An Alphabet Independent Approach to Two Dimensional Matching*

Amihood Amir† Gary Benson‡ Martin Farach§
Georgia Tech Univ. of Maryland DIMACS

Abstract

There are many solutions to the *string matching problem* which are strictly linear in the input size and *independent of alphabet size*. Furthermore, the model of computation for these algorithms is very weak: they allow only simple arithmetic and comparisons of equality between characters of the input. In contrast, algorithm for two dimensional matching have needed stronger models of computation, most notably assuming a totally ordered alphabet. The fastest algorithms for two dimensional matching have therefore had a logarithmic dependence on the alphabet size. In the worst case, this gives an algorithm that runs in $O(n^2 \log m)$ with $O(m^2 \log m)$ preprocessing.

We show an algorithm for two dimensional matching with an $O(n^2)$ text scanning phase. Furthermore, the text scan requires no special assumptions about the alphabet, i.e. it runs on the same model as the standard linear time string matching algorithm. The pattern preprocessing requires an ordered alphabet and runs with the same alphabet dependency as the previously known algorithms.

Key Words: multidimensional matching, period, string

AMS(MOS) subject classifications: 68Q05, 68Q20, 68Q25

Abreviated title: Two Dimensional Matching

---

*The results presented in this paper appeared in the proceedings of the 24_{th} Symposium on Theory of Computing [3].

†College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280; (404) 853-0083; amir@cc.gatech.edu; Partially supported by NSF grant IRI-90-13055.

‡Dept. of Computer Science, University of Maryland, College Park, MD 20742; (301) 405-2715; benson@cs.umd.edu; Partially supported by NSF grant IRI-90-13055.

§DIMACS, Box 1179, Rutgers University, Piscataway, NJ 08855; (908) 932-5928; farach@dimacs.rutgers.edu; Supported by DIMACS under NSF contract STC-88-09648.

1 Introduction

The classical *string matching problem* has as its input a *text* string T of length n and a *pattern* string P of length m. The elements in the text and pattern are taken from an alphabet set Σ and σ is the number of distinct characters in the pattern. The output is all text locations i where there is a character-by-character match with the pattern, i.e. $T[i+j-1] = P[j],\ j = 1, ..., m$.

String matching is one of the most widely studied problems in computer science [12]. Fischer and Paterson [11] gave a convolutions based solution of time complexity $O(n \log m \log \sigma)$ word operations ($O(n \log m \log\log m \log \sigma)$ bit operations). Karp, Miller and Rosenberg [15] gave a parallelizable label doubling algorithm with complexity $O(n \log m)$. Knuth, Morris and Pratt [17] gave the first linear-time solution. A heuristically improved algorithm was presented by Boyer and Moore[9]. Galil and Seiferas [13] showed a real time algorithm using a constant number of registers. The Knuth, Morris and Pratt, and Galil and Seiferas algorithms have time complexity $O(n)$, are alphabet independent and use a weak model of computation where only equality of symbols is tested.

Karp and Rabin [16] came up with a *randomized* linear time algorithm in a stronger arithmetic model. They generate a large random prime number as well as use arithmetic operations (e.g. multiplication, modulo) on the characters. Vishkin [22] introduced a deterministic sampling scheme that allowed using the "signature" idea in a deterministic weak model.

In recent years there has been growing interest in multidimensional pattern matching, largely motivated by problems in low-level image processing [21]. Various algorithms exist for the *exact two dimensional matching* problem. The exact two dimensional matching problem is defined similarly to the string matching problem but the text and pattern are rectangular matrices rather than strings. For simplicity's sake we assume that T is an $n \times n$ matrix and P is an $m \times m$ matrix, although our results apply to rectangular matrices as well.

Baker [7] and, independently, Bird [8] used the Aho and Corasick [1] dictionary matching algorithm to obtain a $O(n^2 \log \sigma)$ algorithm for the exact two dimensional matching problem. Their model requires a totally ordered alphabet (since it uses the Aho and Corasick algorithm as a subroutine), and so the time is dependent on the alphabet size. For an unbounded alphabet, their algorithm's time is $O(n^2 \log m)$. Two other algorithms for exact two dimensional matching appear in [6] and [4]. They both use subword trees and run in time $O(n^2 \log \sigma)$. Note that while these algorithms require no arithmetic operations on the characters, they all assume a total ordering on their alphabets and make order comparisons in addition to checking equality of characters. A convolutions based method was suggested by Amir and Landau [5]. There the time is $O(\sigma n^2 \log m)$ word operations in an arithmetic model (or also $O(n^2 \sqrt{m} \log m)$). The Karp and Rabin algorithm also generalizes to two dimensions but, as we noted before, it is a randomized algorithm with a relatively powerful arithmetic model.

