Separable Attributes: a Technique for Solving the Sub Matrices Character Count Problem

Amihood Amir* — AT&T Labs and Bar-Ilan University

Kenneth W. Church† — AT&T Labs

Emanuel Dar‡ — Bar-Ilan University

Abstract

The *subsequence character count problem* has as its input an array $S = s_1, ..., s_n$ of symbols over alphabet Σ and a natural number m. Its output is: for every i, $i = 1, ..., n - m + 1$, the number of *different* alphabet symbols occurring in the subsequence $s_i, s_{i+1}, ..., s_{i+m-1}$. The subsequence character count problem is a natural problem that has many uses. It can be solved in linear time for fixed finite alphabets and in time $O(n \log m)$ for infinite alphabets.

The character count problem can be generalized to two dimensions and becomes the *submatrix character count problem.* Its input is an $n \times n$ matrix T over alphabet Σ and a natural number m. Its output is: for every i, j, $i, j = 1, ..., n - m + 1$, the number of *different* alphabet symbols occurring in the submatrix $T[i + k, j + \ell]$, $k = 0, ..., m - 1; \ell = 0, ..., m - 1$.

The straightforward one dimensional solution slides a window along the text adding an element and deleting an element at every step. The problem with two dimensions is that at every move of the window there are m elements added and m deleted.

In this paper we present an alternate one dimensional solution that generalizes to two dimensions. We achieve a $O(n^2)$ time solution to the submatrix character count problem over finite fixed alphabet and a $O(n^2 \log m)$ solution over an infinite alphabet.

1 Introduction

The "sliding window" technique has proven useful in a number of one dimensional array problems. The idea is to consider a certain property within a consecutive subarray, and then maintain that property as the "window" slides by one location. As an example consider the *subsequence sums* problem.

*AT&T Labs – Research, Shannon Laboratory, E278, 180 Park Avenue, Florham Park, NJ 07932-0971, (973)360-8120; aamihood@research.att.com; Part of this work was done while the author was at Georgia Tech, College of Computing and partially supported by NSF grant CCR-01-04494.

†AT&T Labs – Research, Shannon Laboratory, D235, 180 Park Avenue, Florham Park, NJ 07932-0971, (973)360-8620, Fax: (973)360-8077, kwc@research.att.com.

‡Department of Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8408; dar@cs.biu.ac.il; This work is part of E. Dar's Ph.D. Thesis.

Definition 1 *The* subsequence sums *problem is defined as follows:*
INPUT: Array $S = s_1, ..., s_n$ *of numbers and a natural number* m.
OUTPUT: For every i, $i = 1, ..., n - m + 1$, *the sum* $\sum_{j=i}^{i+m-1} s_j$.

This problem can be solved trivially by the "sliding window" technique in time $O(n)$. Simply take the sum of the first m numbers, $\sum_{i=1}^{m} s_i$, as the solution for $i = 1$. Subsequently compute the solution to $i + 1$ by subtracting s_i and adding s_{i+m} to the solution of i.

A trifle more challenging is the *subsequence character count problem.*

Definition 2 *The* subsequence character count *problem is defined as follows:*
INPUT: Array $S = s_1, ..., s_n$ *of symbols over alphabet* Σ *and a natural number* m.
OUTPUT: For every i, $i = 1, ..., n - m + 1$, *the number of* different *alphabet symbols occurring in the subsequence* $s_i, s_{i+1}, ..., s_{i+m-1}$.

In [4] the problem was used to solve the *parameterized matching* problem. The "sliding window" technique keeps a list of counters of the number of occurrences of every symbol in the window, and a variable D designating the number of different symbols in the window. When sliding from position i to position $i + 1$, subtract 1 from the counter of the symbol in location i. If the counter drops to 0 then subtract 1 from D. Add 1 to the counter of the symbol in location $i + m$ and, if it changed from 0 to 1 then add 1 to D.

The time analysis depends on the complexity of accessing the counters in the list. For alphabet $\{1, ..., n\}$ this can clearly be done in time $O(n)$. It was also shown that in the comparison model the problem is solved in time $\Theta(n \log m)$ if the alphabet is ordered. For unordered alphabets the problem can not be solved faster than $\Omega(nm)$. The lower bounds were achieved via reduction to the element distinctness problem [12].

