Algorithmica (2011) 60: 335–348
DOI 10.1007/s00453-009-9345-9

Approximate Pattern Matching with the L_1, L_2 and L_∞ Metrics

Ohad Lipsky · Ely Porat

Received: 21 June 2007 / Accepted: 16 July 2009 / Published online: 5 August 2009

Abstract Given an alphabet $\Sigma = \{1, 2, \ldots, |\Sigma|\}$ text string $T \in \Sigma^n$ and a pattern string $P \in \Sigma^m$, for each $i = 1, 2, \ldots, n - m + 1$ define $L_p(i)$ as the p-norm distance when the pattern is aligned below the text and starts at position i of the text. The problem of pattern matching with L_p distance is to compute $L_p(i)$ for every $i = 1, 2, \ldots, n - m + 1$. We discuss the problem for $d = 1, 2, \infty$. First, in the case of L_1 matching (pattern matching with an L_1 distance) we show a reduction of the *string matching with mismatches* problem to the L_1 matching problem and we present an algorithm that approximates the L_1 matching up to a factor of $1 + \varepsilon$, which has an $O(\frac{1}{\varepsilon^2} n \log m \log |\Sigma|)$ run time. Then, the L_2 matching problem (pattern matching with an L_2 distance) is solved with a simple $O(n \log m)$ time algorithm. Finally, we provide an algorithm that approximates the L_∞ matching up to a factor of $1 + \varepsilon$ with a run time of $O(\frac{1}{\varepsilon} n \log m \log |\Sigma|)$. We also generalize the problem of String Matching with mismatches to have weighted mismatches and present an $O(n \log^4 m)$ algorithm that approximates the results of this problem up to a factor of $O(\log m)$ in the case that the weight function is a metric.

Keywords Design and analysis of algorithms · Combinatorial algorithms on words · Approximate string matching · Hamming distance

Research supported in part by US-Israel Binational Science Foundation.

O. Lipsky · E. Porat (✉)
Department of Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel
e-mail: porately@cs.biu.ac.il

O. Lipsky
e-mail: lipsky@gmail.com

1 Introduction

The last few decades have seen the evolution of pattern matching from a concern with combinatorial solution of the exact string matching problem [12, 20] to an area concerned with approximate matching of various relationships motivated by a wide range of scientific and business applications. To this end, two new paradigms were needed—"Generalized matching" and "Approximate matching".

In generalized matching the input is still a text and pattern but the "matching" relation is defined differently. The output consists of all start locations in the text where the pattern "*matches*" under the new definition of "*match*". The different applications define the matching relation. An early generalized matching was the string matching with don't cares defined by Fischer and Paterson [12]. Another example of a generalized matching problem is the *less-than* matching [2] problem defined by Amir and Farach. In this problem both texts and patterns are strings of numbers. All the text locations, where every pattern numbers is less than its corresponding text numbers, are sought. Amir and Farach showed that the less-than-matching problem can be solved in $O(n\sqrt{m}\log m)$ time. Other examples of generalized matching are parameterized matching [3, 6] and shift matching [8]. Lower bound results for generalized matching can be found in [25].

The second important pattern matching paradigm and the one with which we are concerned in this paper is approximate matching. In approximate matching, a tolerance is specified and a distance function is defined; then all the locations in the text are sought where the distance between the pattern and the aligned substrings of the text is within the tolerated bound, under the given distance function. Some examples that motivate approximate matching include finding a "close" mutation in computational biology, transmission with noise in communications, compensating for common typing errors in texts, and adjusting for lossy compressions, occlusions, scaling, affine transformations or dimension loss in multimedia.

When dealing with strings of characters the most widely studied distance function is Levenshtein's edit distance [13, 21]. The edit distance between two strings is defined as the minimum number of character insertions, deletions and changes needed to convert one string to the other. Let n be the text length and m the pattern length. The string edit distance matching problem is to compute the minimum string edit distance between the pattern and any prefix of $T[i,\dots,n]$ for each i. Masek and Paterson [23] provide an $O(nm/\log n)$ algorithm for this problem. A generalization of this problem is the *Weighted Edit Distance*, where each edit operation is given a different cost which can also depend on the symbol inserted, deleted or substituted. However there is no better time bound than $O(\min(nm/\log n, nk))$ even for the simple case of equal weights.

Hamming distance is another well known distance function which concern strings of characters. It is the number of characters substitutions to convert one string to the other. The string matching with mismatches problem is to find the Hamming distance between the pattern and every substring of the text of equal length. By applying methods similar to those of Fischer and Paterson [12] we can show that the string matching with mismatches problem can be solved in $O(\min(|\Sigma|, m)n\log m)$ time. For given constant size alphabets the time needed is $O(n\log m)$. Abrahamson [1] developed an algorithm that solves this problem for general alphabets in $O(n\sqrt{m \log m})$ time. It is an open question whether this bound is tight or it can be reduced. Karloff [19] introduced an algorithm that approximate the Hamming distance up to a factor of $1+\epsilon$ and works in time $O(\frac{1}{\epsilon^2} n \log^3 m)$. Indyk [14] provided an improved algorithm with the same approximation factor, that runs in time $O(\frac{1}{\epsilon^2} n \log n)$. Atallah, Chyzak and Dumas [5] present an algorithm for estimating the number of matches for all text locations in $O(n \log m)$ running time.