In this paper we present what is, to our knowledge, the first deterministic algorithm for two dimensional exact matching where the text scanning is alphabet independent and thus truly linear. Moreover, our algorithm is comparison based using a weak model of computation. During the text scan, the only character comparisons made are of the equality type, thus the model is weaker than in the above mentioned two dimensional matching algorithms. As opposed to previous algorithms, our algorithm is inherently two dimensional, and uses a novel technique in two dimensional matching - *two dimensional periodicity.*

The two-dimensional periodicity idea: A periodic pattern contains locations, other than the origin, where the pattern can be superimposed on itself without mismatch. Suppose our pattern is *non periodic*, i.e. there are no such locations, other than the origin. We could then narrow down the number of *potential candidates* for a pattern appearance in the text in a fashion that insures that all such candidates are "sufficiently far" from each other. Verification of a candidate could then be done in the naive character-by-character comparison, but the time would still be linear because the candidates do not overlap.

The problem with implementing this idea is that there is no guarantee that the pattern is non periodic. Indeed it has been shown [2] that there are four different types of two dimensional periodicity and that a pattern may contain many locations where it can superimpose on itself without mismatch. Moreover, it is not possible to subdivide all patterns into non-periodic subunits, as is the case with one dimensional strings. In this paper, we make use of the very strong property that superimposable patterns can not disagree in the area of overlap, and we present a new method for exploiting the pattern's periodicity.

In contrast, previous algorithms have all, to a lesser or greater degree, shared a common weakness. They all treat a matrix as a *set* of rows, rather than as a sequence of rows. That is, they only consider periodicity one dimension at a time. Thus, while exploiting periodicity *within* rows, information about periodicity *amongst* rows is disregarded. The extra log factor can be seen as a way to recompute information which was discarded in earlier stages of the algorithm. Our unified approach to two dimensional periodicity allows us to use all periodicity information throughout the text scanning algorithm.

Our algorithm consists of a *pattern analysis* stage and a *text scanning* stage. In the pattern analysis we construct a *WITNESS* array that allows a constant time decision of whether two overlapping pattern appearances conflict. This stage is done in time $O(m^2 \log \sigma)$. The text scanning stage has two phases, the *compatibility phase* and the *verification phase*.

We begin by assuming that the pattern could occur anywhere in the text. In the compatibility phase we eliminate candidate locations until all remaining candidates agree on the expected text characters. We are left with potential candidates that are all *compatible* with each other. In the verification phase we verify which of these potential candidates are indeed a match. The entire text scanning stage is done in time $O(n^2)$.

The paper is organized as follows. The pattern analysis is described in section 2. Section 3 consists of the text scan. We conclude with some open problems.

2 Pattern Preprocessing

The idea of array overlap or *periodicity* and the pattern preprocessing algorithm are given in [2]. For completeness, we review the algorithm here. Our goal is to determine where two copies of an array A can overlap without conflict. Such sites are called *sources* (figure 1). For each location that is not a source, there exists a *witness* that proves that the overlapping copies of A mismatch.

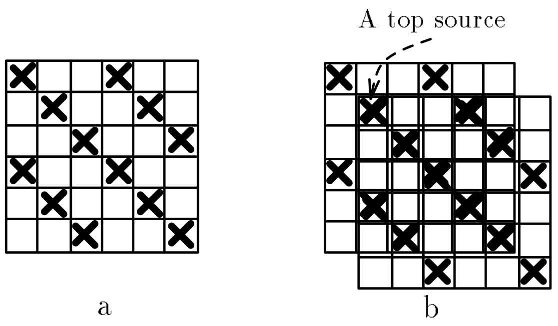

Figure 1: a) An array A b) A overlaps itself without a mismatch.

