Two Dimensional Dictionary Matching

Amihood Amir* Martin Farach†
Georgia Tech DIMACS

September 10, 1992

Abstract

Most traditional pattern matching algorithms solve the problem of finding all occurrences of a given pattern string P in a given text T. Another important paradigm is the *dictionary matching* problem. Let $D = \{P_1, ..., P_k\}$ be the *dictionary*. We seek all locations of dictionary patterns that appear in a given text T.

Previous dictionary matching algorithms have all involved exact matching of a set of strings. In this paper, we present an algorithm for the *Two Dimensional Dictionary Problem*.

The two dimensional dictionary problem is that of finding each occurrence of a set of two dimensional patterns in a text. Our algorithm runs in time $O(|D| \log k)$ preprocessing, $O(|T| \log k)$ text processing.

*College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280; (404) 894-3152; amir@cc.gatech.edu; Partially supported by NSF grant IRI-9013055.

†DIMACS, Rutgers University, P.O.Box 1179, Piscataway, NJ 08855-1179; farach@dimacs.rutgers.edu; Supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center under NSF contract STC-8809648.

1 Introduction

Traditional Pattern Matching has dealt with the problem of finding all occurrences of a single pattern in a text (under some definition of the word "occurrence"). The most basic instance of this problem is the *Exact String Matching Problem*, i.e. the problem of finding all *exact* occurrences of a pattern in a text. This problem has been extensively studied. The earliest linear time algorithms include [KMP77] and [BM77]. For two dimensional patterns and texts, the first linear time algorithms for bounded alphabets were given in [Bir77] and [Bak78]. The first linear time algorithm for the unbounded alphabet case appears in [**?**].

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

Any pattern matching algorithm can be trivially extended to a set of patterns by matching for each pattern separately. If a given algorithm runs on a text T and a pattern P in time $M(t,p)$, were t is the length of T and p the length of P, then the trivial scheme runs on a text T and a dictionary $D = \{P_1, P_2, \ldots, P_k\}$ in time $\sum_{i=1}^{k} M(t,p_i)$. Throughout the rest of this paper, we will refer to the size of some object X by its lower case equivalent x. The only exception will be when referring to dictionaries, in which $d = \sum_{P_i \in D} p_i$, while $k = |D|$, the cardinality of the dictionary. In addition, for alphabet Σ, we will take σ_P and σ_D to mean the number of distinct characters that occur in pattern P or dictionary D respectively. In general, when there is no ambiguity, we will drop the subscript and simply refer to σ.

In certain setting, however, one can do much better than this brute force approach. Aho and Corasick [AC75] solved the Static Dictionary Matching problem. Given a dictionary D whose characters are taken from the alphabet Σ, they preprocess the dictionary in time $O(d \log \sigma_D)$ and then process text T in time $O(t \log \sigma_D)$. The algorithm is for static dictionaries because inserting or deleting a pattern may require processing the entire dictionary again.

Their complexity is perhaps surprising because the text scanning time is *independent* of the dictionary size (for finite alphabet). Aho and Corasick reported their complexity in terms of output size. This is because more than one pattern might match at any given location. They chose to represent their output as a list at each location of all patterns that match at that location. However, when dealing with exact matching, one needs simply to output the *longest* pattern since all information about shorter patterns is contained implicitly in this representation and can be retrieved in time linear in the number of matching patterns. We choose this representation since it is computationally equivalent but the output size is linear.

In [AF91a] and [**?**], we showed a dynamic dictionary matching algorithm with the following complexities:

Adding Pattern P:	$O(p \log(d+p))$
Deleting Pattern P:	$O(p \log d)$
Scanning Text T:	$O(t \log d)$.

Note that for unbounded alphabets this algorithm matches the complexity of the Aho-Corasick static algorithm, since $\sigma_D = O(D)$.

In this paper we present a *two dimensional* dictionary matching algorithm that preprocesses the dictionary in time $O(d(\log k + \log \sigma)$ and subsequently finds all appearances of all dictionary patterns in text T in time $O(t(\log k + \log \sigma))$.

There are two main ideas behind our algorithm. First, we linearize the pattern matrices along the diagonal to produce a dictionary of strings over an alphabet of subrow/subcolumn pairs. We analyze the KMP algorithm to show how to apply the automaton technique to our new strings while preserving the two dimensionality of the underlying problem. We also explore efficient methods of dynamically inserting new elements into suffix trees.

