10
Two Dimensional Matching

String matching is a basic theoretical problem in computer science, but has been useful in implementating various text editing tasks. The explosion of multimedia requires an appropriate generalization of string matching to higher dimensions. The first natural generalization is that of seeking the occurrences of a pattern in a text where both pattern and text are *rectangles*.

The last few years saw a tremendous activity in two dimensional pattern matching algorithms. We naturally had to limit the amount of information that entered this chapter. We chose to concentrate on serial deterministic algorithms for some of the basic issues of two dimensional matching.

Throughout this chapter we define our problems in terms of *squares* rather than *rectangles*, however, all results presented easily generalize to rectangles.

10.1 Exact Matching

The *Exact Two Dimensional Matching Problem* is defined as follows:

INPUT: *Text* array $T[n \times n]$ and *pattern* array $P[m \times m]$.
OUTPUT: All locations $[i, j]$ in T where there is an *occurrence* of P, i.e. $T[i+k, j+l] = P[k+1, l+1]$ $0 \leq k, l \leq n-1$.

10.1.1 LINEAR REDUCTIONS

A natural way of solving any generalized problem is by reducing it to a special case whose solution is known. It is therefore not surprising that most solutions to the two dimensional exact matching problem use exact string matching algorithms in one way or another. In this section, we present an algorithm for two dimensional matching which relies on reducing a matrix of characters into a one dimensional array.

Let $P'[1 \ldots m]$ be a pattern which is derived from P by setting $P'[i] = P[i,1]P[i,2] \cdots P[i,m]$, that is, the i^{th} character of P' is the i^{th} row of P. Let $T_i[1 \ldots n-m+1]$, for $1 \leq i \leq n$, be a set of arrays such that $T_i[j] = T[i,j]T[i,j+1] \cdots T[i,j+m-1]$. Clearly, P occurs at $T[i,j]$ iff P' occurs at $T_i[j]$.

We can now use any linear time string matching algorithm to find, for each i, the occurrences of P' in T_i. We have, in fact, *linearized* the matrix, that is, we have taken an inherently two dimensional problem and made it into a set of one dimensional problems.

However, string matching algorithms assume that characters can be compared in constant time. A straightforward implementation of the algorithm suggested above would require $O(m)$ to compare two "characters" since the characters of P' and the T_i's are really sequences of m characters from the input alphabet Σ. Thus for each i, it takes $O(nm)$ to find occurrences of P' in T_i, making the total time for two dimensional matching $O(n^2m)$. This is clearly an improvement over the naive $O(n^2m^2)$ algorithm, but not good enough. The obvious bottleneck involves comparing pattern rows with length m text subrows. Below is a solution.

An Automaton Based Method Suppose that we take each pattern row and find all its occurrences in the text rows. We now set $T_i[j] = k$ if k is the smallest number such that the k^{th} row of P matches the subrow starting at $T[i,j]$. Similarly, we set $P'[i] = k$ if k is the smallest number such that the i^{th} row of P equals the k^{th} row of P. After such a preprocessing, we can then make constant time "comparisons" between characters of P' and T_i. This second phase takes linear time. It takes $O(n^2)$ time to find occurrences of pattern rows in the text during the preprocessing. There are m such rows, so the complexity is once again $O(n^2m)$.

Note however, that we can treat the pattern rows as a set of pattern strings. Such a set of pattern strings is called a *dictionary.*

The *Dictionary Matching Problem* is the following:

Preprocess: $D = \{P_i | 1 \le i \le k, P_i = P_{i,1} \cdots P_{i,m_i}\}$

Input: Text $T[1,\ldots,n]$.

Output: For each text location i, all $j \in \{1,\ldots,k\}$ such that P_j occurs at T_i.

Let σ be the number of distinct characters that occur in D, and let $d = \sum_{i=1}^{k} |P_i|$. The classical solution, which is a generalization of the Knuth-Morris-Pratt (henceforth KMP) automaton approach, solves the dictionary matching problem in time $O(d\log\sigma)$, to preprocess the dictionary, and $O(n\log\sigma + tocc)$ to scan the text, where $tocc$, which stands for *total occurrences*, is the size of the output. Note that since all the patterns in our dictionary are the same size, the size of the output is linear. That is, no pattern is a proper prefix of another so two distinct patterns cannot both match at some location.

The details of this algorithm are beyond the scope of this chapter, however, the interested reader is referred to this chapter's conclusion for a reference.

Returning to two dimensional matching, we derive the following algorithm:

1. Let the set of distinct rows in the pattern be a dictionary (if some rows are identical, include only the lowest numbered such row in the dictionary). Construct P'. Use the algorithm for dictionary matching to construct the T_i's.
2. Use any linear time string matching algorithm to find the occurrences of P' in each T_i. For any pair (i, j) such that P' occurs at $T_i[j]$, mark an occurrence of P at $T[i, j]$.

Time: 1. $O(n^2 \log \sigma)$. 2. $O(n^2)$.
Total Time: $O(n^2 \log \sigma)$.

10.1.2 PERIODICITY ANALYSIS

The above two dimensional matching algorithm reduces the problem of two dimensional matching into one dimension. While the complexity achieved is favorable compared to the brute force method, it still contains a term dependent on the alphabet size. In contrast, there are one dimensional string matching algorithms which are linear and do not have an alphabet dependent term, so there is some gap between the complexity of one dimensional matching and two dimensional matching. This gap can be closed somewhat by exploiting the periodicity, or lack thereof, in a two dimensional array.

The periodicity of strings has been well studied and the periodic nature of strings has been used to generate efficient algorithms for string matching. By analyzing the two dimensional periodicity of the pattern and using that knowledge in the text scanning step, we not only obtain a more efficient algorithm, but we will also see that it will prove useful in compressed matching (see §10.3).

A string s can be said to be periodic if there exists a string w such that $s = w^k$ for some $k \geq 2$. This notion of periodicity generalizes nicely to two dimensions if we consider an unbounded pattern which extends in all directions. Then we can define a parallelogram of periodicity with which we can tile the plain to generate the (infinite) pattern. However, for bounded patterns, the notion does not prove as useful since it has been shown that even intuitively periodic patterns may not have such a tiling parallelogram.

