Finding Approximate Matches in Large Lexicons

JUSTIN ZOBEL
Department of Computer Science, RMIT, P.O. Box 2476V, Melbourne 3001, Australia.
Email: jz@cs.rmit.oz.au

AND

PHILIP DART
Department of Computer Science, The University of Melbourne, Parkville 3052, Australia.
Email: philip@cs.mu.oz.au

SUMMARY

Approximate string matching is used for spelling correction and personal name matching. In this paper we show how to use string matching techniques in conjunction with lexicon indexes to find approximate matches in a large lexicon. We test several lexicon indexing techniques, including n-grams and permuted lexicons, and several string matching techniques, including string similarity measures and phonetic coding. We propose methods for combining these techniques, and show experimentally that these combinations yield good retrieval effectiveness while keeping index size and retrieval time low. Our experiments also suggest that, in contrast to previous claims, phonetic codings are markedly inferior to string distance measures, which are demonstrated to be suitable for both spelling correction and personal name matching.

KEY WORDS Pattern matching string indexing approximate matching compressed inverted files Soundex

INTRODUCTION

Approximate string matching is used when a query string is similar to but not identical with desired matches.[1,2,3] In personal name matching, a name might be known by its pronounciation rather than its spelling, and data entry errors can lead to the same name being entered into a database in several forms. In text processing, words can be misspelt; approximate matching can Be used to find the correct form in a dictionary.

When using words for retrieval in document database systems, both personal name matching and spelling correction can be required on the same data. The exact spelling may be unknown, spelling of a given word can vary, or there may simply be no single accepted spelling. In the TREC database,[4] for example, there are at least six "correct" spellings of Gorbachev, not including the feminine forms of these spellings. Spelling in stored documents can be unreliable; we estimate that around one-fifth of the distinct words in an electronic form of the Commonwealth Acts of Australia are spelling errors. Moreover, there is no reliable way to distinguish between names and other words—the stored documents are English text, without fields or other meta-information to distinguish one kind of word from another.

In this paper we consider how, given a query string, to find approximate matches in a large lexicon. Given that a complete search of a lexicon is prohibitively expensive, we propose that some form of

Received May 1994
Revised October 1994

indexing be used to extract likely candidates from the lexicon, in a *coarse* search, and that, once retrieved, these candidates be processed more carefully in a subsequent *fine* search. How best to combine indexing and approximate matching methods into coarse and fine searching techniques is the principal problem addressed in this paper.

There are two broad classes of schemes for approximate string matching that could be used for fine searches: string similarity measures and phonetic coding. String similarity measures compute a numerical estimate of the similarity between two strings; such computation might be based on the number of characters they have in common, or the number of steps required to transform one into another. These measures are often referred to as *edit distances*.[2] These measures can be used to rank a set of strings—that is, a *lexicon*—with respect to a query string. Such schemes are generally regarded as appropriate to spelling correction, where around 80% of human errors are a single insertion, omission, or exchange.[2,3] Phonetic codings, on the other hand, assign a phonetic code to each string; two strings are judged to be similar if they have the same code, and dissimilar otherwise. Phonetic schemes have been regarded as appropriate to personal name matching because it is possible for names that sound similar to have very different written forms.[1,5]

There are also several candidate coarse search schemes. One scheme would be to bucket the lexicon strings according to their phonetic code, and retrieve the bucket with the same code as the query string. Another scheme is the use of n*-grams*, that is, indexing each string in a lexicon according to the n character substrings it contains. Yet another scheme, which to our knowledge has not previously been applied to approximate matching, is to "permute" a lexicon by adding to it every rotation of every word, and find answers by binary search in the lexicon that results.

We evaluated several combinations of coarse and fine search techniques with regard to speed, space overhead, and effectiveness—that is, ability to find answers that a human judges to be correct. The techniques were selected as representative rather than exhaustive. The evaluation was based on two test lexicons: a large set of personal names and a publicly available dictionary. Surprisingly, we found edit-distance techniques to be far more effective (that is, better able to identify good matches) than phonetic techniques for both spelling correction and personal name matching, with at best only one answer in four correct, compared to one in two for edit distances. With regard to performance, n-grams work well for both coarse and fine searching, and would usually be preferable to permutation because they have better update characteristics. A third lexicon, of a 3 Gb text database, was used to test whether our preferred methods scale up, with excellent results.

TEST DATA

Three test lexicons were used in the approximate string matching experiments described in the following sections. The first, DICT, is the dictionary distributed with the *ispell* interactive spelling checker, which is publicly available on Internet. Our version of DICT contains 113,212 words, at an average of 9.1 letters per word, and so occupies 1,114 Kb including a one-byte terminator for each string. Although DICT is not perfect—it contains some spelling errors and there are surprising omissions—it is large, mostly correct, and contains the plural form and other variants of most words.