Both the Hamming and the Levenshtein edit distances give a mismatch error a fixed weight, independent of the characters that were exchanged. We discuss a generalization for the problem of String Matching with Mismatches. The problem of *String Matching with Weighted Mismatches* defined in [24], informally, is to sum up the weighted mismatches between the pattern and every substring of the text of equal length. Cormode and Muthukrishnan [10] used technique of embedding into metric vector spaces in order to approximate the string edit distance up to a factor of $O(\log n \log^* n)$ with a running time of $O(n \log n)$. One of the results of this paper is that the String Matching with Weighted Mismatches can be solved deterministically in $O(n|\Sigma| \log m)$ time, and can be approximated up to a factor of $O(\log \Sigma)$, using embedding in a time of $\tilde{O}(n)$ (linear up to a logarithmic factor), if the weight function is a metric.

In many aspects of science researchers deal with strings of numbers, which are also referred to as *time-series data*. In meteorology they measure temperature, air pressure, wind speed, etc. over time. In the Stock Market or in the Foreign Exchange market we keep a record of the stock price or the exchange rate over time. Another area which deals with such strings is Music Information Retrieval (MIR) [26]. If we wish to seek patterns in such strings of numbers, it would not be realistic to seek the exact same values, rather than "close" instances of a pattern. When dealing with strings of numbers the most common distance metrics are the L_1, L_2 and L_∞ Minkowsky norm metrics. The 1-norm distance is more colorfully called the taxicab norm or Manhattan distance, because it is the distance a car would drive in a city laid out in square blocks. The 2-norm distance is the standard Euclidean distance, which is a generalization of the Pythagorean theorem to more than two coordinates. It is what would be obtained if the distance between two points were measured with a ruler: the "intuitive" idea of distance. L_∞ is known as the Maximum metric.

Therefore, we propose new pattern matching problems: The string L_1 distance matching problem (L_1 Matching, for short) ask for the L_1 distance between the pattern and every substring of the same length of the text. Similarly we define the String L_2 Distance Matching (L_2 Matching) and the String L_∞ Distance Matching (L_∞ Matching) problems.

The problem of searching for patterns in time-series data has been extensively studied for indexing and searching time-series databases. In this case, the problem is to preprocess the data so as to support fast searches for patterns [7, 11]. Another similar problem is looking for some periodicity along the data [17]. Most of the algorithms used for these variants apply L_2 as the distance function. Moreover, most of them are also based on heuristics, and do not have any "worst case" time analysis.

In this paper we provide an algorithm for the L_2 matching problem, and approximation algortihms for L_1 and L_∞. Our algorithm for L_2 work in time $O(n \log m)$ and our approximation algortihms work in time $\tilde{O}(n)$.

A deterministic algorithm for the L_1 matching problem, which works in $O(|\Sigma| n \log m)$ time can be constructed using a method similar to the one used in [18]. For given constant size alphabets, the runtime is $O(n \log m)$, but for large alphabets (i.e. $|\Sigma| = n$) it is worse than the naive algorithm which works in $O(nm)$ time. This time complexity leads us to assume that the main obstacle to constructing an efficient algorithm (i.e. $\tilde{O}(n)$ time) is the alphabet size. Therefore, we have sought a method to reduce the alphabet size. In [18] techniques for dividing the alphabet into frequent and non-frequent symbols, similar to those in [1], provided an algorithm that runs in $O(n\sqrt{m \log m})$ time. In [22] we show that the string matching with mismatches problem can be linearly reduced to the L_1 matching problem. This fact has led us to look for an approximation algorithm rather than an exact solution as that seems to be difficult to construct in $\tilde{O}(n)$ time. A randomized algorithm that approximates the L_1 matching results up to a factor of $1+\varepsilon$ with probability $1-\delta$ can be constructed based on [15], with a run time of $O(\frac{1}{\varepsilon^2} n \log \frac{1}{\delta} \log m)$. We present a deterministic algorithm for approximating the L_1 distances which has a runtime of $O(\frac{1}{\epsilon^2} n \log m \log |\Sigma|)$.

We also present an algorithm that approximates the L_∞ matching results up to a factor of $1+\varepsilon$ and also runs in $O(\frac{1}{\epsilon} n \log m log |\Sigma|)$ time.

All the algorithms, except the approximation method for String Matching with Weighted Mismatches, are based on convolutions, and therefore can easily be extended to include *don't care* symbols within the alphabet.

This paper is organized as follows: Sect. 2 provides definitions and preliminaries, Sect. 3 presents the algorithms for string matching with weighted mismatches and Sect. 4 describes the algorithm for the L_2 matching problem. Section 5 deals with the problem of L_1 Matching—it describes the reduction of the string matching with mismatches problem to the L_1 Matching problem and presents an approximation algorithm for the L_1 matching problem. Finally Sect. 6 discusses the problem of L_∞ Matching and presents an approximation algorithm for the problem.

2 Preliminaries and Problem Definition

All algorithms in this paper, except those in Sect. 3, deal with alphabets of the form $\Sigma = \{1, 2, \ldots |\Sigma|\}$. We assume the RAM model of computation, which allows arithmetic on $\log N$ bit numbers in $O(1)$ time, where N is the order of the maximum problem size. In addition we use n to denote $|T|$ and m to denote $|P|$. We can assume w.l.o.g. that $n \le 2m$, for otherwise we can use the method of cutting the text into n/m overlapping segments, each having length $2m$.

Let $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$ be two vectors over Σ. Then the Minkowsky L_p distance metric between x and y is defined as: $L_p(x, y) = \sqrt[p]{\Sigma_{i=1}^{n} |x_i - y_i|^p}$. The L_1, L_2 and L_∞ metrics are the well-known Manhattan, Euclidean and Max metrics, respectively. The approximation algorithms we construct for L_1 and L_∞ Matching have an approximation factor of $1 \pm \varepsilon$ and are easily changed to have a factor of $1+\varepsilon$ by running the algorithms with error bound $\frac{\varepsilon}{3}$ and multiplying the results by $1 + \frac{\varepsilon}{2}$.

