High-error approximate dictionary search using estimate hash comparisons

This is a preprint of an article published in Software Practice and Experience.
DOI: 10.1002/spe.797 http://dx.doi.org/10.1002/spe.797
http://www.interscience.wiley.com
Copyright © 2006 John Wiley & Sons Ltd.

JOHAN RÖNNBLOM <johan@ronnblom.net>

SUMMARY

A method for finding all matches in a pre-processed dictionary for a query string *q* and with at most *k* differences is presented. A very fast constant-time estimate using hashes is presented. A tree structure is used to minimise the number of estimates made. Practical tests are performed, showing that the estimate can filter out 99% of the full comparisons for 40% error rates and dictionaries of up to four million words. The tree is found efficient up to a 50% error rate.

KEY WORDS: edit distance, approximate, string matching, dictionary, hash

INTRODUCTION

This paper considers the problem of finding text strings in a pre-processed dictionary that are close to some query string. Closeness is defined here according to the commonly used *Edit Distance*, allowing the following edit operations:

- deleting a character in the query, at a cost c_d;
- inserting a character in the query, at a cost c_i;
- substituting a character in the query, at a cost c_s;
- transposing two characters adjacent in both compared strings, at a cost c_t.

Two common cost setups have been given names: $c_d = c_i = c_s = 1$, $c_t = 2$ is known as the *Levenshtein distance*, while $c_d = c_i = c_s = c_t = 1$ is the *Damerau distance*.

A query is formulated as a query string, *q*, and a maximum cost for edit operations, *k*. The search algorithm must then return all strings that can be matched under these conditions. This problem has many practical applications, often with slightly different requirements. Some examples:

- Spell checking for bad spellers. Dictionary words and query words are often transformed into pseudo-phonetic script, as bad spellers may have problems choosing the correct characters for a particular sound.
- Checking for typing errors. Similar, although not equivalent, to spell checking, this application may assume that the user knows the correct sequence of characters to type, but may sometimes, owing to motor errors, fail to press keys, accidentally hit additional keys, hit the wrong keys or hit the correct keys but in the wrong order. The search is therefore best done with unmodified query words and dictionaries. In practice, spell checkers also check for typing errors. For both these problems, the Damerau distance may be appropriate.
- Searching in phone or address registries. This application is similar to spell checking, in that the user is likely to be unaware of the correct characters to spell a name, where only the pronunciation may be known. On the other hand, the user can be expected to type the query carefully, as it is short. Thus, the Levenshtein distance may be most suitable.
- Refining initial word hypotheses obtained by a text recognition system, such as Optical Character Recognition (OCR) or handwriting recognition. Here, substitutions are very common, while insertions or deletions are more unlikely, especially when the text to be recognised is machine printed. Transpositions do not occur. It is therefore desirable to set c_i and c_d higher than c_s.

In light of these differences, the algorithms discussed in this paper will for the most part be kept general with respect to the cost of edit operations. It is however assumed that all costs are at least 1, and since a replacement can be considered as an insertion and a deletion, and a transposition is two replacements, this places some restrictions on the possible values.

Where previous published methods have often been limited to low error rates, or even one error [1 - 5], the method discussed in this paper focuses primarily on error rates between 20 and 50%. The paper focuses exclusively on natural language

dictionaries, for lack of known practical uses of other types of dictionaries and their relevant properties.

Notation

The string $s_{1...n}$ has n characters where the first is $s[1]$ and the last $s[n]$. The set of available characters (alphabet) is denoted Σ. A C-like notation is used for bit operations: '~' denotes the bitwise inversion, '&' a bitwise AND operation, '|' a bitwise OR and '^' a bitwise XOR (exclusive-or). The operators '<<' and '>>' shift bits left and right, filling in zero bits from the edges. Bit 1 is assumed to be the rightmost, least significant bit. The notation $|x|$ is used to signify the number of '1' bits in x, known as the bit count or population count. For the BPM algorithm (see below), bit patterns written on the form '1^x' signify that the bit value is repeated x times. These patterns are concatenated such that '$01^40^21^0$' = '0111100'. Comparisons assume that only the specified bits are compared, thus, an implicit masking operation may be needed if the computer word is longer.

COMPARISONS OF STRINGS

There are several known algorithms that check whether two strings $q_{1...m}$, $e_{1...n}$ match given a maximum edit cost k. The basis of these algorithms is to build an $m \times n$ matrix M where $M(i, j)$ is the aggregated minimum cost after comparing the characters $q[i]$, $e[j]$. Moving to the right in the matrix means inserting a character in q, moving down means deleting a character in q, and moving down diagonally means substituting or transposing characters. The value $M(m, n)$ is the edit distance. Jokinen et al [6] present an algorithm that can be adapted for different edit costs.

