Boyer-Moore Strategy to Efficient Approximate String Matching

Nadia El-Mabrouk and Maxime Crochemore

IGM, Université Marne la Vallée, 2 rue de la Butte Verte, 93166 Noisy Le Grand Cedex

Abstract. We propose a simple but efficient algorithm for searching all occurrences of a pattern or a class of patterns (length $m$) in a text (length $n$) with at most $k$ mismatches.

This algorithm relies on the Shift-Add algorithm of Baeza-Yates and Gonnet [6], which involves representing by a bit number the current state of the search and uses the ability of programming languages to handle bit words. State representation should not, therefore, exceed the word size $\omega$, that is, $m(\lceil \log_2 (k+1) \rceil + 1) \leq \omega$. This algorithm consists in a preprocessing step and a searching step. It is linear and performs $3n$ operations during the searching step.

Notions of shift and character skip found in the Boyer-Moore (BM) [9] approach, are introduced in this algorithm. Provided that the considered alphabet is large enough (compared to the Pattern length), the average number of operations performed by our algorithm during the searching step becomes $n(2 + \frac{k+1}{m-k})$.

1 Introduction

Our purpose is approximate matching of a pattern or a class of patterns in a text, all sequences of characters or classes of characters from a finite alphabet $\Sigma$. Errors considered here are mismatches. A class of patterns, is a set of patterns with don’t care symbols, patterns containing the complementary of a character or any other class of characters. Such a problem has a lot of applications, in particular in molecular biology for predicting potential nuclear gene-coding sequences in genomic DNA sequences. In fact, exact string matching is not sufficient since gene-coding sequences are in general only partially and approximately specified.

Concerning exact string matching, algorithms based on the Boyer-Moore (BM) [9, 13] approach are the fastest in practice. Such algorithms are linear and may even have a sublinear behaviour, in the sense that every character in the text need not be checked. In certain cases, text characters can be “skipped” without missing a pattern occurrence. The larger the alphabet and the longer the pattern, the faster the algorithm works.

Various algorithms have been developed for searching with $k$ mismatches all occurrences of a pattern (length $m$) in a text (length $n$), both defined over an alphabet $\Sigma$ (length $c$). Running times have ranged from $O(mn)$ for the naive algorithm, to $O(kn)$ [15, 11] or $O(n \log m)$ [12]. The first two algorithms consist in a preprocessing step and a searching step. Grossi and Luccio algorithm [12]
uses the suffix tree. Other algorithms have used the BM approach in approximate string matching [4, 18]. Running times are \(O(kn)\) for Baeza-Yates and Gonnet [4] and \(O(kn(\frac{1}{m-k} + \frac{k}{k}))\) for Tarhio and Ukkonen [18]. The problem of approximate matching of a class of patterns was also studied [2, 1, 5], especially in the case of patterns with don’t care symbols [10, 17, 16, 3, 8, 14]. Fisher et Paterson [10] developed an \(O(n \log c \log^2 m \log \log m)\) time algorithm based on the linear product. Abrahamson [1] extended this method for generalized string pattern. Pinter [17] has used the Aho and Corasick automaton [2] for searching a set of patterns. Other algorithms have considered the problem of exact matching of patterns with variable length don’t cares [16, 8, 14]. As for Akutsu [3], he developed an \(O(\sqrt{kn} \log c \log^2 m \log \log \frac{m}{k})\) time algorithm for searching a pattern with don’t cares in a text with don’t cares.

In 1992, several new algorithms for approximate string matching were published [6, 20, 7]. They combine both speed and programming practicality, in contrast with older results, most of which being mainly of theoretical interest. Moreover, they are flexible enough to allow searching for a class of patterns. These algorithms consist in a pattern preprocessing step and a searching step. They are all based on the same approach, consisting in finding, at a given position in the text, all approximate pattern prefixes ending at this position. Speed is increased by representing the state of the search as a bit number [6, 20] or an array [7], and by using the ability of programming languages to handle bit words.

Nevertheless, these algorithms are based on a naive approach and process each character of the text. Our goal is to speed up searching by using a BM strategy and including notions of shift and character skip.

We have chosen to consider such an improvement in the case of the Shift-Add algorithm of Baeza-Yates and Gonnet [6]. The main idea of Shift-Add is to represent the state of the search as a bit number, and perform a few simple arithmetic and logical operations. Provided that representations don’t exceed the word size \(\omega\), that is \(m(\lceil \log_2 (k + 1) \rceil + 1) \leq \omega\), each search step does exactly a shift, a test and an addition. Therefore, this algorithm runs in \(O(n)\) time and the searching step does \(3n\) operations. We developed an algorithm combining the practicality of the Shift-Add method and the speed of the BM approach. Provided that the considered alphabet is large enough compared to \(m\), our new algorithm performs on average \(n \left(2 + \frac{k+4}{m-k}\right)\) operations during the searching step.

The paper is organized as follows. Section 2 summarises the algorithm Shift-Add, in the case of exact or approximate matching of a pattern or a class of patterns. Section 3 develops the adaptation of the BM approach to the Shift-Add method. An improvement of this last algorithm is given in Section 4. Finally, section 5 gives experimental results obtained with both algorithms.

## 2 Shift-Add Algorithm

Let \(P = p_1 \cdots p_m\) be a pattern and \(t = t_1 \cdots t_n\) be a text over a finite alphabet \(\Sigma\). The problem is to find in \(t\) all occurrences of \(P\) with at most \(k\) mismatches \((0 \leq k \leq m)\). The main idea is to represent the state of the search as a bit number, and to perform a few simple arithmetic and logical operations. Provided that representations don’t exceed the word size \(\omega\), that is \(m(\lceil \log_2 (k + 1) \rceil + 1) \leq \omega\), each search step does exactly a shift, a test and an addition. Therefore, this algorithm runs in \(O(n)\) time and the searching step does \(3n\) operations. We developed an algorithm combining the practicality of the Shift-Add method and the speed of the BM approach. Provided that the considered alphabet is large enough compared to \(m\), our new algorithm performs on average \(n \left(2 + \frac{k+4}{m-k}\right)\) operations during the searching step.