Convolutions The convolution vector of two vectors t and p, denoted by $t \otimes p$, is defined as the vector w such that

$$w[i] = \sum_{j=1}^{m} t[i+j-1]p[j].$$

The convolution can be computed in $O(n \log m)$ time, in a computational model with a word size of $O(\log m)$, by using the Fast Fourier Transform (FFT) [9].

2.1 Problem Definitions

We provide more convenient definitions of the problems that are equivalent to the form in which they were introduced above. The following problems all take two input vectors, a text vector $t = t_1, t_2, \dots, t_n$, and a pattern vector $p = p_1, p_2, \dots, p_m$, where $t_i, p_j \in \{1, 2, \dots, |\Sigma|\}$. In addition, the approximate matching problems all have an exactness parameter ϵ, $0 < \epsilon < 1$. It remains to specify the output vector $O[1 : n - m + 1]$ for each problem.

String Matching with L_1 Distance (*L_1 Matching* for short)

$$O[i] = \sum_{j=1}^{m} |t_{i+j-1} - p_j|.$$

Approximate String Matching with L_1 Distance (*Approximate L_1 Matching*)

$$\hat{O}[i] \le O[i] \le (1+\epsilon)\hat{O}[i], \quad \text{where } \hat{O}[i] = \sum_{j=1}^{m} |t_{i+j-1} - p_j|.$$

String Matching with L_2 Distance (*L_2 Matching*)

$$O[i] = \sqrt{\sum_{j=1}^{m} |t_{i+j-1} - p_j|^2}.$$

String Matching with L_∞ Distance (*L_∞ Matching*)

$$O[i] = \mathrm{Max}_{j=1}^{m} |t_{i+j-1} - p_j|.$$

Approximate String Matching with L_∞ Distance (*Approximate L_∞ Matching*)

$$\hat{O}[i] \le O[i] \le (1+\epsilon)\hat{O}[i], \quad \text{where } \hat{O}[i] = \mathrm{Max}_{j=1}^{m} |t_{i+j-1} - p_j|.$$

String Matching with Weighted Mismatches

$$O[i] = \sum_{j=1}^{m} f(t_{i+j-1}, p_j).$$

3 String Matching with Weighted Mismatches

We describe an algorithm with time complexity $O(|\Sigma|n\log m)$. It work even for non-metric alphabet distance functions. This algorithm is a direct extension of the algorithm given by Fischer and Paterson in [12]. They examined a case where the distance function defined on the alphabet is:

$$f(\sigma_1,\sigma_2)=\begin{cases}1 & \sigma_1\neq\sigma_2\\ 0 & \sigma_1=\sigma_2\end{cases}$$

We generalize the problem to any given distance function $f:\Sigma\times\Sigma\to\Re$.

Algorithm steps:

1. For each $\sigma\in\Sigma$:
   a) Create t^σ by replacing every σ by 1 and every other symbol by 0.
   b) Create p^σ by replacing every $p_j\neq\phi$ by $f(p_j,\sigma)$, and ϕ by 0.[a]
   c) Compute $O_\sigma=t^\sigma\otimes p^\sigma$.
2. Compute $O\leftarrow\sum_{\sigma\in\Sigma}O_\sigma$

[a] ϕ denotes a "don't care" symbol i.e. a symbol that matches any other symbol.

It is easily seen that the algorithm computes the desired results. The time complexity of this algorithm is derived from the time needed to compute a convolution, and the size of the alphabet. Therefore the overall time needed for this algorithm is $O(|\Sigma|n\log m)$. For given constant size alphabets, the time needed is $O(n\log m)$, but for large alphabets (i.e. $|\Sigma|=m$) it is worse than the naive algorithm which works in $O(nm)$ time.

If the alphabet distance function f is a metric we can approximate the results up to a factor of $\log\Sigma$ in the $\tilde{O}(n)$ time algorithm, by embedding into L_1 (using the method of [16]) and then using the L_1 Matching approximation algorithm.

4 L_2 Matching Deterministic Algorithm

In this section we present a deterministic algorithm that solves the problem of L_2 Matching in $O(n\log m)$ time. The time complexity is derived from the time needed to compute a convolution. We use the trick of cutting the text into $2m$ consecutive overlapping pieces in order to obtain a time of $O(n\log m)$ instead of $O(n\log n)$ time, for more details see [4]. By definition we need to compute:

$$O[i]=\sqrt{\sum_{j=1}^{m}|t_{i+j-1}-p_j|^2}$$

for every $i = 1, 2, \ldots, n - m + 1$. We show how to compute

$$O[i]^2 = \sum_{j=1}^{m} |t_{i+j-1} - p_j|^2$$

and in one linear time pass on this vector we compute the desired results vector. Note that

$$O[i]^2 = \sum_{j=1}^{m} |t_{i+j-1} - p_j|^2 = \underbrace{\sum_{j=1}^{m} t_{i+j-1}^2}_{(1)} + \underbrace{\sum_{j=1}^{m} p_j^2}_{(2)} - \underbrace{2\sum_{j=1}^{m} t_{i+j-1} p_j}_{(3)} .$$

We compute (1), (2) and (3) separately, each of them for all $i = 1, 2, \ldots, n - m + 1$, and finish by summing them (for each i). It is a simple task to compute each of them separately.

Computing (1) for all i's is done by first calculating $t^2 = t_1^2, t_2^2, \ldots, t_n^2$ in linear time and then sliding a size m window over it, where for the first position it takes m steps, and for each following position the steps are computed from the predecessor position by adding the new number that is covered by the window and subtracting the number that was just uncovered.