For the common case where the edit operations are all of unit cost, Myers [7] devised a method called *BPM* to express the columns being calculated with each cell represented by single bits in five computer words. Hyyrö and Navarro [8] extended this method to allow for an early exit when k has been exceeded. In listing 1, their algorithm has been modified for matching of entire strings not longer than 32 characters, with the Levenshtein distance. It is similar to the Myers (cutoff) algorithm found in [9].

BPMPreProcess($q_{1\ldots m}$, k)

1. **For** ($j \in \sum$) $B[j] \leftarrow 0^m$
2. **For** ($j \in 1\ldots m$) $B[q[j]] \leftarrow B[q[j]] \mid 0^{m-j}10^{j-1}$
3. **Return** B

BPMCompare(B, $q_{1\ldots m}$, $e_{1\ldots n}$, k)

4. $VP \leftarrow 1^m$, $VN \leftarrow 0$, $top \leftarrow k$
5. $d \leftarrow 0^{m-k}10^{k-1}$
6. **For** ($j \in 1\ldots n$)
7. $\quad X \leftarrow B[e[j]] \mid VN$
8. $\quad D0 \leftarrow ((VP + (X \,\&\, VP)) \wedge VP) \mid X$
9. $\quad HN \leftarrow VP \,\&\, D0$
10. $\quad HP \leftarrow VN \mid \sim(VP \mid D0)$
11. $\quad X \leftarrow (HP << 1) \mid 0^{m-1}1$
12. $\quad VN \leftarrow X \,\&\, D0$
13. $\quad VP \leftarrow (HN << 1) \mid \sim(X \mid D0)$
14. $\quad d \leftarrow d << 1$
15. $\quad$ **If** ($D0 \,\&\, d = 0^m$)
16. $\quad\quad$ **If** ($d \neq 0^m$) $val \leftarrow k + 1$
17. $\quad\quad$ **Else**
18. $\quad\quad\quad d \leftarrow 10^{m-1}$
19. $\quad\quad\quad$ **If** ($(HP \,\&\, d) \neq 0^m$) $top \leftarrow top + 1$
20. $\quad\quad\quad$ **If** ($(HN \,\&\, d) \neq 0^m$) $top \leftarrow top - 1$
21. $\quad\quad\quad val \leftarrow top$, $top \leftarrow$ **Min**(top, k)
22. $\quad\quad$ **While** ($val > k$) **Do**
23. $\quad\quad\quad$ **If** $d = 0^{m-1}1$ **Then Return** FALSE
24. $\quad\quad\quad$ **If** ($(VP \,\&\, d) \neq 0^m$) $val \leftarrow val + 1$
25. $\quad\quad\quad$ **If** ($(VN \,\&\, d) \neq 0^m$) $val \leftarrow val - 1$
26. $\quad\quad\quad d \leftarrow d >> 1$
27. **Return** ($d = 10^{m-1}$)

Listing 1. A modified BPM algorithm for comparison of entire strings.

OPTIMISING THE SEARCH

Although the BPM algorithm of listing 1 is heavily optimised, performing a brute-force search by comparing a query string to each word in a dictionary is far from optimal. Grossi and Luccio [10] proposed a method for quickly filtering out non-matching strings. This method was improved by Navarro [11]. It counts the frequencies of characters within a potential match and is very elegant when searching for a string at any location inside a text. For matching of entire words in a dictionary it is not so elegant, but still turns out faster than the brute-force approach, as will be seen below. Listing 2 shows code for this algorithm as adapted for dictionary search. It will be referred to as the *counting* algorithm. It is possible to implement the bit-parallel algorithm of [11] by considering the dictionary words to be the search templates, and then comparing them to the query string. This requires precalculating table entries for each alphabet character and dictionary word. However, the single-pattern approach is far more straightforward implementation-wise and is therefore used as a reference.

CountPrepare($q_{1\ldots m}$)

1. **For** ($c \in \Sigma$) $A[c] \leftarrow 0$
2. **For** ($i \in 1\ldots m$) $A[q[i]]$++
3. **Return** A

CountFilter($q_{1\ldots m}$, $e_{1\ldots n}$, A, k)

4. **For** ($i \in 1\ldots n$) $B[e[i]] \leftarrow A[e[i]]$
5. **For** ($i \in 1\ldots n$)
6. **If** ($B[e[i]]$-- > 0) k++
7. **Return** ($k \geq$ **Max**(m, n))

Listing 2. The counting filter algorithm applied to dictionary words.

Estimating edit distance through a simple hash comparison

The main idea of this paper is to perform a similar filtration as that of the counting algorithm, but adapted to a dictionary by pre-computing hash values for each word. The filtration can then be performed using very few operations per word, without accessing the words themselves. The idea may be compared to [12].