While the "sliding window" works simply and efficiently for subsequences of vectors, it is sometimes necessary to apply it to two dimensional arrays. Of particular interest to us was the *submatrix character count* problem. This problem was motivated by parameterized matching in two dimensions which is a good model for seeking a pattern in an image with a change of color map. The number of different colors in a subarea of an image is considered a "signature". There are many image processing tools that use this measure (see e.g. [9]).

The two dimensional generalizations of the above two problems are formally defined below.

Definition 3

a. *The* submatrix sum *problem is:*
   INPUT: Two dimensional $n \times n$ *array* T *of numbers and a natural number* m.
   OUTPUT: For every $i, j \in \{1, ..., n - m + 1\}$, *the sum* $\sum_{k=i}^{i+m-1} \sum_{\ell=j}^{j+m-1} T[k, \ell]$.

b. *The* submatrix character count *problem is:*
   INPUT: Two dimensional $n \times n$ *array* T *of symbols over alphabet* Σ *and a natural number* m.
   OUTPUT: For every $i, j \in \{1, ..., n - m + 1\}$, *the number of* different *alphabet symbols occurring in the submatrix* $T[k, \ell]$, $k = i, ..., i + m - 1$, $\ell = j, ..., j + m - 1$.

A straightforward generalization of the "sliding window" idea will not work since a shift from position $[i, j]$ to, say $[i, j+1]$ in the text, requires m operations to keep the window property satisfied.

The submatrix character count problem is a special case of a *color range query.*

Definition 4 *The* color range query *problem is:*
PREPROCESS: Two dimensional $n \times n$ array T of symbols over alphabet Σ – the colors.
Subsequently we are interested in answers to:
QUERY: Given intervals $[i_1, j_1]$ and $[i_2, j_2]$, $i_1, i_2, j_1, j_2 \in \{1, ..., n\}$ and $i_1 \leq j_1$, $i_2 \leq j_2$ give the number of different *alphabet symbols (colors) occurring in the submatrix $T[k, \ell]$, $k = i_1, ..., i_2$, $\ell = j_1, ..., j_2$.*

Jonardan and Lopez [10] showed that with a $O(n^2 \log^2 n)$ preprocessing one can answer queries in time $O(log^2 n)$. This means that the submatrices character count problem can be solved in time $O(n^2 \log^2 n)$ by preprocessing and then querying, for every location, the $m \times m$ submatrix starting at that location.

We are not aware of a faster direct approach for solving the submatrix character count problem. However, problems with a similar flavor, where the desired calculation is a convolution, are solved in electrical engineering by a method called *Separable Convolutions* or *Separable Filters* [8]. A similar notion was used by Bird [6] and Baker [5] to solve the two dimensional pattern matching problem.

The contributions of this paper are two-fold.

1. We generalize the notion of *separable convolutions* to *separable attributes.* We believe it is important to keep this method in mind as a "labeled" element of the basic algorithmic toolkit. It has proven useful in the past and, we think, will prove useful for solving various two-dimensional problems in the future.

2. We use the separable attributes method on the submatrices character count problem. This is a basic problem that, surprisingly, did not seem to be solved in the literature. It has turned out to be deceptively tricky. Separable attributes enabled an efficient solution. However, the attribute we ended up using, while simple, is by no means obvious.

This paper is organized as follows. In section 2 we introduce the notion of *separable convolutions* and show how it can be used to solve the character count problem for fixed alphabets of bounded finite size. In section 3 we generalize the notion of separable convolutions to separable attributes that are not convolutions. We introduce the attribute that will efficiently solve the submatrices character count problem. Section 4 gives the formal definition of the attribute used. The outline of our algorithm is presented in section 5. The main technical difficulties of efficiently updating the attribute are handled in section 6.

2 Separable Convolutions

Convolutions are used for filtering in signal processing and other applications. When it is necessary to perform convolutions in two dimensions, a technique commonly used is *separable convolutions.*

In essence, the convolution is performed twice: once vertically on all columns, and then horizontally on all rows.

Intuitively, a convolution uses two initial functions, f and g, to produce a third function h. We formally define a discrete convolution.