In strings, we can equivalently define periodicity in terms of *borders*. A border is a proper prefix of a string which is also a suffix. A string is periodic iff it has a border which is at least half its length. While it is not clear what constitutes a prefix or suffix in a two dimensional pattern, the idea of overlapping regions of pattern can be easily generalized.

The idea of string overlap has been used to derive fast parallel string matching algorithms (see Chapter 2). The well know notion of a *witness* can also be used in the context of two dimensional matching to produce an efficient algorithm for exact matching. For a offset (i,j), the pair (k,l) is a witness if $P[k,l] \neq P[i+k,j+l]$. That is, if the pattern does not overlap with itself when offset by (i,j), then there must be some place where the two copies of the pattern disagree. Any such position of disagreement is a witness, so the witness of an offset need not be unique.

A *witness table* is an array W such that $W[i,j] = (k,l)$ if (k,l) is a witness for offset (i,j). If no such witness exists for (i,j), then $W[i,j]$ is undefined. Such a table can be constructed in time $O(m^2)$. See exercise 1 for a simple version of witness table construction. .

The witness table for the pattern provide some important understanding of its two dimensional structure. This knowledge can be used to provide an alphabet independent $O(n^2)$ text scanning phase.

Alphabet Independent Search 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* (denoted $(r,c) \doteq (x,y)$) if they expect the same text characters in their region of overlap (two candidates with no overlap are trivially consistent). Equivalently, two overlapping candidates are consistent if they have no witness.

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).

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 10.1. *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 $(r_1,c_1) \not\doteq (r_3,c_3)$. 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 $(r_3, c_3) \doteq (r_1, c_1)$. □

Lemma 10.2. *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* $(r_1, c_1) \doteq (r_3, c_3)$*.*

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 A 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 10.1, 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 A. *Eliminate inconsistent candidates within a column*

Step A.1: Get column number, c.

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

Step A.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 10.1, consistency within a single column is transitive.

- If $(r, c) \not\doteq (x, c)$ 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 A.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 A.4.3: Return S.

Theorem 10.3. *Algorithm A 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 10.1.

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.

As noted above, by calling Algorithm A 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 A. 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.

Algorithm B. *Candidate Consistency*

Step B.1: For $i \leftarrow 1$ to n $C_i \leftarrow$ Algo A(i)
Step B.2: For $i \leftarrow 1$ to n initialize R_i to be an empty list of candidates for each row i.
Step B.3: Put the candidates on C_n onto their appropriate R_i lists.
Step B.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 B.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 B.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 B.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.

Step B1. *Bottom-Up(c)*

Step **B1.1:** Initialize: cur gets bottom value from C_c. $row \leftarrow n$ is a pointer to the last row

Step **B1.2:** While not at the top of C_c do

Step **B1.2.1:** 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 10.1, 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 **B1.2.2:** 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 previous item on C_c, otherwise do nothing.

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 **B1.2.3:** If the row counter points to a row above cur's row, set cur to the previous candidate in C_c.

Theorem 10.4. *The Algorithm B is correct and runs in $O(n^2)$.*

Proof: As in algorithm A, 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 10.3. 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 10.1, 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 10.4.1. *$r' \geq r_1$ and $(r', c_1) \doteq (r_2, c')$.*

Proof: Suppose that $(r', c_1) \not\doteq (r_2, c')$. 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 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. □
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 10.1, 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 10.2 rather than lemma 10.1.
The argument that shows the running time to be $O(n^2)$ is similar to the complexity analysis in Theorem 10.3. 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 $O(n^2)$. □

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

The candidate verification algorithm follows:

Algorithm C. *Candidate Verification*

> Step C.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 C1).

> Step C.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 C.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 C.1.

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

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

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:

Step C1. *The Wave*

> Step **C1.1: Initialization:** Mark every candidate with $\langle 1,1\rangle$.

> Step **C1.2: 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 **C1.3: 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.

Theorem 10.5. *Algorithm C 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 C1 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, the row wave will ensure correct marking of all element with column distance $d+1$ to the source.

Time: Each of the steps of algorithm C is easily implementable in time $O(n^2)$. Note that in each of steps C.1 and C.4 is single call to procedure C1, which clearly takes $O(n^2)$ time. □

10.2 Scaled Matching

An interesting extension of exact matching is exact *scaled matching*. Such a problem is motivated by the fact that one may be interested in matching patterns whose occurrence in the text is of different scale than provided by the pattern. For example, if the text is a newspaper that we would like to scan, then we encounter letters of the alphabet in various sizes.

A "clean" version of the problem may be defined as follows:

The string $aa...a$ where the symbol a is repeated k times (to be denoted a^k), is referred to as *'a' scaled to k*. Similarly, consider a string $A=a_1\cdots a_l$. *'A' scaled to k* (A^k) is the string $a_1^k,...,a_l^k$.

Let $P[m \times m]$ be a two-dimensional matrix over a finite alphabet Σ. Then P *scaled to* k (P^k) is the $km \times km$ matrix where every symbol $P[i,j]$ of P is replaced by a $k \times k$ matrix whose elements all equal the symbol in $P[i,j]$. More precisely,

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

The problem of *two-dimensional pattern matching with scaling* is defined as follows:

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

OUTPUT: all locations in T where an occurrence of P scaled to k starts, henceforth a k-*occurence*, for any $k = 1, ..., \lfloor \frac{n}{m} \rfloor$.

The basic algorithmic design strategy described below can be viewed as realizing the following approach: For each scale k, try to select only a fraction of $\frac{1}{k}$ among the n columns and seek k-occurrences only in these columns. Since each selected column intersects n rows, this leads to consideration of $O(\frac{n^2}{k})$ elements. Summing over all scales, we get $O(n^2)$ multiplied by the harmonic sum $\sum_{i=1}^{\frac{n}{m}} \frac{1}{i}$, whose limit is $\log \frac{n}{m}$, making the total number of elements scanned $O(n^2 \log \frac{n}{m})$.