Performing the edit operations can be considered as inserting, adding, replacing or transposing various *features* of the string. To simulate this with a hash consisting of a fixed-size computer word, the idea is to map features of strings to the bits of the hash. The operations then become equivalent to adding, removing or substituting bits in the hash. It is important that the features are selected such that a strict minimum of operations can be calculated. It is further desirable that the estimate be as close as possible to the real distance.

Constructing hashes for strings in a dictionary

Statistical means are used to achieve hashes that are suitable for a particular dictionary. The algorithm starts by counting the occurrence of each character in the dictionary. It keeps separate counts based on the number of instances of the character in the currently examined string. Thus, for the string *'referral'*, the following features are counted: *r*1, *e*1, *f*1, *e*2, *r*2, *r*3, *a*1 and *l*1. For convenience, each character/instance pair will be called a *feature*. Any string, then, has a number of features equal to the number of characters in the string.

The second step of the algorithm is to group the features in a number of buckets, where, ideally, the features in each bucket should occur equally often in the dictionary, and each bucket should have an equal number of features in it. The optimum number of buckets is dependant on the nature of the dictionary and practical implementation considerations. In this paper, 32 or 64 buckets were used.

Table I shows an allocation of features for an English dictionary. It can be noted that, because of the highly uneven character distribution in English, it is impossible to spread the features evenly. The algorithm successively takes the most common unallocated feature and adds it to the bucket with the lowest aggregate feature count, taking the first one in case of a tie. The individually most common features are thus assigned to the first bits of the hash value.

Each bucket corresponds to a bit in the hash, and if any of the features in a bucket occurs in the string, that bit is set. If a string contains a feature that does not occur in the dictionary, a fallback method must be used to assign a suitable bit. The following formula was used for this purpose:

$$b = (w_h/2) + ((character + instance) \bmod (w_h/2)) \quad (1)$$

where w_h is the width of the hash in bits. This formula spreads such features over the second half of the hash. The motivation for using only the second half, representing individually less common features, was found in practical experiments.

The alert reader will have noticed that this method of constructing a hash discards any information about the order of characters within a string. Thus, anagrams have the same hash. In practice, however, anagrams are rare. Furthermore, it can be noted that the Counting algorithm provides an upper bound for the filtration rate of this method. The difference in efficiency depends on how many bits in a hash are typically used to represent multiple features of a word.

1: *s*1	**9**: *l*1	**17**: *s*2	**25**: *t*2 *x*1 *q*1 *r*3 *s*5 *i*5 *f*3 *g*4 *y*3 *x*4
2: *e*1	**10**: *c*1	**18**: *p*1	**26**: *f*1 *p*2 *n*3 *o*3 *x*2 *a*5 *z*3
3: *i*1	**11**: *'*1	**19**: *h*1	**27**: *r*2 *i*3 *a*3 *l*3 *w*2 *l*4 *j*2 *d*4 *k*4 *f*4
4: *a*1	**12**: *d*1	**20**: *b*1	**28**: *o*2 *d*2 *b*2 *k*2 *z*2 *t*4 *s*6 *u*4 *p*4 *a*6
5: *r*1	**13**: *u*1	**21**: *i*2	**29**: *k*1 *z*1 *j*1 *i*4 *d*3 *b*3
6: *n*1	**14**: *e*2	**22**: *y*1 *v*2 *a*4 *n*4 *o*5	**30**: *v*1 *c*2 *h*2 *t*3 *p*3 *u*3
7: *t*1	**15**: *m*1	**23**: *a*2 *s*4 *m*3 *r*4 *x*3 *c*4 *i*6	**31**: *w*1 *e*3 *u*2 *y*2 *g*3 *h*3 *e*5 *q*2 *z*4
8: *o*1	**16**: *g*1	**24**: *n*2 *m*2 *f*2 *k*3 *m*4 *b*4	**32**: *l*2 *s*3 *g*2 *e*4 *c*3 *o*4 *w*3 *n*5 *s*7

Table I. Feature allocation for an English dictionary.

Computing the edit distance estimate for two hashes

Let h_q and h_e be hashes of two strings q and e to be compared. The number of bits in h_q and h_e then constitute a minimum number of features in the respective strings. We further compute the bitwise exclusive-or of h_q and h_e and name it x. The bits in x correspond to the bits that have changed between the two hashes. This may be for three reasons: the feature may need to be inserted in q, meaning that the corresponding bit is only present in h_e; the feature may need to be deleted from q, meaning that the bit is only present in h_q; or a replacement may be necessary, in case the bit should be 'moved' from one position in h_q to another one in h_e.

