ALPHABET INDEPENDENT AND DICTIONARY SCALED MATCHING *

Amihood Amir† Gruia Călinescu‡

Georgia Tech and Bar-Ilan University — Georgia Tech

Abstract

The rapidly growing need for analysis of digitized images in multimedia systems has lead to a variety of interesting problems in multidimensional pattern matching. One of the problems is that of *scaled matching*, finding all appearances of a pattern in a text in all sizes. Another important problem is *dictionary matching*, quick search through a dictionary of preprocessed patterns in order to find all dictionary patterns that appear in the input text.

In this paper we provide a simple algorithm for two dimensional scaled matching. Our algorithm is the first linear-time alphabet-independent scaled matching algorithm. Its running time is $O(|T|)$, where $|T|$ is the text size, and is independent of $|\Sigma|$, the size of the alphabet. The main idea behind our algorithm is identifying and exploiting a *scaling-invariant* property of patterns.

Our technique generalizes to produce the first known algorithm for scaled dictionary matching. We can find all appearances of all dictionary patterns that appear in the input text in any discrete scale. The time bounds of our algorithm are equal to the best known exact (no scaling) two dimensional dictionary matching algorithms.

*A preliminary version of this paper appeared in Combinatorial Pattern Matching 1996

†Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, ISRAEL; (972-3)531-8770; amir@cs.biu.ac.il; Partially supported by NSF grant CCR-96-101709, the Israel Ministry of Science grant number 6297, and a 1998 Bar-Ilan University Internal Research Grant.

‡College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280; (404) 894-9389; gruia@cc.gatech.edu.

1 Introduction

The *Scaled Matching* problem is that of finding all occurrences of an input pattern P in input text T in all possible scales. The problem is both practically and theoretically significant.

Practically, we are witnessing a tremendous thrust of various industries into multimedia and digitized analysis. This makes it imperative to derive a multidimensional equivalent of it "text string searching" in an image database [23]. Scaling is one of several important issues in template matching. The others are *local errors* and *rotation*. However, the scaling problem is important even devoid of rotation as, for example, in the problem of reading a text with different sized letters (e.g. a newspaper) [15].

In reality, the scaling problem is intimately connected with approximation. Therefore, to be truly practical, such an algorithm should be able to cope with non-discrete scales as well. However, there is no known fast algorithm even for *scaled string matching* with general scales. The problem of scaled matching with *natural* scales, albeit a less general model, has been discussed in the literature, and proved to have theoretical importance. The scaled matching problem was studied for the first time by Amir, Landau and Vishkin [11]. They showed that all appearances in text T of a pattern P *scaled* to any discrete size, can be found in linear time for fixed finite alphabets. That solution gave rise to the compressed matching problem [2, 7] which, in turn led to the development of two-dimensional periodicity [3]. Two dimensional periodicity turned out to be the most important tool in two dimensional matching. Its development led to an alphabet independent two dimensional matching algorithm [6, 4, 18] and to optimal parallel two dimensional matching algorithms [5, 13].

Much progress has been made with *dictionary matching* as well. In the traditional pattern matching model a single pattern is sought in a single text. Dictionary matching allows preprocessing of a (possibly vast) dictionary of patterns. Subsequently, appearances of dictionary patterns in various input texts are to be found quickly. Various algorithms for two dimensional matching (static, dynamic, square patterns, rectangular patterns) were developed [8, 9, 19, 21, 24]. The two remaining fundamental problems in dictionary matching are 1) Scaled dictionary matching and 2) Approximate dictionary matching.

This paper addresses the scaled two dimensional dictionary matching problem. The techniques of [11] for scaled matching can not be generalized to dictionary matching. The main idea was analyzing the text with the aid of *power columns*. Those are the text columns appearing $m-1$ columns apart, where P is an $m \times m$ pattern. This dependence on the pattern size make the power columns useless where a dictionary of different sized patterns is involved.

In this paper we present a novel idea for solving the scaled dictionary matching problem for a single pattern. This idea is significantly *simpler* than the algorithm of [11] and has an additional advantage of being *alphabet-independent*. The previous scaled matching algorithm had a worst case running time of $O(|T| \log |\Sigma|)$, where $|T|$ and $|\Sigma|$ are the respective sizes of the text and the alphabet. Our algorithm runs in time $O(|T|)$ for every alphabet. We achieve this time by developing a *scaling-invariant "signature"* of the pattern.

We also show that this idea can be further developed to dictionary matching. We present the first algorithm for *scaled two dimensional dictionary matching*. Our algorithm solves the scaled dictionary matching problem for a static dictionary of square pattern matrices in time similar to that of the non scaled algorithm [8]. Our times are: $O(|D|(\log k + \log \sigma_D))$ preprocessing, where $|D|$ is the total dictionary size, k is the number of patterns in the dictionary, and σ_D is the number of symbols in the dictionary, and $O(|T|(\log k + \log \sigma_D))$ text scanning time, for input text T. This is identical to the time at [8], the best non-scaled matching algorithm for a static dictionary of square patterns. It is somewhat surprising that scaling does not add to the complexity of single matching nor dictionary matching.

This paper is organized as follows. In section 2, we give the problem definition and some preliminaries. In section 3, we give the alphabet independent single pattern two dimensional scaled matching algorithm. In section 4 we give the scaled dictionary matching algorithm. We conclude with open problems and future research. For the sake of completeness, we added an appendix that describes the previously known witness algorithm for (non-scaled) two dimensional pattern matching.

2 Problem Definition

Before we define the scaled matching problem, we introduced the following notation. The string $aa...a$ where the symbol a is repeated k times (to be denoted a^k), is referred to as *scaling of the singleton string a by multiplicative factor k*, or simply as *a scaled to k*. Similarly, consider a string $A = a_1 \cdots a_l$. *A scaled to k* (A^k) is the string $a_1^k, ..., a_l^k$.

The *scaled string matching problem* is defined as follows:

> *Input:* Pattern $P = p_1 \cdots p_m$ and text $T = t_1 \cdots t_n$ where $n > m$.
>
> *Output:* All positions in T where an occurrence of P scaled to k starts, for any $k = 1, ..., \lfloor \frac{n}{m} \rfloor$.
>
> A simple and elegant linear time algorithm for the *scaled string matching problem* can be adapted from Eilam-Tsoreff and Vishkin [16]. One can easily check that the running time of that algorithm is independent of the alphabet size.

Let $P[m \times m]$ be a two-dimensional matrix over alphabet Σ (not necessarily bounded). Then *P scaled to s* (P^s) is the $sm \times sm$ matrix where every symbol $P[i,j]$ of P is replaced by a $s \times s$ matrix whose elements all equal the symbol in $P[i,j]$. More precisely,

$$P^s[i,j] = P[\lceil \frac{i}{s} \rceil, \lceil \frac{j}{s} \rceil]$$

We say that pattern $P[m \times m]$ *occurs* (or an occurrence of P starts) at location (k,l) of the text T if for any $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, m\}$, $T[k+i-1, l+j-1] = P[i,j]$.

Following [11] we define the problem of *two-dimensional pattern matching with scaling* as follows:

> *Input:* Pattern matrix $P[i,j]$ $\quad i = 1, ...m;\ j = 1, ..., m$ and Text matrix $T[i,j]$ $\quad i = 1, ..., n;\ j = 1, ..., n$ where $n > m$.

Output: all locations in T where an occurrence of P scaled to s (an *s-occurrence*) starts, for any $s = 1, ..., \lfloor \frac{n}{m} \rfloor$.

We defined the scaled matching problem on square texts and patterns for the sake of simplicity only. Our alphabet independent result is also true for rectangular texts and patterns.

In [11], this problem was solved in time $O(n^2)$. However, the alphabet was finite. Over an infinite alphabet, the Amir-Landau-Vishkin algorithm will achieve time $O(n^2 \log m)$. Added deficiencies of that algorithm are that it is extremely complicated, and that the square root of the pattern size (m) plays a crucial role in the text processing. Our algorithm is simple and straightforward, and its time complexity is $O(n^2)$ for every alphabet.

Traditional Pattern Matching has dealt with the problem of finding all occurrences of a single pattern in a text (under some definition of the word "occurrence"). While the case of a pattern/text pair is of fundamental importance, the single pattern model is not always appropriate. One would often like to find all occurrences of a *set* of patterns in a text. We call such a set of patterns a *dictionary.* In addition to its theoretical importance, dictionary matching has many practical applications. For example, in computer vision, one is often interested in matching a template to a picture. In practice, one needs to match an enormous set of templates against each picture. Clearly one would like an algorithm which is minimally dependent on the size of the database of templates.

The *static two dimensional dictionary matching problem* is defined as follows. Given a set D (the *dictionary*) of patterns $P_1[m_1 \times m_1], P_2[m_2 \times m_2], \ldots, P_k[m_k \times m_k]$ of total length $d = \sum_{i=1}^{k} m_i^2$ all over alphabet Σ. Preprocess the set in a way that enables solution to the following problem:

Input: Text $T[n \times n]$ over alphabet Σ.

Output: All ordered pairs $([x, y], j)$ such that pattern P_j matches the subsquare of text beginning at location $T[x, y]$.

