Tries for Approximate String Matching

H. Shang and T.H. Merrett*

September 8, 1995

Abstract

Tries offer text searches with costs which are independent of the size of the document being searched, and so are important for large documents requiring spelling checkers), case insensitivity, and limited approximate regular secondary storage. Approximate searches, in which the search pattern differs from the document by k substitutions, transpositions, insertions or deletions, have hitherto been carried out only at costs linear in the size of the document. We present a trie-based method whose cost is independent of document size.

*H. Shang and T.H. Merrett are at the School of Computer Science, McGill University, Montréal, Québec, Canada H3A 2A7, Email: {shang, tim}@cs.mcgill.ca

Our experiments show that this new method significantly outperforms the nearest competitor for k=0 and k=1, which are arguably the most important cases. The linear cost (in k) of the other methods begins to catch up, for our small files, only at k=2. For larger files, complexity arguments indicate that tries will outperform the linear methods for larger values of k.

Trie indexes combine suffixes and so are compact in storage. When the text itself does not need to be stored, as in a spelling checker, we even obtain negative overhead: 50% compression.

We discuss a variety of applications and extensions, including best match (for spelling checkers), case insensitivity, and limited approximate regular expression matching.

1 Introduction

The need to find an approximate match to a string arises in many practical problems. For example, if an optical character reader interprets a "D" as an "O", an automatic checker would need to look up the resulting word, say "eoit" in a dictionary to find that "edit" matches it up to one substitution. Or a writer may transpose two letters at the keyboard, and the intended word,

	worst-case run	**preproc. time**	**extra space**	**ref.**
naive	mn			
KMP	$2n$	m	m	[13]
BM	$2n - m + 1$		$\mathcal{O}(m + \lvert\Sigma\rvert)$	[6, 1]
Shift-or	$\mathcal{O}(n)$	$\mathcal{O}(m + \lvert\Sigma\rvert)$	$\mathcal{O}(\lvert\Sigma\rvert)$	[4]
Patricia	$\mathcal{O}(m)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	[10]

Figure 1: Exact Match Algorithms

say "sent", should be detected instead of the error, "snet". Applications occur with strings other than text: strings of DNA base pairs, strings of musical pitch and duration, strings of edge lengths and displacements in a diagram, and so on. In addition to *substitutions* and *transpositions*, as above, errors can include *insertions* and *deletions*.

The approximate match problem in strings is a development of the simpler problem of exact match: given a text, W_n, of n characters from an alphabet Σ, and a string, P_m, of m characters, $m < n$, find occurrences of P in W. Baeza-Yates [2] reviews exact match algorithms, and we summarize in Figure 1.

Here, all algorithms except the naive approach require some preprocessing. The Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), and Shift-or algorithms all preprocess the search string, P, to save comparisons. The Boyer-Moore algorithms are sublinear in practice, and better the bigger m is, but depend on n. The Patricia method builds a trie and is truly sublinear.[1] The preprocessing is on the text, not the search strings, and although substantially greater than for the linear algorithms, need be done only once for a text. Note that tries of size n can be built in RAM in time $\mathcal{O}(n)$, but that on secondary storage, memory differences make it better to use an $n \log n$ method for all practical sizes of trie. So we quote that complexity.

Trie-based methods are best suited for very large texts, which require secondary storage. We emphasize them in this paper, but will compare our trie-based method experimentally with the linear methods.

Approximate string matching adds a parameter to the above, k: the algorithm reports a match where the string differs from the text by not

[1]The term "sublinear" in this literature has two meanings, which we distinguish as *sublinear* and *truly sublinear*. *Truly sublinear* in n means $\mathcal{O}(f(n))$ where f is a sublinear function, *e.g.,* $\log n$ or 1. *Sublinear* means truly sublinear or $\mathcal{O}(n)$ where the multiplicative constant is less than 1.

more than k changes. A change can be a replacement (or substitution), an insertion, or a deletion. It can also be a transposition, as illustrated above. Such operations were formulated by Damerau [8] and the notion of *edit distances* was given by Levenshtein [15]. A dynamic programming (DP) algorithm was shown by Wagner and Fischer [26] with $\mathcal{O}(mn)$ worst case. Ukkonen [24] improved this to $\mathcal{O}(kn)$ (and clearly $k \leq m$) by finding a cutoff in the DP. Chang and Lawler [7] have the same worst case, but get sublinear expected time, $\mathcal{O}((n/m)k \log m))$ and only $\mathcal{O}(m)$ space, as opposed to $\mathcal{O}(m^2)$ or $\mathcal{O}(n)$ for earlier methods. This they do by building a *suffix tree* [27, 16], which is just a "Patricia" trie (after Morrison [19]), on the *pattern* as a method of detecting common substrings. Kim and Shawe-Taylor [12] propose an $\mathcal{O}(m \log n)$ algorithm with $\mathcal{O}(n)$ preprocessing. They generate *n-grams* for the text and represent them as a trie for compactness. Baeza-Yates and Perlberg [5] propose a counting algorithm which runs in time independent of k, $\mathcal{O}(n + R)$, where R is bounded $\mathcal{O}(n)$ and is zero if all characters in P_m are distinct. Figure 2 summarizes this discussion. *Agrep* [28] is a package based on related ideas, which also does limited regular expression matching, *i.e.*, P_m is a regular expression.

(Regular expression matching and k-approximate string matching solve

	worst-case run	**preproc. time**	**extra space**	**ref.**
DP	$\mathcal{O}(mn)$	$\mathcal{O}(m)$	$\mathcal{O}(mn)$	[26]
cutoff	$\mathcal{O}(kn)$	$\mathcal{O}(k)$	$\mathcal{O}(kn)$	[24]
suffix tree	$\mathcal{O}(kn)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$	[7]
n-gram	$\mathcal{O}(m \log n)$			[12]

Figure 2: k-Approximate Match Algorithms

different problems. The problem areas overlap — *e.g.*, P_5 = "`a#a##`", where `#` is a one-place wildcard, can be written as a regular expression, but is also a 3-approximate match — but they do not coincide.)

