Two Dimensional Matching

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 a pair \( (i, j) \) with \( j < m \) then assign to \( T[r, c] \) the pair \( (i, j + 1) \).

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 from right to left within the rows.

**Theorem 3.5.** 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. If \( (r, c) \) is a candidate, we refer to the candidate origin \( T[r, c] \) as a source. Let \( (r, c) \) be a candidate containing position \( T[r + i, c + j] \). Then \( j \) is the column distance and \( i \) is the row distance between \( T[r + i, c + j] \) and the source for \( (r, c) \). The column-close candidates containing location \( T[r, c] \) have sources whose column distance to \( T[r, c] \) is minimal. The closest candidate containing location \( T[r, c] \) is the column-close candidate whose source has smallest row distance to \( T[r, c] \).

**Claim:** The pattern coordinate pair marked by procedure D1 in location \( T[r, c] \) is the pair \( (i, j) \) 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 \( (i, 1) \) where \( i \) is the row 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 correct, it is easy to see that the row wave will ensure correct marking of all elements with column distance \( d + 1 \) to the closest 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, there is a 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 previously known algorithms for exact two dimensional matching used string matching tools to solve the two dimensional case. However, some succeeded in achieving results similar to the string matching case. Our new idea of analyzing periodicity in two dimensions has been useful in improving results of the most basic two dimensional task - that of exact matching.

**References**


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 Bottom-Up (resp. Top-Down) is called, and they are each called \( O(n) \) times. Therefore the complexity is \( O(n^2) \). □

### 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 in 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 \( (i, j) \), where \( (i, j) \) are the coordinates of the pattern element \( P[i, j] \) with which \( T[r, c] \) should be compared.

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 \( (i, j) \) 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 fan’s 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 origin with \((1, 1)\).

**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 \((i, j)\) with \( i < m \) then assign to \( T[r, c] \) the pair \((i + 1, j)\).
also consistent with all candidates on $R_{\text{row}}$, even if $\text{cur}$ is later deleted as inconsistent with another candidate. We need not consider that row again.

**Step C1.2.2:** If $\text{cur}$ is not consistent with leftmost item on $R_{\text{row}}$, then find a witness to their inconsistency. Check which of them has a mismatch against the text. If the leftmost item on $R_{\text{row}}$ has a mismatch, remove that candidate from its list. If $\text{cur}$ has a mismatch, set $\text{cur}$ to the next item above $\text{cur}$ on $C_c$.

We remove the candidate that has a mismatch against the text. If the item in $R_{\text{row}}$ is removed, then we still need to check if $\text{cur}$ is consistent with the remaining candidates in that row. Thus, we don’t need to update any pointers. Otherwise, if $\text{cur}$ 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 $\text{cur}$ 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 3.4.** 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 consistent candidates are left on the $R_c$ lists (and on the $C_c$), 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 3.3. Let $(r_2, c')$ be the leftmost candidate such that $c' > c_1$ and $c'$ appears on $R_{\text{row}}$ after processing column $c_1$. By lemma 3.1, we need only show that $(r_1, c_1) \sim (r_2, c')$ since $(r_2, c') \sim (r_2, c_2)$.

Let $(r', c_1)$ be the last candidate with which $(r_2, c')$ was compared during $\text{Bottom} - \text{Up}(c_1)$.

**Claim 3.4.1.** $r' \geq r_1$ and $(r', c_1) \sim (r_2, c')$.

**Proof:** Suppose that $(r', c_1) \not\sim (r_2, c')$. 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 $R_{\text{row}}$ 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 $(r', c_1)$, then 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 which $(r_2, c')$ 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 consistent. But then we never would have compared $(r_2, c')$ with $(r', c_1)$ at all. □

Finally, we know that $(r_1, c_1) \sim (r', c_1), (r', c_1) \sim (r_2, c'), (r_2, c') \sim (r_2, c_2)$ and that $r_1 \leq r' \leq r_2$ and that $c_1 \leq c' \leq c_2$. So by lemma 3.1, 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 Top-Down rather than Bottom-Up and lemma 3.2 rather than lemma 3.1.

The argument that shows the running time to be $O(n^2)$ is similar to the complexity analysis in Theorem 3.3. We observe that during Bottom-Up (and Top-Down) in each comparison of candidates results in the removal of a candidate (which can
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 $i ← 1$ to $n - m + 1$ do $C_i ←$ Call Algo B($i$)

Step C.2: For $i ← 1$ to $n - m + 1$ do initialize $R_i$ to be an empty list of candidates for each row $i$.

Step C.3: Put the candidates on $C_{n-m+1}$ onto their appropriate $R_i$ lists.

Step C.4: For $i ← n - m$ down to $1$ 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 $i + 1, \ldots, n$.

**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 $i + 1, \ldots, n$.

**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 $C_c$. $row ← n - m + 1$ 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 $R_{row}$ or $R_{row}$ is empty, then $row ← row - 1$.

We compare the current candidate with the leftmost candidate in some row $row$ below it. If they are consistent, then by lemma 3.1, all candidates above $cur$ on $C_c$ are
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 3.1, 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 \( (n - m + 1, c) \) to the top of \( S \).

**Step B.3:** For row \( r = n - m \) 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 array:

* If \( (r, c) \sim (x, c) \), then add \( (r, c) \) to the top of \( S \).

**Step B.3.2:** 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 3.1, consistency within a single column is transitive.