Amir and Farach [8] provided an algorithm for the static two dimensional dictionary matching problem where the dictionary preprocessing was done in time $O(d \log k)$ and subsequent text scans took time $O(n^2 \log k)$. In [9], the dynamic two dimensional dictionary matching problem was considered. In the dynamic version, patterns may be inserted to and deleted from the dictionary. An algorithm was provided where a dictionary update takes time $O(|P| \log d)$ (P is the pattern inserted or deleted) and the text scanning takes time $O(n^2 \log d)$. Giancarlo [19] also obtained an algorithm for the dynamic two dimensional dictionary matching problem, but his algorithm is less efficient. In [10] simple randomized algorithm for this problem were presented.

All above algorithms assume that the patterns are squares (although the text need not be). Idury and Schäffer [21] showed an algorithm for dynamic two dimensional dictionary matching of general rectangular patterns.

In this paper we show that our new scaled matching algorithm can be adapted to solve the static two dimensional dictionary matching problem for square patterns in the same time complexity as that of the exact matching case.

3 Alphabet Independent Scaled Matching

3.1 Algorithm's Idea and Data Structures

Our algorithm is based on the alphabet independent exact two dimensional matching algorithm of Amir, Benson and Farach [6]. That algorithm has two stages, the *candidate consistency* stage and the *candidate verification* stage.

In the candidate consistency stage, we check for every pair of text location if both locations can be the start of a pattern appearance. We say the two locations are consistent if both can be the start of a pattern appearance. A preprocessed witness table allows constant time elimination of one of the two candidate sources, if they do not agree in every location of the overlap.

In [4, 18] there are two different algorithms that construct the witness table in time $O(m^2)$. The candidate consistency stage takes time $O(n^2)$ [6].

In the candidate verification stage, a *wave* is employed to verify which of the non-conflicting sources are indeed starts of pattern appearances. This stage is also done in time $O(n^2)$.

Main Idea of Algorithm:

For every scale $s = 1, \ldots, \lfloor \frac{n}{m} \rfloor$ we divide the text into a $\lfloor \frac{n}{s} \rfloor \times \lfloor \frac{n}{s} \rfloor$ grid of $s \times s$ squares. Call such a square an *s-block*. We do a constant c amount of work per s-block. The total amount of time is then

$$\sum_{s=1}^{\lfloor \frac{n}{m} \rfloor} c\frac{n^2}{s^2} = O(n^2).$$

It should be noted that the Amir-Landau-Vishkin scaled matching algorithm also tries to employ this strategy. The problem is that the division into s-blocks is very complicated there. We simply divide into the natural grid, and adjust the exact location at the end.

Our algorithm assumes the pattern is *non-trivial*. A *trivial pattern* is a pattern where all rows are equal or all columns are equal. Finding all occurrences of a trivial pattern is simply achieved. One can search for all scaled occurrences of the single repeating row by the one-dimensional alphabet-independent scaled matching algorithm presented in Amir, Landau and Vishkin. For every such location one needs to verify that the scaled row repeats sufficiently many times. The machinery which we will introduce allows such a verification in constant time.

It is clear that in a non-trivial pattern there exist at least two consecutive rows that are different and one of which has at least two different symbols. Notice that non-trivial patterns have $m \geq 2$.

First we introduce several definitions and indicate how to preprocess the text.

Definition 1 We say that position $[i, j]$ of pattern P *occurs s-scaled* at position $[p, q]$ in text T if P^s occurs at position $[p - s(i-1), q - s(j-1)]$.

Definition 2 Let $[i, j]$ be a position of $n \times n$ text T.

1. Let $B_r[i, j]$ be the largest integer l for which the two strings $T[i-1; j, j+1, ..., j+l-1]$ and $T[i; j, j+1, ..., j+l-1]$ are equal. In words, $B_r[i, j]$ gives the length of the longest common prefix of rows $i-1$ and i starting at column j.
2. Let $C_r[i, j]$ be the largest integer l for which $T[i, j] = T[i, j+1] \cdots = T[i, j+l-1]$. In words, $C_r[i, j]$ gives the number of times that the symbol in $T[i, j]$ appears consecutively starting at position $[i, j]$.

We similarly define and construct B_r^P and C_r^P for positions of the pattern P.

It is easy to construct B_r and C_r in time $O(n^2)$. We preprocess the columns of B_r and C_r for *range minimum queries.*

Let $L = [l_1, ..., l_n]$ be an array of n numbers. A *Range Minimum* query is of the form:

> Given a range of indices $[i, ..., j]$, where $1 \leq i \leq j \leq n$, return an index k $i \leq k \leq j$ such that $l_k = \min\{l_i, ..., l_j\}$.

In [17] it was shown that a list of length n can be preprocessed in time $O(n)$ such that subsequent range minimum queries can be answered in constant time.

We augment the witness table to answer the following question: for each position $[i, j]$ of the pattern, we want to know if the array $P[i \dots m, j \dots m]$ has one of the following properties:

1. All lines are equal, and if not provide a location $[p, q]$ such that $i \leq p \leq m-1$, $j \leq q \leq m$ and $P[p, q] \neq P[p+1, q]$.
2. All columns are equal, and if not provide a location $[p, q]$ such that $i \leq p \leq m$, $j \leq q \leq m-1$ and $P[p, q] \neq P[p, q+1]$.

This augmented table can be constructed in time $O(m^2)$ using appropriate minimum range queries on B_r^P and C_r^P.

Definition 3 Position $[i, j]$ of P is a *pivot* if the following conditions hold:

1. $i > 1$, $j > 1$.
2. The strings $P[i-1; j-1, \dots, m]$ and $P[i; j-1, \dots, m]$ are different.

3. One of the strings $P[i-1; j-1, \ldots, m]$ and $P[i; j-1, \ldots, m]$ contains two different symbols.

Clearly every non-trivial pattern has a pivot.

The main property exploited by our algorithm (stated in corollary 2) is the fact that an s-scaled occurrence of the pivot might start in at most one position of an s-block.

Choose a pivot $[i, j]$ of P, and call it *the* pivot of P. Consider an s-block starting at position $T[p, q]$ (i.e. $T[p, \ldots, p+s-1; q, \ldots, q+s-1]$). The routine below finds the unique position in the s-block where the pivot might occur s-scaled.

Routine A

Find l, $p \leq l \leq p+s-1$ such that $B_r[l, q-1]$ is the minimum. There are two cases:

1. $B_r[l, q-1] > 0$. Output $[l, q + ((B_r[l, q-1] - 1) \bmod s)]$
2. $B_r[l, q] = 0$. Compute $r = \min(C_r[l-1, q-1], C_r[l, q-1])$. Output $[l, q + ((r-1) \bmod s)]$.

Notice that the output of the routine is a position in the s-block starting at position $T[p, q]$.

Time: The above routine may be implemented using minimum range queries. The total running time is constant and independent on the pattern.

Proposition 1 *Routine A correctly outputs the (unique) position in the s-block where the pivot might occur s-scaled.*

Note that the proposition only claims that the pivot cannot occur s-scaled at any other position in the s-block; it does not promise that there is indeed an s-scaled pivot occurrence in the output position.

Proof. We follow the cases of the routine. The first case is $B_r[l, q-1] > 0$.

The first case has two subcases:

1. $B_r[l, q-1]$ is not relevant to the pattern, because it is too "far". Formally $B_r[l, q-1] > (m+1-j)s$. Recall that $[i, j]$ is the (fixed) pivot of the pattern. In this case the pivot cannot occur s-scaled at any position in the s-block. Suppose, to the contrary, that the pivot occurs s-scaled at position $[a, b]$, where $p \leq a < p+s-1$ and $q \leq b < q+s-1$. Since we know that $P[i-1; j-1, \ldots, m]$ and $P[i; j-1, \ldots, m]$ are different, there is a c, such that $j-1 \leq c \leq m$ and $P[i-1, c] \neq P[i, c]$. Then the s-scaled letter corresponding to $P[i-1, c]$ contains $T[a-1, q-1+(c+1-j)s]$ and the s-scaled letter corresponding to $P[i, c]$ contains $T[a, q-1+(c+1-j)s]$. Then $T[a-1, q-1+(c+1-j)s] \neq T[a, q-1+(c+1-j)s]$. It follows that $B_r[a, q-1] \leq (c+1-j)s \leq (m+1-j)s$, since $j-1 \leq c \leq m$, and by the choice of l we have $B_r[l, q-1] \leq B_r[a, q-1] \leq (m+1-j)s$.

   This contradicts the fact that $B_r[l, q-1] > (m+1-j)s$.