Definition 5 *Let T be a function whose domain is $\{0, ..., n-1\}$ and P a function whose domain is $\{0, ..., m-1\}$. We may view T and P as arrays of numbers, whose lengths are n and m, respectively. The* discrete convolution of T and P *is the polynomial multiplication*

$$M[j] = \sum_{i=0}^{m-1} T[j+i]P[i].$$

Using the Fast Fourier Transform (FFT) [7], the polynomial multiplication can be computed in time $O(n \log m)$, in a computational model with word size $O(\log m)$. However, for special cases of P, the convolution can be computed in linear time. For example, the subsequence sum problem in definition 1 is a convolution where $P[i] = 1, \quad i = 0, ..., m-1$.

In a two dimensional convolution, both T and P are two dimensional functions, i.e., T is an $n_1 \times n_2$ matrix and P is an $m_1 \times m_2$ matrix. For ease of notation we will assume that T and P are squares, although all definitions and results in this paper apply to rectangles.

Definition 6 a. *Let T be a function whose domain is $\{[i, j] | i, j = 0, ..., n-1\}$ and P a function whose domain is $\{[i, j] | i, j = 0, ..., m-1\}$. We may view T and P as square matrices of numbers, whose sizes are n^2 and m^2, respectively. The* two dimensional discrete convolution of T and P *is*

$$M[i, j] = \sum_{k=0}^{m-1} \sum_{\ell=0}^{m-1} T[i+\ell, j+k]P[i, j].$$

b. *A convolution is* separable *in the special case where the matrix P, whose size is m^2 can be expressed as the multiplication of two vectors of length m: P_{row} and P_{column}, such that: $P[i, j] = P_{row}[i]P_{column}[j]$.*

Corollary 1 *If P is separable then the convolution of T and P is*

$$M[i, j] = \sum_{k=0}^{m-1} P_{column}[k] \sum_{\ell=0}^{m-1} T[i+\ell, j+k]P_{row}[\ell].$$

Proof: By definition,

$$M[i, j] = \sum_{k=0}^{m-1} \sum_{\ell=0}^{m-1} T[i+\ell, j+k]P[i, j]$$

But P is separable, so this equals:

$$\sum_{k=0}^{m-1} \sum_{\ell=0}^{m-1} T[i+\ell, j+k]P_{row}[i]P_{column}[j] = \sum_{k=0}^{m-1} P_{column}[k] \sum_{\ell=0}^{m-1} T[i+\ell, j+k]P_{row}[\ell]. \square$$

The submatrix sum problem in definition 3.a. is a separable convolution. For every column $T[i, j_0]$, $i = 1, ..., n$, solve the subsequence sum problem and create a new $n \times n$ matrix T' such that in location $T'[i_0, j_0]$ is the sum of the subsequence $T[i, j_0]$, $i = i_0, ..., i_0 + m - 1$. Now solve the subsequence sum problem for every row of T'. The subsequence sum of $T'[i, j]$ is the submatrix sum of $T[i, j]$. Therefore, if we know how to solve the subsequences problem in time $O(f(n))$ (in this case linear time), we can solve the submatrices problem in time $O(f(n^2))$ (in this case, still linear time).

Corollary 2 *The submatrices character count problem can be solved in time $O(n^2)$ for matrix T over a fixed finite alphabet Σ.*

Proof: Let $a, b \in \Sigma$. Define

$$\chi_a(s) = \begin{cases} 1 & \text{if } a = s; \\ 0 & \text{otherwise.} \end{cases}$$

Extend the definition to arrays in the natural form, i.e. if T is an $n \times n$ array then $\chi_a(T)$ is the $n \times n$ array whose $[i, j]$ element is $\chi_a(T[i, j])$.

For a fixed $c \in \Sigma$, the submatrices sum problem of $\chi_c(T)$ can be computed in linear time $O(n^2)$. The crucial observation here is that a submatrix sum is 0 iff c does not appear in that submatrix. Therefore we conclude that c appears in precisely the submatrices whose sum is non-zero.

Repeating the above operation $|\Sigma|$ times, allows us to solve the submatrices character count problem in time $O(|\Sigma|n^2)$, which is linear where Σ a fixed finite alphabet. □

3 The Idea behind the Algorithm

3.1 The General Idea for Two Dimensional Problems

Another way of viewing the separation of the convolutions is by remarking that the two dimensional problem has been somehow projected into one dimension in a manner that preserves the information crucial to the problem.