A recent review of these techniques is in the book by Stephen [23]. Hall and Dowling [11] give an early survey of approximate match techniques. The work is all directed to searches in relatively small texts, *i.e.*, those not too large to fit into RAM. For texts that require secondary storage, $\mathcal{O}(n)$ is far too slow, and we need $\mathcal{O}(\log n)$ or faster methods, as with conventional files containing separate records [17]. The price we must pay is to store an index, which must be built once for the whole text (unless the text changes). If we are interested in the text as an ordered sequence of characters, we must store the text as well, and the index represents an additional storage requirement. If we are interested in the text only for the substrings it contains, as in a dictionary for spelling check, then we need only store the index, and we can often achieve compression as well as retrieval speed.

Tries have been used to index very large texts [10, 18] and are the only known truly sublinear way to do so. Tries are trees in which nodes are empty but have a potential subtree for each letter of the alphabet, Σ, encoding the data (*e.g.*, 0 and 1 for binary tries). The data is represented not in the nodes but in the path from root to leaf. Thus all strings sharing a prefix will be represented by paths branching from a common initial path, and considerable compression can be achieved.[2] Substring matching just involves finding a path, and the cost is $\mathcal{O}(m + \log n)$ plus terms in the number of resulting matches. (The $\log n$ component reflects only the number of bits required to store pointers to the text, and is unimportant.) Regular expression matching

[2]Note that this compression is on the *index*, which may still be larger than the text. Typically, if we index every character in the text, as we do in Section 4, the index will be five times the size of the text. If we index only every word, the index is smaller and compression results.[18] If we do only dictionary searches, as in Section 6, there is great compression.

simulates the regular expression on the trie, [9] and is also fast $\mathcal{O}(\log^m(n)\ n^\alpha)$ where $\alpha<1$.

This paper proposes a k-approximate match algorithm using Damerau-Levenshtein DP on a text represented as a trie. The insight is that the trie representation of the text drastically shortens the DP. A $m \times n$ DP table is used to match a given P_m with the text, W_n. There would have to be a new table for each suffix in W (of length $n, n-1, \ldots$). But the trie representation of W compresses these suffixes into overlapping paths, and the corresponding column *need be evaluated only once.* Furthermore, the Ukkonen cutoff can be used to terminate unsuccessful searches very early, as soon as the differences exceed k. Chang and Lawler [7] showed Ukkonen's algorithm evaluated $\mathcal{O}(k)$ columns, which implies searching a trie down to depth $\mathcal{O}(k)$. If the fanout of a trie is Σ, the trie method needs only to evaluate $\mathcal{O}(k\ |\Sigma|^k)$ DP table entries.

We present this method in terms of full-text retrieval, for which both the index and the text must be stored. In applications such as spelling checkers [14], the text is a dictionary, a set of words, and need not be stored separately from the index. These are special cases of what we describe. In such cases, our method offers negative storage overhead, by virtue of the compression, in addition to the very fast performance.

We compare our work experimentally with *agrep* [28], and show that tries outperform *agrep* significantly for small k, the number of mismatches. Since *agrep* complexity is linear in k, and trie search complexity is exponential in k, *agrep* is expected to become better than tries for large k. Our experiments show that the breakeven occurs beyond the practically important case of $k = 1$. Since the authors of *agrep* compare their work thoroughly with other approximate search techniques [28], we make no other comparisons here.

This paper is organized as follows. The next section introduces Damerau-Levenshtein DP for approximate string matches. Section 3 briefly describes trie data structures, and gives our new algorithm for approximate search on text tries. Then we give experimental results comparing approximate trie methods with *agrep*. Sections 5 and 6 discuss extensions and advanced applications of our method, including the important case of dictionary checking, where we attain both speedup and compression. We conclude and discuss further possible research.

2 Dynamic Programming

Let $P_m = p_1p_2...p_m$ and $W_\ell = w_1w_2...w_\ell$ be a pattern and a target string respectively. We use $D(P_m, W_\ell)$ for *edit distance*, the minimum number of edit operations to change P_m to W_ℓ. Here, an edit operation is either to insert w_j after p_i, delete p_i, replace p_i by w_j, or to transpose two adjacent symbols in P_m. We assume symbols are drawn from a *finite* alphabet, Σ.

Given an example $P_7 = \texttt{exsambl}$ and $W_7 = \texttt{example}$. We have $D(P_7, W_7) = 3$ since changing P_7 to W_7 needs to: (1) delete $p_3 = \texttt{s}$, (2) replace $p_3 = \texttt{b}$ by $w_5 = \texttt{p}$, and (3) add $w_7 = \texttt{e}$ after $p_7 = \texttt{l}$. The edit distance, $D(P_i, W_j)$, can be recursively defined as follows:

$$D = \begin{cases} 0 & i = j = 0 \\ \infty & i < 0 \text{ or } j < 0 \\ \min \begin{pmatrix} D(P_i, W_{j-1}) + 1 \\ D(P_{i-1}, W_j) + 1 \\ D(P_{i-1}, W_{j-1}) + S_{ij} \\ D(P_{i-2}, W_{j-2}) + R_{ij} \end{pmatrix} & \text{else} \end{cases}$$

where $p_i = w_j = \phi$ when $i, j \leq 0$ (the null character), and

$$S_{ij} = \begin{cases} 0 & p_i = w_j \\ 1 & \text{else} \end{cases}, \; R_{ij} = \begin{cases} 1 & p_{i-1} = w_j \wedge p_i = w_{j-1} \\ \infty & \text{else} \end{cases}.$$

To evaluate $D(P_m, W_\ell)$, we need to invoke D four times with both subscripts decreasing by no more than two. Thus, a brute force evaluation must take $\mathcal{O}(2^{\min(m,\ell)})$ calls. However, for $D(P_m, W_\ell)$, there are only $(m+1)\times(\ell+1)$ possible values. DP evaluates $D(P_m, W_\ell)$ by storing each possible D value in a $m\times\ell$ table. Table 1 shows a 3×4 DP table for P_2=`ab` and W_3=`bbc`.