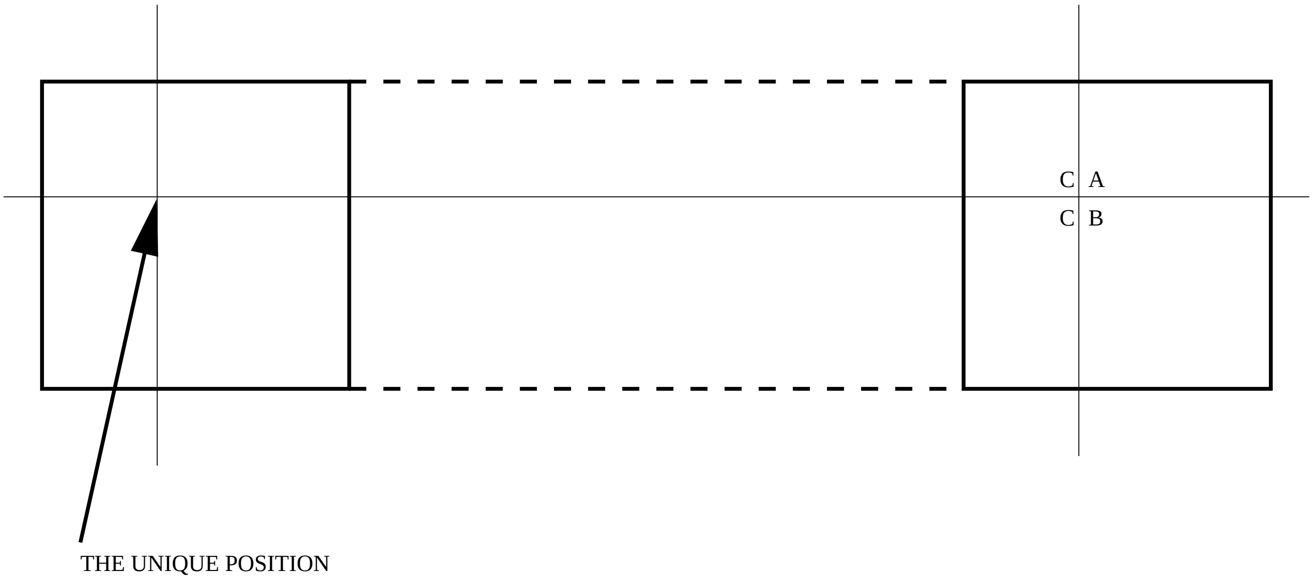

Figure 1: The first case, relevant subcase, of proposition 1. $A \neq B$, but the symbol C might be A or B.

2. $0 < B_r[l, q-1] \leq (m+1-j)s$. In this case it is relevant to the pattern.

   Suppose that the pivot occurs s-scaled at position $[a, b]$. We cannot have $a \neq l$, since otherwise a s-scaled letter of P will contain both $T[l-1, q-1+B_r[l, q-1]]$ and $T[l, q-1+B_r[l, q-1]]$. But by definition $T[l-1, q-1+B_r[l, q-1]] \neq T[l, q-1+B_r[l, q-1]]$. Also notice that since $T[l-1, q-1+B_r[l, q-1]-1] = T[l, q-1+B_r[l, q-1]-1]$, either $T[l-1, q-1+B_r[l, q-1]] \neq T[l-1, q-1+B_r[l, q-1]-1]$ or $T[l, q-1+B_r[l, q-1]] \neq T[l, q-1+B_r[l, q-1]-1]$. We cannot have $b \neq q + ((B_r[l, q-1]-1) \bmod s)$, since otherwise a s-scaled letter of P will contain both $T[l-1, q-1+B_r[l, q-1]]$ and $T[l-1, q-1+B_r[l, q-1]-1]$ and another s-scaled letter will contain both $T[l, q-1+B_r[l, q-1]]$ and $T[l, q-1+B_r[l, q-1]-1]$. See figure 1.

In the second case, from the fact that $T[l-1, q] \neq T[l, q]$ (and recalling that, for the pivot $[i, j]$, $i > 1$ and $j > 1$) we can deduce that the pivot can occur s-scaled only at line l. We have two subcases, just as before. See figure 2.

1. In the first subcase r is not relevant to the pattern. Formally $r > (m-j+1)s$. Informally, at both lines $l-1$ and l, on the right of column q, the first change in symbol appears too far away. Using the fact that one of the strings $P[i-1; j-1, \ldots, m]$ and $P[i; j-1, \ldots, m]$ contains two different symbols, as in the first subcase above, we can deduce that no pivot can occur s scaled in the s-block.

2. In the second subcase $r \leq (m-j+1)s$ is relevant to the pattern.

   Since either $T[l-1, q-1+r-1] \neq T[l-1, q-1+r]$ or $T[l, q-1+r-1] \neq T[l, q-1+r]$ and all these positions are inside the s-scaled occurrence of the pattern, it follows that

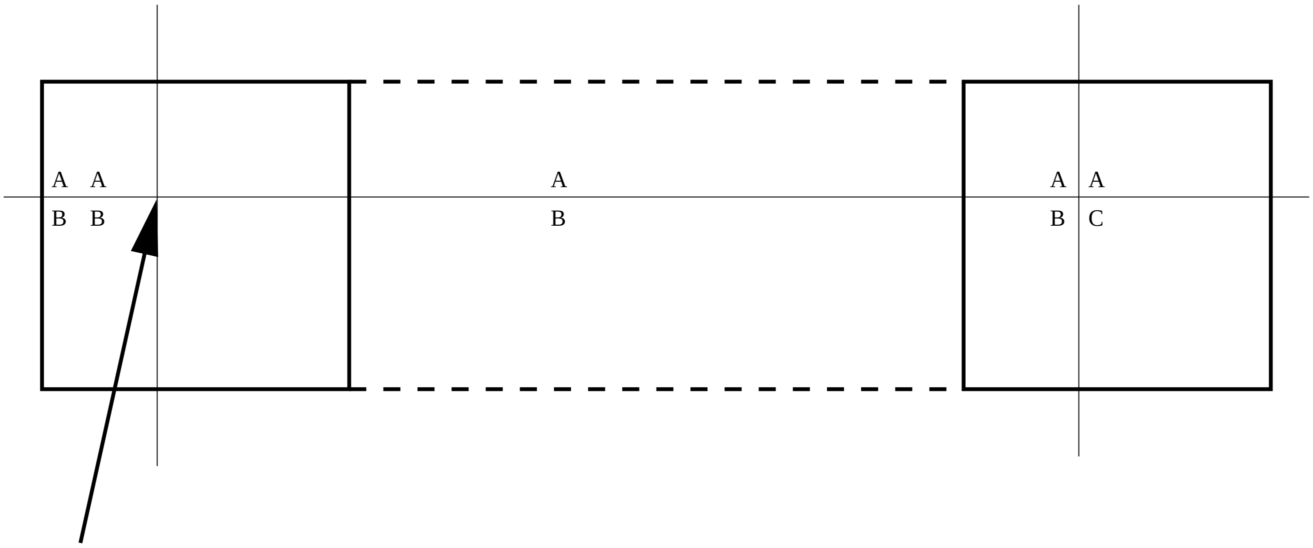

Figure 2: The second case, relevant subcase, of proposition 1. $A \neq B$, but the symbol C might be A.

> $[l, q - 1 + ((r-1) \bmod s)]$ is the only column in the s-block where the pivot might occur s-scaled.

■

Corollary 2 *Let $[i, j]$ be a pivot location in P and let $[p, q]$ be the starting location of an s-block in T. Then there is at most one possible position in the s-block where an s-scaled occurrence of the pivot may start.*

3.2 The Algorithm

The algorithm is a modification of the alphabet independent two dimensional matching algorithm of [6]. Although that algorithm is textbook material by now (see [14, 12]) we bring the algorithm in the appendix for completeness' sake. The modification allows running the algorithm on an *s-coarse text*. An s-coarse text is a $\lfloor \frac{n}{s} \rfloor \times \lfloor \frac{n}{s} \rfloor$ grid of s-blocks. We will discover all s-occurrences of the pattern by a constant amount of work per s-block in the $\lfloor \frac{n}{s} \rfloor \times \lfloor \frac{n}{s} \rfloor$ grid. The algorithm below is to be run separately for every $s = 1, \ldots, \lfloor \frac{n}{m} \rfloor$.

Algorithm for Exact Matching of P^s in an s-coarse Text

Step 1 Calculate the unique pivot position for every s-block.

Step 2 Calculate the unique possible starting position of P^s for every s-block. (Done by copying the pivot position of that pattern appearance.)

Step 3 Candidate Consistency Stage.

Step 4 Candidate Verification Stage.

end Algorithm

As was shown in section 3.1 Steps 1 and 2 can be done in time $O(\lfloor \frac{n}{s} \rfloor^2)$. We will now show a $O(\lfloor \frac{n}{s} \rfloor^2)$ implementation of the Consistency and Verification Stages.

Consistency Stage

In the consistency stage, all pairs of possible pattern starting positions (*sources*) are compared using the witness table. Either the two sources are compatible, i.e. agree on the overlap, or one of the sources is eliminated.

Conceptually, our algorithm should be identical to the consistency algorithm of [6], however, there is one difference we need to account for. When comparing two s-blocks, each one has a unique possible starting position. We call this position its *offset*. Formally, for an s-block starting at position $T[p, q]$, if $T[l, r]$ is the unique position where the pattern can occur in the block, the offset of the block is defined to be $[l - p, r - q]$.

The first extra consideration necessitated by the possibility of different offsets is the maximum distance between sources. In the unscaled version consistency between two candidate sources is checked when the L_0 distance between them is up to $m-1$. We will need to check till distance m, to allow for differing offsets. This can result in more consistency checks than in the unscaled version, but the overall complexity is still linear.