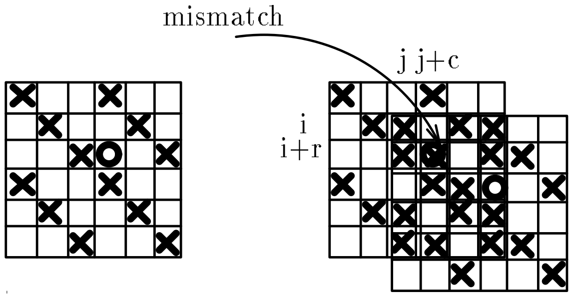

Figure 2: The witness table gives the location of a mismatch (if one exists) for two overlapping copies of the pattern. Here TOP-WITNESS$[i, j] = (r + 1, c + 1)$.

Given two copies of an $m \times m$ array $A[1 \ldots m, 1 \ldots m]$ one directly on top of the other, the two copies are said to be *in register* when all of the corresponding elements in the area of overlap contain the same symbol. Clearly, A is in register with itself when $A[1, 1]$ is aligned with $A[1, 1]$. If we can slide the upper copy over the lower copy to a point where the copies are again in register, then at least one of the corner elements $A[1, 1]$ or $A[m, 1]$ in one copy overlaps an element of the other copy. If the overlapping corner is $A[1, 1]$ then we have a *top source*. Otherwise, we have a *bottom source*.

We want to fill out two WITNESS arrays. For each location $A[i, j]$, TOP-WITNESS$[i, j] = (m + 1, m + 1)$ if A is in register with itself when element $A[1, 1]$ overlaps element $A[i, j]$. Otherwise, TOP-WITNESS$[i, j] = (r, c)$ where (r, c) identifies some mismatch. Specifically $A[r, c] \neq A[i + r - 1, j + c - 1]$ (figure 2). BOTTOM-WITNESS$[i, j]$ is filled out similarly except element $A[m, 1]$ overlaps element $A[i, j]$.

The Pattern Preprocessing Algorithm

Our pattern preprocessing algorithm (Algorithm A) makes use of two algorithms (Algorithms 1 and 2) from [19] which are themselves variations of the KMP algorithm [17] for string matching.

Algorithm 1 takes as input a pattern string w of length m and builds a table *lppattern*$[1,\ldots,m]$ where *lppattern*$[i]$ is the length of the longest prefix of w starting at w_i. Algorithm 2 takes as input a text string t of length n and the table produced by Algorithm 1 and produces a table *lptext*$[1..n]$ where *lptext*$[i]$ is the length of the longest prefix of w starting at t_i.

The idea behind Algorithm A is the following: We convert the two-dimensional problem into a problem on strings (figure 3). Let the array A be processed column by column and suppose we are processing column j. Assume we can convert the block $A[1..m, j..m]$ into a string $T_j = t_1 \ldots t_m$ where t_i represents the suffix of row i starting in column j. This will serve as the text string. Assume also that we can convert the block $A[1..m, 1..m-j]$ into a string $W_j = w_1 \ldots w_m$ where w_i represents the prefix of row i of length $m-j$. This will serve as the pattern string. Now, use Algorithm 1 to produce the table *lppattern* for W_j and Algorithm 2 to produce the table *lptext* for T_j. If the longest prefix of the pattern in the text starting at t_i runs through the last row of the text (*lptext*$[i] = m - i$), then $A[i,j]$ is a source. If the longest prefix stops before the last row (*lptext*$[i] < m - i$), then there is a mismatch between the prefix of row *lptext*$[i]$ and the suffix of row $i +$ *lptext*$[i]$. We need merely locate the mismatch to obtain the witness. In order to treat the suffix and prefix of a row as a single character, we will build a *suffix tree* for the array.

A suffix tree is a compacted trie of the suffixes of a string ([20, 23]). The suffix tree is perhaps the most widely used data stucture in string matching. A thorough description of suffix trees and their properties appears in [10]. We note that since a suffix tree is a trie, each node v has associated with it some string $S(v)$. In [18], it was pointed out that if l is the Least Common Ancestor (LCA) of two nodes v and w, then $S(l)$ is the longest common prefix of $S(v)$ and $S(w)$. In [14], an algorithm was given which preprocesses a tree in linear time and answers LCA queries in constant time. Thus a suffix tree, in conjuction with LCA queries, is a powerful tool for comparing the substrings of a string.

Algorithm A *For building witness array*

Step A.1: Build a suffix tree by concatenating the rows of the array. Preprocess the suffix tree for least common ancestor queries in order to answer questions about the length of the common prefix of any two suffixes.