In section 2, we present a novel algorithm for two dimensional matching. In section 3, we present an algorithm for two dimensional dictionary matching. We conclude with open problems in section **??**.

2 A Two-dimensional Matching Algorithm

For the sake of clarity of exposition, we first present a novel algorithm for matching a single two dimensional pattern and in the next section show how to generalize to a set of patterns.

We start with definitions. We say that b is a *border* of x if $|b| < |x|$ and if b is a prefix and a suffix of x. We set $B(x) = |b|$, where b is the longest border of x.

Consider the KMP algorithm [KMP77] for exact one dimensional matching. In the **preprocessing stage**, the pattern string is used to construct an automaton with the following properties. Each node, i, representing the i^{th} prefix of $P[1, \ldots, n]$, has two links, a forward or success link to node $i+1$ which is traversed if the next character read equals $P[i+1]$, and a failure link whose value is $B(P[1, \ldots, i])$. That is, if $FL(i)$ is the failure link from the i^{th} node of the automaton, then $P[1, \ldots, FL(i)] = P[i - FL(i) + 1, \ldots, i]$.

KMP showed the following lemma:

Lemma 1 *For string $X \in \Sigma^+$ and $\alpha \in \Sigma$, if $X[B(X)+1] = \alpha$, then $B(X\alpha) = B(X)+1$. We will call this the* border extensibility property.

Based on this lemma, they showed how to construct such a KMP automaton in linear time, under the assumption that letters of the alphabet can be compared in constant time.

In the **text scanning stage**, the text is scanned linearly, following success or failure links in the automaton.

The KMP idea can be directly used for 2-dimensional exact matching. Consider every pattern row as a single character and construct the KMP automaton of the "linearized" matrix. Now scan the text down each column considering the subrow of length m starting at each position as the (single) character to be scanned. If a comparison between rows can be done in constant time, we have a linear-time exact 2-dimensional algorithm. This idea was used in [Bir77] and [Bak78] to produce a linear time two dimensional matching algorithm. The same idea was used differently for scaled matching in [ALV90]. These algorithms rely on the fact that the rows of a matrix are all the same length. If all the rows are the same length then exactly one of them can match at any given location of the text. The KMP algorithm only allows for a single forward transition state, thus if more than one row pattern matches at a text location (e.g. if the row patterns are of different lengths), this approach breaks down. The issue of non-rectangular pattern matching was addressed in [AF91b].

We propose a different approach. Instead of linearizing the 2-dimensional matrix along one of its axes, we linearize along the diagonal. Note that when we cut a matrix along the diagonal we divide the matrix into an upper right half and a lower left half. We construct a new alphabet from the subrows of the lower left hand corner and their matching subcolumns from the upper right hand corner. Note that we now have "characters" of different length rows and columns. As noted above, more than one such character can match at some location, therefore a direct application of the KMP algorithm will not work. However, we show how to modify the border extensibility property so that KMP will still work.

2.1 Preprocessing: (Automaton construction)

Convert the matrix $M[1,\ldots,m;1,\ldots,m]$ into string $M'[1,\ldots,m]$ as follows. Let $c(i) = M[i,\ldots,1;i]$ (this is a subcolumn of M in *reverse order*). Similarly, let $r(i) = M[i;i,\ldots,1]$ (also in reverse order). We now set $M'[i] = \langle c(i), r(i)\rangle$.

We define the i^{th} prefix of the array M to be $\mathcal{P}(i) = M[1,\ldots,i;1,\ldots,i]$ and the i^{th} suffix to be $\mathcal{S}(i) = M[i,\ldots,n;i,\ldots,n]$. We will use the notational shorthand $\mathcal{S}(i,j)$ to mean $M[i,\ldots,j;i,\ldots,j]$, that is, the i^{th} prefix of the j^{th} suffix of M. Note that $\mathcal{P}(i) = \mathcal{S}(1,i)$ and that $\mathcal{S}(i) = \mathcal{S}(i,n)$.

Recall that for any string X, if $X[B(X)+1] = \alpha$ then $B(X\alpha) = B(X)+1$. This straightforward statement about strings raises two issues when we generalize to two dimensions. First note the following about our new string M'. $M'[i]$ consists of a row and column each of length exactly i. Therefore, when we check in the KMP algorithm if $X[B(X)+1] = \alpha$, we note that $X[B(X)+1]$ is a character of "size" $B(X)+1$ and α is a character of "size" $|X|+1$ (since $X\alpha$ must be a valid string). But $B(X) < |X|$, so α is never exactly the same row/column pair as $X[B(X)+1]$. However, we need not abandon the border extensibility property and the KMP algorithm. We define the operator $\doteq$ to replace exact equality.

