# Scaled and Permuted String Matching

Ayelet Butman* Holon Academic Institute of Technology

Revital Eres† University of Haifa

Gad M. Landau‡ University of Haifa & Polytechnic University

**Abstract**

The goal of *scaled permuted string matching* is to find all occurrences of a pattern in a text, in all possible scales and permutations. Given a text of length $n$ and a pattern of length $m$ we present an $O(n)$ algorithm.

*Keyword:* Algorithms; Approximate String Matching; Permutations; Scalings

## 1 Introduction

The well known string matching problem that appears in all algorithm textbooks has as its input a text $T$ of length $n$ and a pattern $P$ of length $m$ over a given alphabet $\Sigma$. The output is all text locations where there is an exact match of the pattern. This problem has received much attention, and many algorithms have been developed to solve it (e.g. [6, 13, 14]). A detailed modern view of stringology can be found in a number of published books [5, 7, 11].

Most recent work has dealt with inexact matches. Many types of differences between the patterns were defined, for example, errors (Hamming distance, LCS [12], Edit distance [15]), rotations [1, 3, 9, 10], scaling [2, 4], or permutation. Most of the theoretical work has dealt with one type of difference at a time. This paper is one of the first attempts to deal with two types of differences together – scaling and permutation.

**Definition 1 Scaled permuted string matching**

**Input:** *A pattern* $P = p_1 \cdots p_m$ *and a text* $T = t_1 \cdots t_n$ *both over alphabet* $\Sigma$.

**Output:** *All positions in* $T$ *where an occurrence of a permuted copy of the pattern* $P$*, scaled to* $k$ *starts* ($k = 1, \ldots, \lfloor \frac{n}{m} \rfloor$)*. The pattern is first permuted and then scaled.*

**Example:** The string *bbbbaabbaaccaacc* is a scaled (to 2) permutation of *baabbacc*.

*Holon Academic Institute of Technology, Israel, `butmosh@zahav.net.il`.

†Department of Computer Science, Haifa University, Haifa 31905, Israel, phone: (972-4) 828-8376, FAX: (972-4) 824-9331; email: `revitale@cslx.haifa.ac.il`; partially supported by the Israel Science Foundation grant 282/01.

‡Department of Computer Science, Haifa University, Haifa 31905, Israel, phone: (972-4) 824-0103, FAX: (972-4) 824-9331; Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840; email: `landau@poly.edu`; partially supported by NSF grant CCR-0104307, by the Israel Science Foundation grant 282/01, and by IBM Faculty Award.

The scaled (only) string matching problem is a well studied problem. The algorithm presented in [4], which follows the method described in [8], achieves a linear running time for the scaled string matching problem. In [2] the case where the scaling of the pattern is by real numbers was considered, and a linear time algorithm was introduced.

An algorithm for the permuted string matching problem over run-length encoded strings is described in section 2. In section 3 we present the main result of this paper, an algorithm that solves the scaled permuted string matching problem in $O(n)$ time and space. Open problems are given in section 4.

## 2 Permuted string matching over run-length encoded text

The permuted string matching problem over uncompressed text is simply solved. A sliding window of size $|P|$ can be moved over $T$ to count, for each location of $T$, the order of statistics of the characters. Obviously, this can be done in $O(n)$ time.

The run-length of a string $S$ is a popular encoding method. According to this encoding $S$ can be described as a sequence of ordered pairs $(\sigma, i)$, often denoted by the *symbol* $\sigma^i$, each consisting of an alphabet *character* $\sigma$ and an integer $i$. Each pair corresponds to a run in $S$, consisting of $i$ consecutive occurrences of $\sigma$.