A final intuitive step is to select also a $\frac{1}{k}$ fraction of the rows. Since $\sum_{i=1}^{\frac{n}{m}} \frac{1}{i^2}$ is bounded by a constant, the number of elements decreases now to

$$O(n^2 \sum_{i=1}^{\frac{n}{m}} \frac{1}{i^2}) = O(n^2).$$

We will show here how to achieve the $O(n^2 \log \frac{n}{m})$ bound and hint at the method used to eliminate the logarithmic factor.

10.2.1 STRING SCALED MATCHING

Let us consider the scaled matching problem in one dimension. The solution to this problem will shed light on one aspect of the two dimensional scaled matching problem.

The problem of *one-dimensional string matching with scaling* is the following:

INPUT: Pattern $P = p_1 \cdots p_m$ and text $T = t_1 \cdots t_n$ where $n > m$.

OUTPUT: All positions in T where a k-occurrence of P starts, for any $k = 1, ..., \lfloor \frac{n}{m} \rfloor$.

We can use the inherent repetitions in scaled appearances of strings to provide a simple scaled matching algorithm.

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

We denote by $S^{\Sigma} = \sigma_1'\sigma_2'\cdots\sigma_{\hat{n}}'$, the *symbol part* of S, and by $S^{\#}$, the vector of natural numbers $r_1, r_2, ..., r_{\hat{n}}$, the *repetition part* of S.

Example 1: For $S = AAABABBCCACAAAA$, $S' = A^3B^1A^1B^2C^2A^1C^1A^4$,

$S^{\Sigma} = ABABCACA$ and $S^{\#} = [3, 1, 1, 2, 2, 1, 1, 4]$.

Algorithm D. *Algorithm for the one dimensional scaled matching problem*

> Step D.1: Derive the symbol string T^{Σ} and the repetition string $T^{\#}(= t_1, ..., t_{\hat{n}})$ from the text string T. Similarly, derive P^{Σ} and $P^{\#}(= p_1, \ldots, p_{\hat{m}})$ from the pattern string P.

Observation 1. Finding all occurrences of P in T scaled to k is equivalent to finding all locations i that satisfy conditions A and B below.

Condition A. There is an exact occurrence of P^{Σ} in location i of T^{Σ}.

Condition B.1. $t_i \geq kp_1$

Condition B.2. $t_{i+1} = kp_2, \ldots, t_{i+\hat{m}-2} = kp_{\hat{m}-1}$

Condition B.3. $t_{i+\hat{m}-1} \geq kp_{\hat{m}}$

> Step D.2: Suppose $\hat{m} \geq 3$. Derive the *quotient* string $T' = \frac{t_2}{t_1}, \frac{t_3}{t_2}, \ldots, \frac{t_{\hat{n}}}{t_{\hat{n}-1}}$ from $T^{\#}$ and the quotient string $P' = \frac{p_3}{p_2}, \frac{p_4}{p_3}, \ldots, \frac{p_{\hat{m}-1}}{p_{\hat{m}-2}}$ from $P^{\#}$.

Observation 2. Suppose $\hat{m} \geq 3$. Condition B.2 from Observation 1 is satisfied for $k = \frac{t_2}{p_2}$ if and only if an occurrence of string P' starts at location $i + 1$ of string T'.

> Step D.3.1: Find all occurrences of string P^{Σ} in the string T^{Σ} and all occurrences of the string P' in the string T'.

This is done by applying any linear time string matching algorithm.

> Step D.3.2: For each location i in T^{Σ}, such that P^{Σ} starts at i and P' starts at location $i + 1$ of T', check whether conditions B.1 and B.3 extend to locations i and $i + \hat{m} - 1$ in T'.

This will take $O(1)$ time per location i.

Comment: Extension to the case where $\hat{m} < 3$ within the same complexity bound as claimed below is trivial.

Time: Step D.1 needs $O(|P| + |T|)$ time. All other steps need $O(|P^{\Sigma}| + |T^{\Sigma}|)$ time. If the input is already provided in the run-length form, then the running time of the algorithm will be linear in the length of the compressed input, and possibly sublinear in $|P| + |T|$.

10.2.2 TWO DIMENSIONAL SCALED MATCHING

The main idea behind the $O(n^2 \log \frac{n}{m})$ algorithm is similar to the string matching algorithm presented above. Consider any column j and scale k. If P^k (the pattern scaled to k) starts in column j then every row of the pattern is translated into a *k-block* of k consecutive equal rows where each row is of length km. The maximum number of possible blocks in a column is $\frac{n}{k}$. Suppose we succeed in dividing column j into b_j^k k-blocks in time $O(b_j^k)(\leq O(\frac{n}{k}))$. Finding all appearances of P^k in column j can now be done in time $O(b_j^k)$ by running a KMP-like algorithm where every pattern row is compared only to the first row in the appropriate block (constant time using LCA on suffix trees) and verifying that the number of rows in the block indeed matches the expected number (comparison of numbers, also constant time).

Thus the total time for finding all appearances of P^k is $O(\frac{n^2}{k})$ and the total time for all scales is $O(\sum_{k=1}^{\lfloor \frac{n}{m} \rfloor} \frac{n^2}{k}) = O(n^2 \log \frac{n}{m})$.

We now present a detailed algorithm for scaled pattern matching, assuming that the block division can be done efficiently (we will later show how the block division is done).

Let us first formally define a k-block.

k-blocks: Let k be a positive integer. A *k-block* at *position i_1 of column j_1* of T is a submatrix $T' = T[i_1, ..., i_2; j_1, ..., j_1 + km - 1]$, that satisfies the following:

(1) all rows of T' are equal.

(2) no rows can be added to T' without disrupting condition (1) above (formally, substring $T[i_2; j_1, ..., j_1+km-1]$ is not equal to substring $T[i_2+1; j_1, ..., j_1 + km - 1]$ and substring $T[i_1; j_1, ..., j_1 + km - 1]$ is not equal to substring $T[i_1 - 1; j_1, ..., j_1 + km - 1]$).