Computing (2) takes m steps which is the same value for all i's.

Computing (3) is done by computing $-2t \otimes p$ and therefore can be computed in $O(n \log m)$ time.

This can "easily" support "wild card" symbols. "wild card" symbol is a symbol that matches exactly any corresponding symbol in the text. We will use the notation ϕ for a wildcard symbol. In order to support wildcard symbols in our algorithm the following steps are needed.

If we have wildcard symbols in the pattern: use $\sum_{j=1}^{m} t_{i+j-1}^2 \hat{p_j}$ for the values of (1), where $\hat{p_j} = 1$ if $p_j \neq \phi$ and $\hat{p_j} = 0$ for $p_j = \phi$. (It will take another convolution which costs $O(n \log m)$.) In case we have wildcard symbols in the text as well, a similar method is used to compute modified versions of (2) and (3).

5 String Matching with L_1 Distance

The problem of Exact L_1 Matching can be solved in $O(|\Sigma| n \log m)$ time using the algorithm described in Sect. 3. An $O(n\sqrt{m \log m})$ time algorithm can be designed using the method of dividing the alphabet into frequent and non-frequent symbols, as in [18]. Therefore, the current best time known for Exact L_1 Matching is $O(n\sqrt{\log m} \min(|\Sigma|\sqrt{\log m}, \sqrt{m}))$.

The problem of String Matching with Mismatches can be linearly reduced to the problem of Exact L_1 Matching [22]. This fact led us to search for an approximation algorithm rather than an exact one, which seems hard to design in a better time complexity.

We have designed an algorithm that approximates the results of L_1 Matching up to a factor of $1 + \varepsilon$ and runs in time of $O(\frac{1}{\varepsilon^2} n \log m \log |\Sigma|)$.

5.1 L_1 Approximation Algorithm

We begin by describing an exact algorithm that solves the L_1 Matching problem in $O(n|\Sigma|\log m \log|\Sigma|)$ time and showing its correctness. Then we modify it and insert the exactness parameter ε in order to reduce the time complexity. The total time complexity of our algorithm for Approximated L_1 matching is $O(\frac{1}{\varepsilon^2} n \log m \log|\Sigma|)$. We emphasize the fact that the exact algorithm is not efficient, and is used only for methodological reasons, to construct the approximation algorithm.

Key Idea The idea is to reduce the alphabet size. We identify two special cases that can be solved quickly: The first case arises when for each text location i, all aligned pairs t_{i+j-1}, p_j have the property that $|t_{i+j-1} - p_j| < c$, where c is some constant. This implies that instead of using the given large alphabet we can use a smaller alphabet. We replace each number t_i, p_j with $t_i \bmod 2c$ and $p_j \bmod 2c$, respectively. We use the shortest cyclic distances (since we have possible cases of $t_{i+j-1} = 2c-1, p_j = 2c+1$ but it is impossible that $t_{i+j-1} = 2c-1, p_j < (c-1)$). This creates an alphabet of size $2c$. The second case occurs when for each i, j we have the property that $|t_{i+j-1} - p_j| > 100c$, and we allow an error of $\pm 1\%$. In such cases we can reduce the alphabet by dividing all t_i, p_j by c, and multiplying the results by c. this achieves an alphabet which is c times smaller. To solve the general problem, we divide all possible distances into bounded ranges. On each range we use the "Weighted mismatch" technique, since it will contain only a small alphabet. We use the factors of $4/\epsilon$ and $\epsilon/2$ for the upper and lower bounds respectively.

We have designed an algorithm that approximates the results of L_1 Matching up to a factor of $1+\varepsilon$ which runs in time $O(\frac{1}{\varepsilon^2} n \log m \log|\Sigma|)$.

5.2 L_1 Approximation Algorithm

We begin by describing an exact algorithm that solves the L_1 Matching problem in $O(n|\Sigma|\log m \log|\Sigma|)$ and showing its correctness. Then we modify it and insert the exactness parameter ε in order to reduce the time complexity. The total run time is $O(\frac{1}{\varepsilon^2} n \log m \log|\Sigma|)$. We emphasize the fact that the exact algorithm is not efficient, and is used only for methodological reasons to construct the approximation algorithm.

We use a specific alphabet distance function to construct our algorithm. Given some real number k, define *$k - 2k$ String Matching* to be String Matching with Weighted Mismatches with the specific weight function: $f : \Sigma \times \Sigma \to \Re$ defined as

$$f(\sigma_1, \sigma_2) = \begin{cases} 0 & |\sigma_1 - \sigma_2| < k \\ \lceil |\sigma_1 - \sigma_2| - k \rceil & k \le |\sigma_1 - \sigma_2| \le 2k \\ \lceil k \rceil & 2k < |\sigma_1 - \sigma_2| \end{cases}$$

for $\sigma_1 \neq \phi$ and $\sigma_2 \neq \phi$, and $f(\sigma_1, \sigma_2) = 0$ otherwise. Note that this problem can be solved in time $O(|\Sigma| n \log m)$, because it is a specific case of the String Matching with Weighted Mismatches.

Exact L_1 Matching We provide an algorithm to solve L_1 Matching that uses $k-2k$ Matching as a given subroutine.