We note that a replacement, that is, a bit 'moving', may cause the number of bits set in the hash to increase or decrease. This is because there may be several features in q or e corresponding to the same bit, causing the number of bits in h_q or h_e to be lower than the number of features in q or e. Consider table I and the word 'mammal'. In this case, the features $a2$ and $m3$ are both expressed in bit 23. When a replacement is made, one such 'invisible' feature may become visible, because the corresponding bit has not been set previously--or *vice versa*. Now consider the difference

$$|h_q| - |h_e| \quad (2)$$

which corresponds to the number of operations necessary to get the same number of bits in both hashes. If the difference is positive, it can correspond either to deletions in q or to replacements. If it is negative, its absolute value corresponds to insertions in q or replacements.

With additions or deletions catered for, the value $|x|$ can tell us more about replacements. Since each replacement can cause two bits to move, and therefore increase $|x|$ by two, a lower bound for such replacements is:

$$(|x| - \text{abs}(|h_q| - |h_e|)) / 2 \quad (3)$$

A minimum edit distance can then be summed as

$$d_{min}(q, e) = c_s \times (|x| - \text{abs}(|h_q| - |h_e|) / 2 + \min(c_d, c_i, c_s) \times \text{abs}(|h_q| - |h_e|) \quad (4)$$

or, if $c_d \geq c_s$, $c_i \geq c_s$, which is normally true,

$$d_{min}(q, e) = c_s \times (|x| + \text{abs}(|h_q| - |h_e|)) / 2 \quad (5)$$

Some processor architectures such as DEC Alpha and UltraSPARC have dedicated operations to count the bits of a computer word. For implementations on other CPUs, one or even two of these counts can be pre-computed. As a bit count can be parallelised by using masks and shifts, the resulting function is very fast. Listing 3 shows the minimum edit distance estimate complete with such an optimised bit

count function for 32 bit hashes, using $c_s = 1$. Discussion about a similar bit-count function can be found in [13].

d_{min}32(h_Q, nh_Q, h_E)
1. $x \leftarrow h_Q$ ^ h_E
2. $nh_E \leftarrow$ **CountBits32**(h_E)
3. $nx \leftarrow$ **CountBits32**(x)
4. **Return** (nx + **Abs**($nh_Q - nh_E$)) / 2

CountBits32(a)
5. $mask \leftarrow 011111111111$
6. $a \leftarrow (a$ - ((a & ~$mask$) >> 1)) – ((a >> 2) & $mask$)
7. $a \leftarrow a + (a >> 3)$
8. $a \leftarrow (a$ & 070707) + ((a >> 18) & 070707)
9. $a \leftarrow a \times 010101$
10. **Return** ((a >> 12) & 077)

Listing 3. The algorithm d_{min} and a supporting function. For clarity, constants (except shift counts) are expressed in octal base.

INDEXING STRINGS TO AVOID COMPARISONS

Of the many techniques invented to reduce the number of comparisons necessary for a search operation, the tree structure BKT has been found suitable for dictionary searches [2, 3]. The method presented here is similar to the BKT. More in-depth analysis of the BKT can be found in [14]. The general idea is to group similar strings such that entire branches can be dismissed, rather than having to consider each individual string in the dictionary.

Constructing the tree

Hashes are pre-computed for each string. A pivot string (or, rather, a pivot hash) is selected arbitrarily from the dictionary. It is then compared against all other hashes to find the maximum distance, which becomes the height of the tree. Each node in

the tree has a pivot *p*, which it shares with its first subnode. All strings *e* where $d_{min}(p, e) < L$ are put in a subtree under it, where *L* is the level in the tree that the node belongs to. Then, one of any remaining strings is selected arbitrarily for a second node at that level. This continues until all strings are grouped under some subnode. Subtrees for each node are then created in the same way. This means that on level 1, a pivot is grouped with all hashes at a distance of at most 1 from it, while on level 0, words with identical hashes are grouped. Figure 1*a* shows a small tree. Note again that the order of characters is ignored, thus 'parrot' is identical to 'raptor'.

Space and I/O considerations

As we will see, the estimate is very accurate, which means that traversing the tree itself will likely turn out to be more costly than computing exact edit distances for the few potential matches, even for very high values of *k*. It is desirable to optimise the layout of the tree to make the necessary I/O as fast as possible. This is best achieved by a data layout that results in serial rather than random accesses. Note that 'I/O' here refers to fetching data from the computer's main memory into the cache, although the same principle would hold if the data is held on an even slower medium such as a hard drive or a network.

It is noted that if *k* is so large that every hash matches, we will visit each node in a determined order. If a subtree can be dismissed, we merely skip forward in this sequence. Furthermore, practical experiments led to the conclusion that it is more efficient to omit nodes for level 1. Rejecting a level 1 subtree then means skipping through the subsequent data, going past all level 0 nodes.