Step A.2: For each column j, fill out TOP-WITNESS for column j:

Step A.2.1: Use Algorithm 1 to construct the table *lppattern* for $W_j = w_1 \ldots w_m$. Character w_i is the *prefix* of row i of length $m - j$. We can answer questions about the equality of two characters by consulting the suffix tree. If the common prefix of the two characters is at least $m - j$ then the characters are equal.

Step A.2.2: Use Algorithm 2 to construct the table *lptext* for $T_j = t_1 \ldots t_m$. Character t_i is the *suffix* of row i starting in column j (also of length $m - j$). Again we test for equality by reference to the suffix tree.

Step A.2.3: For each row i, if *lptext*$[i] = m - i$ then we have found a source and TOP-WITNESS$[i,j] = (m+1, m+1)$ otherwise, using the suffix tree, compare the suffix of row $i +$ *lptext*$[i]$ starting in column j with the prefix of row *lptext*$[i]$. The length l of the common prefix will be less than $m - j$, and TOP-WITNESS$[i,j] = ($*lptext*$[i], l+1)$.

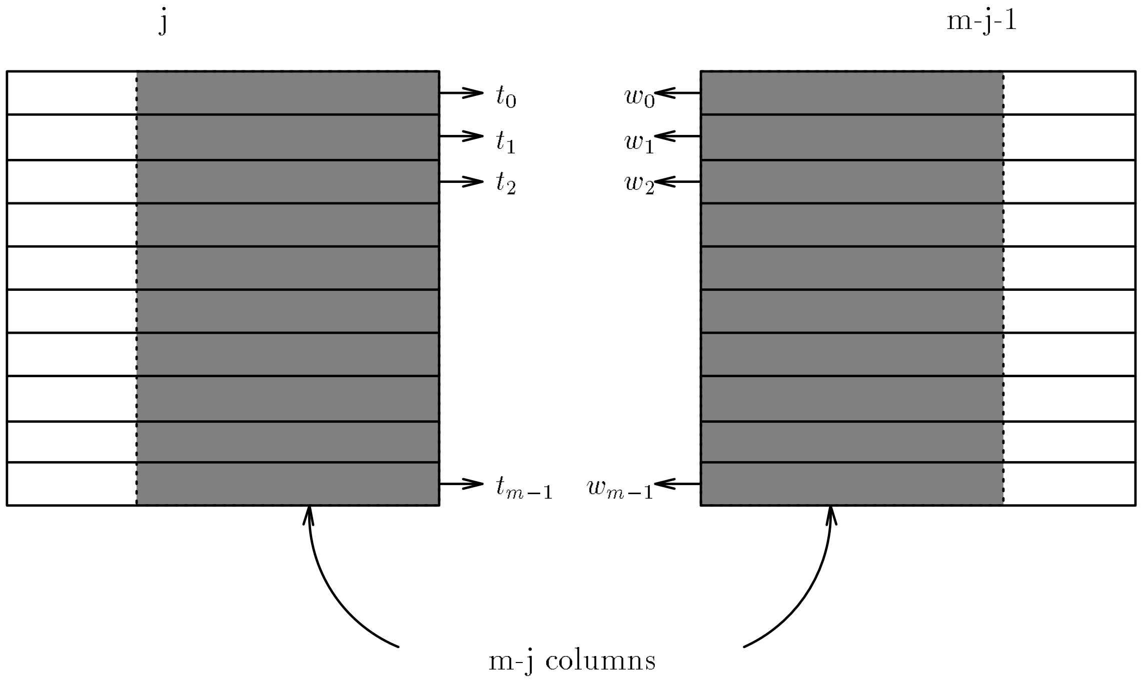

Figure 3: Representing a block of the array by a string. For the preprocessing algorithm, $T_j = t_1 \ldots t_m$ is the text and $W_j = w_1 \ldots w_m$ is the pattern.

Step A.3: Repeat step 2 for BOTTOM-WITNESS by building the automatons and processing the columns from the bottom up.

Theorem 1 *Algorithm A runs in time* $O(m^2 \log \sigma)$.

