A Sub-quadratic Algorithm for Approximate Limited Expression Matching

Sun Wu, Udi Manber, & Gene Myers

TR 92-36

A Sub-quadratic Algorithm for Approximate Limited Expression Matching

Sun Wu, Udi Manber[1]*, and Gene Myers*[2]

TR 92-36

ABSTRACT

In this paper we present an efficient sub-quadratic time algorithm for matching strings and limited expressions in large texts. Limited expressions are a subset of regular expressions that appear often in practice. The generalization from simple strings to limited expressions has a negligible affect on the speed of our algorithm, yet allows much more flexibility. Our algorithm is similar in spirit to that of Masek and Paterson [MP80], but it is much faster in practice. Our experiments show a factor of 4 to 5 speedup against the algorithms of Sellers [Se80] and Ukkonen [Uk85] independent of the sizes of the input strings. Experiments also reveal our algorithm to be faster, in most cases, than a recent improvement by Chang and Lampe [CL92], especially for small alphabet sizes for which it is 2 to 3 time faster.

December 16, 1992

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

[1] Supported in part by a Presidential Young Investigator Award DCR-8451397, with matching funds from AT&T, and by NSF grant CCR-9001619.

[2] Supported in part by NIH grant LM04960, NSF grant CCR-9001619, and the Aspen Center for Physics.

A Sub-quadratic Algorithm for Approximate Limited Expression Matching

1. Introduction

Pattern matching problems are used in many applications including text editors, information retrieval, compilers, command interpreters, computational biology, and object recognition. The basic problem is to find all occurrences of a *pattern* in a *text* where the pattern and the text are both simple strings. The two famous algorithms for this string matching problem are the Boyer-Moore algorithm [BM77] and the Knuth-Morris-Pratt algorithm [KMP77]. The basic problem has been extended to include more complicated patterns, including a set of strings (Aho and Corasick [AC75] and Commentz-Walter [CW79]), strings with "don't care" symbols (Fischer and Paterson [FP74]), and strings with "don't care" and "complement" symbols (Pinter [Pin85]). Abrahamson [Abr87] studied the complexity of several matching problems and presented algorithms for strings with "classes," which are sets of characters, but as he himself noted, even though the worst-case running times were better than quadratic, the algorithms were not practical. Baeza-Yates and Gonnet's shift-or algorithm [BG92] can handle all of the above options and it is very practical but only for short patterns.

The *approximate* string matching problem is a generalization of the exact string-matching problem in that now we are looking for all substrings in the text that are *similar* to the pattern. There are many ways to define the similarity metric; we will concentrate on the *edit distance*. The edit distance between two strings A and B is the minimum number of edit operations for transforming A to B, where an edit operation can be a deletion, an insertion or a substitution.

A classical solution to the approximate string matching problem is a dynamic programming approach (Sellers [Se80]), which is a generalization of the dynamic programming approach for comparing two strings (Wagner and Fischer [WF74]). Its running time is $O(mn)$, where m is the size of the pattern and n is the size of the text. There are many results that improve the dynamic programming approach. For example, Ukkonen [Uk85] gives an algorithm whose expected running time is $O(nk)$, where k is the edit distance, by computing only a part of the dynamic programming table. The same paper also gave an automata approach which after some preprocessing of the pattern can process the text in linear time. However, the preprocessing time can be up to 3^m which is too high for large m. Later, several papers [My86, LV88, GP90, UW90] proposed algorithms that achieve $O(nk)$ worst case time complexity, which is an improvement over the $O(nm)$ approach for small k, but not for the worst case where $k = O(m)$. Very recently, a number of algorithms that are sublinear *in expectation* have been proposed [CL90, Uk92, My90], but the first two are no more effective in practice than former algorithms, and the latter requires an inverted index of the entire text. Wu and Manber [WM92] have developed one of the fastest practical algorithms, which is part of the *agrep* package, and can handle limited expressions as well as arbitrary regular expressions. However, the algorithm's dependence on bit-vector operations limits its use to short patterns. A very recent algorithm by Chang and Lampe [CL92] while theoretically no better than Ukkonen's $O(nk)$ algorithm, has been shown experimentally to depend on the size of the underlying alphabet, Σ, as $1/\sqrt{|\Sigma|}$ which makes it very fast for large alphabets.

The only improvement in the worst case performance of approximate string matching is due to Masek and Patterson [MP80], who used the "Four Russian's" technique [ADKF70] to solve the approximate string matching problem in time $O(nm/\log^2 n)$[3]. The "Four Russian's" paradigm, which we also employ, consists of first building a table of solutions to all possible problems of a small size r, and then consulting the table to solve the original problem in blocks of size r. This paradigm was originally arrived at in the context of multiplying boolean matrices [ADKF70] and can be applied to many problems, for example, regular expression pattern matching [My92] and approximate regular expression matching [WMM92]. In the case of Masek and Paterson's instantiation of the paradigm, there is a hidden dependence on alphabet size in that the log factor is actually $\log_b n$ where $b = O(|\Sigma|^2)$. Thus their result holds only for strings over fixed size alphabets, and n must be very large before the log-factor is large enough that the algorithm actually becomes more efficient than the basic dynamic programming algorithm. Indeed, Masek and Paterson estimated that their algorithm would be superior to dynamic programming only when $nm \geq 262{,}419^2$ for an alphabet of only two symbols. They never ran experiments, presumably, because their conception of the method required a table that would not fit in the memory of any of todays computers. In the experimental section we argue that their approach has the potential to be faster than ours only for alphabets of size 2 or 3 and even then it is unlikely.