Furthermore, it was found efficient to include the hashes into the tree rather than indexing them. Figure 1*b* shows an improved tree layout. The continuous line shows the serial order of the data, while the dashed arrows show offsets stored at tree nodes, indicating where to skip in case the subtree of a pivot is dismissed. Nodes further store information about which level they belong to, as well as a bit to indicate if the node pointed to is at the same level, to reduce memory access. This information can be packed into a 32-bit word for reasonably large dictionaries. The bottom nodes store a hash, and one or several offsets to strings. Two bits of these offsets are used to signify if the subsequent data belongs to the same hash, is a

level 1 implicit node, or a higher node. Listing 4 shows pseudocode that constructs an optimised tree recursively. A drawback of the optimised layout is that it does not lend itself well to insertions and deletions in the tree.

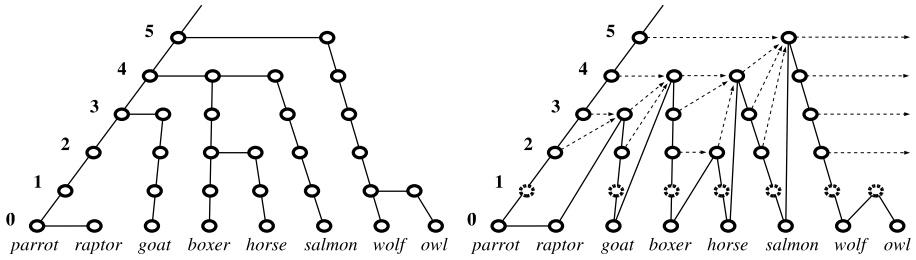

Figure 1a. A small example tree. *1b. The same tree with optimised layout. Level numbers on the left show d_{min} between words connected through that level by an up-right-down path. Words are pivots for all nodes on lines going up from them.*

Searching the tree

The search is entered at the first node of the top level. At any node, q is compared against the pivot e. If $d_{min}(q, e) - k$ is lower than the level of the node, the search is continued from level $d_{min}(q, e) - k$ of the leftmost subtree for the node. If it is higher or equal, the search continues at the next node of the same level. If it is the last node in a subtree, the search is continued at the next node counted from the parent node in the tree, and so on.

If at any time $d_{min}(q, e) \leq k$, the node that has been compared may contain a match. At this point, a full edit distance comparison must be made for all strings belonging to that node. These may be more than one, as several strings may have identical hashes. When a match is found, it is possible to let the search function return that string, and allow subsequent calls to continue searching from the same position in the tree. Listing 5 shows pseudocode that searches an optimised tree.

EXPERIMENTAL RESULTS

The algorithms were implemented in C and compiled with GCC using full optimisations. Critical sections were disassembled and manually inspected to verify code efficiency. Some changes from the pseudo-code shown in this paper were made, for example to minimise memory accesses and avoid unnecessary branches. The test machine is a Pegasos 2 with a 1 GHz 7447 ('G4') 32-bit CPU and the MorphOS 1.4.5 operating system. All required data was held in memory.

MakeTree(*W*)
1. global variables *pos* ← 0, *Tree*
2. create a container *T* with all words *W*
3. *TreeHeight* ← **Max**(**d**$_{\mathbf{min}}$(*T*[0], *e*)) for words *e* ∈ *T*
4. **MakeSubnodes**(*T*, *TreeHeight* - 1, TRUE)
5. *Tree*[*pos*] ← *TreeHeight*
6. **Return** *Tree*

MakeSubnodes(*N*, *L*, *U*)
7. **If** (*L* > 0)
8. **Repeat**
9. move *e* ∈ *N* with **d**$_{\mathbf{min}}$(*N*[0], *e*) < *L* to a container *S*
10. *R* ← (is *N* empty?)
11. **If** (*L* > 1) *Q* ← *pos*++
12. **MakeSubnodes**(*S*, *L* - 1, *R*)
13. **If** (*L* > 1) *Tree*[*Q*] ← *R* << SHIFT$_{\text{highlevelbit}}$ | *L* << SHIFT$_{\text{level}}$ | *pos*
14. **Until** (*R*)
15. **Else**
16. *Tree*[*pos*++] ← the identical hash for all words ∈ *N*
17. **Repeat**
18. *W* ← offset to *N*[0], remove that word
19. *R* ← (is *N* empty?)
20. *Tree*[*pos*++] ← *U* << SHIFT$_{\text{highlevelbit}}$ | *R* << SHIFT$_{\text{levelonebit}}$ | *W*
21. **Until** (*R*)

Listing 4. Construction of the optimised tree.