Let $T'$ be the run-length compressed version of $T$ where $T' = \sigma_1^{r_1} \cdots \sigma_{|T'|}^{r_{|T'|}}$. Similarly, $P'$ is the run-length compressed pattern. The pattern can be permuted, and therefore, in each location of the text we check if the order of statistics of the characters is equal to that of the pattern. As a result, a better compression can be achieved. Symbols with the same character are compressed. For example, let $P = aabbbaccaab$, its run-length compressed version is $P' = a^2b^3a^1c^2a^2b^1$ and a permuted run-length compressed version is $P'' = a^5b^4c^2$. The technique we use is similar to the sliding window technique: a window is shifted on $T'$ from left to right in order to locate all the matches. The window is a substring of $T'$ that represents a candidate for a match. Unlike the simple algorithm, this time the window size is not fixed.

We will define a *valid* window as a substring of $T'$ that fulfills the following two properties:
*sufficient* – The number of times each character appears in the window is at least the number of times it appears in the pattern.
*minimal* – Removing the rightmost or the leftmost symbol of the window violates the *sufficient* property.

Note that: (a) The *valid* window property does not ensure a match. (b) If a permutation of the pattern occurs in a *valid* window of $T'$, ${\sigma_i}^{r_i} \cdots {\sigma_j}^{r_j}$, then only the characters $\sigma_i$ and $\sigma_j$ can appear more times in this window than they appear in $P''$. (c) If $\sigma_i = \sigma_j$ then the pattern may occur more than once in the *valid* window. Also, if $\sigma_{i-1} = \sigma_j$ ($\sigma_i = \sigma_{j+1}$) the pattern may occur more than once in ${\sigma_{i-1}}^{r_{i-1}} \cdots {\sigma_j}^{r_j}$ (${\sigma_i}^{r_i} \cdots {\sigma_{j+1}}^{r_{j+1}}$). (d) A permuted pattern occurs in the text only in a *valid* window (including the symbols on the left and right of the window).

The algorithm scans the text, locates all *valid* windows and finds the ones in which a permuted copy of the pattern occurs. During the scan of the text, given a *valid* window, it is trivial to check if it contains a match. Hence, we will describe only how to locate all *valid* windows.

Note that given a text $T' = \sigma_1^{r_1} \cdots \sigma_{|T'|}^{r_{|T'|}}$: (a) At most one *valid* window may start on each $\sigma_i^{r_i}$. (b)

A *valid* window does not contain another *valid* window.

The *valid* windows are found by scanning the text from left to right, using two pointers, *left* and *right*. To discover each *valid* window, the *right* pointer moves first to find a *sufficient* window and then the *left* pointer moves to find the *valid* window within the *sufficient* window. Each move of the *right* pointer increases the size of the window. The right pointer moves as long as deleting the leftmost symbol of the window violates the *sufficient* property of the window. When this symbol can finally be removed, the *right* pointer stops and the *left* pointer starts moving. Each move of the *left* pointer decreases the size of the window. The pointer moves as long as deleting the leftmost symbol of the window does not violate the *sufficient* property of the window. At this point, a new *valid* window has been found.
**Example:** Let $P'' = a^2b^3c^2d^2$ and $T' = c^3a^2c^2a^3d^2b^3c^1$ then $c^3a^2c^2a^3d^2b^3$ is the first *sufficient* window, and $c^2a^3d^2b^3$ is the first *valid* window (but not a match).

**Claim 1** The algorithm finds all (and only) *valid* windows.

Proof: The algorithm reports only *valid* windows. We will prove by contradiction that the algorithm finds all the *valid* windows. Denote by $\prod_1$ and $\prod_2$ two consecutive *valid* windows that are discovered by the algorithm, and by $i_{left_1}$, $i_{right_1}$, $i_{left_2}$ and $i_{right_2}$ the left and right pointers of those windows respectively. Assume that there exists a *valid* window $\prod_3$ (with left and right pointers $i_{left_3}$ and $i_{right_3}$ respectively) between $\prod_1$ and $\prod_2$ ($i_{right_1} < i_{right_3} < i_{right_2}$) that the algorithm does not discover. By the *minimal* propriety we get that $i_{left_1} < i_{left_3}$. After reporting $\prod_1$ the algorithm looks for the next *valid* window. During the scanning of the right pointer the algorithm passes $i_{right_3}$ and does not stop, which means that the window $i_{left_1} + 1 \cdots i_{right_3}$ does not satisfy the sufficient property. Since, $i_{left_1} + 1 \leq i_{left_3}$ we conclude that the window $\prod_3$ does not satisfy the sufficient property as well, and hence, it is not *valid*. ∎