	$w_0 = \phi$	$w_1 = b$	$w_2 = b$	$w_3 = c$
p_0	0	1	2	3
p_1	1	$D(P_1, W_1)$	$D(P_1, W_2)$	$D(P_1, W_3)$
p_2	2	$D(P_2, W_1)$	$D(P_2, W_2)$	$D(P_2, W_3)$

$\Longrightarrow$

	ϕ	b	b	c
$p_0 = \phi$	0	1	2	3
$p_1 = a$	1	1	2	3
$p_2 = b$	2	1	1	2

Table 1: Dynamic Programming

Furthermore, it is not necessary to evaluate every D values (DP table entries). Ukkonen [24] proposed an algorithm to reduce the table evaluations. His algorithm works as follows: Let C_j be the maximum i such that $D(P_i, W_j) \leq k$ for the given j (C_j=0 if no such i). Given C_{j-1}, compute

$D(P_i, W_j)$ up to $i \leq C_{j-1}+1$, and then set C_j to the largest i ($0 \leq i \leq C_{j-1}+1$) such that $D(P_i, W_j) \leq k$. Chang [7] proved that this algorithm evaluates $\mathcal{O}(k^2)$ expected entries. As shown in Table 2, for P_4=`adfd` and W_7=`acdfbdf` of 5×8=40 entries, Ukkonen's algorithm evaluates only 23 entries for k=1.

Ukkonen's algorithm sets $D(P_1, W_0)$=1, $D(P_2, W_0)$=2, and C_0=1 at initial time. It evaluates the first column up to row C_0+1=2. Since the largest entry value of this column is at row 2, it sets C_1=2. Then, it evaluates the second column up to row C_1+1=3. Since the largest entry value of this column is at at row 2, it sets C_2=2. Similarly, it evaluates the third column up to row C_2+1=3 to get C_3=2, the fourth column to get C_4=3, and the fifth column to get C_5=0, which indicates that it is impossible to change any prefix of `adfd` to `acdfb` in less than one edit operation. Thus, we know $D(P_4, W_7)$>1. We can stop the evaluation if we do not want to know the exact value of $D(P_4, W_7)$.

3 Trie and Approximate Search

We follow Gonnet *et al.* [9] in using semi-infinite strings, or *sistrings*. A sistring is a suffix of the text starting at some position. A text consists of many sistrings. If we assume sistrings start at word boundaries, the text, "`echo enfold sample enface same example`," will have six sistrings of this kind. Figure 3 shows these sistrings and an index trie constructed over these sistrings. To make Figure 3 simpler, we truncate sistrings after the first blank. To index full size sistrings, we simply replace leaf nodes by sistring locations in the text. To prevent a sistring being a proper suffix of another, we can append either arbitrary numbers of the null symbol after the text or a unique *end-of-text* symbol. The index trie has many distinctive properties:

- When conducting a depth-first traverse, we not only get all sistrings, but also get them in lexicographical order.

- When searching a string, say `example`, branching decisions at each node are given by each character of the string being sought. As the trie in Figure 3, we test the first letter `e` to get to the left branch, and the second letter `x` to get to the right branch. As a result, search time is proportional only to the length of the pattern string, and independent of the text size.

```
Text:
  echo enfold sample enface same example
```

```
Sistrings:
  echo enfold sample enface same example
  enfold sample enface same example
  sample enface same example
  enface same example
  same example
  example
```

Trie:

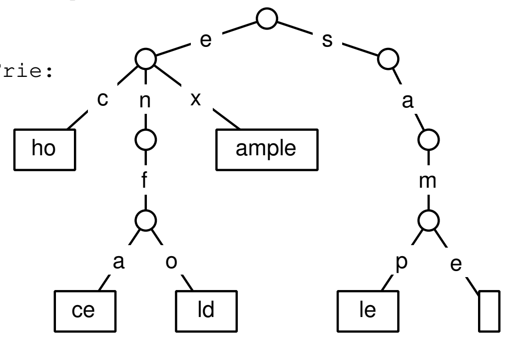


Figure 3: Text, Sistring and Index Trie

- The common prefixes of all sistrings are stored only once in the trie. This gives substantial data compression, and is important when indexing very large texts.

Trie methods for text can be found in [10, 18, 22]. Here we describe them only briefly. When constructing a trie over a large number of and extremely long sistrings, we have to consider the representation of a huge trie on secondary storage. Tries could be represented as trees, with pointers to subtrees, as proposed by Morrison [19], who first came up with the Patricia trie for text searches. Orenstein [21] has a very compact, pointerless representation, which uses two bits per node and which he adapted for secondary storage. Merrett and Shang [18, 22] refined this method and made it workable for Patricia tries with one bit per node. Essentially, both pointerless representations would entail sequential searches through the trie, except that the bits are partitioned into secondary storage blocks, with trie nodes and blocks each grouped into levels such that any level of nodes is either entirely on or entirely off a level of blocks. With the addition of two integers per block, the sequential search is restricted to within the blocks, which may be searched as a tree. For more details of this representation, see [22].

3.1 Two Observations

Before introducing our approximate search algorithm, we give two observations which will link the trie method with the DP technique.

Observation I

Each trie path is a prefix shared by all sistrings in the subtrie. When evaluating DP tables for these sistrings, we will have identical columns up to the prefix. Therefore, these columns need to be evaluated only *once*.

Suppose we are searching for string `sane` in a trie shown in Figure 3. To calculate distances to each word, we need to evaluate six tables. Table 3 shows three of them. For each table, entries of the ith column depend only on entries of the $j \leq i$ th column, or the first i letters of the target word. Words `sample` and `same` have the same prefix `sam`, and therefore, share the table entries up to the third column. And so does the first column of words `echo`, `enface`, `enfold` and `example`, the first three columns of words `enface` and `enfold`. In general, given a path of length x, all DP entries of words in the subtrie are identical up to the xth column.

This observation tells us that edit distances to each indexed word (sistring in general) can be calculated by traversing the trie, and in the meantime, storing and evaluating one DP table. Sharing of common prefixes in a trie structure saves us not only index space but also search time.