Let us first consider what is required of the KMP automaton construction when we move on the diagonals. Suppose that at some stage j of constructing the KMP automaton of M', we have determined that $\mathcal{P}(i) = \mathcal{S}(j-i+1,j)$. To see if $\mathcal{P}(i+1) = \mathcal{S}(j-i+1,j+1)$ we must confirm that $\langle c(i+1), r(i+1)\rangle = \langle M[j+1,\ldots,j-i+1;j+1], M[j+1;j+1,\ldots,j-i+1]\rangle$. But $M[j+1;j+1,\ldots,j-i+1]$ is a prefix of $r(j+1)$ and $M[j+1,\ldots,j-i+1;j+1]$ is a prefix of

$c(j+1)$. Thus, we define the "equality" relation as follows:

For $i \leq j$, we define $\langle c(i), r(i)\rangle \doteq \langle c(j), r(j)\rangle$ iff $c(i)$ is a prefix of $c(j)$ and $r(i)$ is a prefix of $r(j)$.

As noted above, this relation is not symmetric and is thus not an equality relation at all.

Lemma 2 *Given a string M' which is the diagonal linearization of a two dimensional matrix M, then we can construct an automaton to match for M with a linear number of row/column comparisons.*

Proof:

By lemma 1, the KMP construction relies on the border extensibility property. Bye the argument given above, the relation $\doteq$ preserves the border property in diagonally linearized matrices. We can therefore use the KMP automaton construction as a black box for building the automaton for M' using a linear number of meta-character comparisons.□

It remains to be shown how to compare the row/column meta-characters efficiently, as well as how to use the automaton to match against the text.

2.2 Text scanning:

We linearize the given text, $T[1,\ldots,n;1,\ldots,n]$ as follows. Construct a set of $2n-1$ strings $T'_i[1,\ldots,n-|i|]$, $-(n-1) \leq i \leq n-1$, where each T'_i corresponds to the diagonal whose column and row coordinates differ by i, that is, $T[c,r]$ is on the diagonal T'_{c-r}. For $i=0$, we get the main diagonal, and for $i>0$ ($i<0$) we get the diagonals below (above) the main diagonal. Let $m_i = \max\{1,i\}$. We let $T'_i[j] = \langle c_i(j), r_i(j)\rangle$ where

$$c_i(j) = T[m_i + j, \ldots, m_i; j]$$

$$r_i(j) = T[m_i + j; m_i + j, \ldots, m_i]$$

Algorithm 1 2-D matching – scanning the text

Begin

```
for i←−(n−1) to n−1 do
        state←0
        for j←1 to n−|i| do
                if M'[state+1]≐T_i[j]
                        state←state+1
                        if state = m
                                Found a match
                                state←FL(state)
```

else
while($state > 0$ and $M'[state + 1] \dot{\neq} T_i[j]$)
$state \leftarrow FL(state)$

End

Implementation Issues:

Our main concern is making comparisons between subrows or between subcolumns in constant time. We handle the constant-time comparison differently for the automaton construction and for the text scanning parts.

Automaton Construction: To compare $c(i)$ with $c(j)$ we do the following. Take each column in reverse order (that is, from the bottom of the matrix to the top) and concatenate them. Then, in linear time (times log of the effective alphabet size) construct the suffix tree, T_c (for construction see e.g. [McC76, Wei73]), of this new string and preprocess the string for Least Common Ancestor (LCA) queries (see [HT84]). As has been described in many sources, (e.g. [**?**]), it is now possible to compare two subcolumns by finding their corresponding leaves in T_c and finding the LCA of those leaves. The rows are processed analogously to construct the tree T_r. The total preprocessing time is therefore $O(p \log \sigma_P)$.

Text Scanning: Theoretically, we can construct the suffix trees and do the LCA preprocessing for a concatenation of the text columns and rows *together* with those of the patterns. The problem is that then we would need to preprocess the patterns anew for every different input text. For single pattern matching, this is no problem, since the pattern is smaller than the text. However, when we have a dictionary of patterns, any particular text may be much smaller than the total size of the dictionary. We therefore would like to avoid rebuilding the dictionary data structure each time we need to process some text.