This type of projection has also been used before. Bird [6] and Baker [5] solved the two dimensional pattern matching problem by using the Aho and Corasick algorithm [1] for finding all occurrences of all pattern rows in the text, and then the Knuth, Morris and Pratt algorithm [11] for finding, in the text columns, the occurrences of the string representing the "flattening" of the pattern rows.

We can describe this process as a *separation of attributes*, where the vertical and horizontal attributes need not be convolutions and need not be equal. They only need to satisfy the following conditions.

For a fixed row i, identify an attribute A, such that $A[i, j]$ holds for the *subcolumn* of length m starting at $T[i, j]$, i.e. $A[i, j]$ holds for $T[i + \ell, j]$, $\ell = 0, ..., m - 1$. The attribute A should have the following traits:

1. It is possible to compute the required submatrix problem for $T[i + k, j + \ell]$, $k, \ell = 0, ..., m - 1$ from $A[i, j], A[i, j + 1], ..., A[i, j + m - 1]$.

2. Given the computation of the submatrix problem from $A[i,j], A[i,j+1], ..., A[i,j+m-1]$, and given $A[i,j+m]$, it is possible to efficiently compute the submatrix problem for $T[i+k, j+1+\ell],\ k,\ell = 0,...,m-1$.
3. Given $A[i,j+\ell],\ \ell = 1,...,n$, it is possible to compute $A[i+1,j+\ell],\ \ell = 1,...,n$ in time "close to" linear in n.

Example: The separable convolution used for the submatrix sum problem is described in this context quite easily. Simply take $A[i,j]$ to be $\sum_{\ell=0}^{m-1} T[i+\ell, j]$. All above traits hold.

1. The submatrix sum at location $[i,j]$ (denoted by $\sigma[i,j]$) is $\sum_{\ell=0}^{m-1} A[i,j+\ell]$.
2. $\sigma[i,j+1] = \sigma[i,j] - A[i,j] + A[i,j+m]$, thus we can compute the submatrix sum at location $[i,j+1]$ in constant time, using the submatrix sum at location $[i,j]$.
3. $A[i+1,j] = A[i,j] - T[i,j] + T[i+m,j]$, thus the attribute values for a new row can be computed in time $O(n)$ using the previous row.

Unfortunately, such a simple scheme does not work for the submatrix character count problem since there is an interdependence between subcolumns. We therefore propose a different scheme for the subsequence character count problem. This new scheme generalizes to two dimensions.

3.2 New Scheme for the Subsequence Character Count Problem

We will employ for one dimension an idea similar to the projection described above. We will define an attribute A with traits that enable a linear time computation of the subsequence character count problem. However, since the problem is one dimensional, A is a vector.

For the first m positions, attribute $A[i]$ is $+1$ if the character in $S[i]$ does not appear previously, otherwise it is 0. Subsequently, Attribute A is $+1$ at a location i if the subsequence $S[i-m+1..i]$ has one more character than the subsequence $S[i-m..i-1]$. It is -1 if the subsequence $S[i-m+1..i]$ has one less character than the subsequence $S[i-m..i-1]$. It is 0 if both subsequences have the same number of symbols.

A formal local definition of A is:
For $i = 1,...,m$:

$$A[i] = \begin{cases} +1 & \text{if } S[i] \neq S[i-\ell],\ \ell = 1,...i; \\ 0 & \text{otherwise.} \end{cases}$$

For other i Define

$$A[i] = \begin{cases} +1 & \text{if } S[i] \neq S[i-\ell],\ \ell = 1,...,m \text{ but } \exists \ell \in \{1,...,m-1\} \text{ such that } S[i-m] = S[i-\ell]; \\ -1 & \text{if } S[i-m] \neq S[i-\ell],\ \ell = 0,...,m-1 \text{ but } \exists \ell \in \{0,...,m-1\} \text{ such that } S[i] = S[i-\ell]; \\ 0 & \text{otherwise.} \end{cases}$$

It is easy to see that the required traits hold for attribute A (only the first two are relevant since the problem is one dimensional):

1. The subsequence character count at location i is $\sum_{j=1}^{i+m-1} A[j]$.

2. There are a number of straightforward ways to implement the efficient computation of $A[i+1]$. We briefly describe one that we later generalize to the submatrix character count problem. $A[i]$, $i = 1, ..., m$ are computed in a manner similar to the one described in the introduction. For $i > m$, when symbol $S[i]$ is encountered a few actions are taken:

   (a) Write -1 in $A[i+m]$ (that is where the new introduced symbol will "drop off" assuming it does not occurs again between now and then).