If the witness table for the two sources identifies a conflict, it is sufficient to check the bottom rightmost element of the s-block where the conflict lies. Regardless of the offsets of the sources, the bottom rightmost element lies in both scaled version of the conflicting positions.

The situation is more complicated where the witness table indicates consistency. Here different offsets may still cause a conflict. We handle the three possible cases:

1. Equal offsets. There is no conflict between the sources.
2. The two offsets differ in one index. Without loss of generality we may assume that index is the column index. The situation is: Source c_1 (as location of the s-coarse text) has offset $[x, y_1]$, and source c_2 has offset $[x, y_2]$. See figure 3.

   In this case the two sources are consistent if the overlapping part of the pattern has the property that all columns are equal (or in other words each line has only one character). A lookup in the augmented witness table will tell us if this is the case, and if the property that all columns are equal does not hold, will point to an appropriate (in the overlapping region) location $[p, q]$ such that $P[p, q] \neq P[p, q+1]$. Using the location $[p, q]$ we find a location of the text T where the two sources expect a different character. Then a single check of the text will eliminate one of the two sources.
3. The two offsets differ in their two indices. The situation here is very similar to the previous one. Either the entire area of the overlap consists of a single symbol, in which

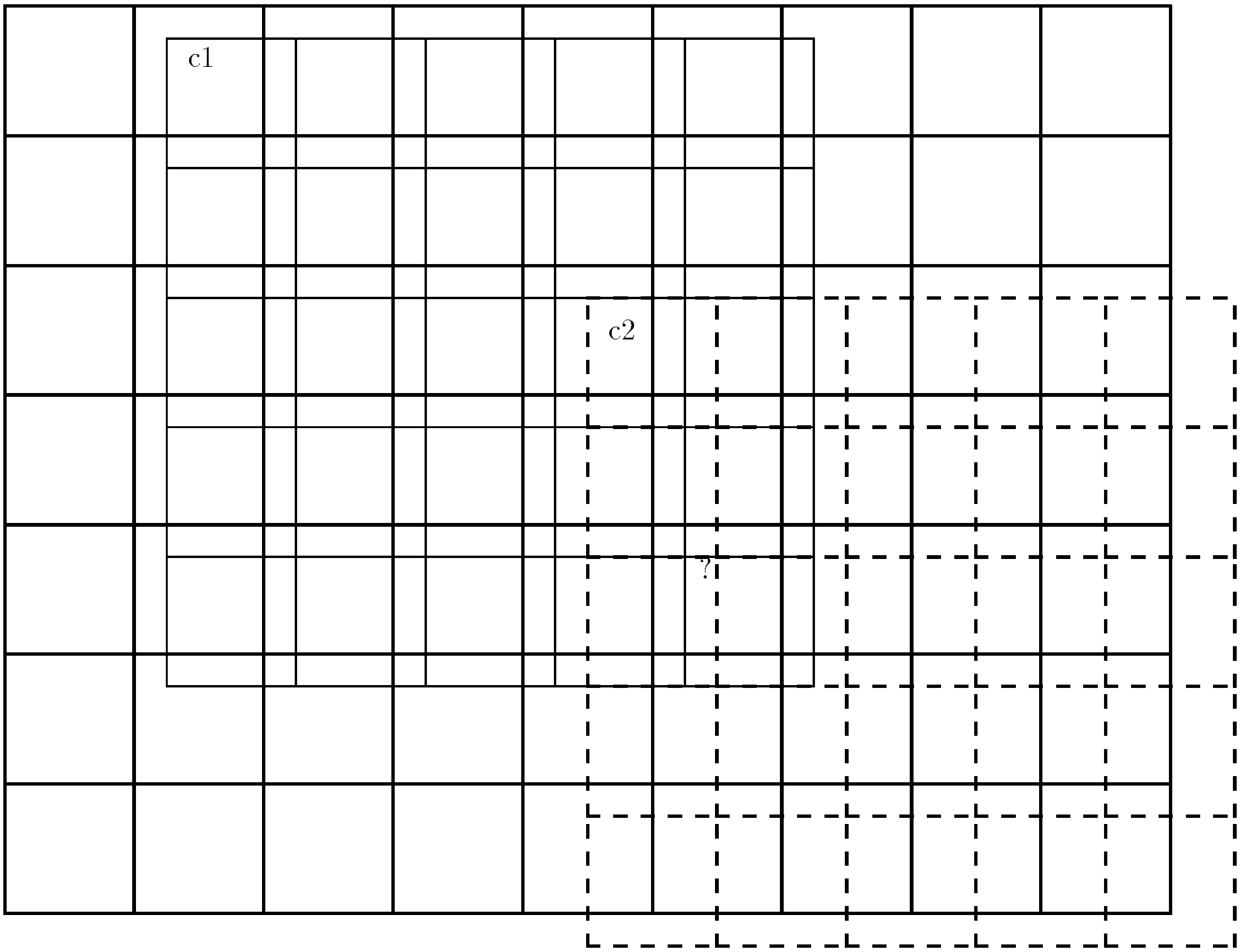

Figure 3: In the case when the offsets of the two candidates c_1 (solid thin line) and c_2 (dotted line) differ in one index consistency is still possible if the region they overlap has all lines made by only one character. If not, a lookup table will give us a location - for example the one denoted by X - where the two candidates expect a different character

case there is no conflict, or we can decide in constant time where two different symbols should occur and eliminate one of the two options.

Time: $O(\frac{n^2}{s^2})$ since we do a constant amount of work for every s-block and the underlying source consistency algorithm has linear time complexity.

Correctness: At the conclusion of the Consistency Stage we have in each s-block at most one possible source. Moreover, we also know that if there are pattern appearances at any of these sources, they can not conflict on the overlap. This follows immediately from the correctness of the consistency stage of the [6] algorithm and from the augmented witness table to handle offsets. Note that transitivity of nested consistency is kept even when the offsets are different.

From now on we will refer to a candidate as a location (s-block) of the s-coarse text together with the unique offset at that location. The region covered by the candidate is the region of the text where the s-scaled pattern starting at the candidate position is to match the text. Formally, the region covered by candidate $[a, b]$ (as a position in the s-coarse text) with offset $[x, y]$ is $T[(a-1)s+x+1 \ldots s(m+a-1)+x; (b-1)s+y+1 \ldots s(m+b-1)+y]$.

Verification Stage

The verification stage of the unscaled two dimensional matching algorithm (as described in [6], see also the appendix) has three phases: first each location of the text is marked with (at most) one candidate source containing that location. Then each location does a single character checking. A single check is sufficient since by consistency all sources expect the same character at the same location. If the text symbol checked does not equal the expected pattern symbol, a flag is raised at that location. Any candidate source in whose area a flag is raised is discarded, and the algorithm outputs the remaining sources. A *wave* algorithm is used to mark text locations with the appropriate pattern index with which they should be compared and to discard all candidates which contain a flagged location. As shown in [6], the *wave* runs in linear time.

We intend to follow the above outline on the s-coarse text. This means that only a constant number of comparisons is allowed for every s-block. This is not a problem with s-blocks that are well within the various pattern s-occurrences that cover that s-block. However, care should be taken with s-blocks on the edges, since different pattern s-occurrences may start at different offsets. Thus, the s-block area checked by one candidate may not be precisely the one that a different candidate would need to check.

Below is a modification to the verification stage of [6]. The main idea behind this modification is insuring that the edges are taken care of, and that all "internal" (the inside area) s-blocks are guaranteed in their *entirety* to be correct for all candidates of which they are internal.

Definition 4 Let $[i, j]$ (as location of the s-coarse text) be a candidate source.

The *inside area* (see figure 4) of the candidate is the region of the text covered by the s-blocks $[p, q]$ where $i < p < i + m$ and $j < q < j + m$. In terms of text T, the inside area of the candidate $[i, j]$ is $T[is + 1 \ldots (i + m - 1)s; js + 1 \ldots (j + m - 1)s]$.