Observation II

If all entries of a column are $> k$, no word with the same prefix can have a distance $\leq k$. Therefore, we can stop searching down the subtrie.

For the last table of Table 3, all entries of the second column are > 1.

If searching for words with $k = 1$ differences, we can stop evaluating strings in the subtrie because for sure $D(\texttt{sane}, \texttt{en}...) > 1$. For the same reason, after evaluating the fourth column of table `sample`, we find all entries of the column are > 1, and therefore, stop the evaluation.

This observation tells us that it is not necessary to evaluate every sistring in a trie. Many subtries will be bypassed. In an extreme case, the exact search, all but one of the subtries are trimmed.

3.2 Search Algorithm

The algorithm of Figure 4 shows two functions: *DFSearch*(*TrieRoot*, 1) traverses an index trie depth-first, and *EditDist*(j) evaluates the jth column of the DP table for pattern string P and target string W. For the purpose of illustration, we start and stop evaluation at the word boundary in the following explanation.

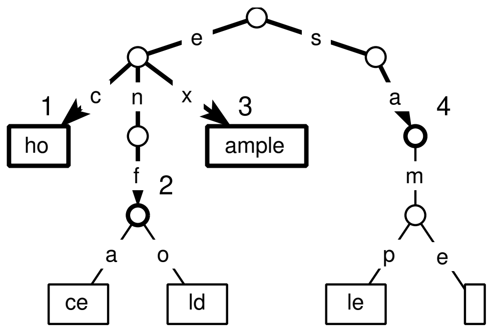
Essentially, this algorithm is a trie walker with cutoffs (rejects before reaching leaves). Given a node c, its root-to-c path, $w_1w_2...w_x$, is a prefix shared by all strings in $SubTrie(c)$. If changing $w_1w_2...w_x$ to any possible prefix of P costs more than k, there will be no string in $SubTrie(c)$ with

```
T   :array [-1..max, -1..max] of integer; /* [i,0] = [0,j] = i+j, [-1,] = [,-1] = ∞ */
C   :array [0..max] of integer;        /* variables for Ukkonen's cutoff, C[0] = k */
P,W :array [0..max] of character; /* pattern and target string, W[0] = P[0] = φ */
k   :integer;                                   /* number of allowable errors */

Procedure DFSearch( TrieNode :Anode, Level :integer);
  begin                                            /* depth-first trie search */
    if (TrieNode in a leaf node) then
      for each character in the node do        /* retrieve characters one by one */
        W[Level] := the retrieved character;
        if (W[Level] = ' ') then                          /* find a target word */
          output W[1]W[2]...W[j-1];
          return;
        if (EditDist( Level) = ∞ ) then                 /* more than k mistakes */
          return;
        Level := Level + 1;
    else
      for each child node do                    /* retrieve child node one by one */
        ChildNode := the retrieved node;
        W[Level] := the retrieved character;
        if (W[Level] = ' ') then                          /* find a target word */
          output W[1]W[2]...W[j-1];
          return;
        if (EditDist( Level) = ∞ ) then /* cut off here! don't search subtrie down */
          return;
        DFSearch( ChildNode, Level+1)                 /* search down the subtrie */
  end;

Function EditDist( j :integer) :integer;
  begin                                      /* evaluate one column of DP table */
    C[j] := 0;
    for i:=1 to Min( C[j-1]+1, length(p)) do
      s := if (P[i]=W[j]) then 0 else 1;           /* evaluate one table entry */
      r := if (P[i-1]=W[j] and P[i]=W[j-1]) then 1 else ∞;
      T[i,j] := Min(T[i,j-1]+1,T[i-1,j]+1,T[i-1,j-1]+s,T[i-2,j-2]+r);
      C[j] := if (T[i,j] <= k) then i else C[j];

    return (if (C[j]=0) then ∞ else T[i-1,j]);
  end;
```

Figure 4: Approximate Trie Search Algorithm

$\leq k$ mismatches. Hence, there is no need to walk down $Subtrie(c)$. A cutoff occurs. Each letter w_j ($1 \leq j \leq x$) on the path will cause a call to $EditDist(j)$. We use Ukkonen's algorithm to minimize row evaluations.

Suppose we have a misspelled word P=`exsample` and want all words with k=1 mismatches. Figure 5 shows the index trie and some intermediate results of the search. After evaluating $D(P,$ `ech`$)$, we find that entries on the third column are all $\geq$2. According to observation II, no word W with the prefix `ech` can have $D(P, W) \leq 1$. We reject word `echo` and continue traversing. After evaluating $D(P,$ `enf`$)$, we know, once again, no word W with prefix `enf` can have $D(P, W) \leq 1$, and therefore, there is no need to walk down this subtrie. We cut off the subtrie. Since `ech` and `enf` share the same prefix `e`, we copy the first column of `ech` when evaluating `enf` (observation I). After evaluating path 3, we find $D(P,$ `example`$) = 1$ and accept the word. The search stops after cutting at path 4, `sa`. Figure 5 shows some intermediate results of the search.



Pattern String:	exsample	$k \leq 1$

Depth First	String	Distance	Action
Search Path 1:	ech	≥ 2	reject
Search Path 2:	enf	≥ 2	cutoff
Search Path 3:	example	$= 1$	accept
Search Path 4:	sa	≥ 2	cutoff

Figure 5: Approximate Trie Search Example

4 Experimental Results

We built tries for five texts: (1) *The King James' Bible* retrieved from akbar.cac.washington.edu, (2) *Shakespeare's complete works* provided by Oxford University Press for NeXT Inc., (3) section one of *UNIX manual pages* from Solbourne Computer Inc., (4) *C source programs* selected randomly from a departmental teaching machine, and (5) randomly selected *ftp file names* provided by Bunyip Information System. Sistrings start at any character except the word boundary, such as blank and tab characters. Table 4 shows the sizes of the five texts and their index tries.

4.1 Search Time

We randomly picked up 5 substrings from each of the five texts, and then searched for the substrings using both *agrep* [28] and our trie algorithm. Both elapsed time and CPU time are measured on two 25MHz NeXT machines, one with 28MB RAM and the other with 8MB RAM. Table 5 shows measured times, averaged on the five substrings, in seconds.