The suffix tree construction [23] takes time $O(m^2 \log \sigma)$ while the preprocessing for least common ancestor queries [14] can be done in time linear in the size of the array. Queries to the suffix tree are processed in constant time. The tables *lppattern* and *lptext* can be constructed in time $O(m)$ [19]. For each of m columns, we construct two tables so the total time for steps 2 and 3 is $O(m^2)$. The total complexity of the pattern preprocessing is therefore $O(m^2 \log \sigma)$. $\square$

3 Text Processing

Text processing is accomplished in two stages: Candidate Consistency and Candidate Verification. A *candidate* is a location in the text where the pattern may occur. We denote a candidate starting at text location $T[r,c]$ by (r,c). We say that two candidates (r,c) and (x,y) are *consistent* if they expect the same text characters in their region of overlap (two candidates with no overlap are trivially consistent). In terms of witnesses, the candidates are consistent if the witness array indicates no witness, i.e. if $r \leq x$ and $c \leq y$ then $TOP-WITNESS[x-r+1, y-c+1] = (m+1, m+1)$. If $r > x$ and $c \leq y$ then $BOTTOM-WITNESS[x-r+1, y-c+1] = (m+1, m+1)$. consistent if they have no witness. We use the shorthand $(r,c) \doteq (x,y)$ to mean that the candidates (r,c) and (x,y) are consistent. If the two candidates are inconsistent, then we write $(r,c) \bowtie (x,y)$.

Initially, we have no information about the text and therefore all text locations are candidates. However, not all text locations are consistent. During the candidate consistency phase, we eliminate candidates until all remaining candidates are pairwise consistent. During the candidate verification phase, we check the candidates against the text to see which candidates represent actual occurrences of patterns. We exploit the consistency of the surviving candidates to rule out large sets of candidates with single text comparisons (since all consistent candidates expect the same text character).

3.1 Candidate Consistency

As stated above, the goal of the *candidate consistency algorithm* presented in this subsection is to produce a set of candidates for the given text such that the candidates are all consistent.

We begin with some transitivity lemmas for the $\doteq$ relation.

Lemma 1 *For any* $1 \leq r_1 \leq r_2 \leq r_3 \leq n$ *and for any* $1 \leq c_1 \leq c_2 \leq c_3 \leq n$, *if* $(r_1, c_1) \doteq (r_2, c_2)$ *and* $(r_2, c_2) \doteq (r_3, c_3)$, *then* $(r_1, c_1) \doteq (r_3, c_3)$.

Proof: Suppose that $(r_1, c_1) \bowtie (r_3, c_3)$. Then, there exists an $x \leq m - r_3 + r_1$ and a $y \leq m - c_3 + c_1$ such that $P[x,y] \neq P[x+r_3-r_1, y+c_3-c_1]$. But $r_3 \geq r_2$ so $x+r_3 \geq r_2$ and $m \geq x+r_3-r_1 \geq r_2-r_1$. Similarly, $m \geq y+c_3-c_1 \geq c_2-c_1$. Since $(r_1, c_1) \doteq (r_2, c_2)$, we have that $P[x+r_3-r_1, y+c_3-c_1] =$

$P[x+r_3-r_2, y+c_3-c_2]$. A similar argument shows that $P[x,y] = P[x+r_3-r_2, y+c_3-c_2]$ since $(r_3,c_3) \doteq (r_2,c_2)$. We conclude that $P[x,y] = P[x+r_3-r_1, y+c_3-c_1]$. This is a contradiction. Therefore $(r_3,c_3) \doteq (r_1,c_1)$. $\square$

Lemma 2 *For any* $1 \le r_1 \le r_2 \le r_3 \le n$ *and for any* $1 \le c_3 \le c_2 \le c_1 \le n$, *if* $(r_1,c_1) \doteq (r_2,c_2)$ *and* $(r_2,c_2) \doteq (r_3,c_3)$, *then* $(r_1,c_1) \doteq (r_3,c_3)$.

Proof: The proof is analogous to that of Lemma 1.$\square$

A one dimensional consistency algorithm

Let c be some column of the text. Initially, all positions in this column are candidates. We would like to remove candidates until all candidates within the column are consistent. Further, we would like to preserve any candidate which might actually represent an occurrence of the pattern in the text. Thus, we will only remove candidates when we find some specific text location with which they mismatch. The idea of algorithm B is the following. Suppose we have eliminated inconsistent candidates from the last i rows of column c. The surviving candidates are placed on a list. Notice that by lemma 1, if the candidate in row $n-i$ is consistent with the top candidate on the list, it is consistent with all of them. This check takes constant time using the witness array. This principle is used to produce an $O(n)$ algorithm for column consistency.