We present a new algorithm for approximate string matching that achieves subquadratic worst case running time and very good expected running time. We made comprehensive experiments and we demonstrate that our algorithm is faster in practice than any of the previous algorithms by Sellers, Ukkonen, and Chang. In addition, our algorithm works for patterns that contain "don't care," "complement," and "classes" symbols. Most text searching applications require these types of patterns, therefore algorithms that cannot support them have limited applicability. Our algorithm has an expected running time of $O(nk/\log n)$, which is a log factor improvement over Ukkonen's $O(nk)$ expected time algorithm [Uk85]. One of the key differences of our method over that of Masek and Paterson is that we build a single "universal" Four Russian's table, that serves all problem instances. Thus our log-factor does not contain a hidden dependence on alphabet size and in practice yields an algorithm that is a factor of 4 or 5 times faster than Ukkonen's algorithm regardless of the size of n, k, and $|\Sigma|$.

---

[3] Their result was originally stated as $O(nm/\log n)$ under a logarithmic cost model where operations on large integers take time proportional to the number of bits in the operands. In this paper we assume a unit cost model.

2. Preliminaries

Let Σ denote the alphabet set. We assume that the symbols in Σ are ordered and every symbol has a unique index. A *literal* is a subset of symbols from Σ. A *limited expression* is a sequence of literals. For convenience of notation, we use the operators '–', '[', ']', '^', and '.', to denote common types of literals as follows (this is consistent with *agrep*).

1. Any symbol in Σ is by itself a literal.
2. A '.' (don't care symbol) is a literal representing all of Σ.
3. A pair of '[' and ']' define a literal that represents the union of symbols inside the brackets. For example, $[abcdx]$ represents $\{a, b, c, d, x\}$.
4. A '-' inside the brackets is used to represent a range of symbols in Σ. If the index of p is less than that of q, then $[p-q]$ represents the range of symbols between p and q (including p and q). For example, [a-dx] is the same as [abcdx].
5. A '^' immediately after the '[' represents a complement operation of the literal inside the brackets. For example, [^abc] represents the set of symbols from Σ excluding a, b, and c.

The length of a limited expression is the number of literals in it and will be denoted throughout by m. Let $A = a_1 a_2 \cdots a_m$ be a string, and let $P = B_1 B_2 \cdots B_m$ be a limited expression where $B_i, 1 \le i \le m$ is a literal. A matches P if $a_i \in B_i$ for all $1 \le i \le m$. We denote by $L(P)$ the language generated by pattern P, i.e., $L(P)$ consists of all the strings that match P. We define the k-neighborhood of P, $L_k(P)$, as follows: $L_k(P) = \{x \mid y,\ y \in L(P),\ \delta(x,y) \le k)\}$, where $\delta(x,y)$ denotes the edit distance between string x and y. The approximate limited expression pattern matching problem is defined as follows: Given a limited expression $P = B_1 B_2 \cdots B_m$, a text string $T = a_1 a_2 \cdots a_n$, and an integer k, find substrings in the text that are in $L_k(P)$.

It seems to us that it is not easy to extend some of the $O(nk)$ algorithms for approximate string matching that depend on tries or suffix trees, e.g. [LV88, GP90, CL90], to approximate limited expression matching. But for methods based directly on dynamic programming, e.g. [Se80, Uk85], the extension to limited expressions is quite simple under the assumption that the size of the alphabet set is fixed. In the next section, we present the basic dynamic programming approach extended to solve the approximate limited expression matching problem in time complexity $O(nm)$. We then apply our version of the Four Russian's paradigm [ADKF70] to this algorithm, to obtain an $O(\log n)$ factor of improvement. For the analysis of our Four Russian's variations, we assume a unit cost RAM model. In this model, the addressing of an $O(n)$ memory can be done in constant time, and the arithmetic operation with integers representable with $O(\log n)$ bits can be done in constant time too. This model is satisfied by most computers.

3. An O(nm/logn) Algorithm

Let $T[j]$ denote the prefix $a_1 \cdots a_j$ of T and let $P[i]$ denote the prefix $B_1 \cdots B_i$ of P. We can extend the dynamic programming approach to handle limited expressions in a straightforward way. We compute an $(m+1)\times(n+1)$ matrix E such that $E[i,j]$ is the minimum edit distance between $P[i]$ and a suffix of $T[j]$. The matrix is computed using the following recurrence.

$$
\begin{array}{l}
\text{For all } 0 \le i \le m, \quad E[i,0] = i. \\
\text{For all } 1 \le j \le n, \\
\quad E[0,j] = 0 \text{ and} \\
\quad \text{For all } 1 \le i \le m, \\
\qquad E[i,j] = \min\,(E[i-1,j-1] + s_{i,j}, E[i-1,j]+1, E[i,j-1]+1) \\
\qquad\quad \text{where } s_{i,j} = \text{if } a_j \in B_i \text{ then 0 else 1.}
\end{array}
\tag{3.1}
$$

This basic recurrence is attributed to Sellers [Se80], the only difference from the original being the definition of $s_{i,j}$. In order for the algorithm to run in time $O(nm)$, we must be able to determine $s_{i,j}$ in constant time. This is done by precomputing an $m \times |\Sigma|$ table, IN, where $IN[i,\sigma]$ is 1 or 0 depending on whether $\sigma \in B_i$ or not, respectively. This table is built in $O(m\,|\Sigma|)$ time and space. Given the table, $s_{i,j}$ is simply $IN[i,a_j]$.

The matrix E can be evaluated either column by column or row by row. In practice, column by column is more common, because m is in general much smaller than n. Whenever $E[m,j] \le k$, it means that there are some substrings ending at position j in the text that match the pattern with no more than k differences. The computation of column j depends only on the values of column $j-1$. Therefore, there is no need to store the whole matrix. Two columns — the previous column and the current column — suffice. In fact, the algorithm may be viewed as a scan of T in which each column of E is computed from the previous column and the currently scanned character.

To develop a sub-quadratic algorithm, we begin with a review of Ukkonen's automata approach which consists of two phases: (1) preprocessing the pattern to generate an automaton, and (2) scanning the text using the automaton. The preprocessing time may be exponential in m, but after the preprocessing the scanning of the text can be done in linear time.