The second lexicon, NAMES, is a set of distinct personal names compiled from student records and a bibliographic database. NAMES contains 31,763 words, at an average of 6.8 characters per word, and so occupies 242 Kb including a one-byte terminator for each string. Most of the entries in NAMES are surnames, but, due to data entry errors in the original databases, there are many given names and a fair number of misspelt names. Some of the data was supplied without indication of case or spacing, so for uniformity we suppressed spaces and diacritical marks and folded all uppercase letters to lowercase; for example, "Van Der Haufen" was transformed to "vanderhaufen".

The third lexicon, TREC, is the distinct words occurring in the 3 Gb TREC document collection,[4] a heterogeneous set of newspaper articles, regulations, journal articles, and patent applications. This lexicon consists of 1,073,727 words and occupies 9,933 Kb, at an average of 9.3 characters per string including a terminator.

Note that, in the context of database systems, lexicons with errors are the norm—it is not common for control to be exercised over the words entered into a database. Moreover, it is often impossible to even distinguish between names and other text; consider for example the lexicon of a database of newspaper articles.

To compare the performance of the different matching techniques we decided to use test query sets and relevance judgements to determine retrieval effectiveness. This methodology is widely used in information retrieval for the similar task of evaluating methods for identifying the closeness of correspondence between documents and a query,[6] and has been used to evaluate approximate string matching methods.[2,5] Relevance judgements are a human assignment of a binary value ("match" or "doesn't match") to each "query, potential match" pair; in the context of approximate string matching, every lexicon entry is a potential match. For a given matching technique, relevance judgements are used to compute *recall* (the proportion of correct matches that have been identified) and *precision* (the proportion of correct matches amongst the retrieved strings). For example, to a surname database the query `person` might yield the answers `pierson`, `persson`, `persoon`, `persin`, `perron`, `pehrson`, and `pearson`. If four of these (the second, fourth, six, and seventh, say) sound sufficiently similar to be judged relevant, precision is 57%; if a further six answers are known to be in the database, recall is 40%. Recall and precision figures are combined into a single measure of effectiveness by averaging precision at 0%, 10%, $\ldots$, 100% recall.[6] Loosely speaking, an effectiveness of 50% means that every second returned answer is correct.

The test query set used for DICT was the Kukich set of 170 human generated spelling errors.[3] The answers are the correct spelling corresponding to each of these errors. The test query set used for NAMES was a random selection of 48 of the names. Because we did not have the resources to manually compare every query term with every entry in NAMES, for each query term we merged the output of several matching techniques and manually evaluated the top 200 responses. We found an average of 6.4 matches per query.

For TREC, we did not have the resources needed for the necessary relevance judgements, and so cannot estimate effectiveness for this collection. We can, however, examine space and cpu time, using the same query set as for NAMES.

APPROXIMATE STRING MATCHING

In this section we review approximate matching techniques and experimentally compare them on our test databases.

String similarity measures

One class of approximate string matching techniques are the string similarity measures. A simple form of string similarity measure is an edit-distance such as the number of single character insertions and deletions needed to transform one string into another. For two strings s and t of length m and n respectively, the minimal such edit distance can be determined by computing $\mathit{edit}(m, n)$ with the recurrence relation shown in Figure 1, in which the function $d(a, b)$ returns 0 if a and b are identical, and 1 otherwise.[2] For example, the edit distance between `hordes` and `lords` is 2, and the distance between `water` and `wine` is 3.

There are also more sophisticated versions of edit distance that consider moves, operations on blocks of letters, and allocation of different costs to each kind of transformation.[2,7] For example, if the expres-

$$\begin{aligned}
\mathit{edit}(0,0) &= 0\\
\mathit{edit}(i,0) &= i\\
\mathit{edit}(0,j) &= j\\
\mathit{edit}(i,j) &= \mathit{min}[\mathit{edit}(i-1,j)+1,\\
&\qquad\quad \mathit{edit}(i,j-1)+1,\\
&\qquad\quad \mathit{edit}(i-1,j-1)+d(s_i,t_j)]
\end{aligned}$$

Figure 1. Recurrence relation for minimal edit distance

sion

$$\mathit{edit}(i-2,j-2)+d(s_i,t_{j-1})+d(s_{i-1},t_j)+1$$

is added as an argument to the call to *min*, then the resulting *modified edit* allows for transposition of characters with a cost of 1, the same cost as an insertion, deletion, or substitution. This modified *edit* can be regarded as more powerful than the original, but is more expensive to compute.

A related class of technique is used by the approximate string matchers, such as *agrep*,[7] which report strings as matching if they are identical but for at most K errors, where K is specified by the user. That is, matching is binary, with no ranking of the returned strings.

Other distance measures are based on n-gram similarities, where an n-gram of a string s is any substring of s of some fixed length n. A simple such measure is to choose n and count the number of n-grams two strings have in common. That is, the similarity between strings s and t is given by

$$\mathit{gram\text{-}count}(s,t)=|G_s\cap G_t|\,,$$

where G_x is the set of n-grams in string x. For example, with $n=2$, the similarity of `hordes` and `lords` is 2, because both contain the 2-grams `or` and `rd`, and the similarity of `water` and `wine` is 0. At first glance it might seem that this measure ignores the importance of character ordering within strings, but since the majority of words and names do not contain repeated characters, the importance of ordering is largely retained. For larger n, ordering is more strongly enforced; few words, for example, would contain `bes` and `est` but not `best`.