   (b) If $S[i] \neq S[i-\ell]$, $\ell = 1, ..., m-1$ then $A[i] \leftarrow A[i] + 1$ (recall that it is possible that there was a -1 in $A[i+1]$).

   (c) if $\ell \in \{i-m+1, ..., i-1\}$, $S[i-\ell] = S[i]$, and $S[i] \neq S[j], \forall \ell > j > 0$ (i.e. the character $S[j]$ appears another time within the last window of size m, and its last appearance is in $S[i-\ell]$) then do nothing to $A[i]$ but remove the -1 from $A[i-\ell+m]$, i.e. set $A[i-\ell+m] \leftarrow 0$.

   The actions above guarantee the desired property A. Clearly all operations are constant time except for the verification of $S[i]$'s occurrence in $S[i-\ell]$, $\ell = 1, ..., m-1$ which can be done (as in the introduction) in time $O(1)$ for alphabet $\{1, ..., m\}$, $O(\log m)$ for ordered alphabets in the comparison model, and $O(m)$ in unordered alphabets.

3.3 Idea for the Submatrix Count Problem

We generalize this new scheme for the subsequence character count problem to two dimensions. We are now considering an $n \times n$ matrix T. Whereas $A[i]$ before told us if symbol $S[i]$ adds to the character count or the loss of $S[i-m]$ subtracts from it (or both), we now incorporate the effect of subcolumn $T[i, j+\ell]$, $\ell = 0, ..., m-1$. We will call the attribute array the *Delta Vector* since $D[i, j]$ will be the difference between the number of new characters introduced by subcolumn $T[i, j+\ell]$, $\ell = 0, ..., m-1$ and the number of characters that are lost in the submatrix as a result of dropping subcolumn $T[i-m, j+\ell]$, $\ell = 0, ..., m-1$.

It is clear that the number of different characters in the submatrix whose origin is $[i, j]$ ($T[i+k, j+\ell]$, $k, \ell = 0, ..., m-1$) is

$$\sum_{k=1}^{j+m-1} D[i, k].$$

The remainder of this paper and the main effort herein is devoted to the formal definition and efficient computation of the attribute – the delta vector $D[i, j]$.

4 Formal Attribute Definition

Definition 7 *Let T be a $n \times n$ array. A* strip *is an m row submatrix of T, with the first strip being the first m rows, the second strip is rows 2 to $m+1$, and the ith strip is rows i to $m+i-1$.*

Note that every two consecutive strips have $m-1$ rows in common. In fact, moving from strip i to strip $i+1$ means "peeling off" row i and adding row $i+m$.

Before formally defining the delta vector, we need the concept of *character effective region.*

Definition 8 *The* effective region *of character c in strip i is the set of intervals $E_c^i = \{[a_\ell, b_\ell] | 1 \leq a_\ell < b_\ell \leq n,\ \ell = 1, ..., k\}$ such that for all ℓ:*

1. *c appears in column a_ℓ of the strip.*
2. *c does not appear in any column not in the intervals.*
3. *For $b_\ell < n$, c does not appear in columns $b_{\ell-j}$, $j = 0, ..., m-2$ but does appear in column $b_{\ell-m+1}$.*
4. *For any $d \in [a_\ell, b_\ell]$ c appears in one of the columns $d - j$, $j = 0, ..., m-1$.*

The intuitive meaning of the effective region of a character is the area of the strip where that character appears in an $m \times m$ matrix. A character that appears in a strip column, no matter on which line, can effect all the sub matrices that contain this column. Suppose that a character appears first in column j then the effective region of this character will be $[j, j+m-1]$, means that every matrix that starts within the interval contains this character. Now suppose that a new instance of this character appears later, say at column k, $k > j$. If $k > j+m-1$ then k does not belong to the previous interval and it creates a new interval $[k, k+m-1]$. If k belongs to the previous interval $(k \leq j+m-1)$ then k will enlarge the previous interval to $[j, k+m-1]$. The set of these intervals is that character effective region.