The states of the automaton, call it M, correspond to all possible columns that can occur in the computation of the dynamic programming matrix. In the preprocessing, all such possible states, together with all possible state transitions on every possible input character in Σ, are precomputed and stored in a transition table modeling M. That is, the algorithm precomputes all the possible column transitions in the dynamic programming table before it scans the text, such that when the text string is processed the column transition can be computed in constant time just by table lookup.

At first glance, it seems that the number of possible states of M could be as high as $O((m+1)^{(m+1)})$, since there are $m+1$ entries per column and each entry can have value from 0 to

m. Ukkonen showed that the number of possible states is actually bounded by 3^m by showing the following lemma actually proven earlier by Masek and Paterson.

Lemma 3.1: [Uk85, MP80]

$$-1 \le E[i,j]-E[i-1,j] \le 1 \quad \text{for } 1 \le i \le m, 0 \le j \le n. \tag{3.2}$$

We carry this idea a little further by using the differences $D_j[i]=E[i,j]-E[i-1,j]$ directly in the recurrence and maintaining only these values. To express the column D_j of differences in terms of column D_{j-1}, the recurrence below requires the quantities $c_{i,j}=E[i,j]-E[i,j-1]$ at each row. Note that computationally given an array $D[1..m]$ holding the column D_{j-1}, the recurrence permits column D_j to be computed in place with a single auxiliary scalar c holding the $c_{i,j}$ values as the column is produced in increasing row order.

Lemma 3.2: For $0 \le i \le m$, $D[i,0]=1$.
For all $1 \le j \le n$,
$c_{0,j}=0$ and
For all $1 \le i \le m$,

$$D_j[i] = \min(1, s_{i,j}-c_{i-1,j}, D_{j-1}[i]-c_{i-1,j}+1) \tag{3.3}$$
and $c_{i,j}=c_{i-1,j}+D_j[i]-D[i,j-1]$.

Proof: We have to prove that (3.3) follows from (3.1). The proof is by a simple algebraic manipulation. First observe that

$$\begin{aligned}
D_j[i] &= E[i,j]-E[i-1,j]\\
&= \min(E[i-1,j]+1, E[i-1,j-1]+s_{i,j}, E[i,j-1]+1) - E[i-1,j] \quad \text{(by (3.1))}\\
&= \min(1, s_{i,j}-(E[i-1,j]-E[i-1,j-1]), (E[i,j-1]-E[i-1,j-1])-(E[i-1,j]-E[i-1,j-1])+1)\\
&= \min(1, s_{i,j}-c_{i-1,j}, D_{j-1}[i]-c_{i-1,j}+1).
\end{aligned}$$

Moreover,

$$\begin{aligned}
c_{i,j} &= E[i,j]-E[i,j-1]\\
&= (D_j[i]+E[i-1,j]) - (D_{j-1}[i]+E[i-1,j-1])\\
&= (E[i-1,j]-E[i-1,j-1]) + D_j[i]-D_{j-1}[i]\\
&= c_{i-1,j}+D_j[i]-D_{j-1}[i].
\end{aligned}$$

Thus (3.3) is correct. □

Lemma 3.1 shows that a state of the automaton M can be represented by a vector of size m whose entries have values –1, 0, or 1. Lemma 3.2 shows how to compute the transition table for M directly in this representation framework. Of course the problem is that M has 3^m states. We now proceed to show how to simulate M using a combination of smaller automata.

Our approach is to partition each column into regions and build a smaller automaton for each region. We then combine these smaller automata to simulate the function of the original automaton M. Each m-vector is partitioned into sub-vectors, called *regions*, of size r (to be determined later). Suppose for simplicity that r divides m evenly. Let region p where $1 \le p \le m/r$, of m-vector $D[1..m]$ be the sub-vector $D[(p-1)\cdot r+1, (p-1)\cdot r+2, .. p\cdot r]$. Denote by

$D_j<p>$ the p region of the j column vector; that is, $D_j<p> = D_j[(p-1)\cdot r+1 \,..\, p\cdot r]$. Each region can be one of the 3^r possible vectors of r differences. From here on we will use the term state to describe the values of a region, as opposed to the global state of the automaton M. We encode the state of a region by an integer between 0 and 3^r-1 in the obvious way. What we would like to do is precompute all the state transitions for a region p, so that $D_j<p>$ can be computed in a single step from $D_{j-1}<p>$ during the scan of the text string T.

Let's study (3.3) more closely. The state of $D_j<p>$ depends on three factors: (1) the state of $D_{j-1}<p>$, (2) the value of $c_{(p-1)\cdot r,j}$, and (3) the values of $s_{(p-1)\cdot r+1,j}$, $s_{(p-1)\cdot r+2,j}$, $\cdots$ $s_{p\cdot r,j}$, called the *characteristic vector* of region p on input a_j. Overall, $D_j<p>$ depends on two vectors of size r, and a 'carry in' value $c_{(p-1)\cdot r,j}$. The 'carry out', $c_{p\cdot r,j}$, needed as input for region $p+1$, is also a function of these same three factors. Both the column vector and characteristic vector (of size r) can be encoded by an integer in the obvious way, as long as r is bounded by $O(\log n)$ (recall the assumption of unit cost RAM). Thus we can build a transduction automaton whose states are the 3^r possible difference vectors of size r. Given a characteristic vector cv encoded as an integer between 0 and 2^r-1, and a carry-in value $c \in \{-1,0,1\}$, the automaton moves from its current states s to a uniquely determined next state and outputs a uniquely determined carry-out value as it does so. This automaton can be modeled as two tables, $GOTO[cv,s,c]$ and $COUT[cv,s,c]$, that take $2r+1$ such values encoded as three integers, and return the next state and carry-out integers, respectively. If $r=\log_6 n$, then these tables have exactly $2^{\log_6 n}\cdot 3^{\log_6 n}\cdot 3$ = $3n$ possible index triples and we will show that they can also be computed in time $O(n)$. These tables can be precomputed and stored in $O(n)$ space. Note that by using the characteristic vector instead of input characters and literals in the transition, every region can use this same automaton and so we call it a *universal automaton*. One advantage of such a universal automaton is that it needs to be computed only once for a given region size r, and can then be used for any pattern and input string. Thus in practice, one builds an automaton whose size is dependent on the amount of available memory and not on n, since the time to build the table need no longer precede every search. However, the characteristic vectors for a particular search are specific to the pattern and alphabet, and so must be constructed each time. We will talk about constructing this function in the next paragraph.