The *height* of a submatrix is its number of rows. The height of such k-block is $i_2 - i_1 + 1$. If a k-block is contained in a k-occurrence then its height must be at least k. (Because each pattern row must appear k successive times.)

Let S_j^k be a list whose elements are lists of consecutive k-blocks in column j. The k-blocks in each list are adjacent and in consecutive order. The last k-block in a list does not have an adjacent k-block below it. The first k-block below it starts the next list. Every list entry is a *starting row* of a k-block and its *height*. The sum of the heights of all blocks in a list is no less than km.

Algorithm E. *Scaled Matching Algorithm*

Pattern analysis: Let $P_1, P_2, ..., P_m$ be the rows of the pattern matrix P. P can be described as row P_{l_1} repeating r_1 times (the block at row l_1), followed by row $P_{l_2}(\neq P_{l_1})$ repeating r_2 times, followed by additional blocks until finally we have row $P_{l_\alpha}(\neq P_{l_{\alpha-1}})$ repeating r_α times, where $l_1 = 1$, $l_k = r_1 + \cdots + r_{k-1} + 1$ and $r_1 + r_2 + \cdots + r_\alpha = m$. The above description is a run-length representation of the sequence $P_1, P_2, \ldots, P_m$ when viewed as a one dimensional string R with m symbols. Formally a pair (P_{l_h}, r_h) is a block of r_h rows at row l_h.

> Step E.1: Compress the string R into its run length representation, which is denoted FR.

(This will take time proportional to $|P_1| + |P_2| + ... + |P_m|$.)

In the remainder of this presentation we assume that $\alpha > 2$. We leave it to the interested reader to see why there is a solution within the claimed bound for the case $\alpha \leq 2$. Let $\hat{FR}$ be the subsequence of FR that starts at its second element and ends at the predecessor of the last element. That is, $\hat{FR} = (P_{l_2}, r_2)...(P_{l_{\alpha-1}}, r_{\alpha-1})$.

> Step E.2: Compute the failure array $FAIL$ for string $\hat{FR}$.

This step is done precisely as in the KMP algorithm with one exception. Comparison of two pairs (P_{l_i}, r_i) and (P_{l_j}, r_j) is done by comparing P_{l_i} and P_{l_j} (using the suffix tree of the concatenated pattern rows, as in Chapter 4) and by comparing the numbers r_i and r_j.

Scanning the text: Scan each *column* j of T separately, as if it was a one-dimensional string. The scan is done in a similar way to KMP, with a few modifications. Most notable is the fact that instead of advancing from a row i to its successive row $i + 1$ in column j, our advancement is guided by the list S_j^k. The other modification is due to the fact that, as in the pattern preprocessing, comparisons between pattern and text symbols are now comparisons between blocks.

Input for Step E.3: Pattern automaton for $\hat{FR}$, lists S_j^k (recall that the elements of S_j^k are lists of consecutive k-blocks).

Output of Step E.3: All indices of T where a copy of rows $P_{l_2}^k ... P_{l_{\alpha-1}}^k$ of P^k begins.

In other words, we are ignoring the first and last blocks of P. This is done because all blocks except the first and last are sharply delineated in the text, but the first and last blocks may appear in a text location that has leading and trailing rows equal to the rows in the first and last pattern blocks. Thus, after finding the appearances of P^k without the first and last blocks, we will verify that the beginning and end also match (step E.4). Step E.3 is a modified version of the main loop of KMP. We provide somewhat more detailed psuedocode so emphasize the similarity.

> Step E.3:

```
for column col := 1 to n − km + 1 do
   l = start of first list in S^k_col
   While l is still a list in S^k_col do
   row := l (start of first block in list); h := 2
   last := start of last block in list
   While  row ≤ last do
      if COMPARE(k-block at T[row, col], block at row kl_h of P^k)
      then row := row + kr_h (start of next block in S^k_col);
      h := h + 1
         if h = α
         then
            Output: there is a match at [row − k(m − l_α) + 1, col] ;
            h := FAIL(h)
      else h := FAIL(h)
   end { while}
   l = next list in S^k_col
   end { while}
end { for}
```

Step E.4: For every occurrence of $\hat{F}R$ check whether it extends into an occurrence of the whole pattern

.

Time: Assuming that comparisons can be done in constant time, step E.3 is basically the Knuth-Morris-Pratt algorithm. Its time for column *col* is $O(|S^k_{col}|) \leq O(\frac{n}{k})$. The time for all n columns is $O(\frac{n^2}{k})$. Step E.4 takes constant time per occurrence so its total contribution is also $O(\frac{n^2}{k})$.

We now show how the constant time comparison is achieved.

Step E1. *Compare*

$INPUT$ (1) Block at $T[i, j]$. (The algorithm guarantees that row i is in the list S^k_j.)

(2) The k-block at row kl_h of P^k (represented by the pair (P_{l_h}, r_h)).

$OUTPUT$ Determine in $O(1)$ time whether $T[i; j, j+1, ..., j+km-1] = P_{l_h}$ and the height of the S^k_j block is r_h.

The main tool in comparing the strings $T[i; j, j+1, ..., j+km-1]$ and $P^k_{l_h}$ is an LCA query with respect to the suffix tree ST.

We need to make the suffix tree ST more explicit.

Constructing Suffix Tree ST:

Step **E1.1:** Form a long string C as follows. Concatenate all rows of T and append to them the following $\lfloor \frac{n}{m} \rfloor$ strings:

* A concatenation of the rows of P.

* A concatenation of the rows of P, where each row is scaled to 2.
* For k, $3 \leq k \leq \lfloor \frac{n}{m} \rfloor$, a concatenation of the rows of P, where each row is scaled to k.

The length of string C is equal to

$$|T| + \sum_{j=1}^{m} \sum_{k=1}^{\lfloor \frac{n}{m} \rfloor} |(P_j^k)|$$

which is $O(n^2)$, since

$$|T| = O(n^2)$$

and

$$\sum_{j=1}^{m} \sum_{k=1}^{\lfloor \frac{n}{m} \rfloor} |(P_j^k)| \leq m \sum_{k=1}^{\lfloor \frac{n}{m} \rfloor} km \leq m^2 (\frac{n}{m})^2 = O(n^2)$$

Time: The initial preprocessing of ST is $O(n^2 \log \sigma)$, for effective alphabet size σ. Initial preprocessing for LCA takes also $O(n^2)$ time. Subsequently, every query (i.e. comparison) is constant time.