We are now ready to formally define the delta vector.

Definition 9 $D[i,j] = |\{c | j \in E_c^i\}|$

In words, the number of different characters whose effective regions encompass column j in the strip.

As mentioned in Subsection 3.3, for the $m \times m$ submatrix whose top leftmost corner is in location $[i,j]$, the number of different characters is:

$$\sum_{k=1}^{j+m-1} D[i,k].$$

This can clearly be computed in linear time for every row. Our task is to efficiently compute the delta vector D. We distinguish between constructing the delta vector for strip 1 and updating the delta vector for strip i to become the delta vector for strip $i+1$.

5 Algorithm Outline

5.1 Constructing the Delta Vector For Strip 1

The delta vector is constructed considering the effective regions of all the characters. For each interval $[j_1, j_2]$ that belongs to one of the character's effective regions, the delta vector is updated. Conceptually, there are two necessary steps per update:

1. increment the delta vector at index j_1 $(D[j_1]++)$.

2. decrement at index $j_2 + 1$ ($D[j_2 + 1] - -$).

The delta vector is the result of the above updates for all effective regions of all characters.

In reality, we will not be maintaining an effective regions data structure. Rather, the delta vector is updated as symbols are added to or deleted from the strip. The algorithm idea is to move column by column in the strip. For each column go over all the different characters. For each character recalculate the character's effective region and update the delta vector accordingly.

Efficient construction of the delta vector requires the usage and maintenance of some auxiliary variables and data structures. These auxiliary data structures are described in detail in the next section.

Algorithm outline

```
For j = 1 to n do

    For each different character in column j
        If it appears in the last m − 1 column then
            We are already in one interval of this character effective region.
            Update D - Enlarge this region.
        Else
            Update D - Create a new effective region.
        Endif
    End for each

Endfor j
```

end Algorithm

In the next section we will show that the delta vector for the first strip and all the auxiliary variables can be constructed in time $O(mn)$.

5.2 Update Delta Vector for Next Strip

The delta vector for the $(i + 1)$st strip is constructed by updating the delta vector of the ith strip. Moving to the next strip means that line i of the matrix T replaced by line $i+m$. From the columns point of view, for each column j, $j = 1 \dots n$, the old character $T[i, j]$ is replaced by a new character $T[i, j + m]$. This replacement may generate two changes to the delta vector. If the new character didn't appear in this column before and if the old character is the last instance of the character in this column (includes the new character). In these cases we have to recalculate the effective regions of the old and new characters and update the delta vector accordingly.

Algorithm outline

For $j = 1$ to n do

If new char didn't appear before in this column then

Recalculate new character effective region and update D

If old char doesn't appear any more in this column then

Recalculate old character effective region and update D

Endfor j

end Algorithm

In the next section we will show that we can build the next strip delta vector and update all its auxiliary variables in time of $O(n \log n)$.

Total Algorithm Time: There are $n - m + 1$ strips in matrix T. The delta vector for the first strip is constructed in time $O(nm)$. Subsequently, we update each of the $n - m$ remaining strips in time $O(n \log m)$. The total is, then $O(mn + (n - m)n \log m) = O(n^2 \log m)$.

6 Algorithm Details

The efficiency of our algorithm lies in the auxiliary data structures. This section presents the definition of the data structures and their maintenance. For simplicity's sake we assume a bounded fixed alphabet. We later mention how to generalize to alphabets of unbounded size.

6.1 The Auxiliary Data Structures

The data structures below are initialized for the first strip and updated as we move from strip to strip.

1. $V_{j,c}$: the number of instances of char c in column j.

   Purpose: This variable helps us keep track of whether a new character is introduced or dropped from a column as the strip is moved.

   Maximum Size: For fixed bounded alphabet $O(n)$ since there are at most a constant number of characters per column. If the alphabet is infinite, there may be n^2 different characters in T. However, in any strip there are at most $mn + 1$ different characters active. Therefore this data structure can be implemented in space $O(nm)$ with a $O(\log m)$ access time (by keeping the columns' characters in balanced search trees (see, e.g. [7]).

2. H_c: the number of columns, among the $m - 1$ columns to the left of the current active column, that character c appears in.