However, simply counting n-grams does not allow for length differences between strings; for example, `water` has exactly as many n-grams in common with itself as it does with `waterline`. To address this problem, Ukkonen[8] has proposed an n-gram distance that can be defined as

$$\mathit{gram\text{-}dist}(s,t)=\sum_{g\in G_s\cup G_t}|s[g]-t[g]|\,,$$

where $x[g]$ is the number of occurrences of n-gram g in string x. For example, *gram-dist*(`hordes`, `lords`) is 5 if n is 2 or 3.

Ukkonen's function allows for strings to contain repeated occurrences of the same n-gram; however, in our test data sets, less than 2% of strings contain a repeated 2-gram and almost none contain a repeated 3-gram. Assuming that neither s nor t contains a repeated n-gram, Ukkonen's *gram-dist* function can be expressed as

$$\mathit{gram\text{-}dist}'(s,t)=|G_s|+|G_t|-2|G_s\cap G_t|\,.$$

As shown later, this form is useful for retrieval, as $|G_s\cap G_t|$ can be computed from an inverted index and $|G_s|$ is a trivial function on the length of s. Even where a string contains a repeated n-gram, the error introduced by this approximation is small.

Readers who are familiar with information retrieval may wonder whether document ranking techniques, such as the cosine measure,[6] can be applied to the problem of approximate matching with n-grams. Although these problems are superficially similar, there is an important respect in which they differ: the more effective document ranking techniques factor out document length, to allow documents of very different lengths to be ranked as similar. For words, this behaviour is undesirable. The cosine measure would, for example, regard a word as being as similar to any one of its n-grams as it is to itself.

Phonetic methods

The other major class of approximate string matching schemes are the phonetic methods. Of these, by far the oldest is the *Soundex* method of Odell and Russell, first patented in 1918.[2] *Soundex* uses codes based on the sound of each letter to translate a string into a canonical form. These codes are shown in Figure I. The *Soundex* algorithm itself, in Figure 2, transforms all but the first letter of each string into the code, then truncates the result to be at most four characters long.

Code	Characters
0	a e h i o u w y
1	b f p v
2	c g j k q s x z
3	d t
4	l
5	m n
6	r

Table I. *Soundex* phonetic codes

For example, `king` and `khyngge` reduce to `k52`, but `knight` reduces to `k523` while `night` reduces to `n23`—strings that are more different than the original forms. Conversely, very different strings can transform to the same code, such as `pulpit` and `phlebotomy`.

1. Replace all but the first letter of s by its phonetic code.
2. Eliminate any consecutive repetitions of codes.
3. Eliminate all occurrences of code 0 (that is, eliminate vowels).
4. Return the first four characters of the resulting string.

Figure 2. The Soundex algorithm

The *Soundex* algorithm is designed to bucket together names of similar pronounciation,* but, as the above examples show, is fairly crude, and for this reason adaptations to *Soundex* have been proposed. One of the more ambitious *Soundex* variants is Gadd's *Phonix* algorithm.[9,10]

* Presumably to simplify management of personnel records in a filing cabinet, in which the time required for any kind of comprehensive search is prohibitive.

Phonix is similar to *Soundex* in that letters are mapped to a set of codes, but, prior to this mapping, letter-group transformations are applied. For example, `gn`, `ghn`, and `gne` are mapped to `n`, the sequence `tjV` (where `V` is any vowel) is mapped to `chV` if it occurs at the start of a string, and `x` is transformed to `ecs`. Altogether about 160 of these transformations are used. These transformations provide a certain degree of context for the phonetic coding and allow, for example, `c` and `s` to be distinguished, which is not possible under *Soundex*. The *Phonix* codes are shown in Figure II. The first stage of the *Phonix* algorithm—the letter-group substitutions—can be regarded as a separate algorithm, and is included as *partial Phonix* in our experiments.

Code	Characters
0	a e h i o u w y
1	b p
2	c g j k q
3	d t
4	l
5	m n
6	r
7	f v
8	s x z

Table II. *Phonix* phonetic codes

As originally described, *Phonix* is expensive to evaluate, as each of the letter-group transformations are applied in turn. There are however close approximations to *Phonix* that can be evaluated much more efficiently.[5]

There are other techniques for approximate string matching,[2] as well as spelling and OCR correction techniques[3] and name matching techniques.[1] However, the techniques described in this section are typical of the techniques that apply to approximate matching of queries against a large lexicon.

Effectiveness

The effectiveness of the approximate string matching techniques described above can be compared for NAMES and DICT, the test collections for which we have queries and relevance judgements. We compared effectiveness by using each technique to order the whole of each collection with respect to each query, then compute recall-precision figures for each ordering.