Our only remaining task is constructing the lists S_j^k.

10.2.3 PARTITIONING A COLUMN INTO K-BLOCKS

The main idea in constructing list S_j^k is the *power columns*. The algorithm's efficiency is achieved by grouping the n columns of the text into sets of m successive columns each, as follows $\{1, ..., m\}, \{m+1, ..., 2m\}, ...\{(\lfloor \frac{n}{m} \rfloor - 1)m+1, ..., \lfloor \frac{n}{m} \rfloor m\}$ (possibly followed by a final set of less than m elements $\{\lfloor \frac{n}{m} \rfloor m+1, ..., n\}$). Instead of handling each of the columns separately, the effort for processing all m columns in a set is combined. The key player in each such set is its rightmost column, called *power column*. Columns $m, 2m, ..., \lfloor \frac{n}{m} \rfloor m$ are the *power columns*.

We now need an additional data structure that enables retrieval of the following queries: *Given a power column c and another column $j \geq c$ (*alternatively, $j < c$*), find all rows $1 \leq i \leq n-1$ such that rows i and $i+1$ differ between columns c and j (*alternatively, between columns j and $c-1$*). Formally, $T[i; c, ..., j] \neq T[i+1; c, ..., j]$ (*alternatively, $T[i; j, j+1, ..., c-1] \neq T[i+1; j, j+1, ..., c-1]$*).*

We would like the time to process this query to be proportional to the length of the output (that is, the number of such rows i). The following data structure will enable such processing.

Let $[i, c]$ be a position on a power column c in T. Let $B_r[i, c]$ be the largest integer k for which the two strings $T[i; c, c+1, ..., c+k-1]$ and $T[i+1; c, c+1, ..., c+k-1]$ are equal. Similarly, let $B_l[i, c]$ be the largest integer k for which the two strings $T[i; c-k, c-(k-1), ..., c-1]$ and

$T[i+1; c-k, c-(k-1), ..., c-1]$ are equal. In words, $B_r[i,c]$ gives the longest common prefix of rows i and $i+1$ starting at column c and $B_l[i,c]$ gives the longest common suffix of rows i and $i+1$ ending at column $c-1$.

It is easy to see that $B_r[i,c]$ and $B_l[i,c]$ can be constructed for every row i and power column c in time totalling $O(n^2)$.

Example 2: Let row i in T be $a^3b^4c^5a^2b^5$ and row $i+1$ be $c^2a^1b^4c^1a^1c^3b^1a^1b^5$. Suppose columns 5,10 and 15 are power columns. The figure below illustrates rows i and $i+1$ in T, as well as the power columns:

$$\begin{array}{ccccccccccccccccccc}
a & a & a & b & b & b & b & c & c & c & c & c & a & a & b & b & b & b & b \\
 & & & & | & & & & & | & & & & & | & & & & \\
c & c & a & b & b & b & b & c & a & c & c & c & b & a & b & b & b & b & b
\end{array}$$

Then $B_r[i,5]=4$, $B_r[i,10]=3$, $B_r[i,15]=5$ and $B_l[i,5]=2$, $B_l[i,10]=0$, $B_l[i,15]=1$.

For a fixed column j and scale k, let c be the power column immediately to the right of j. Consider the arrays $B_r[1,...,n-1;c]$ and $B_l[1,...,n-1;c]$. Every i where $B_l[i,c] < c-j+1$ or $B_r[i,c] < km-(c-j+1)$ indicates a start of a new block. We call such a row a *seam*. We can find the seams using *range minimum queries*.

Definition: Let $L=[l_1,...,l_n]$ be an array of n numbers.

A *Range Minimum* query is of the form:

Input: Given a range of indices $[i,...,j]$, where $1 \le i \le j \le n$,
Output: Return an index k $i \le k \le j$ such that $l_k = \min\{l_i,...,l_j\}$.

Following a linear time preprocessing of L, each range minimum query can be processed in $O(1)$ time.

The only difficulty is that we want to find all the k-blocks in time $O(\frac{n}{k})$. Direct use of range minimum queries will find **all** blocks, even the small ones. The time then may exceed the allowed $O(\frac{n}{k})$.

However, since we are only interested in **consecutive** k-blocks, we are assured that, except for the first and last block, all k-blocks have only one seam every k rows. In other words, there are k seamless rows in each side of a seam. This observation leads to the following algorithm.

Algorithm F. *Algorithm for constructing S_j^k*

Step F.1: Divide column j into $\frac{n}{2k}$ intervals of $\frac{k}{2}$ consecutive rows each.

Step F.2: For each interval, do 3 range minimum queries, to find a seam and possibly another seam to its right or left within the interval. If two seams or more are found in an interval it is discarded.

Step F.3: Every single seam interval, followed by non-seam intervals and eventually another single seam interval (with at least k rows between the two seams), defines the beginning of a k-block. Its height is the number of rows between the seams.

Step F.4: Discard all S_j^k entries where the total height of consecutive k-blocks is less than $k(l_{\alpha-1} - l_2)$.

Time: Since there is only a constant amount of work per interval, the time is $O(\frac{n}{k})$ for every column, or $O(\frac{n^2}{k})$ for the entire text.

Total Algorithm Time: We have seen that the time for searching all occurrences of P^k is $O(\frac{n^2}{k})$. Since k ranges over $\{1, \ldots, \lfloor \frac{n}{m} \rfloor\}$ the total time is

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

Remarks: Note that for scale k, even a list of consecutive k-blocks do not necessarily indicate a possible appearance of P^k. In particular, character changes within rows force upon us the knowledge that a scaled occurrence can appear only within multiples of k from that change (because scale k means that every pattern row now consists of k-length strings, each consisting of a single repeating character). Thus for every scale k it is only necessary to construct the S_j^k for $\frac{n}{k}$ **columns**. This brings the total algorithm time to $O(n^2)$.