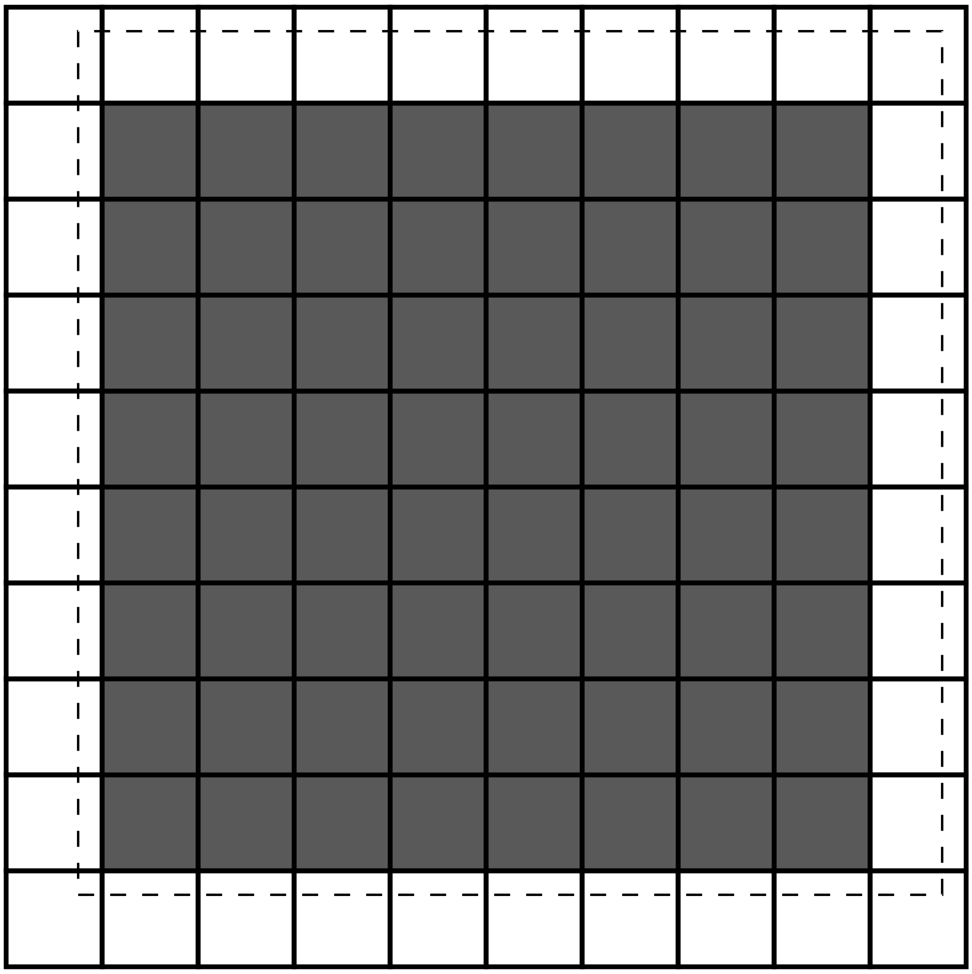

Figure 4: The solid line represents the s-coarse text. A candidate is represented by the dotted line and its *inside area* is filled.

In general the inside of an $m \times m$ pattern scaled to s is the area covered by the $(m-1)\times(m-1)$ *whole* s-blocks that appear inside it. The only exception is when the candidate source starts at the integral boundary of an s-block. We still define the inside as $(m-1)\times(m-1)$ s-blocks.

We say that a location in the s-coarse text is *inside* a candidate source if it is in the inside area of that source.

Let $c = [i, j]$ be the location in T of a candidate source for an s-occurrence of P. The *boundary* of source c is composed of the text locations where the s-occurrence of the first row, first column, last row and last column of c should occur. Formally, it is locations $T[i \ldots i+(s-1);\ j \ldots j+ms-1]$, $T[i \ldots i+ms-1;\ j \ldots j+s-1]$, $T[i+(m-1)s \ldots i+ms-1;\ j \ldots j+ms-1]$, $T[i \ldots i+ms-1;\ j+(m-1)s \ldots j+ms-1]$.

Since the candidate consistency stage guarantees that nested sources agree on their inside areas, the only problem may occur in the edges. Our algorithm, therefore, checks the edges of every source separately. We will see how this can be done in constant time per candidate.

Verification of the inside area is done as in the verification stage of the unscaled algorithm. However, every s-block may be composed of parts of four different s-scaled symbols. All four parts need to be checked.

Verification Algorithm Outline (For s-coarse Text)

1. For every source c in s-block $[i, j]$ and offset $[x, y]$, mark s-block $[i+1, j+1]$ as $\langle 2, 2\rangle$ with offset $[x, y]$.
2. Do a vertical wave with horizon $m-1$ from top to bottom on the s-coarse text, incrementing the second coordinate of $\langle 2, k\rangle$ and passing the offset of the wave source.
3. Do a horizontal wave with horizon $m-1$ from left to right on the s-coarse text, incrementing the first coordinate of $\langle \ell, k\rangle$ and passing the offset of the wave source.
4. For each s-block do the following: Let $[p, q]$ be the starting location in T of an s-block. Let the coordinate pair produced by the wave be $\langle \ell, k\rangle$, and let its offset be $[a, b]$. Verify that all the symbols in $T[p \ldots p+a-1, q \ldots q+b-1]$ equal $P[\ell-1, k-1]$ (with the convention that if a region of the text is empty, the equality is satisfied), that all the symbols in $T[p+a \ldots p+s-1, q \ldots q+b-1]$ equal $P[\ell, k-1]$, that all the symbols in $T[p \ldots p+a-1, q+b \ldots q+s-1]$ equal $P[\ell-1, k]$, and that all the symbols in $T[p+a \ldots p+s-1, q+b \ldots q+s-1]$ equal $P[\ell, k]$. If any of the above is not true, raise a flag.

   In words, we made sure that the part of the s-occurrence of all four appropriate symbols within the s-block is correct.
5. Do a backward propagating wave. The sources of the wave are positions $[i, j]$ for which $[i+1, j+1]$ was flagged. The horizon is $m-1$. The action is eliminating all candidate sources that get hit by the wave.
6. For all remaining sources, if the edge matches the pattern, retain the source, else discard it.

end Algorithm

Verification Time: Every inside s-block is checked in constant time by four applications of range minimum queries to B_r and C_r, and four symbol comparisons. We formally describe the comparison of $T[p+a \ldots p+s-1, q+b \ldots q+s-1]$ with $P[\ell, k]$. The other cases are similar. The flag is raised if one of the following conditions is not met:

(i) $\min(C_r[p+a, q+b],\ C_r[p+a+1, q+b],\ \ldots\ ,\ C_r[p+s-1, q+b] \geq s-b$

(ii) $\min(B_r[p+a+1, q+b],\ B_r[p+a+2, q+b],\ \ldots\ ,\ B_r[p+s-1, q+b] \geq s-b$

(iii) $T[p+a, q+b] = P[\ell, k]$

Clearly, this means a constant amount of work per s-block.

The edges can also be checked in constant time by the following method. In a preprocessing stage, find all s-scaled occurrences of the first (last) row (column). This can be done in time $O(n^2)$ by the algorithm described in [11]. Store the results in four $n \times n$ tables. To check if a row occurs s-scaled starting at location $T[i, j]$, check if entry $T[i, j]$ is at the table for s-scaled rows, and verify via a range minimum query to B_r that the row repeats s consecutive times. Note that for the columns, we will need similar B_c and C_c data structures.

Conclude: The total verification time for scale s is $O(\lfloor \frac{n}{s} \rfloor^2)$.

Correctness: The edges get checked explicitly. The internal s-blocks are consistent for all candidate sources to which they are internal, and they report exactly to the sources for which they are internal. ■

4 Dictionary Scaled Matching

We are now ready to solve the problem of scaled dictionary matching for a static dictionary of squares. This is the scaled version of the problem solved by Amir and Farach in [8]. In that paper an algorithm (henceforth refered to as AF) was presented that preprocesses the dictionary in time $O(d(\log k + \log \sigma_D))$, where d is the sum of the sizes of all dictionary patterns, k is the number of patterns and σ_D is the number of symbols in the dictionary. The AF algorithm then finds all occurrences of all dictionary patterns that appear in text T in time $O(|T|(\log k + \log \sigma_D))$. Note that this result is not alphabet independent. The scaled algorithm we present has exactly the same complexity.

For completeness' sake we present an overview of the AF algorithm.

The main idea behind the AF algorithm is linearizing the pattern matrices along the diagonal to produce a dictionary of strings over an alphabet of subrow/subcolumn pairs. The Aho and Corasick automaton [1] is then constructed, where every symbol is a subrow/subcolumn pair.

An input text is similarly linearized along all the diagonals, and the Aho and Corasick automaton is run on the linearized text. Symbol comparisons are done in constant time via *lowest common ancestor* (LCA) queries on suffix trees.

An outline of the algorithm appears below.

Algorithm AF for 2-dimensional exact dictionary matching

Preprocessing:

1. Construct suffix trees T_c (for the concatenation of all the columns of all dictionary patterns, from bottom to top) and T_r (for the concatenation of all the rows of all dictionary patterns, from right to left). Preprocess T_c and T_r for LCA.

2. Construct an Aho and Corasick automaton on the linearized patterns.

Text Scanning:

1. Insert text rows and columns into T_r and T_c respectively.

2. Scan the linearized text using the Aho and Corasick automaton.

3. Remove text rows and columns from T_r and T_c respectively.

End Algorithm

Note that the Aho and Corasick (henceforth AC) automaton is constructed precisely in the manner that Aho and Corasick describe, and the text scanning using the automaton is also identical to the Aho and Corasick text scanning. The only change introduced by the AF algorithm is that whenever the AC algorithm checks equality between two symbols, the AF algorithm calls a subroutine that, in constant time, checks equality of a subrow/subcolumn pair.

The main idea for scaling the AF algorithm is similar to the single pattern scaled matching we presented before. Simply divide the text into a grid of s-blocks for each scale $s = 1, \ldots, \lfloor \frac{n}{m} \rfloor$. Subsequently, run an adapted version of AF for each scale s, where the time complexity per s-block is the same.

However, we need to answer a major difficulty. The "pivotal" reason our single pattern algorithm worked is that there could be at most one position in an s-block where the pivot can start. While that unique position was found independent of the pattern (Proposition 1), it did assume knowledge of the pivot's location in the pattern. When dealing with a dictionary, the pivot of different patterns may occur in different locations, and then Proposition 1 will no longer guarantee at most one starting location per s-block.