The results are given in Table III. Each recall-precision figure is an average across the set of queries. The lines *edit*, modified *edit*, *gram-dist*, and *gram-count* are as described above, and as can be seen the first three are about equally effective, with *edit* slightly better for name matching and the other two better for spelling correction. Although *gram-count* is less effective, its performance is at least acceptable.

The approximate matching utility *agrep* was also tested, by applying it to the test files with combinations of parameters and selecting the best results. This utility does not rank answers and therefore cannot be subjected to quite the same experimental methodology as the edit distances. Results are shown in the table. Performance is not particularly good, but it should be noted that *agrep* is not designed for this application.

The lines labelled *Soundex* and *Phonix* are the standard definitions of these methods as given above.

Technique	NAMES	DICT
	Precision (%)	
edit	63.7	39.9
Modified *edit*	61.2	45.5
gram-dist	61.5	45.5
gram-count	55.9	21.7
agrep	32.8	21.5
Soundex	27.4	4.5
Phonix	25.0	3.3
Modified *Soundex*+*edit*	32.9	9.1
Partial *Phonix*+*edit*	60.7	36.1

Table III. Recall-precision of approximate string matching techniques

Both are practically useless for spelling correction; but this is not so surprising, as they are not designed for this task. More seriously, both are also poor at personal name matching. Curiously, for both NAMES and DICT we found *Phonix* to be worse than *Soundex*, despite the fact that it is some seventy years more recent and is explicitly designed as a *Soundex* replacement.

We believe that *Soundex* and *Phonix* are unlikely to perform well on any test set. The notion of phonetic coding is attractive because distinct letters can conflate to the same sound, but these schemes allocate the same code to letters that can represent different sounds; and even where the letters of a code sound similar, that does not imply that words containing them are likely to be confused with one another—consider `mad` and `not`, for example. Moreover, in our test collections we have found very few cases of words with the same pronounciation but no common spelling, but many cases of words with the same pronounciation and similar spelling; so that, statistically at least, we would expect edit-distance methods to perform better than phonetic methods.

We also experimented with a variant of the *Soundex* algorithm, in which the codes were not truncated to four letters, but instead kept at their original length. In conjunction with *edit*, this method provides a mix of similarity methods, but is significantly worse than *edit* applied to the original strings.

Similarly, partial *Phonix* with *edit*—that is, using the letter-group transformations of phonix then comparing strings with the *edit* function—is less effective than using *edit* alone. Experience with English suggests that there are circumstances in which the letter-group transformations would bring together different spellings of names with the same pronounciation, but our experiments indicate that such instances are swamped by cases in which the letter-group transformations introduce noise.

Throughout the rest of the paper we concentrate on *edit*, *gram-count*, *gram-dist*, *Soundex*, modified *Soundex*, and partial *Phonix*, as they are representative of the different techniques discussed in this section. It would be possible to incorporate the *agrep* string comparison technique as a fine search mechanism, but due to its relatively weak performance as a ranking tool we did not pursue this option; we do however consider its performance in the context of the coarse search schemes.

COARSE SEARCH TECHNIQUES

Where approximate matches to a query term are to be found in a large lexicon, some form of indexing must be used. Without indexing, a complete search of the lexicon is required for each query term, an impractical solution in most situations. An index is a fixed assignment of a binary value to each "property, item" pair, indicating whether the indexed item possesses the property. For example, the item `water` includes amongst its properties that it has *Soundex* code `w36`, contains 2-gram `at`, and is at edit-distance 3 from the item `wine`. It is unlikely to be feasible to index each term in a lexicon according to its edit-distance from every possible string. However, other methods—*Soundex* codes, n-grams, and generalisation of n-grams to arbitrary length substrings—present possibilities for indexing items in a lexicon, and are thus candidate coarse search techniques.

Throughout this paper, index sizes are shown as a percentage of the original lexicon size, so that a 200 Kb index for NAMES would have size 82.6%; the CPU times are average per query, on a Sun Sparc 10 Model 512; and effectiveness is average per query. In the experiments, the lexicon was in lexicographical order, the top 50 words were returned in response to each query, and the lexicon was held in memory. Holding the lexicon in memory is desirable because it improves document retrieval efficiency and permits high-performance compression of stored documents.[11,12] We consider the consequences of storing the lexicon on disk later.

N-gram indexing

Several authors have considered the use of n-grams for lexicon indexing.[8,13,14,15,16] Zobel, Moffat, and Sacks-Davis have described an n-gram lexicon indexing scheme, based on compressed inverted files,[17] that is both fast and compact.[18] In this scheme, an ordinal number is associated with each string in the lexicon, and associated with each n-gram is a postings list, that is, a list of the ordinal numbers of the words containing that n-gram. The compression reduces the size of the index from around three times the size of the indexed lexicon to around half to twice the size of the indexed lexicon, depending on choice of n. To find the strings that match a pattern, the postings lists for the n-grams in the pattern are retrieved and intersected—matching strings must contain all of the pattern's n-grams. Some of the retrieved strings will be false matches, which can be eliminated by direct comparison with the pattern.