10.3 Compressed Matching

Data compression is a well-studied area in information science. The two most common motivations for the study of data compression are:

- *Data storage:* Compression allows more data to be stored on a given device.
- *Data Communications:* Communication links, be they cables, phone lines or satellite channels, are both limited and, usually, slower than the data gathering and data processing ends. Compressing the data prior to transmission, and decompressing on the receiving end can increase the speed (and thus the volume) of the transmission process.

The main thrust in the study of data compression has been to achieve compression that is efficient in packing while also being practical in time and space. Compression of strings has had a long history of study, starting with the naive *run-length* encoding and encompassing such diverse algorithmic methods as the Huffman code, on-line textual substitution methods and stochastic methods. The various methods of string compression generalize to multidimensions. An example of a simple and widely used two-dimensional compression scheme is quadtrees. Other examples include arithmetic methods and textual substitution methods.

In the context of pattern matching, one would like a compression scheme to have the property of allowing pattern matching directly in the *compressed data*. We refer to the problem of finding patterns in compressed text representations as *compressed matching*. If a pattern can be found without the need to decompress the data, noticeable savings in time may arise. For example, in the communications arena, systems of small networked processors could enjoy a boost in efficiency if data could be both *transmitted* and *analysed* in a compressed form, especially if the process requires transmission and analysis of the same data in many nodes.

The *compressed matching problem* is the following: Let $\mathcal{C}$ be a given compression algorithm, let $c(D)$ be the size of the output of $\mathcal{C}(\mathcal{D})$.

INPUT: Compressed text $\mathcal{C}(\mathcal{T})$ and compressed pattern $\mathcal{C}(\mathcal{P})$.
OUTPUT: All locations in T where pattern P occurs.

A compressed matching algorithm is *optimal* if its time complexity is $O(c(T))$. It is *efficient* if for T such that $c(T) \ll |T|$ its time complexity is $o(|T|)$. This definition of *efficient* may seem somewhat strange. It is motivated by the following observation. The size of the compressed data will vary depending on the internal structure of the data. In the worst case it may be $O(|T|)$. Any algorithm that is not optimal would then be worse than first decompressing the text and then matching. However, we implicitely assume that compression is beneficial, i.e. $c(D) \ll |D|$. This leads to our definition of efficient compressed matching. As a further characterization of efficient, we call *almost optimal* any efficient algorithm with complexity $O(c(T)\text{polylog}c(T))$.

In one dimension (strings), an optimal compressed matching algorithm is readily apparent for the run-length compression as defined in section 10.2. Such an algorithm is given by the scaled string matching algorithm by simply considering the 1-occurences in the output of that algorithm.

The run-length compression can be simply generalized to two dimensions by concatenating the run-length compression of all the matrix rows. For simplicity's sake we will refer to this compression as the *two-dimensional run-length compression*, or simply the *run-length compression*.

The classification of two dimensional periodicity has proven to be a powerful tool for this problem, and allowed Amir and Benson to present a $O(c(T)\log|P| + |P|)$ time algorithm for compressed matching in the two-dimensional run-length compression.

For didactic reasons we make some simplifying assumptions and only show this result for the easier, non-periodic case. The more general cases have a similar flavor but need to contend with greater details.

10.3.1 PROBLEM DEFINITION AND ALGORITHM OVERVIEW

The *two-dimensional run-length compressed matching problem* is formally defined as follows:

INPUT: Two-dimensional run-length compressed text matrix T; Compressed two-dimensional pattern matrix P.
OUTPUT: All locations in T where P occurs.

We want the running time of our algorithm to be insensitive to the size of the output, therefore we do not allow trivial patterns. A *trivial matrix* is one in which every element contains the same character. If the text and the pattern are both trivial with the same character, then the output size is $O(|T|)$ which is not efficient for any compression scheme. All other patterns contain a *seam*, that is, an occurence of two different, adjacent characters. Since the number of seams in the text is $O(c(T))$ for run length compression, the output size is optimal. For simplicity's sake we will assume here that *every* pattern *row* contains a seam.

Below is an overview of the algorithm. As in section 10.1, the term *candidate* refers to a location in the *text* that *may* correspond to the element $P[0,0]$ of an actual occurence of the pattern. Recall that a set of candidates is *compatible* if for any two candidates in the set, the two copies of the pattern originating at the sources overlap without any mismatches.

Algorithm Overview:

Following the scheme of many pattern matching algorithms, this algorithm consists of a *pattern preprocessing* part and a *text scanning* part.

Pattern Preprocessing: As in section 10.1.2 and exercise 1, a witness table is constructed for the pattern. We will assume that the if the (uncompressed) pattern is an $m \times m$ matrix, then there is a witness for every offset (i, j) such that $-m < j < m$ and $0 \leq i \leq m/2$. In a sense, this is equivalent to the condition for aperiodicity in one dimension, in which a string is considered to be to aperiodic if its longest border is less than half the length of the string. A similar (but more complicated) algorithm to the one presented below works for cases in which such an aperiocity assumption is not warranted.

Text Scanning: Performed in three phases:

1) The **restriction phase** is a text scan in which preliminary candidates sources are selected. This phase is not needed in the uncompressed exact matching algorithm because all text locations are assumed to be candidates.

2) The **compatibility phase**, partitions the portion of the text having candidate sources into blocks of size $\frac{m}{2} \times \frac{m}{2}$. Within each block, a single candidate remains, by the aperiodicity assumption.

3) The **verification phase** is a triple scan of the text to determine which of the remaining candidates are actual occurrences of the pattern.

10.3.2 THE COMPRESSED MATCHING ALGORITHM

To simplify the description of the algorithm, we will assume that the uncompressed text is an $n \times n$ array, the pattern is an $m \times m$ array and both

n and m are powers of 2.