Using the precomputed transition table, we can compute $r=O(\log n)$ values of the dynamic-programming matrix in constant time if the input to the transition table can be obtained in constant time. The carry-in and previous region state are already encoded as integers, so the only question is how to deliver the integer codes for the characteristic vectors. Each characteristic vector consists of r binary values and computing these and encoding them as an integer from scratch every time will take $O(r)$ time which defeats our aims. This difficulty is circumvented by precomputing for each region the integer for the characteristic vector that results from each of the $|\Sigma|$ different symbols that might arise in the text. That is, we build a table, CV of dimensions $m/r \times |\Sigma|$, such that for each region p and each $\sigma \in \Sigma$, $CV[\sigma,p]$ is the integer encoding the vector $[\,IN[(p-1)\cdot r+1,\sigma],\ IN[(p-1)\cdot r+2,\sigma],\ \cdots,\ IN[p\cdot r,\sigma]\,]$.

Once this table is constructed, the characteristic vector of any region on any input character can be looked up in constant time.

After the transition table and the characteristic function table have been built, the algorithm finds the approximate matches as detailed in Figure 1. The principal invariant is that just before the execution of the loop in lines 5 through 11, $D[p] = D_{j-1}\langle p \rangle$ for all $1 \le p \le m/r$ and $e = E[m, j-1]$. That is, the array D contains the states of the regions for column $j-1$ and e is the value of the last entry of column $j-1$. Initially, in lines 1 through 3 the state of each region p, $1 \le p \le m/r$, is set to the integer encoding of $(1, 1, 1, \cdots, 1)$ and e is set to m. By Lemma 3.3, this correctly establishes D and e for column 0. The loop of line 4 scans the text one character at a time. To advance the invariant over a character a_j being scanned, we compute the state transition for every region p in increasing order of p so that the carry-out of one region is available as the carry-in for the next. First, we find the characteristic value for a_j corresponding to region p by a lookup into table CV. Then, we use $c = c_{(p-1)\cdot r, j}$, the current state $D[p]$ of region p, and the characteristic value, to lookup the state of region p in column j and the carry-out of the transition in the tables $GOTO$ and $COUT$. Upon the completion of this computation in the minor loop of lines 6 and 7, the value of c is $c_{m,j}$ and this is added to e in line 8. From the invariant and the fact that $E[m, j] = E[m, j-1] + c_{m,j}$ it follows that after scanning a_j, the value of e is $E[m, j]$. The algorithm is thus correct in reporting a match exactly when $e \le k$ in lines 9 and 10.

Now we analyze the space and time complexity of our algorithm. The characteristic function table CV needs $O(|\Sigma| m/\log n)$ space, since for every symbol we need $O(m/\log n)$ integers. We need $O(m)$ time to precompute the characteristic vectors for a symbol, so the total time to construct the characteristic function table is $O(m|\Sigma|)$. In constructing the transition tables, $GOTO$ and $COUT$, for the universal automaton, each entry needs $O(r) = O(\log n)$ time if computed from scratch. However, once a given entry has been computed, the other 5 entries that differ only in the last characteristic bit and/or region difference can be computed in constant time. Similarly, the 5 entries for which the second-to-last characteristic bit and/or region

```
1.   for p ← 1, 2, ··· m/r do
2.         D[p] ← (1, 1, ··· 1)
3.   e ← m
4.   for j ← 1, 2, ··· n do
5.     {  c ← 0
6.        for p ← 1, 2, ··· m/r do
7.              (D[p], c) ← (GOTO[CV[p, a_j], D[p], c], COUT[CV[p, a_j], D[p], c])
8.        e ← e + c
9.        if e ≤ k then
10.             print "Match at"· j
11.    }
```

Figure 1: An $O(mn/\log n)$ search.

difference is different, can be computed incrementally in 2 steps, and the 30 entries differing in the last part from one of these 6 entries, can be computed incrementally in one step apiece. Continuing this line of reasoning reveals that by computing entries in consecutive order of their integer indices and incrementally computing successive solutions, one can compute each entry in $O(1)$ amortized time. (There is an exact analogy with the observation that counting from 0 to n requires only $2n$ carry operations overall despite that fact that some individual increments take $\log n$ carries.) So the transition tables can be computed in $O(n)$ total time and occupy $O(n)$ space. The total time for scanning a character is $O(m/\log n)$, because each region can be processed in constant time and there are $O(m/\log n)$ regions in each column. The total running time is thus $O(nm/\log n)$ and the total space needed is $O(n + m\,|\Sigma|/\log n)$.

We conclude this section with a very brief sketch of how to handle arbitrarily large alphabet sets. We assume symbols in the alphabet are ordered so that ranges in literals make sense. Suppose there are $p \geq m$ non-meta symbols in the limited expression. Sort the symbols and observe that the characteristic vectors for any symbol between two consecutive symbols in the sorted list are all identical. Thus it suffices to build the characteristic function table for the p symbols in the pattern, and for $p+1$ representative symbols between them in the sorted list. Assuming symbols can be compared in constant time, this preprocessing takes $O(p^2)$ time and the table occupies $O(pm/\log n)$ space. During the search, every time we scan an input character a_j, we first do a binary search against the sorted list of pattern symbols to determine which row of the characteristic function table to fetch characteristic values from. The rest of the algorithm remains the same. Overall, the running time is now $O(p^2 + n\log p + nm/\log n)$.