Algorithm Steps:

```
1. Initialize array $O[1,\dots,n-m+1]$ to zeros.
2. $k := \frac{1}{2}$
3. While $k < |\Sigma|$ do:
   (a) Run $k-2k$ Matching with $t$, $p$ and $k$ obtaining result vector $V[1,\dots,n-m+1]$
   (b) For $i = 1,\dots,n-m+1$ do $O[i] \leftarrow O[i] + V[i]$
   (c) $k = k * 2$
```

Algorithm Correctness By the definition of L_1 Matching, it suffices to show that for every i, $O[i] = \sum_{j=1}^{m} |t_{i+j-1} - p_j|$. It is sufficient to prove that for every i, j we add $|t_{i+j-1} - p_j|$ to $O[i]$. Fix i, j and let $d = |t_{i+j-1} - p_j|$. In every iteration of the while loop the $k-2k$ Matching checks whether $d < k, k \le d \le 2k$ or $2k < d$, and adds $0, d-k$ or k respectively. In the first iteration 1 is added to the running sum if $d \neq 0$, in the ith iterations, $i \ge 2$, 2^{i-2} is added for $d \le 2^{i-1}$; in the jth iteration, where $2^{j-1} \le d < 2^j$, $d - 2^j$ $(= d-k)$ is added. In the subsequent iterations $d < k$ and therefore nothing further is added to $O[i]$. If $d > 0$, the total sum equals $1 +\ 1 + 2 + \cdots + 2^x + d - 2^{x+1} = d$. If $d = 0$ none of the iterations add anything to $O[i]$ since $d < k$ for all k.

Running Time There are $\log|\Sigma|$ iterations of the while loop, each performing one time $k-2k$ Matching and $O(n)$ addition time overall. Let $f(n,m)$ be the time to compute the $k-2k$ Matching. Then the time complexity of the L_1 algorithm is $O((f(n,m)+n)\log|\Sigma|)$. The best currently known time for $k-2k$ Matching is $O(|\Sigma| n \log m)$, therefore the time is $O(|\Sigma| n \log m \log|\Sigma|)$.

5.2.1 First Modification of the Exact Algorithm

Consider the exact algorithm described above. Note that in fact we add every $d = |t_{i+j-1} - p_j|$ to $O[i]$ by iterations that check whether $d > \frac{1}{2}$, $d > 1$, $d > 2$, $d > 4, \dots, d > 2^x, \dots$. Therefore d is computed as the sum $1 + 1 + 2 + 4 + \cdots + 2^x + (d - 2^{x+1}) = d$. The idea behind the following modification is that for d large enough, if we drop the first elements of this sum, we do not err greatly. Now suppose that we are given some $0 < \varepsilon < 1$. Our first modification is to reduce the alphabet size as follows. Each time we run $k-2k$ Matching we run it with t', p' instead of t, p where t' and p' are defined by:

$$t'_i \leftarrow t_i \bmod \frac{4k}{\varepsilon} \quad 1 \le i \le n$$

$$p'_j \leftarrow p_j \bmod \frac{4k}{\varepsilon} \quad 1 \le j \le m$$

The error introduced by this modification arises only when d is very large compared to k. In such cases, in order to err in this iteration of $k-2k$ Matching it must be that $d \bmod \frac{4k}{\varepsilon} < 2k$. In more detail, consider $d = |t_{i+j-1} - p_j|$; if $d > \frac{4k}{\varepsilon}$ then perhaps $d \bmod \frac{4k}{\varepsilon} < 2k$ and then less than k is added to $V[i]$ at this iteration of $k - 2k$ matching. The iteration can err only by adding less than is needed. This is acceptable, because if the error in this iteration is at most k and $d > \frac{4k}{\varepsilon}$, then the error is less than $\varepsilon/4$. Summing all the iterations that can err, the errors are at most $1, 1, 2, \dots, \frac{\varepsilon}{4}d$ which sum to at most $\frac{\varepsilon}{2}d$. The time complexity of this modified version of the algorithm is derived from the alphabet size in each of the iterations of the $k - 2k$ procedure, which leads to: $\sum_{k=\frac{1}{2},1,2,4,\dots,\frac{|\Sigma|}{2}} \frac{4k}{\varepsilon} n \log m = O(\frac{1}{\varepsilon}|\Sigma| n \log m)$. This time complexity is not yet satisfactory, which leads us to the next modification.

5.2.2 Second Modification of the Exact Algorithm

Let us keep in mind that every $d = |t_{i+j-1} - p_j|$ is computed from the sum of $1 + 1 + 2 + 4 + \cdots + 2^x + (d - 2^{x+1})$ and we already allowed error among the small terms. We reduce the alphabet even more, in a way that causes only a small error in the biggest element in the sum, $d - 2^{x+1}$, where $2^{x+1} < d < 2^{x+2}$. The second modification is applied to each stage of $k - 2k$ Matching with t', p' having alphabet size of $\frac{4k}{\varepsilon}$ as follows.

Each time we run $k - 2k$ Matching we run it with t'', p'', k'' instead of t', p', k where t'', p'' and k'' are defined by:

$$t''_i \leftarrow \left\lfloor \frac{t'_i}{\frac{k\varepsilon}{2}} \right\rfloor \quad 1 \le i \le n$$

$$p''_j \leftarrow \left\lfloor \frac{p'_j}{\frac{k\varepsilon}{2}} \right\rfloor \quad 1 \le j \le m$$

$$k'' \leftarrow \frac{2}{\varepsilon}$$

And we change step 3(b) in the exact algorithm to: For $i = 1, \dots, n - m + 1$ do $O[i] \leftarrow O[i] + V[i]\frac{k\varepsilon}{2}$.

Consider some $d = |t_{i+j-1} - p_j|$, and let $d_k = |\lfloor \frac{t_{i+j-1}}{\frac{k\varepsilon}{2}} \rfloor - \lfloor \frac{p_j}{\frac{k\varepsilon}{2}} \rfloor|$. This modification would not change any of the terms in the sequence summing to d apart from the largest one. The modified sequence can be partitioned into three groups based on the value of k: $2k < d$, $k \le d \le 2k$ and $d < k$.

