$  forward (i.e., increment  $i$ ) until  $T[i]$  can match a known occurrence of that character in  $P$ .<sup>4</sup> This algorithm has proven to be effective in practice as well as simple to understand and implement. Its theoretical efficiency, however, depends on its use of the **good suffix rule**, a kind of **match rule** that also preprocesses  $P$  to find *repeated suffixes* (as opposed to the *repeated prefixes* technique used by Main-Lorentz and Knuth-Morris-Pratt).<sup>5</sup>

A number of improvements to the **Boyer-Moore algorithm** have been proposed, all intended to take advantage of information gained on previous iterations to shift  $P$  further forward, then skip comparisons of substrings that can be expected to match.<sup>6</sup> However, the additional logic that these optimizations employ on every iteration inevitably incur a practical cost — including more intricate branching, a larger number of temporary variables, and

so on — whereas modern hardware makes fetching and comparing characters from memory so much cheaper that it seems less imperative to avoid repeated comparisons.

### Horspool and Sunday

In fact, more recently published exact matching algorithms dispense with the suffix rule entirely, and focus on enabling the largest shifts. **Horspool's algorithm**, for example, applies the **bad character rule** only to the rightmost character in the current window.<sup>7</sup> The larger the *alphabet* is relative to the length of  $P$ , the more likely there will be a mismatch at that position; hence large shifts moving the window completely past that position are common. Most searches in text editors using the ASCII alphabet are very short, and  $P$  is unlikely to have repeated suffixes; thus Horspool admits that his algorithm has a worst-case complexity of  $\Theta(mn)$ , although he can also claim that it minimizes the average number of character comparisons.

Sunday achieves even better performance by applying the **bad character rule** to a position in  $T$  that has not yet been considered.<sup>8</sup> The simplest form of this approach, called the **QuickSearch algorithm**, actually employs the **naïve algorithm**, then considers the next character in  $T$  beyond the current window, which would have to be part of subsequent matches. QuickSearch thus benefits from frequent shifts of  $m + 1$ . More subtle versions first compare characters from  $P$  that are most likely to cause a mismatch (e.g., infrequent characters), or most likely to lead to *a long shift in the event of a mismatch*. Conventional benchmarks have shown Sunday's refinements to be superior to previous algorithms, precisely because it makes the fewest character comparisons.

### Seminumerical Algorithms

An entirely different class of algorithms, **seminumerical algorithms**, preprocess  $P$  in such a way as to be able to use bitwise operations on every position in  $T$ . Their complexity is thus strictly linear,  $\Theta(n)$ .

R. Baeza-Yates and G. Gonnet devised a seminumerical method called **Shift-Or**, which searches  $T$  in linear time when  $m$  is less than the machine's word size in bits.<sup>9</sup> As is typical of such algorithms, the search phase has little branching logic, and does not perform any skips, but instead uses inherently fast bitwise operations that make it very effective in practice. The **agrep algorithm**, devised by Wu and Manber, uses a similar approach, but allows wildcards, i.e., it provides a form of inexact matching.<sup>10</sup>

A different approach, using hashes instead of bit masks, is described by Karp and Rabin.<sup>11</sup> They present a **hash function**, using large prime numbers, such that the hash can easily be recalculated at every position in  $T$ . When the current hash is equal to the hash of  $P$ , then there is a match. Note that there is a very small — albeit non-zero — possibility of a false match. The **Karp-Rabin algorithm** is not generally used for exact string matching, but it is nonetheless of considerable theoretical and practical interest in this arena. Unlike previous matching algorithms that use hashes, Karp-Rabin's complexity is provable, and its innovation has proven to be applicable to two-dimensional pattern matching problems. As Baeza-Yates and Gonnet point out, the strict linearity of their algorithm means that it performs in real time: i.e., "the time-delay to process one text character is bounded by a constant." In addition, neither  $P$  nor  $T$  needs to be buffered, which means that seminumerical algorithms are therefore quite applicable to cases where  $T$  is too large to buffer. Finally, seminumerical algorithms are more adaptable to inexact matching problems, where algorithms such as Boyer-Moore have not been.

### Sun Kim's Algorithm

Sun Kim recently published an ingenious algorithm that does not fall into either category.<sup>12</sup> This algorithm divides  $T$  into contiguous partitions of size  $|P|$ . Clearly, if the last character  $c$  of a partition does not occur in  $P$ , then that partition cannot overlap a match. The search can then skip forward at least a constant amount whenever there is such a mismatch, namely,  $m$ . If preprocessing did find an occurrence of  $c$  in  $P$ , then  $P$  is aligned with  $T$  accordingly and compared using the **naïve algorithm**. This insight alone leads to a faster algorithm than Sunday's. The **Sun Kim algorithm's** very simple calculation of a shift is then combined with Sunday's insight to offer an even faster algorithm that can often skip by  $2m$ .