We solve this problem by guaranteeing that the pivot is in the same location in all dictionary patterns. We need an efficient method of converting the patterns to and from a set of patterns with the same pivot location.

Assumption: We again assume that the patterns are *non-trivial.*

Definitions:

1. A *diagonal pivot* of $m \times m$ matrix P is the largest i such that rows $P[1; 1, \ldots, m], \ldots, P[i-1; 1, \ldots m]$ are either all equal to each other or each consists

of one repeating symbol, and columns $P[1,\ldots,m;1],\ldots,P[1,\ldots m;i-1]$ are either all equal to each other or each consists of one repeating symbol.

2. Let P be an $m \times m$ matrix, and let $j = j_P$ be the diagonal pivot of P. The *reduced matrix* $P^{[j]}$ of P is the square matrix $P[j-1,\ldots,m;j-1,\ldots,m]$.

Observations: Every non-trivial square matrix has a diagonal pivot. The diagonal pivot of every reduced pattern $P^{[j]}$ is in location $P^{[j]}[2,2]$.

Note that at least one of the following conditions holds:

1. The partial rows $P[j-1;j-1\ldots m]$ and $P[j;j-1\ldots m]$ are distinct and one of them contains two distinct symbols.
2. The partial columns $P[j-1\ldots m;j-1]$ and $P[j-1\ldots m;j]$ are distinct and one of them contains two distinct symbols.

We partition the dictionary into two according to which of the two conditions above holds (if both hold chose any) and run the algorithm separately on each class. The two classes are symmetric, so we will assume that all the patterns satisfy the first condition.

Lemma 1 *Consider an s-block starting at position $T[p,q]$ (i.e. $T[p,\ldots,p+s-1;q,\ldots,q+s-1]$). Then there is at most one position in the s-block where any of the reduced patterns in the class can occur. This position can be found in constant time.*

Proof. Position $[2,2]$ is a pivot of the reduced pattern for all patterns. We apply Proposition 1. The offset for the s-block starting at position $T[p,q]$ is given by the offset for the pivot in the block starting at position $T[p+s,q+s]$. This is true for all dictionary patterns. ■

We say that a pattern P of size $m \times m$ *ends s-scaled* at the s-block starting at position $T[p,q]$ if P occurs s-scaled in the s-block starting at position $T[p-(m-1)s, q-(m-1)s]$.

Outline of Scaled Dictionary Matching Algorithm

P **Preprocessing:** Partition the dictionary into two classes. For each class, construct dictionary D^R whose elements are the reduced matrices of the patterns in dictionary D. Preprocess this dictionary in a manner similar to the dictionary preprocessing of AF.

S **Text Scanning:**

S.1 For $s = 1,\ldots,\lfloor \frac{n}{m} \rfloor$ construct the grid of s-blocks, find for each s-block the unique offset and run a modification of algorithm AF to find for each s-block the biggest reduced pattern ending at the s-block.

S.2 For every reduced pattern found, find the biggest original pattern which ends at the same position.

end Algorithm Outline

We will show how to implement these operations so that the preprocessing of the dictionary takes time $O(d(\log k + \log \sigma_D))$, the scanning for reduced patterns (step S.1) takes time $O(\lfloor \frac{n}{m} \rfloor \times \lfloor \frac{n}{m} \rfloor (\log k + \log \sigma_D))$, and each extension check in step S.2 takes time $O(\log k + \log \sigma_D)$.

4.1 Dictionary Preprocessing

We distinguish between the preprocessing necessary for the dictionary patterns, and the preprocessing of the text, for every given text.

4.1.1 Pattern Preprocessing – Suffix Trees and LCA's

Recall that the AF algorithm needs to make constant time comparisons of subrows and subcolumns. In order to introduce scaling, we need constant time comparisons of *scaled* subrows and *scaled* subcolumns. This can be achieved by using the *run-length representation* of the rows and columns.

Definition: Let $S = \sigma_1 \sigma_2 \cdots \sigma_n$ be a string over some alphabet Σ. The *run-length representation* of string S is the string $S' = \sigma_1'^{r_1} \sigma_2'^{r_2} \cdots \sigma_k'^{r_k}$ such that: (1) $\sigma_i' \neq \sigma_{i+1}'$ for $1 \leq i < k$; and (2) S can be described as concatenation of the symbol σ_1' repeated r_1 times, the symbol σ_2' repeated r_2 times, ..., and the symbol σ_k' repeated r_k times.

We denote by $S^\Sigma = \sigma_1' \sigma_2' \cdots \sigma_k'$, the *symbol part* of S, and by $S^\#$, the vector of natural numbers $r_1, r_2, ..., r_k$, the *run-length part* of S.

The string $S^{q\#} = \frac{r_2}{r_1}, \frac{r_3}{r_2}, ..., \frac{r_{i+1}}{r_i}, ..., \frac{r_k}{r_{k-1}}$ is the *run-length quotient* of $S^\#$.

Rather than construct suffix trees T_c and T_r (as was done in AF), we construct suffix trees $S_c^\Sigma, S_r^\Sigma, S_c^{q\#}$, and $S_r^{q\#}$. S_c^Σ ($S_c^{q\#}$) is a suffix tree of the concatenation of the symbol (run-length quotient) parts of all columns of all dictionary patterns from bottom to top. S_r^Σ ($S_r^{q\#}$) is a suffix tree of the concatenation of the symbol (run-length quotient) parts of all rows of all dictionary patterns from right to left. (For suffix tree construction see e.g. [22, 25].)

Process all suffix trees for LCA [20].

Time: Suffix tree construction is done in time $O(d(\log k + \log \sigma_D))$. Preprocessing for LCA is done in time $O(d)$.

The reason for constructing the suffix trees of the run-length quotients is that we will need to know in constant time whether a text subrow or column is a *scaled* appearance of a pattern subrow or column. As was observed in [16, 11], if the first and last elements scale, and the symbol part and quotient part of the text and pattern match, we have a scaled matching.

4.1.2 Aho and Corasick Automaton

We construct the Aho and Corasick automaton for dictionary D exactly as is done in the AF algorithm.

4.1.3 Text Preprocessing

1. For every input text, and every scale s, find the offset of every s-block.
2. Construct B_r and C_r and process them for range minimum queries.
3. Insert into the suffix trees $S_c^\Sigma, S_r^\Sigma, S_c^{q\#}$, and $S_r^{q\#}$ the suffixes of the concatenations of the symbol parts of the columns (from bottom to top) and rows (from right to left) of the text, and the suffixes of the concatenations of the run-length quotients of the columns (from bottom to top) and rows (from right to left) of the text, respectively. Dynamically update the LCA preprocessing as in [8].

4.2 Text Scanning

We need to be able to answer the following queries in constant time:

Query 1: Starting from position $T[i,j]$ is the submatrix (notice it is upside down) $T[i, i-1, \ldots, i-sm+1; j, j+1, \ldots, j+s-1]$ equal to P^s where P is the $m \times 1$ matrix $P = p_1 \cdots p_m$, a subcolumn of one of the patterns (also seen bottom up) ?

Meaning of Query 1: Is the submatrix starting at $T[i,j]$ and extending forward for s locations and up for sm locations, an s-occurrence of the given subcolumn P?

Query 2: Starting from position $T[i,j]$ is the submatrix (notice it is from right to left) $T[i, i+1, \ldots, i+s-1; j, j-1, \ldots, j-sm+1]$ equal to P^s where P is the $1 \times m$ matrix $P = p_1 \cdots p_m$, a subrow of one of the patterns (also seen from right to left)?

Meaning of Query 2: Is the submatrix starting at $T[i,j]$ and extending down for s locations and to the left for sm locations, an s-occurrence of the given subrow P?

We will describe how to answer Query 1 in constant time. Query 2 is handled in a similar fashion.

Outline of Query Response: The first, second and last symbol of the pattern (seen as the symbols appear in S_c^Σ - not necessarily the first, second and last positions of the pattern) scaled to s can be checked against the text by range minimum queries on B_r and C_r, and three symbol checks.

We can also make sure in constant time that subcolumns $T[i, i-1, \ldots, i-sm+1; j], T[i, i-1, \ldots, i-sm+1; j+1], \ldots, T[i, i-1, \ldots, i-sm+1; j+s-1]$ are all equal by a range minimum query.

What remains is checking if the subcolumn obtained from $T[i, i-1, \ldots, i-sm+1; j]$ by removing the first and last portion corresponding to p_1 and p_m (which were checked separately before) equals the subcolumn of P^s obtained from P^s by removing the first and last symbol (as the symbols appear in S_c^{Σ} - the symbol part of column, the part without repeating symbols). This can be done by two appropriate LCA queries on S_c^{Σ} and $S_c^{q\#}$.

Conclude: We can check equality and, with a small addition to the above outline, lexicographic order of the text submatrix and scaled pattern substring, in constant time.

We are now ready to describe the text scanning stage.

For every scale s we run the AF automaton on the diagonalized text.