The same n-gram index can be used for approximate string matching, but a different query evaluation mechanism must be used: some matches won't contain all of the n-grams in the query string, so strings containing any of the query n-grams should be considered. Thus, to find matches, the union of the postings lists of the query n-grams is required. This union process should, for each string s, count how many of the postings lists refer to s—that is, computing *gram-count*(q, s), where q is the query string. For our data sets and $n = 1$, the most efficient structure for holding these counts is an array with one element per lexicon entry, since, for our test data, on average over half of the lexicon is contained in at least one of the query's n-grams. For the more practical cases of $n \geq 2$, a dynamic structure such as a hash table is preferable.

To increase precision, an extra n-gram is created for the start and end of each word; for example, the 3-grams of `water` are `|wa`, `wat`, `ate`, `ter`, and `er|`.

Once the *gram-count*(q, s) values have been accrued, a structure such as a heap can be used to identify the $A = 50$ strings with highest count, for presentation to the user. The performance of this strategy is shown in the first block of Table IV.† These results show a clear trade-off between index size and speed, and, to a lesser extent, speed and effectiveness. The decrease in time for large n is because inverted file entries become substantially shorter, from an average of thousands of identifiers per entry for $n = 2$ to an average of tens of identifiers per entry for $n = 4$. That the lexicons were sorted has, for approximate matching, a minimal effect on evaluation time, but, as a consequence of the compression regime, does reduce index size by 25%–35%.

Note that the sizes of n-gram indexes include three components: the postings entries; the n-grams, and pointers from them to the postings entries; and an array of pointers mapping each ordinal string

† Speed improvements of over 30% above those achieved here have been measure for an implementation of this indexing scheme incorporated into the *mg* database system,[19] indicating that our results too could be improved with a production implementation.

Technique	NAMES			DICT		
	Index size (%)	CPU (msec)	Effective (%)	Index size (%)	CPU (msec)	Effective (%)
n						
2	94.3	57.5	55.5	72.2	174.2	22.2
3	137.6	21.5	50.4	92.5	79.4	19.9
4	242.2	16.9	47.5	127.2	66.1	16.3
answers ($n = 2$)						
10	94.3	57.5	51.3	72.2	171.4	21.7
100	94.3	60.0	56.1	72.2	163.1	21.3
1,000	94.3	111.3	55.9	72.2	228.8	21.4
10,000	94.3	542.9	55.9	72.2	888.5	21.4

Index size: as percentage of original lexicon size.

Table IV. Performance of n-gram coarse search strategies

number to a string location.

The times achieved compare well to those of the utility *agrep*, which, for the options that achieved the best effectiveness, required an average of 240 msec on NAMES and 1,860 msec on DICT to search the files for matches. These times were achieved, however, without the use of indexes.

We considered several refinements to the basic n-gram algorithm. The number of answers returned has a pronounced effect on both speed and effectiveness. A small number of matches can usually be returned very quickly indeed, because the cost of sorting the heap becomes insignificant. On the other hand, for a large number of matches the sorting cost can dominate. This is illustrated in the second block of Table IV, in which we vary the number of answers returned for 2-grams.

Owolabi and McGregor have also described approximate string matching with n-gram indexing, using a form of signature file.[16] We have experimented with our techniques on the hardware they used (a Sun 3) and appear to have markedly better performance, but note that Owolabi and McGregor do not indicate how many matches are returned.

We also considered two optimisations shown to be valuable in the context of exact pattern matching.[18] First, to save processing time, once the number of candidate strings had fallen below a fixed threshold the longer postings lists were not processed—thus increasing the number of false matches but potentially decreasing costs overall, because for exact string matching the cost of false match checking is low and no sorting is required. Second, to save index space, strings were grouped into blocks that were indexed as if they were a single string, again increasing the number of false matches. This optimisation can also reduce processing time, as the length (and therefore decompression cost) of each postings list is reduced. We have conducted preliminary experiments applying both of these optimisations to approximate pattern matching, but neither performed satisfactorily.

Permuted lexicons

The n-gram indexing technique is a form of fixed-length substring search; other structures provide access to lexicons via substrings of arbitrary length. One such structure is the "permuted lexicon" scheme,[20] in which the lexicon is extended to contain the rotated form of every word. For example, the rotations of `wine` are `wine|`, `ine|w`, `ne|wi`, `e|win`, and `|wine`. An efficient representation of the permuted lexicon is an array of pointers into the original lexicon, one pointer to each character position (including the one-byte string terminators). A pointer into the middle of a word is interpreted as a pointer to a rotated form of the word; for a example, a pointer to the `n` in `wine` denotes the string `ne`|`wi`. The array of pointers is sorted on the rotated forms, allowing patterns to be found by binary search. The disadvantages of this scheme are that the lexicon itself must be held in memory, to allow binary search, and that the lexicon should be static, because sorting the array of pointers is expensive. However, the use of binary search means that this scheme should be fast.