   Purpose: Utilized in building the first delta vector. Helps to determine whether c enlarges an existing effective interval or creates a new one.

   Maximum Size: For fixed bounded alphabet constant space and access time. For infinite alphabet bounded by $O(m^2)$ space with a $O(\log m)$ access time.

3. L_c: a list of all the columns that in which c appears.

   Purpose: Help recalculate the character effective region and update D accordingly.

   Implementation: The list is implemented in a data structure that supports: *insert, delete, predecessor, successor* and *last*. For example, for each character c, L_c may be maintained as a Red-Black tree. This enables *insert, delete, predecessor* and *successor* in time $O(\log n)$ and *get last* in constant time.

   Maximum Size: As in $V_{j,c}$ the required space is $O(n)$ for a fixed alphabet and $O(mn)$ for infinite alphabet.

6.2 Build the first Delta Vector Algorithm

The first delta vector and its auxiliary data structures can be constructed in a fairly straightforward manner in time $O(nm)$ for bounded alphabets. Simply go down the columns of the strip starting from the leftmost column. For every symbol, check if it is the first appearance in the last size m window, in which case the appropriate variables are updated. For repeating occurrences, update H, V, and L appropriately.

A pseudocode of the algorithm appears below.

Algorithm for First Strip

```
Initialize H, V, L and D.

For j=1 to n do /* For each column of the first strip */

    For i=1 to m do /* For each character in current column */
        char = T[i,j]
        /* If first instance of this char in the last m columns of strip */
        If H_char == 0 do
            D[j]++ /* char is effective from the first instance column */
        /* Elseif first instance of char in curr column (but not in the last m columns */
        Elseif V_{j,char} == 0 do
            col = last(L_char) /* col of prev instance of char */
            D[col+m+1]++ /* This char will be effective only after */
            /* effectiveness of prev instance */
        Endif
        V_{j,char}++
        /* If first instance of char in curr column */
      i. if V_{j,char} == 1 do
            D[j+m+1]-- /* char is effective until column j + m */
            H_char++
            insert(L_char, j)
        endif
    Endfor i
    /* Update H */
    If j > m do
        col = j - m - 1
        For i = 1 to m do /* For each character in column j - m - 1 */
            V_{col, T[col,i]}--
            If V_{col,T[col,i]} == 0 do H_{T[col,i]}--
      i. endfor i
    endif

Endfor j
```

end Algorithm

6.3 Updating the Delta Vector

Equipped with $V_{j,c}$, L_c and D of the ith strip, we are ready to update those variables for the $(i+1)$st strip. Moving to the next strip means that line i is dropped and line $i+m$ is added. From the columns point of view, for each j, $j = 1 \ldots n$, the character $T[i,j]$ is dropped and the character $T[i, j+m]$ is added. If the new character does not appear in its column yet then we have to update D with the character effective region. The update is performed by first, inserting the index j to L_{new_c}, the column list of the new character. Second, calculating the new effective region considering the previous column and next column of the new character appearance. We will deal with the old character in a similar way. If it is the only instance of this character in the column (includes the new character) then we have to delete the index j from L_{old_c} and update D considering the previous column and next column of the old character appearance.

D is updated using V and L in the following manner. Let $T[i, j+m] = c$ and assume $V_{j,c} \neq 0$. Clearly we must increment $V_{j,c}$ by 1. Nothing else needs to be changed, since there is no new effect by $T[i, j+m] = c$.

However, if $V_{j,c} = 0$ then the introduction of c in column j needs to be noted. $V_{j,c}$ is incremented by 1, and j is inserted into L_c. We need to compute the effects on D.

These effects are dependent on the index immediately preceding j in list L_c, denoted by p, and the index immediately following j in L_c, denoted by s. We will say that index k is *far from* index ℓ if they are not within the boundaries of the same submatrix, i.e. $|k - \ell| \geq m$.

The following four cases are possible.