Suppose that at the current moment the automaton is at the following position in the text: in the s-block $[i, j]$, with offset $[x, y]$, and the s-scaled prefix Q (of size $m \times m$) occurs at $T[(i-1)s+x+1, (j-1)s+y+1]$. If Q is actually a reduced pattern or it contains a reduced pattern ending at the same location (the automaton will tell if this is the case) we go to the extension phase.

The AF algorithm does a binary search on all outgoing edges from the current state of the automaton. There are at most k such links, and comparison of the text to each link is constant time by the subroutines used to answer Queries 1 and 2.

Time: The time spent per s-block is $O(\log k)$.

Our algorithm has to be modified however when there is no dictionary pattern that extends prefix Q and therefore the automaton is going to follow a FAIL link. The problem is that the link might lead to another position $[i+f, j+f]$ of the s-coarse text for which another offset $[x_f, y_f]$ has been computed. Here we have to determine if the s-scaled suffix of Q: $Q' = Q[f+1, \ldots, m; f+1, \ldots, m]$ actually occurs at $T(i+f-1)s+x_f, (j+f-1)s+y_f]$, knowing that Q' occurs s-scaled at $T(i+f-1)s+x, (j+f-1)s+y]$.

This is also easily done in constant time using B_r and C_r. It is basically a consistency check. If for example $x = x_f$ and $y \neq y_f$ then Q' occurs at $T(i+f-1)s+x_f, (j+f-1)s+y_f]$ if and only if the area Q' and Q both cover has all the lines made of only one character. One of the cases to be considered is similar to the consistency check described in figure 3.

If Q' does occur s-scaled at position $[i+f, j+f]$ of the s-coarse text with offset $[x_f, y_f]$ then the automaton is resumed from this position, otherwise another FAIL link (from the state in the automaton representing Q') is followed. The offset to which we compare the new Q' (the result of following the FAIL link) is the original offset of Q.

The correctness of the scanning phase then follows directly from the correctness of AF.

4.3 The Extension Step

At the moment we have all locations where s-scaled occurrences of reduced patterns end. We need to find out which of these locations represent scaled occurrences of complete dictionary patterns. A look-up table tells us immediately if a pattern in the dictionary is a suffix of the reduced pattern we are analyzing. Then we check for extensions for each reduced pattern found.

We distinguish between two types of extensions, that follow from the definition of a diagonal pivot j:

1. Each of the first $j-1$ rows is a single repeating symbol In this case, the first $j-1$ columns are all equal. We say that the pattern has a *vertical extension*.

2. Each of the first $j-1$ columns is a single repeating symbol. In this case, the first $j-1$ rows are all equal. We say that the pattern has a *horizontal extension*.

Assume that reduced pattern P' of size $m' \times m'$ starts s-scaled at $T[p,q]$. We will describe how to find all original patterns extending P' in time $O(\log k)$.

During the preprocessing stage, we identify all patterns who extend reduced pattern P'. We split these patterns into two classes, according to their extension type, vertical or horizontal (if a pattern has both extension types, choose one arbitrarily). Since the vertical and horizontal extensions are symmetric, we will describe the vertical case only.

Consider the original patterns $P_1, \ldots P_r$ which have P' as reduced pattern and can be obtained by vertical extension. Let $1 > j_i \geq m_i$ be the diagonal pivot of P_i, so that $P_i[j_i - 1 \ldots m_i; j_i - 1 \ldots m_i] = P'$. Recall that each of the first $j_i - 1$ rows is a single repeating symbol and the first $j_i - 1$ columns are all equal. We call column $j_i - 2$ of P_i ordered from bottom to top, (i.e. $P_i[m \ldots 1; j_i - 2]$), P_i's *extension column*. If $j_i = 2$ then $P_i = P'$ and we can report the occurrence of P_i and then only consider the other patterns in $P_1, \ldots P_r$.

If P_i is to extend the reduced pattern P' starting at $T[p,q]$, we must have that the s-scaled extension column of P_i is a prefix of the string (going upward) $T[p + m's - 1 \ldots 1; q - 1]$. This check is, in essence, Query 1 of subsection 4.2.

During preprocessing the extension columns of the original patterns $P_1, \ldots P_r$ are sorted lexicographically. Comparisons via Query 1 take constant time. We can thus sort the extension columns in time $O(r \log r)$ (which does not increase the overall complexity of the preprocessing).

Now perform two binary searches to find two indices l and b such that the s-scaled extension columns of the patterns $P_l \ldots P_b$ is a prefix of the subcolumn (going from the bottom up) $T[p + m's - 1 \ldots 1; q - 1]$. This takes time $O(\log r) = O(\log k)$.

We now know that the extension column of P_l is the shortest and the extension column of P_b is the longest that match the extension subcolumn of the text occurrence of P'.

Another binary search from l to b allows us to find the biggest (if any) of the original patterns that extends the reduced pattern P'. See figure 5. For each such pattern, an appropriate minimum range query on C_r is necessary, to assure that the top rows are indeed runs of the correct symbols.

5 Future Work

The pivot idea seems to be extendable to scale more general dictionary matching algorithms than AF. We conjecture that every known dictionary matching algorithm can be scaled without degradation in its time complexity.

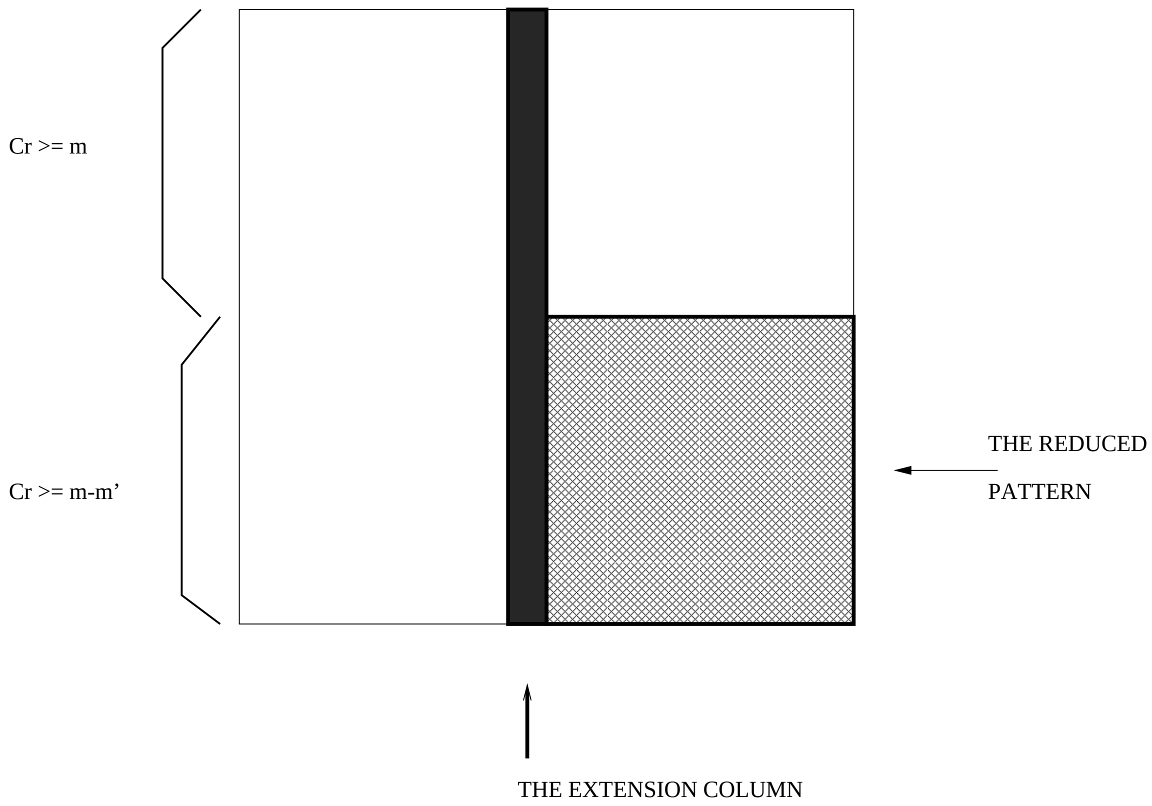

Figure 5: Assuming the reduced pattern (of size $m' \times m'$) and the extension columns both occur checking if the pattern (of size $m \times m$) occurs takes constant time using two minimum range queries.

A very interesting remaining open problem is efficiently finding all scaled occurrences for *all scales*, not just discrete scales. This is open even for one dimensional non-dictionary string matching.

6 Appendix

For the sake of completeness, we bring here the witness algorithm for alphabet independent two dimensional pattern matching [6, 14, 12].

The algorithm consists of a pattern preprocessing part, where the witness table is constructed, and a text scanning part. We do not need to change the witness table construction. The interested reader can find details of the that part in [18, 4]. The additional preprocessing we need has been described in the body of the present paper.

The text scanning part is composed of two phases, text consistency, and text verification. In the consistency phase all consistent potential *sources* are identified. In other words, we eliminate all locations where the pattern may start except for locations that are consistent with each other, i.e. where for each pair of sources it is *possible* for a pattern to start at both sources.

The verification phase verifies which of the consistent sources is indeed a start of a pattern occurrence.