**Time complexity:** We assume that $|\Sigma|$ is $O(|P''|)$, hence, the time complexity of the algorithm is $O(|P''| + |T'|)$. In case the input pattern is not given in a permuted run-length compressed format, an $O(|P|)$ time preprocessing step is added.

## 3 A linear time algorithm for the scaled permuted string matching problem

The algorithm is composed of two stages:
1. Preprocessing the text $T'$. Computing compact copies of the text for each possible scale $1 \leq s \leq \frac{n}{m}$.
2. Applying the permuted string matching over the run-length encoded text algorithm (section 2) on the copies of the text.

**Observation 1** If a permutation of $P$ scaled to $s$ occurs in ${\sigma_i}^{j_i} \cdots {\sigma_k}^{j_k}$ then $j_{i+1}, \ldots, j_{k-1}$ are multiples of $s$, and $j_i, j_k \geq s$.

Following the above observation, we compute for each scale $s$ a compact text $T'_s$ in the following two steps: *Step 1:* Locate all the regions in $T'$ where the symbols appear with multiples of $s$. Add the symbol $ as a separator between the regions. *Step 2:* Expand these regions to include the symbols on their boundaries. In order to simplify the computation of Stage 2, a symbol ${t_j}^{r_j}$ of $T'$ is replaced in $T'_s$ by ${t_j}^{\lfloor \frac{r_j}{s} \rfloor}$.

**Step 1. Locating the regions** – $T'$ is scanned from left to right. Consider a symbol ${t_i}^{r_i}$. A new symbol ${t_i}^{\frac{r_i}{s}}$ is added to $T'_s$ if $r_i$ is a multiple of $s$. The following code describes this idea:

**Step 1 – The parallel construction of the new text**

For every symbol in $T'$ do:
  { *let $a^r$ be the current symbol being examined* }
  $s = 1$
  Repeat Until $s > \sqrt{r}$
    If ($r$ mod $s = 0$) Then
      Add $a^{\frac{r}{s}}$ to $T'_s$
      { *skip the next line if $s = \sqrt{r}$* }
      Add $a^s$ to $T'_{\frac{r}{s}}$
    $s = s + 1$

Note that the efficiency of this procedure depends on the method that finds all the divisors of an integer. In the above example we used a naive method. A new symbol that is added at the end of $T'_s$ may continue a region or start a new one. In the second case we add a separator (\$) between the regions.

**Step 2. Expansion of the regions** – The last refinement is done by scanning each $T'_s$ text from left to right and expanding all the regions we generated in step 1. In the next procedure we deal with symbols that appear on the left side of a \$ separator in $T'_s$. The opposite case is treated in the same way:

**Step 2**

For every \$ separator in $T'_s$ do:
  {let ${t_i}^{\frac{r_i}{s}}$ be the symbol appearing on the left side of the current \$ separator on $T'_s$, and let ${t_{i+1}}^{r_{i+1}}$ be the adjacent symbol to ${t_i}^{r_i}$ on $T'$}
  If $(r_{i+1} > s)$ then
    Add ${t_{i+1}}^{\lfloor \frac{r_{i+1}}{s} \rfloor}$ to $T'_s$ between ${t_i}^{\frac{r_i}{s}}$ and the \$ separator

**Example:** Let $T' = a^6b^2c^4a^3d^5b^9d^2c^8b^4a^7$, the new text after applying step 2 is:

$T'_1 =\$\ a^6b^2c^4a^3d^5b^9d^2c^8b^4a^7\$$ , $T'_2 = \$a^3b^1c^2a^1\$b^4d^1c^4b^2a^3\$$, $T'_3 = \$a^2\$c^1a^1d^1\$d^1b^3\$$, $T'_4 = \$c^1\$c^2b^1a^1\$$, $T'_5 = \$d^1b^1\$$ $T'_6 = \$a^1\$$, $T'_7 = \$a^1\$$, $T'_8 = \$c^1\$$, $T'_9 = \$b^1\$$

Stage 2 runs the permuted string matching over a run-length encoded text algorithm (section 2) on all the new compact texts.