Algorithm B *Eliminate inconsistent candidates within a column*

Step B.1: Get column number, c.

Step B.2: We 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 B.3: For row $r = n-1$ to 1 do:

Step B.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:

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

If the two candidates under consideration are consistent, then they need not be compared with any other candidates on S. This is because, by lemma 1, consistency within a single column is transitive.

- 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 B.3.1 with the new top candidate in S. If no candidates remain in S, add (r,c) to S.

Clearly, if the two candidates are inconsistent, they can't both match the text. Thus the inappropriate one is eliminated.

Step B.4.3: Return S.

Theorem 2 *Algorithm B is correct and runs in time $O(n)$.*

Proof: The correctness of the algorithm follows largely from the comments within the algorithm and from lemma 1.

For the complexity bound, note that S can be initialized in constant time. For each row r in the for loop, there is at most one successful test of consistency. For each unsuccessful test, a candidate is eliminated, either the candidate (r, c) or the top candidate in S. Since the number of candidates is bounded by n the total time is $O(n)$. $\Box$

A two dimensional consistency algorithm

We use the above algorithm as an initial "weeding out" of candidates so that we get a list for each column of consistent candidates. In the two dimensional consistency algorithm, we start with the rightmost column, which we know to be consistent, and add one column at a time from right to left. We will maintain the following loop invariant:

$$P(i) \equiv \text{the candidates remaining in columns } i, \ldots, n \text{ are all pairwise consistent.}$$

As noted above, by calling Algorithm B with value n we are assured of $P(n)$. The approach of the algorithm below is to quickly insure $P(i)$ once $P(i+1)$ is known. When $P(1)$ holds, we are done. We use a similar idea to that of algorithm B. We first have a phase were we make sure that each candidate is consistent with all candidates above and to the right. A symmetric phase makes sure that candidates below and to the right are consistent, thus assuring $P(i)$. To reduce the work, we note that during the first phase, we need only compare a candidate on column i with the leftmost surviving candidate in each row above it. To further reduce the work, once a candidate in column i is found to be consistent with candidates above it, all lower candidates in column i are also consistent (see figure 4).

Algorithm C *Candidate Consistency*

Step C.1: For $i \leftarrow 1$ to n do $C_i \leftarrow$ Call Algo B(i)

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

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

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

Add one row at a time, making sure that it is consistent with all candidates added so far.

Step C.4.1: Call Bottom-Up(i)

Make sure that all candidates in column i are consistent with all candidates below them in columns $i+1, \ldots, m$.

Step C.4.2: Call Top-Down(i)

Make sure that all candidates in column i are consistent with all candidates above them in columns $i+1, \ldots, m$.

Step C.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 C1 *Bottom-Up(c)*

Step C1.1: Initialize: cur gets bottom value from C_c. $row \leftarrow n$ is a pointer to the last row compared so far.

Step C1.2: While not at the top of C_c do

Step C1.2.1: If cur is consistent with leftmost item on R_{row}, then $row \leftarrow row - 1$.

We compare the current candidate with the leftmost candidate in some row row below it. If they are consistent, then by lemma 1, all candidates above cur on C_c are also consistent with all candidates on R_{row}, *even if cur is later deleted as inconsistent with another candidate.* We need not consider that row again.

Step C1.2.2: 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 item above cur on C_c, otherwise do nothing during this traversal of loop.

We remove the candidate that has a mismatch against the text. If the item in R_{row} is removed, then we still need to check if cur is consistent with the remaining candidates in that row. Thus, we don't need to update any pointers. Otherwise, if cur is removed, we move up in C_c. We don't need to change row because of the comment above. None of the rows below row need to be compared against the new candidate cur since we already know they are consistent.

Step C1.2.3: If the row counter points to a row above cur's row, set cur to the next candidate above cur in C_c.

Theorem 3 *The Algorithm C is correct and runs in $O(n^2)$.*

Proof: As in algorithm B, no candidate is removed unless a mismatch is found against the text. Therefore, no valid candidates are removed.