4. An O(nk/logn) Expected-Time Algorithm

The algorithm from the previous section can be improved by applying our Four Russian's universal automaton to Ukkonen's $O(kn)$ algorithm [Uk85] rather than Seller's $O(mn)$ algorithm [Se80]. Ukkonen's algorithm was only recently proven to take $O(kn)$ expected time under suitable distributional assumptions about the input [CL92]. When extended to limited expressions it is not clear that this performance claim continues to hold. We conjecture that as long as the literals in the expression are such that they match text characters with a probability bounded away from 1, then the performance claim does hold. Of course, the constant of proportionality in the $O(kn)$ claim is dependent on these probabilities: the more likely matches are, the more time that is taken.

Ukkonen improved the expected time of the dynamic programming algorithm by computing only a portion of the dynamic programming matrix. The approach is to maintain, for each column j, the smallest row index x_j such that $E[i,j] > k$ for all $i > x_j$. Note that this implies $E[x_j, j] = k$ by Lemma 3.1. If we know the value of x_j, then we do not need to compute entries with greater row index in column j since they cannot lead to a match. Lemma 3.1 implies that $x_j - x_{j-1} \leq 1$ and that x_j can be computed from x_{j-1}. Notice that it is not sufficient to find the first

E-value in a column that is greater than k, because following values can still be less than or equal to k. What does suffice is to compute all entries in column j through row $x_{j-1}+1$ and then set x_j to the greatest row in that range whose value is k [Uk85].

To compute the relevant regions in blocks of size r requires that we maintain the last region, y_j, in column j containing an entry equal to k. To do so requires two additional pieces of information about states not required in the algorithm of the last section. For a state $s = [d_1, d_2, \cdots d_r]$ of r differences, let $d_1+d_2+\cdots+d_r$ be the *block sum* of s and let $\max(d_{k+1}+d_{k+2}+\cdots+d_r : k \in [1,r])$ be the *maximal suffix sum* of s. Let $BKS[s]$ and $MSS[s]$ be two tables that given a state s return its maximal suffix sum and block sum, respectively. Certainly computing these additional tables during construction of the universal automaton does not increase preprocessing time since the tables are of size $3^r = o(n)$ and take a proportional amount of time to compute incrementally.

With these preliminaries, our refinement of Ukkonen's algorithm is given in Figure 2. The principal invariant is that just before the execution of the loop in lines 6 through 18, (1) $y = y_{j-1}$, (2) $D[p] =_{\uparrow} D_{j-1}\langle p \rangle$ for all $p \in [1,y]$, and (3) $e = \Sigma_{p=1}^{y} BKS[D[p]]$. That is, we know y_{j-1}, the states of the relevant regions in column $j-1$, and the column sum of the differences up to row $y \cdot r$. We must be careful not to assert that $D[p]$ is exactly $D_{j-1}\langle p \rangle$ because the entries above row x_{j-1} are not necessarily correct. However, as in Ukkonen's algorithm these differences are not relevant as long as they imply that the corresponding value in E is greater than k. We signal this by using an $=_{\uparrow}$-sign instead of an equal sign in the assertion about $D[p]$. This small detail also requires that the invariant about e be expressed in terms of the values in D and not those of E. Note however, that when $e \le k$ then it is also true that $e = E[y \cdot r, j]$.

```
1.   y ← ⌈k/r⌉
2.   for p ← 1, 2, ··· y do
3.        D[p] ← (1, 1, ··· 1)
4.   e ← y·r
5.   for j ← 1, 2, ··· n do
6.     { if y < m/r then
7.          { (e, y) ← (e+r, y+1)
8.            D[y] ← (1, 1, ··· 1)
9.          }
10.       c ← 0
11.       for p ← 1, 2, ··· y do
12.            (D[p], c) ← ( GOTO[CV[p, a_j], D[p], c], COUT[CV[p, a_j], D[p], c] )
13.       e ← e + c
14.       while y > 0 and e − MSS[D[y]] > k do
15.            (e, y) ← (e − BKS[D[y]], y−1)
16.       if y = m/r and e ≤ k then
17.            print "Match at" j
18.    }
```

Figure 2: An $O(nk/\log n)$ expected-time search.

The algorithm begins by correctly setting up the invariant for column 0 in lines 1 through 4. So suppose we are about to scan a_j and wish to advance the invariant from $j-1$ to j. In exact analogy with Ukkonen's algorithm, in column j we compute all the regions through $\max(y_{j-1}+1, m/r)$ and then set y_j to the largest region that contains an entry equal to k. To do so, the first step is to set the region $y_{j-1}+1$ of column $j-1$ to $(1, 1, \cdots, 1)$ and to add r to e, if $y_{j-1} < m/r$, in lines 6 through 9. Then in lines 10 through 13, the first $y_{j-1}+1$ regions of column j are computed from those of column $j-1$ in the usual fashion, and the last carryout is added to e so that its value stays equal to $\Sigma_{p=1}^{y} BKS[D[p]]$. The variable y at this point is $y_{j-1}+1$. The loop of lines 14 and 15 decrements y until it equals y_j. The logic of this minor loop is as follows. If e minus the maximal suffix sum of region y is greater than k, than region y does not contain an entry equal to k. Decrement e by the block sum of region y and decrement y by one. Subtracting the block sum from e guarantees that the invariant about it is maintained. Ask the question repeatedly and while it is true continue to decrement e and y until either y becomes 0 or the condition is false. For either termination condition, the region y must contain an entry equal to k and must be the largest such region, i.e., $y = y_j$. This completes the computation of column j. If y_j is m/r and $e \le k$ then it must be that $e = E[m, j]$ and the algorithm correctly reports a match in line 17.