The Levenshtein distance was used with the BPM algorithm for verification as long as q was at most 32 characters. For the exceptionally rare longer words, an algorithm similar to one found in [6] was used as a fallback. To improve the efficiency of the tree at the cost of increased pre-processing, 50 attempts were made to select pivots by pseudo-randomly choosing between the available values. Experimentally it was found most efficient to choose the pivot that maximised tree height, and then pivots that grouped as many words as possible into their subtrees, although the speed gain compared to using arbitrary pivots may typically be about ten percent only.

```
FirstSearch(Tree, TreeHeight, q, h_q, k)
 1. level ← TreeHeight – 1, pos ← level - 1
 2. Return RepeatSearch(Tree, TreeHeight, q, h_q, k, pos, level)
RepeatSearch(Tree, TreeHeight, q, h_q, k, pos, level)
 3. While (level < TreeHeight)
 4.     word ← NULL
 5.     If (level > 0)
 6.         level ← Min(level, d_min(h_q, Tree[pos++]) - k)
 7.     If (level > 1)
 8.         node ← Tree[pos - level]
 9.         pos ← node & MASK_pos
10.     Else If (level < 1)
11.         node ← Tree[pos++]
12.         word ← node & MASK_offset
13.         If (node & MASK_levelonebit) level ← 1
14.     Else
15.         While (((node ← Tree[pos++]) & MASK_levelonebit) = 0)
16.     If (node & MASK_levelhighbit) level ← (Tree[pos] & MASK_level) >> SHIFT_level
17.     pos ← pos + Max(0, level – 1)
18.     If (word ≠ NULL)
19.         If (BPMCompare(q, word, k)) Return [pos, level, word]
20. Return NULL
```

Listing 5. Searching the optimised tree.

Fourteen dictionaries from the freely available spell checker Aspell [15] were used as test data. These dictionaries all use a western Latin alphabet. All dictionaries were converted to lower case and any duplicates were removed. Further, all these dictionaries were joined into one large dictionary, again with duplicates removed. The aggregate is here called the 'Monster' dictionary.

Tests were performed with a percentage of errors from 0 to 70%. The maximum query distance *k* was set to this percentage of the length of query words, rounded upwards. For each test, 10 000 query words were pseudo-randomly selected from the dictionary. Then, *k* different pseudo-randomly chosen characters in the word were replaced by characters pseudo-randomly picked from the alphabet used in the dictionary. Thus, each query was guaranteed to match at least one word.

Dictionary	N	Σ	Ws	Cs		Tp		W/h		TS30	
				32bit	64bit	32bit	64bit	32bit	64bit	32bit	64bit
Tagalog	14	29	138	329	388	0.22	0.30	1.08	1.08	0.30	0.33
Breton	33	34	282	688	822	0.65	0.98	1.07	1.07	0.58	0.75
Swedish	118	31	1243	2749	3319	2.79	4.43	1.10	1.08	3.30	3.80
English	135	27	1304	2949	3508	3.49	5.08	1.12	1.10	3.72	4.43
Icelandic	222	38	2434	5463	6711	5.86	9.01	1.10	1.06	6.09	6.59
Dutch	227	41	2639	5651	6809	5.62	8.82	1.09	1.07	7.63	6.98
Irish	296	33	3371	7146	8558	7.75	13.4	1.09	1.06	9.28	9.40
Norwegian	296	37	3966	8267	10043	8.15	12.7	1.08	1.05	13.1	9.93
German	309	35	3999	8101	9940	8.41	13.3	1.11	1.05	12.5	11.1
Portuguese	386	39	4348	8568	10185	10.2	16.2	1.21	1.16	11.4	10.3
Catalan	603	43	6685	13195	15764	16.8	27.0	1.23	1.16	15.9	13.5
Spanish	596	33	6791	12943	15412	16.9	27.7	1.30	1.19	17.4	15.5
French	630	43	7651	15289	18221	19.8	31.5	1.14	1.08	21.4	16.6
Finnish	731	31	9992	17627	21424	15.8	27.9	1.37	1.21	29.2	18.5
Monster	4374	60	52793	96118	119934	133.5	138.6	1.47	1.22	114.6	65.8

Table II. Dictionaries. Legend: N = number of words, thousands. Σ = Alphabet size. Ws = Raw size of words, Cs = Combined size of words and tree (thousands of bytes). Tp = Time to compute the tree (s). W/h = Average number of words per hash. TS30 = Average search time with 30% error rate (ms).

Table II displays some information about these dictionaries and some test results. Space usage for the trees ranges from about 80% of the uncompressed dictionary size for the largest dictionaries and 32 bit hashes, to about 200% for the smallest

dictionaries with 64 bit hashes. The pre-processing was not optimised for speed. In practice it appears to exhibit a linear relationship with the dictionary size. Even with the largest dictionary and 32 bit hashes, only 1.47 words share the same hash on average. Finally, average search times for 30% errors are shown. For 'small' dictionaries, it is faster (although never dramatically) to use 32 bit hashes, while for 'large' dictionaries, 64 bit hashes become faster.