To show that at the end of the algorithm, only mutually compatible candidates are left on the R_i lists (and on the C_i), we pick two arbitrary surviving candidates (r_1, c_1) and (r_2, c_2) such that $c_1 < c_2$. We have two cases:

Case $r_1 \leq r_2$**:** We show this case by induction. Suppose that after processing column $c_1 + 1$ that $P(c_1 + 1)$ holds. The base case is true by Theorem 2. Let (r_2, c') be the leftmost candidate such that $c' > c_1$ and c' appears on R_{r_2} after processing column c_1. By lemma 1, we need only show that $(r_1, c_1) \doteq (r_2, c')$ since $(r_2, c') \doteq (r_2, c_2)$.

Let (r', c_1) be the last candidate with which (r_2, c') was compared during $BottomUp(c_1)$.

Claim 3.1 *$r' \geq r_1$ and $(r', c_1) \doteq (r_2, c')$.*

Proof: Suppose that $(r', c_1) \bowtie (r_2, c')$. Then we either delete (r', c_1) or (r_2, c') from the candidate list. If we remove (r_2, c') from the list, then we would compare the next candidate on R_{r_2} with (r', c_1), thus violating the assumption that (r_2, c') was the leftmost candidate compared with a c_1 candidate. If we remove (r', c_1), the we would compare (r_2, c') with the next candidate above (r', c_1), thus violating the assumption that (r', c_1) was the last candidate on column c_1 with (r_2, c') was compared.

To show that $r' \geq r_1$ we observe that if $r_1 > r'$, then we couldn't have compared (r_2, c') with (r', c_1) without first comparing (r_1, c_1) with (r_2, c'). Since they both survived, they would have had to have been compatible. But then we never would have compared (r_2, c') with (r', c_1) at all. $\Box$

Finally, we know that $(r_1, c_1) \doteq (r', c_1)$, $(r', c_1) \doteq (r_2, c')$, $(r_2, c') \doteq (r_2, c_2)$ and that $r_1 \leq r' \leq r_2$ and that $c_1 \leq c' \leq c_2$. So by lemma 1, we have proved the case.

Case $r_1 > r_2$**:** This case is very similar to the one above, however, we refer the reader to procedure *TopDown* rather than *BottomUp* and lemma 2 rather than lemma 1.

The argument that shows the running time to be $O(n^2)$ is similar to the complexity analysis in Theorem 2. We observe that during *BottomUp* (and *TopDown*) in each comparison of candidates results in the removal of a candidate (which can only happen n^2 times in all calls to these procedures), or in the *cur* pointer being decremented (resp. incremented). This can only happen $O(n)$ time each time *BottomUp* (resp. *TopDown*) is called, and they are each called $O(n)$ times. Therefore the complexity is $O(n^2)$. $\Box$

3.2 Candidate Verification

All remaining candidates are now mutually consistent. Each text element $t = T[r, c]$ may be contained by several candidates, the *relevant* candidates. However, compatible candidates that share the same text element must agree on the expected character in that element. This leads to the following crucial observation: Every element 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:

Algorithm D *Candidate Verification*

Step D.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. We will later see how this step is implemented (procedure D1).

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

This flagging is done by the same method as in step D.1.

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

Our only remaining task is to show how to mark the text elements with the appropriate pattern coordinate pairs. We adopt the popular sports fans technique - *the 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 size of the pattern, 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 wave follow:

Procedure D1 *The Wave*

Step D1.1: **Initialization:** Mark every candidate with $\langle 1,1\rangle$.

Step D1.2: **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 $i < m$ then assign to $T[r,c]$ the pair $\langle i+1,j\rangle$.

Step D1.3: **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 $j < m$ 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. The propagation is bottom-up in the columns and then right-left in the rows.

Theorem 4 *Algorithm D is correct and runs in time $O(n^2)$.*

Correctness: The only non-trivial fact is that the wave correctly marks all elements. We need the following terminology. Let (r, c) be a candidate containing position $T[r+i, c+j]$. Then j is the *column distance* between $T[r+i, c+j]$ and (r, c) and i is the *row distance* between $T[r+i, c+j]$ and (r, c). The *column-close* sources containing location $T[r, c]$ are the sources whose column distance to $T[r, c]$ is minimal. The *closest* source containing location $T[r, c]$ is the column-close source whose row distance to $T[r, c]$ is smallest.

Claim: The pattern coordinate pair marked by procedure D1 in location $T[r, c]$ is the pair $\langle i, j\rangle$ where $(r - i + 1, c - j + 1)$ is the closest source to $T[r, c]$.