In essence our algorithm exactly mimics Ukkonen's algorithm except that it performs the computation in blocks of size r. Choosing $r = \log_6 n$ as before gives an algorithm that takes $O(n)$ space for the universal tables. Since x_j is $O(k)$ in expectation it follows that y_j is $O(k/\log n)$ in expectation. Thus the search of Figure 2 takes $O(nk/\log n)$ expected-time.

5. Implementation and Experimental Results

We turn our attention now to practice. First we make several small changes in our algorithm to make it faster in practice. Then we report empirical data and compare the resulting implementation to previous algorithms.

In the algorithm of Figure 2, the tables *GOTO* and *COUT* are presumed to be three dimensional arrays indexed by (1) an integer in the range 0 to 2^r-1 representing a characteristic vector, (2) an integer in the range 0 to 3^r-1 representing the current state of a region, and (3) a carry-in which is an integer in the range -1 to 1. Indexing such arrays intrinsically involves computing a linear address with two multiplications. We can remove these by a couple of simple modifications to the preprocessing step. Namely, we transform every state integer s into $3s+1$, and every characteristic-vector integer cv into $cv \cdot 3^{r+1}$. This requires modifying the contents of the tables *GOTO* and *CV*, and since states are now every third integer in the range 1 to $3^{r+1}-2$, the domain of the table *BKS* must be enlarged by a factor of 3. We will not use table *MSS* and the space occupied by *BKS* is negligible compared to *GOTO* and *COUT*, even after the expansion. With this modification, the lookup "$GOTO[cv, s, c]$" becomes "$GOTO[cv+s+c]$" where *GOTO* is now a 1-dimensional array over domain $0 \cdots 3 \cdot 6^r - 1$. We choose this particular translation over the other possibilities because it does not modify carry values, and the domain of *BKS* — and hence the space it occupies — is only expanded by a factor of 3. With this choice, there is never a need to reverse the transformations applied to states and characteristic vectors.

The next improvement is one of rearranging the logic of the algorithm to reduce unecessary computations. Rather than compute regions 1 through $y_{j-1}+1$ in column j, consider computing just those through y_{j-1} initially. y_j can be one greater than y_{j-1} only if the column sum e of column $j-1$ equals k and y_{j-1} is not the last region. This is checked in line 11 of the refinement in Figure 3. If the condition is true, then the advance is made, and one does not bother to whittle regions down to y_j since at most only the region just added will be removed. On the other hand, if the condition is not true, then there is no need to compute an extra region. Instead, one sees if regions can be pealed away to get closer to y_j. Rather than use the table *MSS* to ascertain precisely if there is a value of k-or-less in the top region, we conservatively ask if the column sum is greater than $k+r$. Although this is less precise, it is much more efficient because it removes a table lookup, an addition, and the need to check that y is positive. Note that the algorithm is no longer determining y_j but only an upper bound of y_j. Correctness still follows as the algorithm correctly computes states within the slightly expanded region it explores. Moreover, it was found that the algorithm actually ran faster: the cost of the extra

```
0.  (Y, I, K, R) ← (⌈m/r⌉, 3^(r+1)−2, k+r, r − m (mod r))
1.  y ← ⌈k/r⌉
2.  for p ← 1, 2, ··· y do
3.        D[p] ← I
4.  e ← y·r
5.  for j ← 1, 2, ··· n+R do
6.    {  c ← 0
7.        for p ← 1, 2, ··· y do
8.          {  i ← CV[p, a_j] + D[p] + c
9.             (D[p], c) ← (GOTO[i], COUT[i])
10.         }
11.       if e = k and y < Y then
12.         {  (i, y) ← (CV[y+1, a_j] + I + c, y+1)
13.            (D[y], e) ← (GOTO[i], e+r+COUT[i])
14.         }
15.       else
16.         {  e ← e + c
17.            while e > K do
18.                  (e, y) ← (e − BKS[D[y]], y−1)
19.         }
20.       if y = Y and e ≤ k then
21.            print "Match at" j − R
22.   }
```

Figure 3: The practical refinement of Figure 2.

logic to avoid computing a few regions, outweighed the savings gained.

The final implementation issue is one of detail, rather than efficiency. Consider what happens when r does not divide m. The range of regions naturally runs from 1 to $\lceil m/r \rceil$, where only the first $m \ (mod\ r)$ positions of the last region are directly relevant. When computing characteristic vectors for this last region in the preprocessing step, the "trick" is to always set the last $R = r - m \ (mod\ r)$ entries to 0, which effectively pads the pattern with R don't care symbols. The only difficutly is that the right end of the true match will be reported R characters after the fact. This is easily corrected for in line 21 of our refinement. In addition, the text must also be padded with R don't care symbols in order that matches at its very end are reported.

The algorithm of Figure 3 was coded in C and full advantage was taken of the language's pointer arithmetic capabilities and open interface to the operating system's I/O routines. The algorithms against which we compare ours were also implemented in the same framework and manner, so that idiosyncratic coding differences are minimized. All software was run on a Sun 4/490 with 64 megabytes that is rated at 22mips. The machine was running version 4.1.1 of the SunOS Unix operating system, and all code was compiled with Version 1.1 of Sun's unbundled C compiler, with the optimization option on. The table $COUT$ was coded as an array of **char**, and $GOTO$ and BKS as arrays of **int**, which on a Sun 4/490 translates to $15 \cdot 6^r + 12 \cdot 3^r$ bytes of memory. The table below gives the size and time to generate the tables for r from 1 to 7. They were generated using the incremental approach described earlier.

r	Size (in bytes)	Time (in secs.)
1	126	<.001
2	643	<.001
3	3,564	.001
4	20,412	.005
5	119,356	.031
6	708,588	.31
7	4,225,284	2.1

Despite the fact that size and generation time grow exponentially, observe that even for r as large as 5 or 6, table size is quite reasonable and the generation time is small enough that it is not necessary to have prebuilt tables. They can be constructed from scratch as part of each search without contributing a significant amount of time to the search.