We propose that a permuted lexicon be used to find approximate matches as follows. Binary search can be used to find the locations of the $|q| + 1$ rotations of query string q, then the strings in the neighbourhood of these locations can be returned as matches. For example, the immediate neighbours of rotations of `wine` include rotations of `waistline` and `swine`. Strings should be weighted according to their proximity to each of the locations. The performance of this scheme is shown in the second block of Table V. As can be seen, indexes are large and effectiveness is somewhat low, but matches are found quickly.

NAMES			DICT		
Index size (%)	CPU (msec)	Effective (%)	Index size (%)	CPU (msec)	Effective (%)
237.5	2.1	44.5	262.6	2.6	8.1

Index size: as percentage of original lexicon size.

Table V. Performance of permuted index coarse search strategy

Another substring access structure is the trie,[21,22] which, however, has the disadvantage of high space overheads. For either of our test data sets, a trie index would require at least two pointers per character of lexicon—an overhead of at least five to six bytes per byte of lexicon with even an optimal representation—so we have not pursued the use of tries in our experiments.

Phonetic coding

Another class of candidate for coarse search scheme, and indeed a traditional choice for this task, are codings such as the *Soundex* method. However, they have marked limitations. In particular, matching is binary, with no notion of proximity; and the codes are highly generalised and do not reflect any special characteristics of particular lexicons. Performance of standard *Soundex* is shown in Table VI. Although fast and space-efficient, these results show that the method of bucketing words according to phonetic codes is easily the worst we used for finding matches.

However, phonetic coding can easily be combined with n-grams, by using postings lists derived from encoded strings. This approach discards the "bucketing" aspect of phonetic coding, and thus the need to truncate after a fixed number of characters. Such an approach is therefore suited to the modified *Soundex* algorithm and to partial *Phonix*, in which there is no truncation. Results are shown in Table VI. Overall there seems to be no reason to use *Soundex*, modified *Soundex*, or partial *Phonix* in conjunction with n-grams, when n-grams work better by themselves.

Phonetic coding could be incorporated into the permuted lexicon scheme, by storing, together with each string, its encoded form. Because the performance of the permuted lexicon and n-gram indexes were similar for strings that were not encoded, and because the phonetic codes were not particularly successful with n-grams, we have not pursued this combination.

Technique	NAMES			DICT		
	Index size (%)	CPU (msec)	Effective (%)	Index size (%)	CPU (msec)	Effective (%)
Soundex	56.8	16.7	26.9	37.1	60.8	5.4
modified *Soundex*						
$n = 2$	58.2	62.1	31.0	42.0	207.4	6.6
$n = 3$	69.9	21.8	31.6	45.5	80.9	8.9
$n = 4$	80.5	16.9	31.6	48.2	59.0	9.5
partial *Phonix*						
$n = 2$	92.1	61.5	50.0	70.8	170.8	16.3
$n = 3$	131.2	24.4	47.2	88.7	75.8	18.9
$n = 4$	221.7	18.3	45.4	119.2	60.3	17.6

Index size: as percentage of original lexicon size.

Table VI. Performance of phonetic coarse search strategies

The indexing techniques described in this section allocate weights to strings on the basis of index information only (number of matching n-grams, physical proximity). In the next section we consider how to refine the weighting using the strings themselves.

FINE SEARCH TECHNIQUES

Using n-gram coarse search techniques, every string is allocated a *gram-count* value. The strings with non-zero *gram-count* are candidates for processing by a fine search scheme, but the number to be processed needs to be kept small, to avoid processing the bulk of the lexicon at each query. A simple heuristic is to enlarge the heap that is used to extract the highest *gram-count* values, and hope that if A answers are required they will occur in among the kA highest *gram-count* values, for some k. For example, suppose 50 answers are required and $k = 5$. A heap of size 250 will only cost around 30% more to maintain than a heap of size 50, and maintaining the larger heap is far more efficient than returning to the structure of *gram-count* values to find further answers. We have used an enlarged heap in the experiments described below, arbitrarily choosing $k = 3$ since, for most techniques, performance was about as good as $k = 10$ and better than $k = 1$ or $k = 2$.

Given that the n-gram coarse search computes *gram-count*$(q, s) = |G_s \cap G_q|$ and that $|G_q|$ is known, only $|G_s|$ is required to give the modified form *gram-dist*$'$ of Ukkonen's measure. If the strings are held in memory their lengths can be computed on the fly, as we chose to do in our experiments; alternatively, lengths could be precomputed and stored, at a cost of around four bits per string. Results are shown in Table VII for *gram-dist* for $n = 2$, 3, and 4; for no phonetic coding; for *Soundex*; and for partial *Phonix*. The times shown include coarse and fine searching, and space is as in Tables IV, V, and VI.

The most startling feature of these results is the massive improvement in effectiveness for DICT. The other features are the continued poor performance of the phonetic codes, and the further evidence that index size, speed, and effectiveness trade off against each other.