The testing results show that our trie search algorithm significantly outperforms *agrep* in exact match and approximate match with one error. For the exact match, trie methods usually give search time proportional only to the length of the search string. Our measurements show that trie search times for exact match do not directly relate to the text size. It requires few data transfers (only one search path), and therefore, is insensitive to the RAM size.

Let $\rho(k)$ be the average trie search depth. It is the average number of columns to be evaluated before assuring that $D(P, W) > k$. It has been proven that $\rho(k) > k$ if k is less than the target string length, and $\rho(k) = \mathcal{O}(k)$ [24, 7]. For a complete trie, the worst case of a text trie, the trie search algorithm can find all substrings with k mismatches in $\mathcal{O}(k\ |\Sigma|^k)$ expected time: there are $|\Sigma|^k$ paths up to depth k, and each column of the DP table has k rows. The time is independent of the trie size. In fact the trie algorithm is better than the *agrep* for small k, but not for large k, because *agrep* scans text linearly but the trie grows exponentially. For our measured texts, which are relatively small, the trie search brings more data into RAM than *agrep* when $k \geq 2$,

When RAM size is larger than data size, measured CPU times are closer to the elapsed times. Since each query is tested repeatedly, most of data (text and trie) are cached in RAM, and therefore, the searches are CPU-bound.

However, for a smaller RAM size (or larger text data), the searches have to wait for data to be transferred from secondary storage. Since *agrep* scans the entire text, its search time is linearly proportional to the text size.

File names are different from the other tested texts. File names are all pairwise distinct. Any two substrings resemble each other less, which helps *agrep* to stop evaluation more quickly. This does not help the trie search because it makes the trie shallow (toward a complete trie) and takes more time to scan the top trie levels.

5 Extensions

Our trie search algorithm can be extended in various ways. For example, spelling checkers are more likely to ask for the best matches, rather than the words with a fixed number of errors. The optical character recognizers may search for words with substitutions only. When searching for telephone numbers, license numbers, postal codes, etc., users require not only penalties for certain types of edit operations, but also a combination of the exact search and the approximate search because they often remember some numbers for sure. In text searching, patterns are more often expressed in terms of regular expressions. Extensions described in this section (except Section 5.5) have been discussed in [28]. We present them here using DP.

5.1 Best Match

In some applications, we do not know the exact number of errors before a search. We want strings with the minimal number of mismatches, i.e., strings with $0 \leq k$ mismatches and no other string in the text having $k' < k$ mismatches.

To use our algorithm, we define a preset k, which is a small number but no less than the minimal distance, i.e., there exists a string, s, in the text such that $D(\text{pattern}, s) \leq k$. A simple method to set k is to let s be an arbitrary string in the text, and then set $k = D(\text{pattern}, s)$. A better way is to search for the pattern using deletions (or insertions, or substitutions) only. This is to traverse the trie by following the pattern string. Whenever no subtrie corresponds to a character of the pattern, we skip the character in the pattern and look for a subtrie for the next character, and so on. The number of skipped characters will be used as an initial k.

During the traverse, we will have $k' = D(\text{pattern}, s)$ for a leaf node, where

s is the path from the root to the leaf node. Whenever we have $k > k'$, we set $k = k'$ and clear the strings that have been found. For best match searching, k decreases monotonically.

5.2 Weighted Costs

The distances evaluated before are assumed to have cost 1 for any edit operation. Sometimes, we may want to have a different cost. For example, to have substitution costs at least the same as one deletion and one insertion, or to disallow deletions completely.

To make edit operations cost differently, we need only to modify the distance function. Let I, D, S and R be the costs of an insertion, a deletion, a substitution, and a transposition respectively. We assume costs are all > 0. To disallow an operation, say insertions, we set $I = \infty$. As before, $D(P_0, W_0) = 0$ and $D(P_i, W_j) = \infty$ if i or $j < 0$. Otherwise, we redefine $D(P_i, W_j)$ as follows:

$$D(P_i, W_j) = \min \begin{pmatrix} D(P_i, W_{j-1}) + I_{ij} \\ D(P_{i-1}, W_j) + D_{ij} \\ D(P_{i-1}, W_{j-1}) + S_{ij} \\ D(P_{i-2}, W_{j-2}) + R_{ij} \end{pmatrix}$$

Here $I_{ij} = I$, $D_{ij} = D$, and
$S_{ij} = \begin{cases} 0 & p_i = w_j \\ S & \text{else} \end{cases}$, $R_{ij} = \begin{cases} R & p_{i-1} = w_j \wedge p_i = w_{j-1} \\ \infty & \text{else} \end{cases}$.

Furthermore, we may add a cost, C, for changing the case. For example, for case insensitive searches, we set $C = 0$, and for case sensitive searches, we set $C = 1$. We may even disallow case changes by setting $C = \infty$. Let $a \simeq b$ be $a = b$ without checking the case difference, and let $a \sim b$ mean that a and b are of the same case. Now, we define, $C_{ij} = \begin{cases} 0 & p_i \sim w_j \\ C & \text{else} \end{cases}$, and replace:

$$S_{ij} = C_{ij} + \begin{cases} 0 & p_i \simeq w_j \\ S & \text{else} \end{cases},$$

$$R_{ij} = C_{i-1,j} + C_{i,j-1} + \begin{cases} R & p_{i-1} \simeq w_j \wedge p_i \simeq w_{j-1} \\ \infty & \text{else} \end{cases}.$$

The concept of changing cases can be extended even more generally. For example, when searching a white page for telephone numbers, we don't want an apartment number, such as 304B, to be recognized as a telephone number, i.e., do not replace a character unless it is a digit to a digit. For the same reason, we may not want to mix letters, digits and punctuation with each other when searching for license plates, such as RMP-167, or postal codes, such as H3A 2A7. For those applications, we can use above definitions for S_{ij} and R_{ij}, but give a new interpretation of C. We will not elaborate them here.