The consistency phase is done in two parts. The first is consistency within columns and the second is consistency throughout the two dimensional text.

Notation: We denote that sources s_1 and s_2 are consistent by $s_1 \doteq s_2$ and inconsistent by $s_1 \bowtie s_2$.

Column Consistency Algorithm $\mathsf{Consist}(c)$
Eliminate inconsistent candidates within a column, producing $\mathsf{Consist}(c)$, the list (in order, from the bottom to the top) of the surviving candidates.

Step 1 Get column number, c.

Step 2 Create a doubly linked list, S, of consistent candidates in column c. Initialize S by adding candidate (n, c) to the top of S.

Step 3 For row $r = n - 1$ to 1 do:

3.1 Let (x, c) be the top candidate in S. Test if candidates (r, c) and (x, c) are consistent by reference to the witness arrays:

3.1.1 If $(r, c) \doteq (x, c)$, then add (r, c) to the top of S.

3.1.2 If $(r, c) \bowtie (x, c)$ then use the witness character in the text to eliminate one of the candidates. If (x, c) is eliminated, remove it from S and repeat step 3.1 with the new top candidate in S. If no candidates remain in S, add (r, c) to S.

Step 4 Return S.

Two Dimensional Consistency Algorithm

Step 1 For $i \leftarrow 1$ to n $C_i \leftarrow$ Consist(i)

Step 2 For $i \leftarrow 1$ to n initialize R_i to be an empty list of candidates for each row i.

Step 3 Put the candidates on C_n onto their appropriate R_i lists.

Step 4 For $i \leftarrow n-1$ downto 1 do

- 4.1 Call Bottom-Up(i)
- 4.2 Call Top-Down(i)
- 4.3 Add surviving candidates from column i to the appropriate R_j lists.

We describe procedure Bottom-Up only, since procedure Top-Down is symmetric.

Procedure Bottom-up(c)

1 *Initialize:* cur gets bottom value from C_c. $row \leftarrow n$ is a pointer to the last row

2 Repeat until over the top of C_c

- 2.1 If cur is consistent with leftmost item on R_{row}, then $row \leftarrow row + 1$
- 2.2 If cur is not consistent with leftmost item on R_{row}, then find a witness to their inconsistency. Check which of them is inconsistent with the text and remove candidate from its list. If cur is removed, set cur to the next higher item on C_c, otherwise do nothing.
- 2.3 If the row counter points to a row above cur's row, set cur to the next higher candidate in C_c.

All remaining candidates are now mutually consistent. Note that each text element $t = T[r,c]$ may be contained by several candidates, and that all candidates agree on their area of overlap. We say a candidate (i,j) is *relevant* to a position (k,l) if $0 \leq k-i \leq m-1$ and $0 \leq l-j \leq m-1$, that is, if candidate (i,j) covers position (k,l). This leads to the following crucial observation: Every position in T can be labeled as either *true* or *false*, where *true* means that it equals the unique pattern symbol expected by all relevant candidates, and *false* is all other cases. Thus, every text element needs to be compared to a *single* pattern element, and every candidate source that contains a *false* element within it is not a pattern appearance and can be discarded.

The candidate verification algorithm follows:

Candidate Verification Algorithm

Step 1 Mark every text location $T[r,c]$ with a *pattern coordinate pair* $\langle i,j\rangle$, where $\langle i,j\rangle$ are the coordinates of the pattern element $P[i,j]$ that $T[r,c]$ should be compared with.

There may be several options for some locations, namely, the position of the scanned text element relative to each of its relevant candidates. However, any will do since all candidate sources are now compatible. If a location is not contained in any candidate source it is left unmarked. This step is implemented by the *wave* method (procedure Wave).

Step 2 Compare each text location $T[r,c]$ with $P[i,j]$, where $\langle i,j\rangle$ is the pattern coordinate pair of $T[r,c]$. If $T[r,c]=P[i,j]$ then label $T[r,c]$ as *true*, else label it *false*.

Step 3 Flag with a *discard* every candidate that contains a *false* location within its bounds.

This flagging is also done by the wave method.

Step 4 Discard every candidate source flagged with a *discard*. The remaining candidates represent all pattern appearances.

The only remaining task is showing how to mark the text elements with the appropriate pattern coordinate pairs. We adopt the popular sports fans technique - *the wave*.

A *wave* has an *axis*, a *direction*, a *source*, a *horizon* and an *action*.

1. The *axis* is either horizontal or vertical.
2. The *direction* is left-to-right or right-to-left, if the axis is horizontal, and top-to-bottom or bottom-to-top if the axis is vertical.
3. The *source* is the location where a wave starts.
4. The *horizon* is the distance from the wave source that the wave propagates until it dies. The wave source is counted in the horizon.
5. The action taken by the wave is the activity done at each location as the wave propagates.

Given the above parameters, the wave starts at the first source and propagates in the defined direction along the given axis. It keeps a counter from 1 (the source location) to *horizon*. Every location advanced increases the counter by one, unless the new location is a source, in which case the counter is initialized to 1. If the counter reaches *horizon* the wave dies and starts again at the next source. At each location that the wave touches, it does the required action.

Example: The verification wave.

Starting at the top (left) of each column (row), a wave is propagated going down (to the right) as follows. The first element stands and waves its pattern coordinate pair, if such exists. This nudges the neighbor below (to the right of) it to jump and raise its own pair. If it does not have a pair, it borrows its antecedent's pair, incrementing by 1 its row (column) coordinate, to adjust for its position relative to the same source. If the pair assigned to some position exceeds the horizon, that position is left unmarked.

Thus in two sweeps of the text, column waves and row waves, each text element is given an appropriate pattern coordinate pair. Details of the verification wave follow (h denotes the horizon):

The Wave Algorithm

1 *Initialization:* Mark every candidate with $\langle 1, 1\rangle$.

2 *Column Waves:* For each column c, and for all positions r from 1 to n in column c do the following step: If $T[r, c]$ does not have a pair, and $T[r-1, c]$ has pair $\langle i, j\rangle$ with $i < h$ then assign to $T[r, c]$ the pair $\langle i+1, j\rangle$.

3 *Row Waves:* For each row r, and for all positions c from 1 to n in row r do the following step: If $T[r, c]$ does not have a pair, and $T[r, c-1]$ has pair $\langle i, j\rangle$ with $j < h$ then assign to $T[r, c]$ the pair $\langle i, j+1\rangle$.

A similar version of the wave can be used to flag candidates with *discard*. What is propagated there is the *discard* flag, along with a counter pair to make sure the *discard* flag doesn't get propagated too far.

References

[1] A.V. Aho and M.J. Corasick. Efficient string matching. *Comm. ACM*, 18(6):333–340, 1975.

[2] A. Amir and G. Benson. Efficient two dimensional compressed matching. *Proc. of Data Compression Conference, Snow Bird, Utah*, pages 279–288, Mar 1992.

[3] A. Amir and G. Benson. Two-dimensional periodicity and its application. *SIAM J. Comp.*, 27(1):90–106, February 1998.

[4] A. Amir, G. Benson, and M. Farach. The truth, the whole truth, and nothing but the truth: Alphabet independent two dimensional witness table construction. Technical Report GIT-CC-92/52, Georgia Institute of Technology, August 1992.

[5] A. Amir, G. Benson, and M. Farach. Optimal parallel two dimensional pattern matching. *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 79–85, 1993.

[6] A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two dimensional pattern matching. *SIAM J. Comp.*, 23(2):313–323, 1994.

[7] A. Amir, G. Benson, and M. Farach. Optimal two-dimensional compressed matching. *Journal of Algorithms*, 24(2):354–379, August 1997.

[8] A. Amir and M. Farach. Two dimensional dictionary matching. *Information Processing Letters*, 44:233–239, 1992.

[9] A. Amir, M. Farach, R.M. Idury, J.A. La Poutré, and A.A Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995.

[10] A. Amir, M. Farach, and Y. Matias. Efficient randomized dictionary matching algorithms. *Proc. 3rd. Combinatorial Pattern Matching Conference*, pages 259–272, 1992. Tucson, Arizona.

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

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

[13] R. Cole, M. Crochemore, Z. Galil, L. Gąsieniec, R. Harihan, S. Muthukrishnan, K. Park, and W. Rytter. Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions. *Proc. 34th IEEE FOCS*, pages 248–258, 1993.

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

[15] Larry S. Davis. Personal Communication, 1989.

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

[17] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proc. 16th ACM Symposium on Theory of Computing*, 67(135-143), 1984.

[18] Z. Galil and K. Park. Alphabet-independent two-dimensional witness computation. *SIAM J. Computing*, 25(5):907–935, October 1996.

[19] R. Giancarlo and R. Grossi. On the construction of classes of suffix trees for square matrices: Algorithms and applications. *Information and Computation*, 130(2):151–182, 1996.

[20] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestor. *Computer and System Science*, 13:338–355, 1984.

[21] R.M. Idury and A.A Schäffer. Multiple matching of rectangular patterns. *Information and Computation*, 117(1):78–90, 1995.

[22] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[23] A. Pentland. Invited talk. NSF Institutional Infrastructure Workshop, 1992.

[24] S. C. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. *Proc. 26th STOC*, pages 300–309, 1994.

[25] P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.