Algorithm G. *Text Analysis for Non-Periodic Patterns*

Step G.1: Input Compressed representations of the text and the pattern, and the output from the pattern preprocessing.
Step G.2: Restriction Phase

The goal of the restriction phase is to find potential candidate patterns in the text subject to the restriction that each candidate must contain a pattern characteristic seam (in fact, we are even assuming that each row contains a seam). Because there can be no more seams in the text than $c(T)$, this phase provides an important initial limit on the number of candidates considered in the remaining phases of the algorithm. This step scans through the text computing the source position relative to every seam. The list is then bin-sorted by columns. The remaining candidates are now stored on column lists C_i , where i corresponds to the column of the text in which the candidate source occurs. These lists will be used in the compatibility and verification phases.

Step G.3: Compatibility Phase

Here we partition the text into disjoint, occupied blocks of size $\frac{m}{2} \times \frac{m}{2}$. An *occupied* block contains a candidate source. As a result of the restriction phase, the text is partitioned into occupied blocks of size 1×1 each containing one candidate source. We initially combine vertical pairs of these blocks into blocks of size 2×1 and then combine horizontal pairs of these blocks into blocks of size 2×2. Repeating in a similar fashion for $\log m - 1$ stages, we obtain blocks of size $\frac{m}{2} \times \frac{m}{2}$. Blocks are combined vertically by scanning down one column list (steps 3.1 and 3.2) and horizontally by scanning down two adjacent column lists simultaneously (step 3.3). After each stage, the number of lists is reduced by half, until there are $\frac{2n}{m}$ lists at the end of this phase. More formally, each stage partitions the text into disjoint, occupied j-blocks. A *j-block* is a subarray

$$T[k \cdot 2^j \dots (k+1)2^j - 1, l \cdot 2^j \dots (l+1)2^j - 1], \quad k, l = 0 \dots \frac{n}{2^j} - 1$$

containing 2^j rows and columns. A *half-j-block* is a subarray

$$T[k \cdot 2^j \dots (k+1)2^j - 1, l \cdot 2^{j-1} \dots (l+1)2^{j-1} - 1],$$

$$k = 0 \dots \frac{n}{2^j} - 1, \; l = 0 \dots \frac{n}{2^{j-1}} - 1$$

containing 2^j rows and 2^{j-1} columns. A *pair of j-blocks* is two j-blocks that form a half $j+1$-block.

Step G.3: Implementation of Consistency Phase
For $j := 1$ to $\log m - 1$ do:

> Step G.3.1: For each column list $C_i,\ i = 0 \ldots \frac{n}{2^{j-1}} - 1$ scan the list from the top down. If the list contains a pair of occupied $(j-1)$-blocks, look up the witness $w = T[r, c]$ for the candidates in the $Witness$ table from the pattern preprocessing step and put onto the witness list.

The two candidates are *incompatible* and thus have a witness since two compatible sources for non-periodic patterns cannot co-exist within a block of size $\frac{m}{2} \times \frac{m}{2}$ or smaller. As the blocks are combined, we use witness locations from the pattern analysis and refer to the text to eliminate incompatible candidates. Since the text is stored in row lists due to its compressed representation, it is not possible to directly access a witness location in $O(1)$ time. So, for each stage, we first determine all the witness locations, sort them and then do a complete scan of the text to determine which character is in each witness. For each witness, we maintain a pointer to its pair of blocks.

> Step G.3.2: Radix sort the witness list. Scan the text to determine the character in each witness and determine which one or both of the candidates mismatches the text character. Eliminate the appropriate candidate(s) from list C_i.

After step 3.2, each list contains half-j-blocks.

> Step G.3.3: For each pair of lists $C_{2i}, C_{2i+1},\ i = 0 \ldots \frac{n}{2^j} - 1$, scan down both lists simultaneously. If the lists contain two horizontally adjacent occupied half j-blocks, find the witness for the candidates and put on a witness list. Repeat step 3.2 except the remaining candidates are linked onto a new list C_i.

After step 3.3, the number of lists has been reduced by half and each list contains occupied j-blocks.

Step G.4: Verification Phase

In testing the candidates against the text, we employ the following main idea:

- For each segment (run of identical characters) of the text, at most nine candidates contain the first character of the segment.

> Step G.4.1: For each segment S of the text, if a candidate C contains the first character of S, then test if S matches the next untested segment of C. If a mismatch occurs, eliminate C.

Consider the first character of a text segment at location $T[r,c]$. Any candidate that overlaps this character must have its source in one of nine contiguous blocks.Let

$$k \cdot \frac{m}{2} \leq r < (k+1) \cdot \frac{m}{2}$$

$$l \cdot \frac{m}{2} \leq c < (l+1) \cdot \frac{m}{2} \quad k, l = 0 \ldots \frac{2n}{m} - 1$$

Then, the sources for candidates that can contain $T[r,c]$ are in the $\frac{m}{2} \times \frac{m}{2}$ size blocks encompassing text locations

$$T[(k-i) \cdot \frac{m}{2} \ldots (k-i+1) \cdot \frac{m}{2}, (l-j) \cdot \frac{m}{2} \ldots (l-j+1) \cdot \frac{m}{2}] \quad i, j = 0 \ldots 2$$

These blocks are on lists C_{l-j}. Number the blocks consecutively row-by-row from upper left. Then $T[r,c]$ occurs in block B_9. Candidates with sources in blocks $B_4 - B_6$ always overlap row r. Candidates with sources in blocks $B_1 - B_3$ and $B_7 - B_9$ may be, respectively, completely above row r or completely below row r.

Now, compare segment S with first character at $T[r,c]$ and , candidate C with source in one of the nine blocks. Test if S matches the next untested segment in C. (We keep, for each candidate, a pointer to the next untested segment in the compressed representation for P.) On a mismatch, either of character, position or length, eliminate C.

We perform a special test when we begin testing a new row r_C of C, but S is *not* the first segment of that row. C is eliminated unless S matches the second segment of row r_C and the character of the text segment preceding S matches the character of the first segment of row r_C.

We move down each column list C_i as we move down the rows of the text, so there is no time penalty for finding the (possibly) nine candidates to test for each segment.

Theorem 10.6. *The algorithm runs in time* $O(c(T) \log |P|)$

Proof: Restriction Phase: Clearly, $O(c(T))$.