In the iterations with $2k < d$, it is easily seen that $d_k > \frac{2}{\varepsilon}$ and therefore $\frac{2}{\varepsilon} \times \frac{k\varepsilon}{2} = k$ is added to $O[i]$, as in the corresponding iteration with text t' and pattern p'. In the iterations with $k > d$, $d_k < \frac{2}{\varepsilon}$ and thus nothing is added to $O[i]$. The only element that is affected by the modification satisfies $k \le d \le 2k$, i.e. the element $d - 2^{x+1}$. If $k \le d \le 2k$ then $\frac{2}{\varepsilon} \le d_k \le \frac{4}{\varepsilon}$. Therefore, $(d_k - \frac{2}{\varepsilon})\frac{k\varepsilon}{2} = d_k\frac{k\varepsilon}{2} - k$ is added to $O[i]$. We use the following inequality to ensure that this error is bounded by $\frac{k\varepsilon}{2}$:

Lemma 1 *For any integers x, y s.t. $k \le |x - y| \le 2k$:*

$$|x - y|(1 - \varepsilon) \le \left| \left\lfloor \frac{x}{k\varepsilon} \right\rfloor - \left\lfloor \frac{y}{k\varepsilon} \right\rfloor \right| k\varepsilon \le |x - y|(1 + \varepsilon).$$

Proof of Lemma 1 Let $x = c_x k\epsilon + d_x$, where $0 \le d_x < k\epsilon$, and $y = c_y k\epsilon + d_y$, where $0 \le d_y < k\epsilon$. Then $|\lfloor \frac{x}{k\epsilon} \rfloor - \lfloor \frac{y}{k\epsilon} \rfloor| k\epsilon = |c_x - c_y| k\epsilon$ and $|x - y| = |(c_x - c_y)k\epsilon + (d_x - d_y)|$ and the inequality follows.

The total error from both modifications is bounded by $k\varepsilon$ for $k \le d \le 2k$, and therefore is within the tolerated distance of up to a factor of ε. Note that $L_1[i]$ is defined as $\sum_{j=1}^{m} |t_{i+j-1} - p_j|$, so if we err in each pair (t_{i+j-1}, p_j) up to a factor of $1 + \varepsilon$ the total error is also bounded by this factor.

The fact that in each iteration we reduce our alphabet first by $\mathrm{mod}\,\frac{4k}{\varepsilon}$ and then by $\mathrm{div}\,\frac{k\varepsilon}{2}$ leads to an alphabet size of $O(\frac{1}{\varepsilon^2})$. Then the time needed to run the $k - 2k$ procedure is $O(\frac{1}{\varepsilon^2} n \log m)$. The total time taken by this algorithm is $O(\frac{1}{\varepsilon^2} n \log m \log |\Sigma|)$. □

Theorem 1 *Approximate String Matching with L_1 Distance can be done in time $O(\frac{1}{\varepsilon^2} n \log m \log |\Sigma|)$.*

6 String Matching with L_∞ Distance

This problem differs from the problems of L_1 and L_2 Matching since the distance function defined between strings is not the sum of the distances between aligned symbols, but rather is defined as the max distance between aligned symbols. We first construct an $O(n|\Sigma| \log m)$ time algorithm, and then construct an algorithm that approximates L_∞ matching up to a factor of $1 + \varepsilon$ and runs in time of $O(\frac{1}{\varepsilon} n \log m \log |\Sigma|)$.

6.1 $O(n|\Sigma| \log(m + |\Sigma|))$ Algorithm

The method in this algorithm is to encode the text and the pattern in such a way that we find the results using one convolution, followed by a linear time pass on the convolution result.

Key Idea For each offset, for each text-pattern alignment, determine if at least one pair of characters is exactly that far apart. To do this, both the pattern and text are encoded by using length $|\Sigma|$ binary strings to replace each character. The encoding of symbol σ, a number, is all 0's except the σ-th bit which is 1, Let $c(\sigma)$ denote the encoded σ. Now, we begin by aligning $c(p)$ below $c(t)$ and starting at position $-|\Sigma|$ (where $c(t)$ is fixed to start at position 1). We move $c(p)$ to the right till both 1-bits are one below the other. At this position, the distance between the starting position of $c(t)$ and the starting position of $c(p)$ equals the difference $|t - p|$. An example depicting this process is given in Fig. 1. If we look at $r = c(t) \otimes c(p)$ we see either $r[-|t - p|] = 1$ or $r[|t - p|] = 1$. Extending this idea to encode strings of numbers requires us to add leading (or tracing) zeros between the encoded numbers.

In detail: first, define $\chi_{\neq 0}(x) = 1$ if $x \neq 0$ and otherwise 0. Next, define for every $x \in \Sigma = \{1, \dots, n\}$, $c^t(x) = c^t(x)_1, \dots, c^t(x)_{2|\Sigma|}$ where $c^t(x)_i = 1$ if $i = |\Sigma| + x$ and otherwise 0. Similarly define $c^p(x) = c^p(x)_1, \dots, c^p(x)_{2|\Sigma|}$ where $c^p(x)_i = 1$ if $i = x$ and otherwise 0.