To isolate the impact of dictionary size from other factors, subsets of the Monster dictionary were chosen pseudo-randomly for dictionaries with 16 000, 256 000 and 4 096 000 words. Results from these tests are shown in figures 2 and 3.

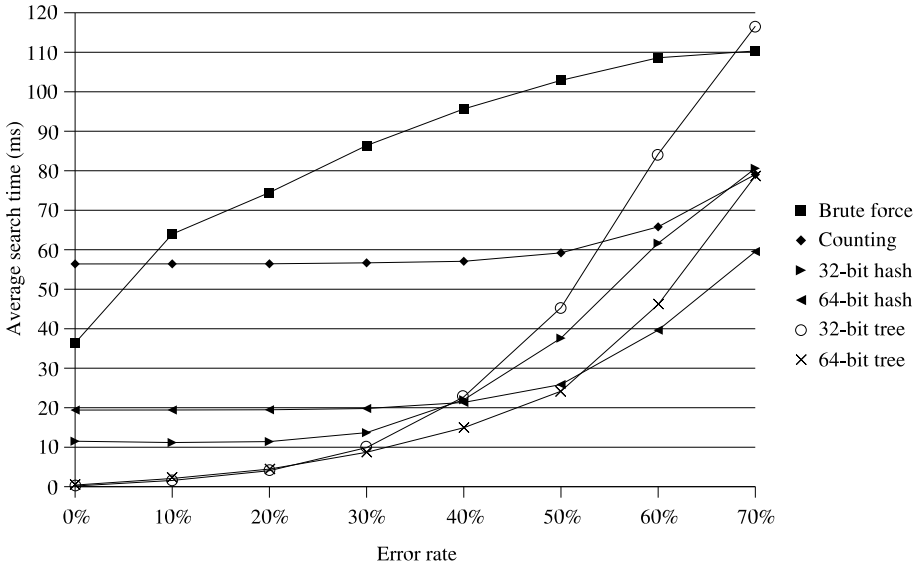

Figure 2. Average search times for a 256,000 word dictionary.

Figure 2 shows average search times for the synthetic 256 000 word dictionary. The *brute force* algorithm uses the BPM to compare each word in the dictionary, or a simple exact comparison for $k = 0$. The counting algorithm works as in listing 2. The 32- and 64-bit hash algorithms store a table with pre-computed hashes for each word. Finally, the tree algorithms operate as in listing 5. It may seem strange that the counting and pure hash algorithms do not require more time as k increases

within moderate levels. The reason is that errors do not necessarily increase the number of estimated matches, even though the number of actual matches grows quickly with *k*. On the contrary, since the character distribution of actual words is not even, a pseudo-random replacement from the entire alphabet is likely to result in a more unique query word, which may make up for the wider query range.

It can be seen that 32-bit hashes are sufficient for low error rates, but quickly deteriorate for high and extremely high number of errors, unlike 64-bit hashes. For larger dictionary sizes, the tree-based algorithms become relatively more efficient. However, the crossing points determining the relative merits of algorithms for each error rate do not move by more than a few percentage units within the examined 16 000--4 096 000 word range.

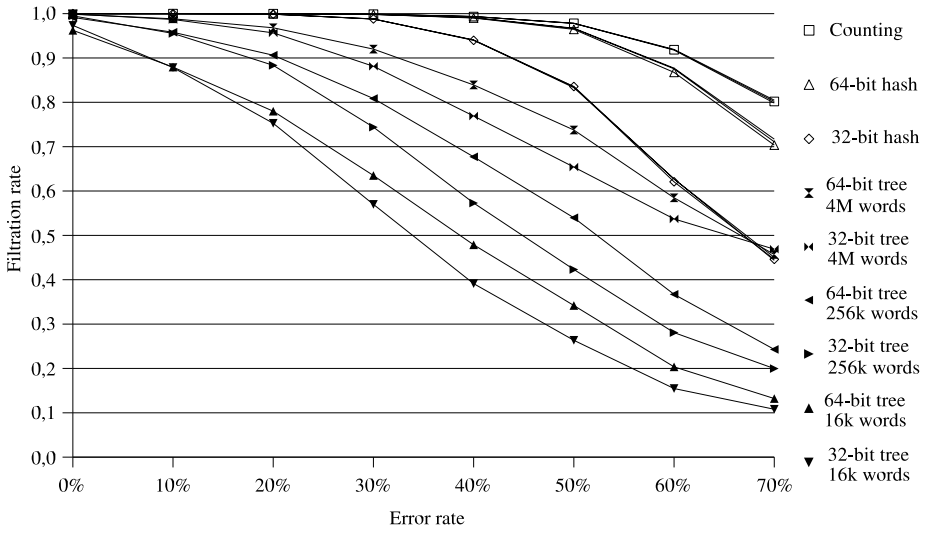

Figure 3. Filtration rates for different algorithms, dictionary sizes and error rates.