In the introduction we mentioned a dozen algorithms for approximate string matching. They divide into two classes, those that are highly efficient only when k is a sufficiently small percentage of m [CL90,My90,Uk92,WM92], and those that do not have such a limit. Our algorithm is in the later class, so we compare it to the algorithms with the same operational capabilities. Of these the only one we need consider is that of Chang and Lampe [CL92], as they have already demonstrated that their algorithm is superior to those of Seller's [Se81], Ukkonen [Uk85], Landau and Vishkin [LV88], Galil and Park [GP90], Ukkonen and Wood [UW90], and Myers [My86]. We also chose to compare our algorithm to that of Ukkonen [Uk85] as a classic reference against which to calibrate our results.

We performed the following empirical trials. A random sequence of one million characters was generated over alphabets of size 2, 4, 8, 16, and 32. Each character was chosen with equal probability. For each of these five texts, a random pattern of length 300 was generated over the same alphabet, and each algorithm was run for all choices of k from 0 to 10 and every even number between 10 and 40. Note that the length of the pattern is irrelevant except that it was chosen to be large enough to make the probability of matching negligble. Our algorithm was run for each choice of r from 1 to 6. We thus ran 8 programs for 26 choices of k and 5 alphabet sizes, for a total of 1040 data points. For each trial, we also made a separate run that accumulated the average number of three parameters per column or character scanned: (1) rows for [Uk85], (2) states for our algorithm, and (3) runs for [CL92]. We call this statistic the *average column height*; it is the average number of times the inner-most loop of the algorithm is repeated per character scanned. As such the running time is expected to be a linear function of average column height. Indeed, we ran a regression against the 130 timing points for each program and obtained the following results. The regresssions show very high correlations and small estimated error.

Algorithm	Regression Line	Correlation Coefficient	Std. Error of Estimate
Uk85	.599x + 1.42	.999	.317
CL92	.448x + 2.45	.999	.340
WMM(r=1)	.556x + 1.10	.999	.258
WMM(r=2)	.548x + 0.89	.999	.094
WMM(r=3)	.547x + 0.85	.999	.086
WMM(r=4)	.549x + 0.82	.999	.089
WMM(r=5)	.560x + 0.79	.999	.072
WMM(r=6)	.879x − 0.24	.986	.706

Graphs illustrating various aspects of our empirical results are shown in Figure 4. A curve of the time taken as a function of k is plotted for each algorithm in Figures 4a, 4b, and 4c, for alphabets of size 2, 8, and 32, respectively. The times plotted are not the actual system clock times reported, but those obtained by applying the appropriate regression function above to the average column height measured for that run. This effectively smooth system clock fluctuations as column height is very precisely determined during the scan of one million characters. Figure 4d shows the average column height as a function of $|\Sigma|$ for each algorithm when k is fixed at 20. Finally, Figure 4e shows the values of k and $|\Sigma|$ for which our algorithm with $r = 5$ is superior to [CL92].

(a) Times for $|\Sigma| = 2$

(d) Alphabet Dependence with $k = 20$.

(b) Times for $|\Sigma| = 8$

(e) Regions of Superiority

(c) Times for $|\Sigma| = 32$

Figure 4: Timing Curves and Other Experimental Results

First note that none of the plots show a curve for $r = 6$. They were left out because, to our surprise, our algorithm is slower with $r = 6$ then it is with $r = 5$. The explanation is that with $r = 6$ the table is large enough and lookups unlocalized enough that the Sun 4/490's 128 kilobyte cache begins to perform poorly. This is actually quite evident in the slope of the regression fit above. However, for large alphabets we found that there is a much greater locality of table lookups because there is a strong bias towards characteristic vectors with very few zeros. Thus for small alphabets our algorithm was much slower for $r = 6$ than for $r = 5$, but for large alphabets it ultimately becomes faster as originally expected. This behavior is further reflected in the fact that the regression fit for $r = 6$ was weaker than the rest since the population contained trials over the spectrum of alphabet sizes. In light of this peculiarity, we decided that in practice $r = 5$ is probably the best choice for our algorithm, especially since the table is small and takes less than .1 second to generate from scratch.

One observes that our algorithm with $r = 1$ and [Uk85] have nearly identical performance in all cases. As r increases, the curves steadily decrease in slope as expected, with $r = 5$ being the best. Also as expected, the algorithm of Chang and Lampe fares better as alphabet size increases. In Figure 4a, where $|\Sigma| = 2$, their algorithm is beaten by ours with r set at just 2. In Figure 4b, where $|\Sigma| = 8$, their algorithm asymptotic performance is between our algorithm with $r = 3$ and $r = 4$. Finally, when $|\Sigma| = 32$ in Figure 4c, their algorithm ultimately becomes faster than ours with $r = 5$ when k is greater than 25. Note in Figures 4b and 4c that their algorithm's timing curve generally starts with a higher y-intercept but has lesser slope than some curves that it eventually overtakes. For a fixed alphabet size, average column height is a linear function of k. Thus the regression above corroborates the higher intercept and smaller slope observation. What happens as alphabet size increases is that average column height drops more rapidly for their algorithm than for Ukkonen's or ours. This is illustrated in Figure 4d.

With this information at our disposal, we went back to determine the choices of k and $|\Sigma|$ for which our algorithm is faster than Chang and Lampe's. The plot in Figure 4e summarizes our results. Basically, our algorithm is always superior when alphabet size is 16 or less, and when k is 24 or less. There is a small hyperbolic region between these vertical and horizontal lines where our algorithm is superior, and the rest of the parameter space belongs to [CL92].