### Implications and Applications

One can expect from experience that most search patterns will be short. In addition, the progress of refinement to Boyer and Moore's **occurrence rule** suggests that most search patterns do not have semiperiods (i.e., repeated suffixes or prefixes), which implies that in practice the **match rule** is less effective than the **occurrence rule**

when the alphabet  $\Sigma$  is of a reasonable size. When the alphabet is smaller, then false partial matches and the attendant small shifts are more likely, and every algorithm's performance degrades.

Patterns from **the genetic alphabet** — which consists of only four characters,  $A$ ,  $C$ ,  $T$  and  $G$  — present a very significant challenge in the exact matching arena. Baeza-Yates' solution to such a problem was to artificially increase the alphabet size by a technique called **k transformation**: each character  $S[i]$  in string  $S$  is encoded as the concatenation of the  $k$  characters from  $S[i]$  to  $S[i + k - 1]$ . This transformation can further be represented in a new, compact encoding, since only two bits are actually needed to represent  $A$ ,  $C$ ,  $T$  and  $G$ .

While work continues on the underlying problem, the wider areas to which these algorithms are currently applied introduce additional problems that are beyond the scope of this paper. First, searching for multiple patterns is an older problem, often with different solutions.<sup>13</sup> The *textual corpora* to be searched have obviously increased in both size and kind. Some very significant areas of current research include **network intrusion detection**, **molecular biology**, and **large, compressed databases**. The amount of information in molecular biology alone is known to double every eighteen months. In this field, as in others, matching algorithms are so fundamental that they have been implemented in hardware.

- 1 An overview of exact matching algorithms, along with Java applets demonstrating their behavior, can be found at <http://www-igm.univ-mlv.fr/~lecroq/string/>.
- 2 See A. V. Aho, "Algorithms for Finding Patterns in Strings," in J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, vol. A, *Algorithms and Complexity* (Amsterdam: Elsevier, 1990): 255-300.
- 3 M. Main and R. Lorentz, "An  $O(n \log n)$  Algorithm for Finding All Repeats in a String," *Journal of Algorithms* 5 (1984): 422-32; D. E. Knuth, J. H. Morris, and V. B. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal of Computing* 6 (1977): 323-50.
- 4 R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM* 20 (1977): 762-772.
- 5 R. Cole, "Tight Bounds on the Complexity of the Boyer-Moore Pattern Matching Algorithm," *SIAM Journal on Computing* 23.5 (1994): 1075-1091.
- 6 Crochemore, M., Czumaj A., Gasieniec L., Jarominek S., Lecroq T., Plandowski W., Rytter W., 1992, "Deux méthodes pour accélérer l'algorithme de Boyer-Moore," in D. Krob ed., *Théorie des Automates et Applications*, Actes des 2e Journées Franco-Belges, Rouen, France, 1991, pp 45-63, pur 176; Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., Rytter, W., 1994, "Speeding up two string matching algorithms," *Algorithmica* 12(4/5):247-267; Hume A. and Sunday D.M. , 1991. "Fast string searching," *Software—Practice & Experience* 21(11):1221-1248; Apostolico A., Giancarlo R., 1986, "The Boyer-Moore-Galil String Searching Strategies Revisited," *SIAM Journal on Computing* 15(1):98-105.
- 7 R. N. Horspool, "Practical Fast Searching in Strings," *Software—Practice & Experience* 10.6: 501-506.
- 8 Daniel M. Sunday, "A Very Fast Substring Search Algorithm," *Communications of the ACM* 33.8 (August 1990): 132-142.
- 9 R. Baeza-Yates, G. H. Gonnet, "A New Approach to Text Searching," *Communications of the ACM* 35.10 (1992): 74-82; R. Baeza-Yates, G. Navarro and B. Ribeiro-Neto, "Indexing and Searching," in *Modern Information Retrieval* (Reading, MA: Addison-Wesley) 191-228.
- 10 S. Wu and U. Manber. "Fast text searching allowing errors." *Communications of the ACM* 35.10 (1992) 83-91. See also <http://glimpse.cs.arizona.edu/>.
- 11 R. M. Karp and M. O. Rabin. "Efficient Randomized Pattern-Matching Algorithms." *IBM J. Res. Dev.* 31.2 (1987) 249-260.
- 12 Sun Kim. "A New String-Pattern Matching Algorithm Using Partitioning and Hashing Effectively." *Journal of Experimental Algorithmics* 4 (December 1999) 1-18.
- 13 The keyword tree was first presented by A. Aho and M. Corasick. "Efficient String Matching: An Aid to Bibliographic Research." *Communications of the ACM* 18 (1975): 333-340. See also Sun Kim and Yanggon Kim, "A Fast Multiple String-Pattern Matching Algorithm," in *Proceedings of the 17th AoM/IAoM Conference on Computer Science 1-7*, which describes a seminumerical algorithm suitable for bibliographic and biomolecular searching.