5.3 Combining Exact and Approximate Searches

We sometimes know in advance that only certain parts of the pattern may have errors. For example, many spelling checkers may give no suggestions for garantee. But suppose we knew the suffix rantee was spelled right. In this case, we want to search part of the pattern exactly. By following *agrep* standards [28], we denote this pattern as ga<rantee>. Characters inside a <> cannot be edited using any one of the four operations.

To support both exact and approximate searches for the same pattern, we need only modify I_{ij}, D_{ij}, C_{ij}, S_{ij} and R_{ij}. Let function $| p_i$ be a predicate that determines whether p_i is a member character inside an exact match <>.

Let function $\perp p_i$ be a predicate that tells whether p_i is the last character inside a `<>`. The new definitions are:

$$I_{ij} = \begin{cases} \infty & \mid p_i \wedge \not\perp p_i \\ I & \text{else} \end{cases}, \quad D_{ij} = \begin{cases} \infty & \mid p_i \\ D & \text{else} \end{cases},$$

$$C_{ij} = \begin{cases} \infty & \mid p_i \wedge p_i \neq w_j \\ C & p_i \not\simeq w_j \vee p_i \simeq w_j \wedge p_i \neq w_j \\ 0 & \text{else} \end{cases},$$

$$S_{ij} = C_{ij} + \begin{cases} 0 & p_i \simeq w_j \\ S & \text{else} \end{cases},$$

$$R_{ij} = C_{i-1,j} + C_{i,j-1} + \begin{cases} R & \mathcal{P} \\ \infty & \text{else} \end{cases},$$

where $\mathcal{P} = (p_{i-1} \simeq w_j \wedge p_i \simeq w_{j-1}) \wedge \not\mid p_{i-1} \wedge \not\mid p_i$.

By above definitions, string `guarantees` also matches `ga<rantee>` with two insertions. To disallow insertions at the end of an exact match, we introduce an anchor symbol, `$` (borrowed from Unix standards). Pattern `ga<rantee>$` means that target strings must have the suffix `rantee`. What needs to be changed is to set $\perp p_i$ *false* when there is a `$` symbol followed p_i, i.e., a pattern looks like `...<...`p_i`>$`. In a similar way, we introduce another anchor symbol, `^`, to prevent insertions at the beginning of an exact match.

For example, ^<g>a<rantee>$ means that target strings must start with the letter g and ended with the suffix rantee. This time, we set $|p_0$ *true*.

5.4 Approximate Regular Expression Search

The ability to match regular expressions with errors is important in practice. Regular expression matching and k-approximate string matching solve different problems. They may overlap but do not coincide. For example, the regular expression a#c, where # is a one-place wildcard, can be written as a 1-approximate match with substitutions and insertions on the second character only. Baeza-Yates [5] proposed an search algorithm for the full regular expression on tries.

In this section, we will extend our trie algorithm to deal with regular expression operators with errors. However, the extension operators work only for single characters, i.e., there is no group operator. For example, we may search for a*b with mismatches, but not (ab)*. Searching tries for the full regular expression with approximation is an open problem.

5.4.1 Alternative Operator

Suppose we want to find all postal codes, `H3A 2A?`, where `?` is either `1, 3`, or `7`. First, we introduce the notation, `[137]` (once again, borrowed from Unix standard), to describe either `1`, `3`, or `7`. Formally, operator `[]` defines a set of alternative characters. Thus, `H3A 2A7` matches pattern `H3A 2A[137]` exactly; while `H3A 2A4` matches the pattern with one mistake.

Substituting one character with a set of allowable characters can be easily achieved by redefining the $=$ and $\simeq$ operators of Section 2 and Section 5.2 respectively. For pattern $P_7 =$`H3A 2A[137]`, we have $p_1 =$`H`, $p_2 =$`3`, ..., and $p_7 =$`[137]`. We define $p_7 = w_j$ as either `1`$= w_j$, or `3`$= w_j$, or `7`$= w_j$. In other words, if p_i is a set of allowable characters, $p_i = w_j$ means w_j matches one of the characters defined by the `[]` operator. $\simeq$ is the case insensitive version of $=$.

As syntactic sugar (Unix standards), we may denote `[a-z]` for all lower case letters, i.e., a range of characters; `[^aeiou]` for anything but vowels, i.e., a complement of the listed characters; and `.` for all characters, i.e., the wild card.

5.4.2 Kleen Star

The kleen star allows its associated characters to be deleted for free, or to be replaced by more than one identical character for free. For example, `ac`, `abc`, `abbc` and `abbbc` all match pattern `ab*c` exactly. `a[0-9]*c` means that an unbounded number of digits can appear between `a` and `c`.

Let function $*p_i$ be a predicate which says there is a Kleen star associated with the pattern character p_i. To support the Kleen star operator, we need only to change I_{ij} and D_{ij}. Remember, p_i* means that we can delete p_i at no cost, and insert any number of $w = p_i$ after p_i at no cost. We now give the new definition as follows:

$$I_{ij} = \begin{cases} \infty & \not{*}p_i \wedge |\, p_i \wedge \not\angle\, p_i \\ C_{ij} & *p_i \wedge p_i \simeq w_j \\ I & \text{else} \end{cases},$$

$$D_{ij} = \begin{cases} \infty & \not{*}p_i \wedge |\, p_i \\ 0 & *p_i \\ D & \text{else} \end{cases}.$$

5.5 Counter

Our algorithm can also be extended to provide counters. Unlike a Kleen star, e.g., `ab*c`, which means that unbounded number of `b`s can appear between `a` and `c`, pattern `ab?c` says that only `ac` and `abc` match exactly. If we want these strings `abbc`, `abbbc`, `abbbbc` and `abbbbbc`, i.e., two to five `b`s between `a` and `c`, we can write the pattern as `abbb?b?b?c`, or `ab{2,5}c` (Unix syntax).