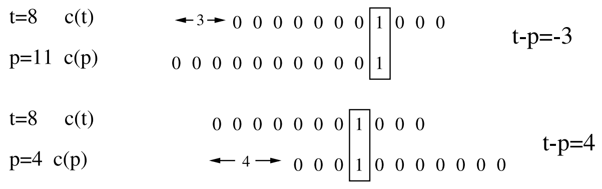
Fig. 1 $c(p)$ moved below $c(t)$ till the 1-bits are aligned

Algorithm Steps

1. Construct $c^t(T) = c^t(t_1) \cdots c^t(t_n)$
2. Construct $c^p(P) = c^p(p_1) \cdots c^p(p_m)$
3. Compute $R = c^t(T) \otimes c^p(P)$
4. For $i = 1, \ldots, n - m + 1$

$$O[i] \leftarrow \max_{s=-|\Sigma|}^{|\Sigma|} \chi_{\neq 0}(R[(2i-1)|\Sigma| + 1 + s])|s|$$

Claim At the end of the algorithm $O[i] = \max_{j=1}^{m} |t_{i+j-1} - p_j|$.

Proof First, we show that $O[i] \geq |t_{i+j-1} - p_j|$ for every $j \in \{1, \ldots, m\}$. In order to depict this, it is enough to show that for every $j \in \{1, \ldots, m\}$ that the following holds: $R[(2i-1)|\Sigma| + 1 + t_{i+j-1} - p_j] \neq 0$ (since, then, the value of $|t_{i+j-1} - p_j|$ is one of the values for the *max* taken in step 4). Now, since $R[(2i-1)|\Sigma| + 1 + t_{i+j-1} - p_j] = \sum_{k=1}^{2m|\Sigma|} c^t(T)_{(2i-1)|\Sigma|+1+t_{i+j-1}-p_j+k-1} c^p(P)_k$ and for $k = p_j$ we have $c^t(T)_{(2i-1)|\Sigma|+t_{i+j-1}} = 1$ and $c^p(P)_{p_j} = 1$ (from the way in which we defined the encoding) $R[(2i-1)|\Sigma| + 1 + t_{i+j-1} - p_j] \neq 0$.

Now, we must only show that $O[i] = |t_{i+j-1} - p_j|$ for some $j \in \{1, \ldots, m\}$. Let s_m be the value for which $O[i] = \chi_{\neq 0}(R[(2i-1)|\Sigma| + 1 + s_m])|s_m|$. We can assume that $R[(2i-1)|\Sigma| + 1 + s_m] \neq 0$ (otherwise $O[i] = 0$ and $0 \leq \max_{j=1}^{m} |t_{i+j-1} - p_j|$ and our proof is completed). The fact that $R[(2i-1)|\Sigma| + 1 + s_m] \neq 0$ implies that for some $j' \in \{1, \ldots, m\}, k' \in \{0, \ldots, 2|\Sigma| - 1\}$ we have $c^t(T)_{(2i-1)|\Sigma|+1+s_m+2j'|\Sigma|+k'-1} = c^t(T)_{(2(i+j')-1)|\Sigma|+s_m+k'} = 1$ and $c^p(P)_{2j'|\Sigma|+k'} = 1$, which in turn implies that for j', $|t_{i+j'-1} - p_{j'}| = s_m$ holds. This completes our proof. □

Time The time needed to convolve 2 strings of size $n|\Sigma|$ and $m|\Sigma|$ is $O(n|\Sigma| \log n)$. The computation of $O[i]$ takes $2|\Sigma|$ steps, thus this step takes $O(n|\Sigma|)$ time. Both steps together take $O(n|\Sigma| \log n)$ time. We can slightly improve the time to a total time of $O(n|\Sigma| \log(m + |\Sigma|))$ by using the technique of cutting the text into n/m overlapping segments, each having length of $2m$.

6.2 L_∞ Approximation Algorithm

Below we present an algorithm that approximates the L_∞ Matching up to a factor of $1 + \varepsilon$.

In each iteration q, the algorithm approximates the L_∞ distance for all text locations with $2^q < L_\infty < 2^{q+1}$.

Algorithm Steps:

```
1. Initialize array $O[1,\dots,n-m+1]$ to zeros.
2. For $q = 1, 2, 4, \dots |\Sigma|$ do:
    (a) Create $t'$ from $t$, where $t'_i = \lfloor \frac{t_i}{q\varepsilon} \rfloor \bmod \frac{4}{\varepsilon}$.
    (b) Create $p'$ from $p$, where $p'_i = \lfloor \frac{p_i}{q\varepsilon} \rfloor \bmod \frac{4}{\varepsilon}$.
    (c) Run the $L_\infty$ exact algorithm with $t'$ and $p'$ to obtain a result vector
        $V[1,\dots,n-m+1]$.
    (d) For $i = 1,\dots,n-m+1$ if $V[i] \ge \frac{1}{\varepsilon}$ then $O[i] \leftarrow V[i]q\varepsilon$.
```

Algorithm Correctness For each $O[i]$, the maximal q where $V[i] > \frac{1}{\varepsilon}$ mean that $L_\infty(p, t[i,\dots,i+m-1]) > q$, but also that $L_\infty(p, t[i,\dots,i+m-1]) < 2q$. Now since we reduce our alphabet by divqε we receive the value computed with an error of $\pm q\varepsilon$, which is acceptable since the exact value is between q and $2q$.

Time Each iteration takes $O(\frac{1}{\epsilon} n \log n)$ time, since we reduced the alphabet to have size $\frac{4}{\epsilon}$. As there are $\log \Sigma$ iterations, this take $O(\frac{1}{\epsilon} n \log n \log |\Sigma|)$ time overall. Applying the technique of cutting the text into n/m overlapping segments, reduces the running time to $O(\frac{1}{\epsilon} n \log m \log |\Sigma|)$.