**Time complexity:** The input to our problem is a compressed text $T' = {t_1}^{r_1}...{t_k}^{r_k}$, whose original length is $n$, and a pattern $P''$ of length $|P''|$ (or a pattern $P$ of length $m$). Both the pattern and the text are over alphabet $\Sigma$. The following claim shows that the total length of all compact new texts is linear.

**Claim 2** *The total length of all the new texts $T'_s$ ($1 \le s \le \frac{n}{m}$) is $O(n)$.*

**Proof:** In step 1, we consider each symbol ${t_i}^{r_i}$ in $T'$, and the number of new symbols that we produce from ${t_i}^{r_i}$ is bounded by $2\sqrt{r_i}$. In addition we may add a \$ separator to each new symbol. In step 2, two new symbols may be added to each \$ separator. Hence the total length of all new texts is: $8 \cdot \sum_{i=1}^{k} \sqrt{r_i} = O(n)$. ∎

The running time of both Stage 1 and Stage 2 is bounded by the length of the new texts, hence the total time complexity is $O(n)$.

## 4 Open problems

The algorithm described in this paper is the first to deal with scaling and permutation. We considered the case in which the pattern is first permuted and then scaled. The first challenge is to design an $o(nm)$ algorithm for the case in which the pattern is first scaled and then permuted. We also dealt with integer scales. The second challenge is to deal with scales that are real numbers.

**Acknowledgments.** The authors are grateful to the referees for their helpful comments.

## References

[1] A. Amir, A. Butman, M. Crocehmore, G.M. Landau, and M. Schaps. Two-dimensional pattern matching with rotations. *Theoretical Computer Science.* 314(1-2):173–187, 2004.

[2] A. Amir, A. Butman and M. Lewenstein. Real scaled matching. *Information Processing Letters.* 70(4):185–190, 1999.

[3] A. Amir, O. Kapah and D. Tsur. Faster two dimensional pattern matching with rotations. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM 2004)*, volume 3109 of *LNCS*. Springer, 2004.

[4] A. Amir, G.M. Landau and U. Vishkin. Efficient pattern matching with scaling. *Journal of Algorithms.* 13(1):2–32, 1992.

[5] A. Apostolico and Z. Galil (editors). *Pattern Matching Algorithms.* Oxford University Press, 1997.

[6] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. ACM.* 20:762–772, 1977.

[7] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

[8] T. Eilam-Tsoreff and U. Vishkin. Matching patterns in strings subject to multilinear transformations. *Theoretical Computer Science.* 60(3):231–254, 1988.

[9] K. Fredriksson, G. Navarro, and E. Ukkonen. Optimal exact and fast approximate two dimensional pattern matching allowing rotations. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002)*, volume 2373 of *LNCS*, pages 235–248. Springer, 2002.

[10] K. Fredriksson and E. Ukkonen. A rotation invariant filter for two-dimensional string matching. In *Proc. 9th Annual Symposium on Combinatorial Pattern Matching (CPM 1998)*, volume 1448 of *LNCS*, pages 118–125. Springer, 1998.

[11] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[12] D.S. Hirshberg. Algorithms for the longest common subsequence problem. *JACM.* 24(4):664–675, 1977.

[13] R. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Res. and Dev.*, pages 249–260, 1987.

[14] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Computing.* 6:323–350, 1977.

[15] V.I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.* 10:707–710, 1966.