If the strings of the lexicon are available in memory, the kA strings with high *gram-count* can be processed with an edit-distance measure. Results for the *edit* function are shown in Table VIII. This table also includes results for permuted lexicon indexing, in which the kA nearest neighbours are processed with *edit*.

The major difference between the use of *edit* and *gram-dist* is that the former adds considerably to

Technique	NAMES		DICT	
	CPU (msec)	Effective (%)	CPU (msec)	Effective (%)
no coding				
$n = 2$	62.9	61.1	173.8	45.1
$n = 3$	26.0	52.4	74.1	39.5
$n = 4$	18.5	49.1	63.2	31.1
modified *Soundex*				
$n = 2$	62.1	32.1	193.6	8.1
$n = 3$	26.5	32.2	72.8	7.6
$n = 4$	19.8	30.3	57.4	7.3
partial *Phonix*				
$n = 2$	68.8	53.1	178.2	29.9
$n = 3$	28.7	47.7	81.9	26.1
$n = 4$	24.0	45.7	65.8	20.6

Table VII. Performance of the *gram-dist* fine search strategy

retrieval cost; and, in the "no coding" case, although *edit* is better for NAMES, it is worse for DICT.

As the results show, there is little advantage to using phonetic coding. It does reduce index size, but at a marked cost to retrieval effectiveness. The results also show that there is little reason to use permuted lexicons. Although they give excellent performance for exact matching,[18,20] the results above show that, for approximate matching, they have speed and retrieval effectiveness similar to that of 3-grams, but with indexes that are about twice as large. That is, their greater speed for coarse ranking is overwhelmed by the cost of fine ranking. Moreover, permuted lexicons are expensive to update and require that the lexicon be held in memory.

SCALING UP

We now consider how well the performance of these techniques scales up for use with the TREC lexicon. Results are given in Table IX. The figures for n-grams are for *gram-dist* fine search and 100 answers—that is, the most successful of the schemes identified in the previous section. The permuted lexicon figures are with edit distance fine search and 100 answers. Because we do not have relevance judgements for TREC, we are only able to present figures for speed and time.

As can be seen, the times compare well to those of the smaller lexicon, particularly for $n = 4$. We see no reason, therefore, why our techniques could not be applied to a lexicon of almost any size.

PRACTICAL CONSIDERATIONS

In our experiments we assumed that lexicons and their indexes are stored in memory. For many systems this will be a reasonable assumption, given the performance gains to be made and given that even the lexicon of the TREC collection occupies less than 10 Mb.

However, for an n-gram index, storing the index and lexicon on disk does not imply a large performance penalty. Typically only five or six disk accesses are required to fetch the postings lists, and perhaps five to twenty disk accesses to fetch the top 50 answers, making the assumption that there will typically be some lexicographic clustering amongst similar words. Even if the lexicon is updated by addition of entries at the end, occasional rebuilds should maintain good clustering. The main problem

Technique	NAMES		DICT	
	CPU (msec)	Effective (%)	CPU (msec)	Effective (%)
no coding				
$n = 2$	66.0	66.6	178.4	41.3
$n = 3$	29.4	62.7	82.4	38.1
$n = 4$	22.7	60.6	68.1	33.8
modified *Soundex*				
$n = 2$	64.6	32.0	196.1	9.4
$n = 3$	26.3	32.2	86.6	8.9
$n = 4$	19.0	31.9	53.5	9.5
partial *Phonix*				
$n = 2$	70.4	60.4	185.2	31.6
$n = 3$	36.0	59.0	82.9	30.8
$n = 4$	27.5	58.7	67.1	27.7
permuted lexicon	42.9	63.6	42.4	40.2

Table VIII. Performance of the *edit* fine search strategy

presented by storage on disks is fine searching. Use of *edit* would be out of the question, as it requires access to kA strings. Similarly, *gram-dist* requires access to kA of the $|G_s|$ values, but these values could be explicitly stored in memory, at a cost of around four bits per lexicon entry.

The mapping from ordinal number to location of lexicon entry also has to be considered. On disk, another five to ten accesses would be required. In memory, this mapping would occupy around three bytes per string.

Another practical issue is cost of update: in a text database, new words will be added to the lexicon as records are inserted or changed. It is best for a new word to be added at the end (that is, allocated the ordinal number following the highest allocated previously), as insertion of a word will change the numbers of other words in the structure and require an index rebuild. (Note that the lexicon entries do not have to be stored in the order implied by their ordinal numbers.) Update of an n-gram index following addition of a word to the end of a lexicon is straightforward, only requiring that a few bits be added to the end of the postings lists of the word's n-grams.

Storing words out of lexicographic order increases index size, however, because compression performance is reduced, so it would be advantageous to occasionally rebuild the n-gram index. Given that even the TREC index can be built in around four minutes on a Sparc 10, such rebuilds would not be onerous.

CONCLUSIONS

We have shown that n-gram indexing provides an excellent coarse search mechanism for identifying approximate matches in a large lexicon. By varying n, the index can be kept small or query evaluation can be fast, with small n giving better effectiveness. Moreover, n-gram indexes present no difficulties with regard to update.