Proof: By induction on the column distance of the closest source. For column distance 0 the column wave assures that the marked pair is the distance to the closest source (+1). Assuming that for every text element whose column distance to its closest source is d, the marked pair is the distance to the closest source, it is easy to see that the row wave will ensure correct marking of all elements with column distance $d + 1$ to the source.

Time: Each of the steps of algorithm D is easily implementable in time $O(n^2)$. Note that in each of steps D.1 and D.4 is single call to procedure D1, which clearly takes $O(n^2)$ time. $\square$

4 Conclusion

While string matching is extremely well studied and understood, multidimensional matching has been somewhat neglected. This neglect does not stem from lack of practical motivation but may be attributed to the fact that string matching techniques do not easily generalize to higher dimensions.

We feel that an inherently multidimensional approach is likely to produce better results. This paper is a step along the way. All four different previously known algorithms for exact two dimensional matching pushed string matching techniques as tools for solving the two dimensional case. However, none succeeded in achieving results similar to the string matching case. Our new idea of analysing periodicity in two dimensions turned out useful in improving results of the most basic two dimensional task - that of exact matching.

References

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

[2] A. Amir and G. Benson. Two-dimensional periodicity and its application. *Proc. of 3rd Symposium on Discrete Algorithms, Orlando, FL*, Jan 1992.

[3] A. Amir, G. Benson, and M. Farach. Alphabet independent two dimensional matching. *Proc. 24th ACM Symposium on Theory of Computation*, 1992.

[4] A. Amir and M. Farach. Two dimensional dictionary matching. Manuscript, 1991.

[5] A. Amir and G. Landau. Fast parallel and serial multidimensional approximate array matching. *Theoretical Computer Science*, 81:97–115, 1991.

[6] A. Amir, G.M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *Proceedings of First Symposium on Discrete Algorithms, San Fransisco, CA*, 1990.

[7] 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.

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

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

[10] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, chapter 12, pages 97–107. NATO ASI Series F: Computer and System Sciences, 1985.

[11] M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.

[12] Z. Galil. Open problems in stringology. In Z. Galil A. Apostolico, editor, *Combinatorial Algorithms on Words*, volume 12, pages 1–8. NATO ASI Series F, 1985.

[13] Z. Galil and J.I. Seiferas. Time-space-optimal string matching. *J. Computer and System Science*, 26:280–294, 1983.

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

[15] R. Karp, R. Miller, and A. Rosenberg. Rapid identification of repeated patterns in strings, arrays and trees. *Symposium on the Theory of Computing*, 4:125–136, 1972.

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

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

[18] G.M. Landau and U. Vishkin. Efficient string matching in the presence of errors. *Proc. 26th IEEE FOCS*, pages 126–126, 1985.

[19] M.G. Main and R.J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. of Algorithms*, pages 422–432, 1984.

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

[21] A. Rosenfeld and A.C. Kak. *Digital Picture Processing*. Academic Press, New York, 1982.

[22] U. Vishkin. Deterministic sampling - a new technique for fast pattern matching. *SIAM J. Comp.*, 20:303–314, 1991.

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

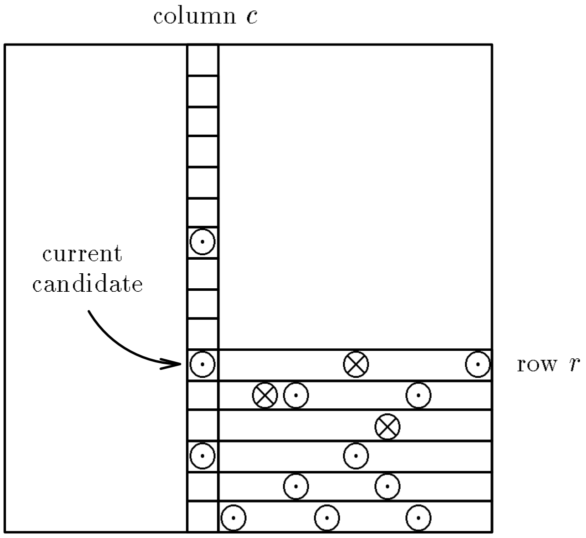

Figure 4: In the bottom up scan, the current candidate in column c need only be tested against the leftmost candidates (marked by $\otimes$) in rows $r + m \ldots r$ *which have not already been tested by candidates below* c.