In figure 3, filtration rates are studied. The filtration rate is defined as the number of non-matches that are filtered out by an algorithm, divided by the total number of non-matches. For Counting and hash filtration rates, this is the ratio of non-matches that are not verified with the BPM algorithm, while for trees it is the ratio of non-matches that are not subjected to a hash estimate. Thus, words grouped because

they have identical hashes are seen to improve the filtration rate of the tree. Filtration rates for counting and pure hash algorithms do not vary with dictionary size. For trees, the filtration rate increases drastically with increased dictionary size. The graph shows that the filtration efficiency, particularly for 64-bit hashes, is very close to the theoretical optimum of the counting filter.

CONCLUSIONS

It is clear that there is a very significant gain to be made from using hash-based estimate comparisons. The space requirement can be made moderate for any dictionary size by choosing the hash length correctly. Furthermore, the high discriminatory power of the estimate means that the dictionary itself can be stored in a compressed form.

The BKT-like tree is not as discriminatory, but still efficient, at least for low-to-moderate error rates, and especially with large dictionaries. Compared to [2] and [3] it appears that the combination of a tree and hash values also improves the efficiency of the tree for higher error rates, as these trees remain efficient for error rates up to 50%.

Many aspects of the presented hashing technique remain open for further research. For example, what makes the hashes more efficient with some dictionaries than with others? Some relation to alphabet size can be suspected, but there may also be other factors. Tests with other error models may prove interesting. It may also be worthwhile to compare with other tree implementations. Since the full comparisons are never needed for decisions on how to traverse the tree, they are ideal for further parallelisation by comparing multiple words at once, ideally using SIMD computations as discussed in reference 3. Furthermore, it may be efficient to traverse the tree once while searching for multiple queries.

Naturally, it would be interesting to see how this algorithm performs compared to algorithms designed specifically for low error rates. As this algorithm was developed mainly with high error rates in mind, and owing to the lack of access to a proven benchmark implementation, this was not done in this study.

REFERENCES

1. Dimov, D. T. An Approximate String Matching Method for Handwriting Recognition Post-Processing Using a Dictionary. *Frontiers in Handwriting Recognition*, Impedovo S (ed.). Springer, 1994; 323-332.
2. Baeza Yates, Ricardo and Navarro, Gonzalo. Fast Approximate String Matching in a Dictionary. *SPIRE '98, String Processing and Information Retrieval*. IEEE Computer Society, 1998; 14-22.
3. Fredriksson, Kimmo. Metric Indexes for Approximate String Matching in a Dictionary. *Proceedings of SPIRE 2004, Lecture Notes in Computer Science*. Springer, 2004; 212-213.
4. Shang, H, Merrettal, T.h, Tries for Approximate String Matching, IEEE *Transactions on Knowledge and Data Engineering*, 1996; 8:4.
5. Jokinen, Petteri and Ukkonen, Esko. Two algorithms for approximate string matching in static texts. *Mathematical Foundations of Computer Science*, Tarlecki, A (ed.). *Lecture Notes in Computer Science 520*. Springer, 1991; 240-248.
6. Jokinen, Petteri, Tarhio, Jorma, and Ukkonen, Esko. A comparison of approximate string matching algorithms. *Software: Practice and Experience* 1996; 26(12): 1439-1458.
7. Myers, G. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM* 1999; 46(3):395-415.
8. Hyyrö, Heikki and Navarro, Gonzalo. Bit-Parallel Witnesses and Their Applications to Approximate String Matching. *Algorithmica* 2005; 41: 203-231.
9. Hyyrö, Heikki. A Bit-Vector Algorithm for Computing Levenshtein and Damerau Edit Distances, *Nordic Journal of Computing*, 2003; 10:1.
10. Grossi, R. and Luccio, F. Simple and efficient string matching with k mismatches. *Information Processing Letters, 1989*; 33(3):113-120.
11. Navarro, Gonzalo. Multiple approximate string matching by counting. P*roceedings of WSP'97*. 1997; 125-139.
12. Faloutsos, C. Signature-based text retrieval methods: a survey. *IEEE Data Engineering* 1990; 13 (1): 25-32.
13. AMD Corporation. *AMD Athlon Processor: x86 Code Optimization Guide*, 2002; 136-139,
14. Chávez, Edgar, Navarro, Gonzalo and Marroquín, José Luis. Searching in Metric Spaces. *ACM Computing Surveys* 2001; 33(3):273-321.
15. Atkinson, Kevin. GNU Aspell. *http://aspell.net/ (Dec 1 2005)*