To support counters, we need only to modify D_{ij} since `p?` means character `p` can deleted for free. Let us define a function $?p_i$ which says there is a counter symbol, `?`, associated with the pattern character p_i. The new definition is:

$$D_{ij} = \begin{cases} \infty & \not{?}p_i \wedge \not{*}p_i \wedge \mid p_i \\ 0 & ?p_i \vee *p_i \\ D & \text{else} \end{cases} .$$

6 Dictionary Search

By a dictionary, we mean a text file which contains keywords only, i.e., a set of strings that are pairwise distinguishable. For dictionary searches, we are only interested in those keywords that relate to the pattern by some measurements (in our case, the edit distance). The orders (or locations) of those keywords are not important to us. For such applications, the text file can be stored entirely in a trie structure. The trie in Figure 3 is a dictionary trie. Experimental results in [22] show that dictionary trie sizes are about 50% of the file sizes for English words. In other words, we are providing not only an algorithm for both exact and approximate searches, but also a data structure for compressing the data up to 50%. Searches are done on the structure without decompression operations.

Searching soundex codes [20] is an example of the dictionary search. By replacing English words with their soundex codes and storing the codes in the dictionary trie, we are able not only to search any given soundex code efficiently (exact trie search) but also to reduce the soundex code size by half.

Searching an inverted file is another example of dictionary search. An inverted file is a sorted list of keywords in a text. The trie structure keeps the order of its keys. By storing keywords in the dictionary trie, we can either search for the keywords or for their location. Furthermore, our trie algorithm provides search methods for various patterns with or without mismatches.

7 Conclusion

Tries have been used to search for exact matches for a long time. In this paper, we have expanded trie methods to solve the k approximate string matching problem. Our approximate search algorithm finds candidate words with k differences in a very large set of n words in $\mathcal{O}(k\,|\Sigma|^k)$ expected worst time. The search time is independent of n. No other algorithm which achieves this time complexity is known.

Our algorithm searches a trie depth first with shortcuts. The smaller k is, the more subtries will be cut off. When $k = 0$, all irrelevant subtries are cut off, and this gives the exact string search in time proportional only to the length of the string being sought. The algorithm can also be used to search full regular expressions [3].

We have proposed a trie structure which uses two bits per node and has no pointers. Our trie structure is designed for storing very large sets of word strings on secondary storage. The trie is partitioned by pages and neighboring nodes, such as parents, children and siblings, are clustered in terms of pages. Pages are organized in a tree like structure and are searched in time logarithmic the file size.

Our trie method outperforms *agrep*, as our results show, by an order of magnitude for k=0, and by a factor of 4 for k=1. Only when $k \geq 2$ does the linear worst case performance of *agrep* begin to beat the trie method for the moderately large documents measured.

8 Future Work

Spelling checkers based on searching minimal edit distance performs excellently for typographic errors and for some phonetic errors. For example, `exsample` to `example` has one difference, but `sinary` to `scenery` has three differences. To deal with phonetic misspellings, we may follow Veronis's work [25] by giving weights to edit operations based on phonetic similarity, or using non-integer distances to obtain finer grained scores on both typographic and phonetic similarities. Another solution is to follow the convention which assumes no mistakes in the first two letters, or gives higher penalty for the first few mistakes. Excluding the first few errors allows us to bypass many subtries near the trie root. This not only gives quicker search time, but also reduces the number of possible candidates. With a small set of candidate words, we can impose a linear phonetic check.

Even with one difference, a short word, say of 2 letters, matches many English words. There are more short words than long words. This type of error is difficult to correct out of context.

Acknowledgments

This work was supported by the Canadian Networks of Centres of Excellence (NCE) through the Institute of Robotics and Intelligent Systems (IRIS) under projects B-3 and IC-2, and by the Natural Sciences and Engineering Research Council of Canada under grant NSERC OGP0004365.

References

[1] A. Apostolico. The myriad virtues of suffix trees. In *Combinatorial Algorithms on Words*, pages 85–96. Springer-Verlay, 1985.

[2] R.A. Baeza-Yates. String searching algorithms. In W.B. Frakes and R.A. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, pages 219–40. Prentice-Hall, 1992.

[3] R.A. Baeza-Yates and G.H. Gonnet. Efficient text searching of regular expressions. In G. Ausiello, M. Dezani-Ciancaglini, and S.R.D. Rocca, editors, *Proceedings of 16th International Colloquium on Automata, Languages and Programming*, LNCS 372, pages 46–62, Stresa, Italy, July 1989. Springer-Verlag.

[4] R.A. Baeza-Yates and G.H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[5] R.A. Baeza-Yates and C.H. Perleberg. Fast and practical approximate string matching. In G. Goos and J. Hartmanis, editors, *Proceedings of 3rd Annual Symposium on Combinatorial Pattern Matching*, LNCS 644, pages 185–92, Tucson, Arizona, April 1992. Springer-Verlag.

[6] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–72, 1977.

[7] W.I. Chang and E.L. Lawler. Approximate string matching in sublinear-expected time. In *31st Annual Symposium on Foundations of Computer Science*, pages 116–24, St. Louis, Missouri, October 1990. IEEE Computer Society Press.

[8] F.J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–6, 1964.

[9] G.H. Gonnet. Efficient searching of text and pictures. Technical Report OED-88-02, Centre for the New OED., University of Waterloo, 1988.

[10] G.H. Gonnet, R.A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W.B. Frakes and R.A. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, pages 66–82. Prentice-Hall, 1992.

[11] P.A.V. Hall and G.R. Dowling. Approximate string matching. *Computing Surveys*, 12(4):381–402, 1980.

[12] J.Y. Kim and J. Shawe-Taylor. An approximate string-matching algorithm. *Theoretical Computer Science*, 92:107–17, 1992.

[13] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *Computer Journal*, 6(2):323–50, 1977.

[14] K. Kukich. Techniques for automatically correcting words in text. *Computing Surveys*, 24(4):377–439, 1992.

[15] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Dokl.*, 6:126–36, 1966.

[16] E.M. McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–72, 1976.

[17] T.H. Merrett. *Relational Information Systems*. Reston Publishing Co., Reston, VA, 1983.

[18] T.H. Merrett and H. Shang. Trie methods for representing text. In *Proceedings of 4th International Conference, FODO'93*, LNCS 730, pages 130–45, Chicago, Ill, October 1993. Springer-Verlag.