Note that it is possible to insert the text rows and columns into the suffix trees in time $O(n^2 \log \sigma_P)$ as follows. Weiner [Wei73] (and McCreight [McC76]) showed a construction for the suffix tree of a string X that runs in $O(x \log \sigma_X)$ time. The details of the Weiner algorithm are very involved (and are largely irrelevant to this discussion) and are therefore omitted (see [CS85] for an excellent treatment of this topic). However, certain details are important for the algorithm presented below. We therefore give a high level overview of the Weiner algorithm.

Let T_X be the suffix tree of the string X. Further, let $T_{\geq i}$ be the suffix tree of the substring $x_i, \ldots, x_n$. The Weiner algorithm proceeds by taking $T_{\geq i}$ and updating it to create $T_{\geq i-1}$. This step takes amortized $O(\log \sigma_X)$ time and therefore the construction takes $O(x \log \sigma_X)$ time.

While the amortized cost is constant, for constant size alphabet, the maximum possible cost for inserting a new suffix is of the order of the depth of the last inserted node in a tree. We denote the depth of the last inserted suffix in T_X as $d(T_X)$. The worst case cost of converting $T_{\geq i}$ to $T_{\geq i-1}$ is $O(d(T_{\geq i}))$. Therefore the cost of transforming T_X to T_{YX} for strings X and Y is $O(y + d(T_X))$. Let \$ be a symbol that does not appear in X. Then building $T_{\$X}$ takes $O(x \log \sigma_X)$. But since the suffix $\$X$ shares no initial characters with any other suffix of $\$X$, $d(T_{\$X}) = 1$. Therefore changing

$T_{\$X}$ to $T_{Y\$X}$ takes time $O(y \log \sigma_{Y\$X})$.

The remaining problem is that the LCA preprocessing is not dynamic, that is, we cannot make LCA queries involving the newly introduced nodes in constant time. However, when we run the automaton on the text we only make comparisons between text rows (columns) and pattern rows (columns). We never compare text rows (columns) with each other. We can therefore insert the text rows and columns into T_c and T_r and whenever we insert a new leaf, we note its closest ancestor that was a node in the original tree. When making LCA queries, we can make queries about these ancestor nodes rather than the new leaf itself and still get the same LCA. We conclude:

Theorem 1 *Given a pattern P of size $m \times m = p$ and a text T of size $n \times n = t$, we can find all occurrences of the pattern in the text in time:*

Preprocessing: $O(p \log \sigma_P)$.

Text Scanning: $O(t \log \sigma_P)$.

Proof:

By lemma 2, we can build the matching automaton with a linear number of comparisons. However, the considerations about show that we can do the comparisons in constant time after we build a suffix tree in $O(p \log \sigma_P)$.

Finally, it is clear that P occurs at $T[r, c]$ iff P', the diagonally linearized pattern, occurs at $T_{c-r}[\max\{c, r\}]$. The correctness of the algorithm then follows from this observation and from the correctness argument for the KMP algorithm in [KMP77].□

3 A Two-dimensional Dictionary Matching Algorithm

Aho and Corasick (henceforth AC) showed in [AC75] that KMP could be modified to match for a set of patterns rather than a single pattern. In their case, they once again constructed an automaton against which the text is run. However, their success and failure links where modified as follows. As before, each node represents some prefix of a pattern. We will let $S(n)$ be the string represented by node n (we will let S^{-1} be S's inverse function, which is obviously only defined over the set of prefixes of the dictionary patterns). A set of patterns may now share a prefix. Thus each node may have up to k success links, one for each letter following its string in the set of patterns. We set $GOTO(n, \alpha)$ to be the node such that $S(GOTO(n, \alpha)) = S(n)\alpha$. Then $GOTO(n, \alpha)$ is only defined if $S(n)\alpha$ is a prefix of some dictionary pattern. For example, suppose our dictionary contains the patterns *aaab* and *aaaca*. Then there will be a node n in the automaton such that $S(n) = aaa$. That node would have a success link for b and c, that is, $GOTO(n, b)$ and $GOTO(n, c)$ will be defined.

In order to describe the failure links in an AC automaton, we define the function β as follows. let a *set border* b of string X with respect to D be a proper suffix of X that is a prefix of some string in dictionary D. We set $\beta(X, D) = b$, where b is the longest set border of X with respect to D.

