A Faster Algorithm for Approximate String Matching

Ricardo Baeza-Yates
Department of Computer Science, University of Chile

July 8, 1996

[syntax by Mireille Régnier]

Approximate string matching is one of the main problems in classical string algorithms. Given a
text of length $n$, a pattern of length $m$, and a maximal number of errors allowed, $k$, we want to find
all text positions where the pattern matches the text up to $k$ errors. Errors can be substituting,
deleting or inserting a character. The solutions to this problem differ if the algorithm has to be
on-line (that is, the text is not known in advance) or off-line (the text can be preprocessed). In
this paper the first case is studied, where the classical dynamic programming solution is $O(mn)$.

In the last years several algorithms have been presented that achieve $O(kn)$ comparisons in
the worst-case [13, 6, 7] or in the average case [14, 6], by taking advantage of the properties of the
dynamic programming matrix. In the same trend is [3], with average complexity $O(kn/\sqrt{c})$
($c$ is the alphabet size). The algorithms which are $O(kn)$ in the worst case tend to involve too
much overhead, and are not competitive in practice. Other approaches attempt to filter the text,
reducing the area in which dynamic programming needs to be used [12, 15, 11, 10, 4, 5]. These
algorithms achieve sublinear expected time in many cases ($O(kn \log c / m)$ is a typical figure) for
moderate $k/m$ ratios, but the filtration is not effective for larger ratios. A simple and fast filtering
technique is shown in [2], which yields an $O(n)$ algorithm for moderate $k/m$ ratios. Yet other
approaches use bit-parallelism [1] in a RAM machine of word length $O(\log n)$ to reduce the number
of operations. [9] achieves $O(kmn / \log n)$, which is competitive for patterns of length $O(\log n)$.
In [16], the cells are packed differently to achieve $O(mn \log c / \log n)$ complexity.

A new algorithm is presented which combines the ideas of taking advantage of the properties
of the matrix, filtering the text and using bit-parallelism, being faster than previous work for
moderate size patterns, as we are interested in text searching. One models the search with a non-
deterministic finite automaton (NFA) built from the pattern and using the text as input. This
automaton is simulated by an algorithm based on bit operations on a RAM machine of word
length $O(\log n)$. The algorithm achieves running time $O(n)$, independently of $k$, for small patterns
(i.e. $mk = O(\log n)$). This restricted algorithm is used to design two general algorithms.

The first one partitions the problem into subproblems, and has average time cost $O(mn / \log n)$ for
small $\alpha = k / m$ (i.e. $\alpha < 1 / \log n$), otherwise it is $O(\sqrt{mk} / \log n)$ (i.e. $O(\sqrt{kn})$ for $m = O(\log n)$,
else $O(kn)$). It involves also a cost to verify potential matches, which is shown to be not significant
for $\alpha < \alpha_1 \approx 1 - m^{1/\log m} / \sqrt{c}$. This algorithm is a generalization of an earlier heuristic [8, 2], that
reduces the problem to exact matching and is shown to be $O(n)$ for $\alpha < \alpha_0 = 1/(3 \log m)$, and
better than problem partitioning for $\alpha < \alpha'_0 \approx 1/(2^{\log m})$.

The second one partitions the automaton into subautomata, being $O(k^2 n / (\sqrt{c} \log n))$ on average.
For $\alpha > 1 - 1 / \sqrt{c}$ its worst case, $O((m - k)kn / \log n)$, dominates. This algorithm is shown to be
better than dynamic programming for $k > \log(n)/(1 - \alpha)$. One studies the optimal way to combine
the algorithms. It is shown experimentally that the hybrid algorithm is faster than previous ones, for moderate $m$. Table 1 shows the complexity.

As a corollary of the analysis, tight bounds are given for the probability of finding an occurrence of a pattern of length $m$ with $k$ errors starting at a fixed position in random text. We also show that the heuristic of [14] works $O(kn)$ on average, with a constant tighter than that of [3].

<table>
<thead>
<tr>
<th>Condition</th>
<th>Complexity</th>
<th>Method used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$mk = O(\log n)$</td>
<td>$O(n)$</td>
<td>the simple algorithm</td>
</tr>
<tr>
<td>$\alpha &lt; \alpha_0$</td>
<td>$O(n)$</td>
<td>reducing to exact match</td>
</tr>
<tr>
<td>$\alpha_0 &lt; \alpha &lt; \alpha_1$</td>
<td>$O(\sqrt{mk/\log n})$</td>
<td>exact match if $\alpha &lt; \alpha_0$, else problem partitioning</td>
</tr>
<tr>
<td>$\alpha &gt; \alpha_1 \land k &lt; \log n/(1 - \alpha)$</td>
<td>$O((m - k)kn/\log n)$</td>
<td>automaton partitioning</td>
</tr>
<tr>
<td>$\alpha &gt; \alpha_1 \land k &gt; \log n/(1 - \alpha)$</td>
<td>$O(mn)$</td>
<td>plain dynamic programming</td>
</tr>
</tbody>
</table>

**Table 1.** Complexity of the hybrid algorithm.

**Bibliography**