[19] D.R. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric. *Journal of the ACM*, 15(4):514–34, 1968.

[20] M.K. Odell and R.C. Russell. U.S. Patent Numbers, 1,261,167 (1918) and 1,435,663, 1922. U.S. Patent Office, Washington, D.C.

[21] J.A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–6, 1982.

[22] H. Shang. *Trie Methods for Text and Spatial Data on Secondary Storage.* PhD Dissertation, School of Computer Science, McGill University, November 1994.

[23] G.A. Stephen. *String Searching Algorithms.* Lecture Notes on Computing. World Scientific Pub., 1994.

[24] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–7, 1985.

[25] J. Veronis. Computerized correction of phonographic errors. *Comput. Hum.*, 22:43–56, 1988.

[26] R.A. Wagner and M.J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–78, 1974.

[27] P. Weiner. Linear pattern matching algorithms. In *IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[28] S. Wu and U. Manber. Fast text searching. *Communications of the ACM*, 35:83–91, 1992.

Heping Shang received a B.S. degree in computer engineering from Changsha Institute of Technology, Changsha, Hunan, China, in 1981, an M.S.

degree in computer science from Concordia University, Montréal, Québec, Canada in 1988, and a Ph.D degree in computer science from McGill University, Montréal, Québec, Canada in 1995. His research interests include data structures and searching techniques for very large textual and spatial database data, database programming languages, parallel processing and concurrency control.

Dr. Shang is now at Replication Server Engineering, Sybase Inc., Emeryville, California, USA.

T. H. Merrett received a B.Sc. in mathematics and physics from Queen's University at Kingston, Ontario, Canada (1964) and a D.Phil. in theoretical physics from Oxford University (1968). After two years with IBM (U.K.), he joined the School of Computer Science at McGill University, where he is a professor. His research interests are in database programming languages and data structures and algorithms for secondary storage.

Dr. Merrett initated and directs the Aldat Project at McGill University, which has been responsible for data structures for multidimensional data, such as multipaging and Z-order, and for trie-based structures for text and spatial data. The database programming language contributions of the Aldat

Project have included the domain algebra; quantified tuple (QT)-selectors; relational mechanisms for multidatabases, metadata, and inheritance; methods for process synchronization and nondeterminism; and the *computation* mechanism, which unifies relations, functions, and aspects of constraint programming.

	ϕ	a	c	d	f	b	d	f
ϕ	0	1	2	3	4	5	6	7
a	1	0	1	2	3	4	5	6
d	2	1	1	1	2	3	4	5
f	3	2	2	2	1	2	3	4
d	4	3	3	2	2	2	2	3

$\Longrightarrow$

	ϕ	a	c	d	f	b	d	f
ϕ	0	1	2	3	4	5		
a	1	0	1	2	3	4		
d	2	1	1	1	2	3		
f			2	2	1	2		
d						2		
	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7

Table 2: Ukkonen's Cutoff

	ϕ	s	a	m	p	l	e
ϕ	0	1	2	3	4	5	6
s	1	0	1	2	3	4	5
a	2	1	0	1	2	3	4
n	3	2	1	1	2	3	4
e	4	3	2	2	2	3	3
column:		1	2	3	4	5	6

	ϕ	s	a	m	e
ϕ	0	1	2	3	4
s	1	0	1	2	3
a	2	1	0	1	2
n	3	2	1	1	2
e	4	2	2	2	1
		1	2	3	4

	ϕ	e	n	$f \cdots$
ϕ	0	1	2	$3 \cdots$
s	1	1	2	
a	2	2	2	
n	3	3	2	
e	4	3	3	
		1	2	$3 \cdots$

Table 3: Dynamic Programming Tables

Text	Text Size	#Sistrings	#Trie Nodes
Bible	4.5MB	3.4 M	46.0 M
Shakespeare	6.4MB	4.4 M	47.9 M
Unix Manual	7.6MB	4.8 M	107.0 M
C Program	8.4MB	5.3 M	157.0 M
File Names	8.4MB	6.6 M	76.9 M

Table 4: Text File and Index Trie

	Text	NeXT with 28MB RAM elapsed (CPU), *sec.*		NeXT with 8MB RAM elapsed (CPU), *sec.*	
		agrep	trie	agrep	trie
$k = 0$	Bible	4.45 (4.32)	0.68 (0.43)	5.98 (4.57)	0.82 (0.43)
	Shakespeare	7.90 (7.76)	0.63 (0.41)	17.50 (9.53)	0.90 (0.42)
	Unix Manual	7.53 (7.43)	1.07 (0.58)	18.72 (9.51)	1.37 (0.58)
	C Program	12.63 (12.50)	0.68 (0.35)	24.13 (14.62)	0.85 (0.37)
	File Names	5.80 (5.68)	0.53 (0.38)	16.82 (7.43)	0.75 (0.37)
$k = 1$	Bible	7.48 (7.37)	2.78 (2.67)	8.58 (7.48)	2.77 (2.55)
	Shakespeare	13.52 (13.37)	2.78 (2.67)	23.53 (15.16)	8.42 (8.20)
	Unix Manual	28.48 (28.29)	4.42 (4.32)	39.58 (30.20)	4.15 (3.90)
	C Program	22.18 (21.93)	4.63 (4.49)	34.08 (23.95)	8.25 (4.68)
	File Names	9.10 (8.86)	7.17 (7.05)	21.07 (11.34)	13.63 (7.48)
$k = 2$	Bible	13.53 (13.21)	22.52 (22.19)	16.42 (13.51)	40.12 (24.32)
	Shakespeare	24.83 (24.50)	28.57 (28.16)	33.90 (26.40)	66.18 (32.93)
	Unix Manual	46.50 (45.87)	41.63 (40.91)	57.22 (47.58)	80.12 (44.17)
	C Program	35.83 (35.40)	62.87 (61.40)	47.87 (37.41)	138.75 (67.59)
	File Names	14.22 (13.77)	98.00 (97.41)	36.40 (16.42)	176.53 (99.20)

Table 5: Approximate Search Time