7 Open Problems and Conclusions

We have presented an $O(\frac{1}{\epsilon^2} n \log m \log |\Sigma|)$ time $1+\epsilon$ approximation algorithm for the L_1 matching problem. L_2 matching is related to convolution computation, therefore any improvement in the time for one of them, directly means improvement for the other as well. The current best time for both L_2 matching and convolution is $O(n \log m)$. The most interesting case is L_∞ matching. In this case we have an $\tilde{O}(n|\Sigma|)$ algorithm, which is a run time worse than that yielded by the naive algorithm for large alphabets. The problem of solving the *exact* L_∞ matching within a better-than-naive time for large alphabets still remains open. We have presented a $1+\epsilon$ approximation algorithm for this problem taking time $O(\frac{1}{\epsilon} n \log m \log |\Sigma|)$.

For the *String Matching with Weighted Mismatches* problem we have provided an $O(n|\Sigma| \log m)$-time exact algorithm. Nonetheless, whether the *exact* solution can be improved or a better approximation ratio can be shown with the $\tilde{O}(n)$-time algorithm remains an open question.

References

1. Abrahamson, K.: Generalized string matching. SIAM J. Comput. **16**(6), 1039–1051 (1987)
2. Amir, A., Farach, M.: Efficient 2-dimensional approximate matching of half-rectangular figures. Inf. Comput. **118**(1), 1–11 (1995)

3. Amir, A., Farach, M., Muthukrishnan, S.: Alphabet dependence in parameterized matching. Inf. Process. Lett. **49**, 111–115 (1994)
4. Amir, A., Lewenstein, M., Porat, E.: Faster algorithms for string matching with k mismatches. J. Algorithms **50**(2), 257–275 (2004). Special SODA 2000 issue
5. Atallah, M.J., Chyzak, F., Dumas, P.: A randomized algorithm for approximate string matching. Algorithmica **29**(3), 468–486 (2001)
6. Baker, B.S.: A theory of parameterized pattern matching: algorithms and applications. In: Proc. 25th Annual ACM Symposium on the Theory of Computing, pp. 71–80 (1993)
7. Chu, K.K.W., Wong, M.H.: Fast time-series searching with scaling and shifting. In: Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31–June 2, 1999, Philadelphia, Pennsylvania, pp. 237–248. ACM, New York (1999)
8. Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching. In: Proc. of the 34th Annual ACM Symposium on Theory of Computing, pp. 592–601. ACM, New York (2002)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1992)
10. Cormode, G., Muthukrishnan, S.: The string edit distance matching problem with moves. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 667–676. ACM, New York (2002)
11. Faloutsos, C., Ranganathan, M., Manolopoulos, Y.: Fast subsequence matching in time-series databases. In: Proceedings 1994 ACM SIGMOD Conference, Mineapolis, MN, pp. 419–429 (1994)
12. Fischer, M.J., Paterson, M.S.: String matching and other products. In: Karp, R.M. (ed.) Complexity of Computation. SIAM-AMS Proceedings, vol.7, pp. 113–125 (1974)
13. Hirschberg, D.S.: Serial computations of Levenshtein distances. In: Apostolico, A., Galil, Z. (eds.) Pattern Matching Algorithms, pp. 123–141. Oxford University Press, London (1997)
14. Indyk, P.: Faster algorithms for string matching problems: matching the convolution bound. In: Proceedings of the 39th IEEE Annual Symposium on Foundations of Computer Science, pp. 166–173. IEEE Comput. Soc., Los Alamitos (1998)
15. Indyk, P.: Stable distributions, pseudorandom generators, embeddings and data stream computation. In: Proc. 41st Symp. Foundations of Computer Science, pp. 189–197. IEEE Press, New York (2000)
16. Indyk, P.: Algorithmic applications of low-distortion geometric embeddings. In: Proceedings of the 42th IEEE Annual Symposium on Foundations of Computer Science. IEEE Comput. Soc., Los Alamitos (2001)
17. Indyk, P., Koudas, N., Muthukrishnan, S.: Identifying representative trends in massive time series data sets using sketches. In: El Abbadi, A., Brodie, M.L., Chakravarthy, S., Dayal, U., Kamel, N., Schlageter, G., Whang, K.-Y. (eds.) VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10–14, 2000, Cairo, Egypt, pp. 363–372. Morgan Kaufmann, San Mateo (2000)
18. Indyk, P., Lewenstein, M., Lipsky, O., Porat, E.: Closest pair problems in very high dimensions. In: Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP). Lecture Notes in Computer Science, vol. 3142, pp. 782–792. Springer, Berlin (2004)
19. Karloff, H.: Fast algorithms for approximately counting mismatches. Inf. Process. Lett. **48**(2), 53–60 (1993)
20. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**, 323–350 (1977)
21. Levenshtein, V.I.: Binary codes capable of correcting, deletions, insertions and reversals. Sov. Phys. Dokl. **10**, 707–710 (1966)
22. Lipsky, O., Porat, E.: L_1 pattern matching lower bound. In: Proceedings of the 12th International Conference on String Processing and Information Retrieval (SPIRE), Bs. As., Argentina. Lecture Notes in Computer Science, vol. 3772, pp. 327–330. Springer, Berlin (2005)
23. Masek, W.J., Paterson, M.S.: A faster algorithm for computing string-edit distances. J. Comput. Syst. Sci. **20**, 18–31 (1980)
24. Muthukrishnan, S.: New results and open problems related to non-standard stringology. In: Proc. 6th Combinatorial Pattern Matching Conference. Lecture Notes in Computer Science, vol. 937, pp. 298–317. Springer, Berlin (1995)
25. Muthukrishnan, S., Ramesh, H.: String matching under a general matching relation. Inf. Comput. **122**(1), 140–148 (1995)
26. Perttu, S.: Combinatorial pattern matching in musical sequences. Master's thesis, Department of Computer Science, University of Helsinki (2000)