For each node n of the automaton, we set the failure link $FL(n) = S^{-1}(\beta(S(n), D))$. The construction of such an automaton is efficient if we can insure the following property, which was proved in [AC75]:

Lemma 3 *For all nodes n, if $GOTO(FL(n), \alpha)$ is defined and $GOTO(n, \alpha)$ is defined, then $FL(GOTO(n, \alpha)) = GOTO(FL(n), \alpha)$.*

This is correspondingly called the *set border extensibility property*. Here we have a case that is analogous to the problem with the border extensibility property in the KMP algorithm. If $GOTO(n, \alpha)$ is defined then α is of size $|S(n)| + 1$, but if $GOTO(FL(n), \alpha)$ is defined, then α is of size $|S(FL(n))| + 1$. Once again we know that $|S(FL(n))| = |\beta(n, D)| < |S(n)|$ so there is no α for which $GOTO$ will be defined in both cases. For our diagonally linearized patterns, we get the following lemma.

Lemma 4 *If $GOTO(n, \alpha)$ is defined and $\exists \alpha'$ such that $\alpha' \doteq \alpha$ and $GOTO(FL(n), \alpha')$ is defined, then $FL(GOTO(n, \alpha)) = GOTO(FL(n), \alpha')$.*

Proof: The correctness follows by the same argument in as in the KMP based algorithm of section 2 by further noting that since $|\alpha'| < |\alpha|$, we know that α' is uniquely defined, i.e. it is the first $|S(FL(n))| + 1$ characters of α.□

Algorithm 2 2-dimensional exact dictionary matching

Begin

[1] Preprocessing:
Construct suffix trees T_c and T_r and preprocess for LCA.
Construct an AC automaton on the linearized patterns.

[2] Text Scanning:
Insert text rows and columns into T_r and T_c respectively.
Scan the linearized text using the AC automaton.
Remove text rows and columns from T_r and T_c. respectively.

End

Implementation Issues:

Constructing an AC automaton (and later using it when scanning the text) involves choosing a success link at every node. If the alphabet is fixed this can be done in constant time. Otherwise it will take $O(\log deg)$, where deg is the maximum possible outdegree at every automaton node. In our case deg is at most k since success at a node means traversing a link labeled with a possible

subrow-subcolumn pair. The size of the next pair is fixed, (hence only one possible per pattern) and there are k patterns, thus there are at most k possible success links.

We summarize with the following theorem.

Theorem 2 *Given a dictionary $D = \{P_1, P_2, \ldots, P_k\}$ of two dimensional patterns such that P_i is a $p_i \times p_i$ matrix with entries taken from the set Σ, then it is possible to find, for each location $T[i,j]$ of a $n_1 \times n_2$ text T, the largest pattern P_i with matches at that location within the following time bounds:*

Step 1 $O(d(\log k + \log \sigma_D))$ *for dictionary preprocessing.*

Step 2 $O(t(\log k + \log \sigma_D))$ *for text scanning.*

References

[AC75] A.V. Aho and M.J. Corasick. Efficient string matching. *C. ACM*, 18(6):333–340, 1975.

[AF91a] A. Amir and M. Farach. Adaptive dictionary matching. *Proc. 32nd IEEE FOCS*, 1991.

[AF91b] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. *Proc. of 2nd Symoposium on Descrete Algorithms, San Fransisco, CA*, Jan 1991.

[ALV90] A. Amir, G.M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *Proceedings of First Symposium on Discrete Algorithms, San Fransisco, CA*, 1990.

[Apo91] A. Apostolico. Efficient crcw-pram algorithms for universal substring searching. Technical Report 91.2, Fibonacci Institute, 1991.

[Bak78] T.J. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comp*, 7:533–541, 1978.

[Bir77] R.S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977.

[BM77] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[CS85] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, chapter 12, pages 97–107. NATO ASI Series F: Computer and System Sciences, 1985.

[GG88] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4:33–72, 1988.

[GS83] Z. Galil and J.I. Seiferas. Time-space-optimal string matching. *J. Computer and System Science*, 26:280–294, 1983.

[HT84] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestor. *Computer and System Science*, 13:338–355, 1984.

[KLP89] Z. M. Kedem, G. M. Landau, and K. V. Palem. Optimal parallel suffix-prefix matching algorithm and application. *Proceedings of the Symposium on Parallel Algorithms and Architectures*, 6:388–398, 1989.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[Ukk85] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

[Vis89] U. Vishkin. Deterministic sampling for fast pattern matching. Manuscript, 1989.

[Wei73] P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.