Finally, we discuss the potential of the Masek and Paterson algorithm. In order for their algorithm to give an $O(log^2 n)$ improvement over Ukkonen's algorithm, the computation must proceed in $r \times r$ squares with tables that are alphabet dependent. Any attempt to make the tables universal, e.g. using r^2 characteristic matrices, leads one back to only an $O(log\ n)$ speedup at the theoretical level. But alphabet dependent tables grow very quickly. Conservatively, each lookup has to produce two integers, one for the bottom and right side of an $r \times r$ square, for a space consumuption of $8 \cdot (3|\Sigma|)^{2r}$ bytes on a machine like the Sun 4/490. For $|\Sigma| = 2$ one can build a 3×3 with 365Kb, and a 4×4 with 14Mb. One can build a 3×3 for $|\Sigma| = 3$ with 4Mb, and for $|\Sigma| = 4$ with 14Mb. For all other alphabet sizes one cannot build greater than a 2×2 without occupying more than 64Mb of memory. But our algorithm with $r = 4$ will certainly perform better in such instances because of its lesser logical complexity. So the only possible cases where the Masek and Paterson approach might give speeds superior to ours are the 4 very special cases above, and even then it is most unlikely because of the increased algorithmic complexity and the caching phenomenon encountered earlier.

References

[AC75]

Aho, A. V., and M. J. Corasick, ‘‘Efficient string matching: an aid to bibliographic search’’, *Comm. of the ACM,* **18** (June 1975), pp. 333–340.

[ADKF70]

Arlazarov, V. L., E. A. Dinic, M. A. Kronrod, and I. A. Faradzev, ‘‘On economic construction of the transitive closure of a directed graph,’’ *Dokl. Acad. Nauk SSSR,* **194** (1970), pp. 487–488 (in Russian). English translation in *Soviet Math. Dokl.,* **11** (1975), pp. 1209–1210.

[BG92]

Baeza-Yates R. A., and G. H. Gonnet, ‘‘A new approach to text searching,’’ *Communications of the ACM* **35** (October 1992), pp. 74–82.

[BM77]

Boyer R. S., and J. S. Moore, ‘‘A fast string searching algorithm,’’ *Comm. of the ACM,* **20** (October 1977), pp. 762–772.

[CL90]

Chang W. I., and E. L. Lawler, ‘‘Approximate string matching in sublinear expected time,’’ *Proc. 31st Symp. on Foundations of Computer Science* (1990), pp. 116–124.

[CL92]

Chang, W. I., and J. Lampe, ‘‘Theoretical and Empirical Comparisons of Approximate String Matching Algorithms,’’ *Proc. 3rd Symp. on Combinatorial Pattern Matching,* Tucson, AZ (April 1992), pp. 172–181.

[CW79]

Commentz-Walter, B, ‘‘A string matching algorithm fast on the average,’’ *Proc. 6th International Colloquium on Automata, Languages, and Programming,* (1979), pp. 118–132.

[GP90]

Galil Z., and K. Park, ‘‘An improved algorithm for approximate string matching,’’ *SIAM J. on Computing,* **19** (December 1990), pp. 989–999.

[KMP77]

Knuth D. E., J. H. Morris, and V. R. Pratt, ‘‘Fast pattern matching in strings,’’ *SIAM J. on Computing,* **6** (June 1977), pp. 323–350.

[LV88]

Landau G. M., and U. Vishkin, ‘‘Fast string matching with k differences,’’ *J. of Computer and System Sciences,* **37** (1988), pp. 63–78.

[MP80]

Masek, W. J., and M. S. Paterson, ‘‘A faster algorithm for computing string edit distances,’’ *J. of Computer and System Sciences,* **20** (1980), pp. 18–31.

[My86]

Myers, E. W., ‘‘Incremental Alignment Algorithms and their Applications,’’ Tech. Rep. 86-22, Dept. of Computer Science, U. of Arizona.

[My90]

Myers, E. W., ‘‘A sublinear algorithm for approximate keywords searching,’’ To appear in *Algorithmica*. Also, Technical report TR-90-25. University of Arizona, Department of Computer Science.

[My92]

Myers, E. W., ‘‘A four-Russians algorithm for regular expression pattern matching,’’ *J. of the ACM,* **39** (1992), pp. 430–448.

[Se80]

Seller P. H., ‘‘The theory and computations of evolutionary distances: Pattern recognition,’’ *J. of Algorithms,* **1** (1980), pp. 359–373.

[Uk85]

Ukkonen E., ‘‘Finding approximate patterns in strings,’’ *J. of Algorithms,* **6** (1985), pp. 132–137.

[Uk92]

Ukkonen E., ‘‘Approximate string-matching with q-grams and maximal matches,’’ *Theoretical Computer Science,* (1992), pp. 191.

[UW90]

Ukkonen, E. and D. Wood, ‘‘Approximate string matching with suffix automata,’’ Technical Report A-1990-4, Department of Computer Science, University of Helsinki, April 1990.

[WF74]

Wagner, R. A., and M. J. Fischer, ‘‘The string to string correction problem,’’ *J. of the ACM,* **21** (1974), pp. 168–173.

[WM92a]

Wu S., and U. Manber, ‘‘Agrep — A Fast Approximate Pattern-Matching Tool,’’ *Usenix Winter 1992 Technical Conference,* San Francisco (January 1992), pp. 153–162.

[WM92b]

Wu S., and U. Manber, ‘‘Fast Text Searching Allowing Errors,’’ *Communications of the*

ACM **35** (October 1992), pp. 83–91.

[WMM92]
S. Wu, U. Manber, and R. E. W. Myers, ‘‘A Sub-Quadratic Algorithm for Approximate Regular Expression Matching,’’ submitted for publication (May 1992) to *J. of Algorithms*.