Compatibility Phase: Each stage requires two passes through the text (size $c(T)$) and two passes through the lists C_i (size $O(c(T))$. Sorting the witnesses is done by radix sort in time $O(c(T))$. Each of $\log m$ stages thus requires time $O(c(T))$ and the entire compatibility phase requires time $O(c(T) \log m) = O(c(T) \log |P|)$.

Verification Phase: There are $c(T)$ text segments and $O(c(T))$ candidates. In the scan of the text rows, each segment is tested against at most 9 candidates. The entire verification phase requires time $O(c(T))$.

The total time for the algorithm is therefore $O(c(T) \log |P|)$. □

10.4 Bibliographic Notes

We have scanned some of the recent results in two dimensional matching. This area is clearly too big to be comprehensively studied in such a brief description. We have therefore concentrated in efficient worst-case sequential deterministic algorithms for exact matching and its variants. Given the long list of modifiers, it is not surprising that a vast amount of work falls outside of the scope of this chapter.

For the interested reader, we can recommend the following papers. For the automata approach described in Section10.1, see Bird [1977] and Baker [1978]. For the alphabet independent approach of Section 10.1.2, see Amir, Benson and Farach [1994a]. The first $O(m^2)$ witness table construction appeared in Park and Galil [1992]. See Amir, Landau and Vishkin [1992] for a description of the scaled matching algorithm of Section 10.2, and Amir and Calinescu [1996] for an alphabet independent and dictionary scaled matching algorithm. The compressed matching algorithm of Section 10.3 appeared in Amir and Benson [1992], and an improvement can be found in Amir, Benson and Farach [1994b].

See Aho and Corasick [1975] for a description of the the automaton based dictionary matching algorithm. Suffix trees and their contruction are described in Weiner [1973] and McCreight [1976]. The first optimal algorithm for Least Common Ancestor queries appears in Harel and Tarjan [1984]. Gabow, Bentely and Tarjan [1984] discuss range minimum queries and cartesian trees.

10.5 Exercises

1. Write a $O(m)$ time algorithm that constructs the witness table of a pattern string $P = p_1 \cdots p_m$. More precisely,

$$WITNESS(i) = \begin{cases} 0, & \text{if } p_{i+j} = p_{1+j}, \quad j = 0, ..., m-i; \\ k, & \text{if } p_k \neq p_{k-i+1}, \text{ for some } i \leq k \leq m. \end{cases}$$

2. Recall the Bird and Baker two dimensional pattern matching algorithm. It first used the Aho and Corasick dictionary matching algorithm to find all pattern rows, and then did a Knuth-Morris-Pratt (KMP) scan vertically, to find all locations were the pattern rows appear consecutively, exactly below each other.
   Using suffix trees, modify the KMP algorithm to find all consecutive appearances of the pattern rows in the text *without needing* the Aho and Corasick algorithm.
3. The two dimensional scaling algorithm presented here had time complexity $O(n^2 \log m)$. Modify the algorithm to achieve time $O(n^2)$. (*Hint:* Assume non-trivial patterns, i.e. patterns with at least two different alphabet symbols. Use symbol changes in the text as "an-

chors" for possible pattern occurrences.)

4. A *prefix encoding* of string $T = t_1 \cdots t_n$ over alphabet Σ is a mapping $c(T) = c(t_1) \cdots c(t_n)$ where $c : \Sigma \to \{0,1\}^*$ and where for any $a \neq b,\ a, b \in \Sigma\ c(a)$ is not a prefix of $c(b)$ and $c(b)$ is not a prefix of $c(a)$. Consider the following compressed matching problem:

   INPUT: Prefix encoding c, $c(T)$, and pattern P.
   OUTPUT: All locations in T where there is an occurrence of P.

   Give a $O(|c(T)| + |P|)$ algorithm for the above problem.

5. In the compressed matching algorithm we assumed that every row (or every column) has a *seam*. In fact, it is sufficient to assume that **one** row (or column) has a seam. Let us assume that we will encode by run-length of rows if every row has a seam, otherwise, we encode by run-length of columns, if every column has a seam. If there are at least one row and at least one column without a seam (i.e. a *stripe*), then we encode by run-length compression of the rows.

   (a) Prove that if the pattern is non-trivial (has more than one symbol) then there are at least one row and at least one column with a seam.
   (b) Prove that if there are stripes in the compression, then all stripes are of the same "color" (have the same repeating symbol).

6. Generalize the compressed matching algorithm to handle compressions with stripes, by adding a preliminary scan that eliminates all text locations that have a stripe of the wrong color.

10.6 Bibliography

A. Amir and G. Benson [1992]. "Two dimensional periodicity and its application", *Proc. of 3rd Symoposium on Discrete Algorithms, Orlando, Florida*, 440–452.

A. Amir, G. Benson, and M. Farach [1994a]. "An alphabet independent approach to two dimensional matching", *SIAM J. Comp.*, **23**:2, 313-323.

A. Amir, G. Benson, and M. Farach [1994b]. "Optimal two-dimensional compressed matching", *Proc. 21st Intl. Col. on Automata, Languages and Programming (ICALP 94)*, Springer-Verlag LNCS 820, 215-226.

A. Amir and G. Calinescu [1996]. "Alphabet independent and dictionary scaled matching", *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM 96)*, Springer-Verlag LNCS 1075, 320-334.

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

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

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

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

H. N. Gabow, J. L. Bentley, and R. E. Tarjan [1984]. "Scaling and related techniques for geometry problems", *Proc. 16th Symposium on the Theory of Computing (STOC 84)*, 135 – 143.

D. Harel and R.E. Tarjan [1984]. "Fast algorithms for finding nearest common ancestor", *Journal of Computer and System Sciences*, **13**, 338–355.

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

G. M. Landau and U. Vishkin [1989]. "Fast parallel and serial approximate string matching", *Journal of Algorithms*, **10**:2, 157–169.

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

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

K. Park and Z. Galil [1992]. "Truly alphabet-independent two-dimensional pattern matching", *Proc. 33rd Symposium on the Foundations of Computer Science (FOCS 92)*, 247-256.

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