N-gram indexes should, however, be used in conjunction with a fine search mechanism, which for spelling correction can more than double effectiveness. The n-gram string distance proposed by Ukkonen is a suitable fine search mechanism, because it is effective and simple to compute. It has the added advantage that it can be adapted to a lexicon that is stored on disk, while other edit distances

Technique	Index size (%)	CPU (msec)
n-grams		
$n = 2$	87.6	1,735.0
$n = 3$	115.4	335.0
$n = 4$	162.2	183.3
permuted lexicons	300.0	111.9

Index size: as percentage of original lexicon size.

Table IX. Performance of search techniques on TREC

may not have this flexibility.

The most dramatic aspect of our results is that they demonstrate that phonetic coding is a poor string matching mechanism. It can provide some speed and space advantages for coarse search, but at low effectiveness, even for personal name matching. For fine search, the speed advantage is lost. Unless there are most unusual constraints on the matching mechanism, string similarity based on n-grams is the approximate string matching technique of choice.

ACKNOWLEDGEMENTS

This work was supported by the Australian Research Council, the Collaborative Information Technology Research Institute, and the Centre for Intelligent Decision Systems.

REFERENCES

1. C.L. Borgman and S.L. Siegfried, 'Getty's Synoname and its cousins: A survey of applications of personal name-matching algorithms', *Journal of the American Society for Information Science*, **43**, (7), 459–476, (1992).
2. P.A.V. Hall and G.R. Dowling, 'Approximate string matching', *Computing Surveys*, **12**, (4), 381–402, (1980).
3. K. Kukich, 'Techniques for automatically correcting words in text', *Computing Surveys*, **24**, (4), 377–440, (1992).
4. National Institute of Standards and Technology. *Proc. Text Retrieval Conference (TREC)*, Washington, November 1992. Special Publication 500-207.
5. H.J. Rogers and P. Willett, 'Searching for historical word forms in text databases using spelling correction methods: reverse error and phonetic coding methods', *Journal of Documentation*, **47**, (4), 333–353, (1991).
6. G. Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*, Addison-Wesley, Reading, MA, 1989.
7. S. Wu and U. Manber, 'Fast text searching allowing errors', *Communications of the ACM*, **35**, (10), 83–91, (1992).
8. E. Ukkonen, 'Approximate string-matching with q-grams and maximal matches', *Theoretical Computer Science*, **92**, 191–211, (1992).
9. T.N. Gadd, ''Fisching fore werds': Phonetic retrieval of written text in information systems', *Program: automated library and information systems*, **22**, (3), 222–237, (1988).
10. T.N. Gadd, 'PHONIX: The algorithm', *Program: automated library and information systems*, **24**, (4), 363–366, (1990).
11. T.C. Bell, A. Moffat, C.G. Nevill-Manning, I.H. Witten, and J. Zobel, 'Data compression in full-text retrieval systems', *Journal of the American Society for Information Science*, **44**, (9), 508–531, (October 1993).
12. J. Zobel, A. Moffat, and R. Sacks-Davis, 'An efficient indexing technique for full-text database systems', *Proc. International Conference on Very Large Databases*, Vancouver, Canada, August 1992, pp. 352–362.

13. R. Anglell, G. Freund, and P. Willett, ‘Automatic spelling correction using a trigram similarity measure’, *Information Processing & Management*, **19**, (4), 305–316, (1983).
14. J. Y. Kim and J. Shawe-Taylor, ‘An approximate string-matching algorithm’, *Theoretical Computer Science*, **92**, 107–117, (1992).
15. K. Kukich, ‘Spelling correction for the telecommunications network for the deaf’, *Communications of the ACM*, **35**, (5), 81–89, (1992).
16. O. Owolabi and D.R. McGregor, ‘Fast approximate string matching’, *Software—Practice and Experience*, **18**, 387–393, (1988).
17. A. Moffat and J. Zobel, ‘Parameterised compression for sparse bitmaps’, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, Copenhagen, Denmark, June 1992, pp. 274–285. ACM Press.
18. J. Zobel, A. Moffat, and R. Sacks-Davis, ‘Searching large lexicons for partially specified terms using compressed inverted files’, *Proc. International Conference on Very Large Databases*, Dublin, Ireland, 1993, pp. 290–301.
19. I.H. Witten, A. Moffat, and T.C. Bell, *Managing gigabytes: Compressing and indexing documents and images*, Van Nostrand Reinhold, New York, 1994.
20. P. Bratley and Y. Choueka, ‘Processing truncated terms in document retrieval systems’, *Information Processing & Management*, **18**, (5), 257–266, (1982).
21. G. Gonnet and R. Baeza-Yates, *Handbook of data structures and algorithms*, Addison-Wesley, Reading, Massachusetts, 2 edition, 1991.
22. D.R. Morrison, ‘PATRICIA—Practical algorithm to retrieve information coded in alphanumeric’, *Journal of the ACM*, **15**, (4), 514–534, (1968).