1. j is far from both p and s. This means that we need to add a new interval $[j, j+m-1]$ to the effective region of c. The implementation of this action is incrementing $D[j]$ by 1 and decrementing $D[j+m-1]$ by 1.
2. j is far from p but not far from s. This means that the interval including p ends before j, thus is unchanged. But the interval starting at s now needs to start at j. Thus we decrement $D[s]$ by 1 and increment $D[j]$ by 1.
3. j is far from s but not far from p. The closeness to p means that the interval starting at p will not end at $p+m-1$ but rather will extend to $j+m-1$. This is implemented by incrementing $D[p+m-1]$ and decrementing $D[j+m-1]$. However, the interval starting at s is unchanged.
4. If j is not far from either p or s there are two possibilities.
   (a) p and s are themselves not far from each other. In this case nothing needs to be done since j does not change the effective region interval at all.
   (b) p and s are far from each other. This means that the interval of p ends at $p+m-1$ which is before the start of the interval that starts at s. Adding c at j unifies those intervals into a single interval that includes also the entire area between p and s. Thus we increment $D[p+m-1]$ and decrement $D[s]$.

The above cases precisely define the necessary updates to D resulting from the addition of $T[i+m, j]$. The changes to D resulting from the deletion of $T[i, j]$ are complementary. Simply exchange every "increment" and "decrement" in the above cases.

Using the cases described above to encode "update D", the pseudocode for updating D at a change of strips is:

Algorithm for Updating D at Change of Strip

```
For j = 1 to n do /* For each column of the strip */
    /* Add T[i + m, j] */
    char = T[i + m, j]
    V_{j,char} ++
    If V_{j,char} == 1 do /* First instance of char in this column */
        insert(L_char, j)
        prev_col = predecessor(L_char, j)
        next_col = successor(L_char, j)
        Update D
    endif
    /* Remove T[i, j] */
    char = T[i, j]
    V_{j,char} --
    If V_{j,char} == 0 do /* Only instance of char in this column */
        delete(L_char, j) /* L_char must have column j */
        prev_col = predecessor(L_char, j)
        next_col = successor(L_char, j)
        Update D
    endif

endfor j
```

end Algorithm

Time for updating D at strip change: This algorithm goes over all the columns and for each column it calculates the functions *insert, delete, predecessor* and *successor* a constant number of times. These functions can be implemented in time $O(\log n)$ and this makes the total time for updating D at every strip change $O(n \log n)$

6.4 Reducing the $\log n$ to $\log m$

Note that the height of a strip is m, thus the $\log n$ factor is a result of the length of the strip. If the strip lengths were $2m$ then our time would be $O(n \log m)$. However, we can easily assume that the strip lengths are $2m$ because of the following observation that is commonly used in pattern matching papers (e.g. [3, 2]).

Observation 1 *The reason that assuming that $n \leq 2m$ does not limit generality is that if the submatrices character count problem can be solved in time $f(m)$ for strips of length $n \leq 2m$, then it can be solved in time $\frac{n}{m} f(m)$ for any n-length strip. Simply divide the text into $2\frac{n}{2m}$ overlapping $2m$-length segments and solve the submatrices character count problem separately for each. Clearly, all locations are covered.*

6.5 Unbounded Alphabet Size

Note that our time of $O(n \log m)$ for update of D after strip change is necessary even for bounded alphabet because of our need to maintain the L list's operations. However, the V structure can be maintained in linear time. An infinite alphabet size will cause maintaining the V data structure to also cost $\log m$ per operation, however, this will not change the total algorithm's time.

Acknowledgments:

The authors thank S. Muthukrishnan for pointing out the work on color range queries.

References

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

[2] A. Amir, Y. Aumann, G. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. *Proc. 38th IEEE FOCS*, pages 144–153, 1997.

[3] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of half-rectangular figures. *Information and Computation*, 118(1):1–11, April 1995.

[4] A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49:111–115, 1994.

[5] T.J. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comp.*, 7:533–541, 1978.

[6] R.S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 1992.

[8] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley Publishing Company, 1990.

[9] J. Hafner, H. S. Sawhney, W. Equitz, M. Flickner, and W. Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE Trans. Pattern Anal. Machine Intell.*, 17(7):729–736, 1995.

[10] R. Janardan and M. A. Lopez. Generalized intersection searching problems. *nternat. J. Comput. Geom. Appl.*, 3:39–69, 1993.

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

[12] A. C. C. Yao. Near-optimal time-space tradeoff for element distinctness. *Proc. 29th IEEE Annual Symposium on Foundations of Computer Science*, pages 91–97, 1988.