* If \( (r, c) \not\sim (x, c) \), then use the witness character in the text to eliminate at least one of the candidates. If \( (x, c) \) is eliminated, remove it from \( S \) and if \( (r, c) \) is not eliminated, 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:** Return \( S \).

**Theorem 3.3.** 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 3.1.

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 \( O(n) \).

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:

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

No candidates can occur in columns \( n - m + 2, \ldots, n \) so \( P(n - m + 2), \ldots P(n) \) are trivially satisfied. As noted above, by calling Algorithm B with value \( n - m + 1 \) we are assured of \( P(n - m + 1) \). 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
and \( \text{TOP-Witness}[i,j] = (l\text{p}_i[l], l+1) \).

Step A.3: Repeat step 2 for \( \text{BOTTOM-Witness} \) by building the automata
and processing the columns from the bottom up.

**Theorem 2.1.** Algorithm A runs in time \( O(m^2 \log \sigma) \).

The suffix tree construction [25] takes time \( O(m^2 \log \sigma) \) while the preprocessing
for least common ancestor queries [16] can be done in time linear in the size of the
array. Queries to the suffix tree are processed in constant time. The tables \( \text{lppattern} \)
and \( \text{lptext} \) can be constructed in time \( O(m) \) [21]. 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 \( O(m^2 \log \sigma) \).

### 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 with origin 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, two candidates are consistent if they have
no witness, i.e. if \( r \leq x \) and \( c \leq y \) then \( \text{TOP-Witness}[x-r+1, y-c+1] = (m+1, m+1) \).
We use the shorthand \( (r,c) \sim (x,y) \) to mean that the candidates
\( (r,c) \) and \( (x,y) \) are consistent. If the two candidates are inconsistent, then we write
\( (r,c) \not\sim (x,y) \).

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 \( \sim \) relation.

**Lemma 3.1.** 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
\( (r_1,c_1) \sim (r_2,c_2) \) and \( (r_2,c_2) \sim (r_3,c_3) \), then \( (r_1,c_1) \sim (r_3,c_3) \).

**Proof:** Suppose that \( (r_1,c_1) \not\sim (r_3,c_3) \). 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
\( x+r_3 \geq x+r_2 \) 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 \( (r_1,c_1) \sim (r_2,c_2) \), 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 \( (r_3,c_3) \sim (r_2,c_2) \). We conclude that \( P[x,y] = P[x+r_3-r_1, y+c_3-c_1] \). This is a contradiction. Therefore
\( (r_3,c_3) \sim (r_1,c_1) \). \( \Box \)

**Lemma 3.2.** 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
\( (r_1,c_1) \sim (r_2,c_2) \) and \( (r_2,c_2) \not\sim (r_3,c_3) \), then \( (r_1,c_1) \not\sim (r_3,c_3) \).

**Proof:** The proof is analogous to that of Lemma 3.1. \( \Box \)

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
The witness array is a powerful tool for comparing the substrings of a string. For each column \( j \), fill out the top-witness for column \( j \):

- 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 the top-witness for column \( j \):
  
  1. **Step A.2.1**: Use Algorithm 1 to construct the table \( lptext \) for \( T_j = t_1 \ldots t_m \). Character \( t_i \) 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 length of the common prefix of the two characters is at least \( m - j \) then the characters are equal.
  
  2. **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 \( \text{TOP-WITNESS}[i, j] = (m + 1, m + 1) \) 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 \( m - j \).
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] ≠ 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 [21] which are themselves variations of the KMP algorithm [19] for string matching. Algorithm 1 takes as input a pattern string w of length m and builds a table Ip(pattern)[1, ..., m] where Ip(pattern)[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 Itext[1..n] where Itext[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 Ip(pattern) for W_j and Algorithm 2 to produce the table Itext 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 (Itext[i] = m - i), then A[i, j] is a source. If the longest prefix stops before the last row (Itext[i] < m - i), then there is a mismatch between the prefix of row Itext[i] and the suffix of row i + Itext[i]. We need merely locate the
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, 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 $O(m^2 \log \sigma)$, $O(m^2 \log m)$ in the worst case, and assumes an ordered alphabet.
Note that very recently, there have been several advances in two dimensional string
matching. In [14] and independently in [3], it was shown how to compute a witness
table in $O(m^2)$ using the unordered alphabet model of computation.

The text scanning stage has two phases, the compatibility phase and the verifica-
tion 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 can-
didates 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.

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.

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-
AN ALPHABET INDEPENDENT APPROACH TO TWO Dimensional Pattern Matching

AMIHOD AMIR*, GARY BENSON†, AND MARTIN FARACH‡

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, algorithms 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

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 $\Sigma$ and $\sigma_P$ is the number of distinct characters in pattern $P$, so in particular, $\sigma \leq \min(|\Sigma|, m)$. We will in general drop the subscript $P$ and simply refer to $\sigma$. The output is all text locations $i$ where there is a character-by-character match with the pattern, i.e. $T[i+j-1] = P[j]$, $j = 1, \ldots, m$.

String matching is one of the most widely studied problems in computer science [13]. Fischer and Paterson [12] 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 [17] gave a parallelizable label doubling algorithm with complexity $O(n \log m)$. Knuth, Morris and Pratt [19] gave the first linear-time solution. A heuristically improved algorithm was presented by Boyer and Moore[10]. Galil and Seiferas [15] 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 [18] devised 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 [24] 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 [23]. 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 [8] and, independently, Bird [9] 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.