Fast Approximate String Matching in a Dictionary

Ricardo Baeza-Yates Gonzalo Navarro

Dept. of Computer Science, University of Chile
Blanco Encalada 2120 - Santiago - Chile
{rbaeza,gnavarro}@dcc.uchile.cl

Abstract

A successful technique to search large textual databases allowing errors relies on an online search in the vocabulary of the text. To reduce the time of that online search, we index the vocabulary as a metric space. We show that with reasonable space overhead we can improve by a factor of two over the fastest online algorithms, when the tolerated error level is low (which is reasonable in text searching).

1 Introduction

Approximate string matching is a recurrent problem in many branches of computer science, with applications to text searching, computational biology, pattern recognition, signal processing, etc.

The problem can be stated as follows: given a long text of length n, and a (comparatively short) pattern of length m, retrieve all the segments (or "occurrences") of the text whose *edit distance* to the pattern is at most k. The edit distance $ed()$ between two strings is defined as the minimum number of character insertions, deletions and replacements needed to make them equal.

In the online version of the problem, the pattern can be preprocessed but the text cannot. The classical solution uses dynamic programming and is $O(mn)$ time [16]. Nowadays, the best practical results are $O(kn)$ time in the worst case and $O(kn \log_\sigma(m)/m)$ on average (where σ is the alphabet size), e.g. [13, 20, 9, 4, 15, 18, 6]. The average case mentioned is "sublinear" in the sense that not all the text characters are inspected, but the online problem is $\Omega(n)$ if m is taken as constant.

We are interested in large textual databases in this work, where the main motivations for approximate string matching come from the low quality of the text (e.g. because of optical character recognition (OCR) or typing errors), heterogeneousness of the databases (different languages which the users may not spell correctly), spelling errors in the pattern or the text, searching for foreign names and searching with uncertainty. Those texts take gigabytes and are relatively static. Even the fastest online algorithms are not practical for this case, since they process a few megabytes per second. Preprocessing the text and building an index to speed up the search becomes necessary.

However, only a few years ago indexing text for approximate string matching was considered one of the main open problems in this area [22, 2]. The practical indices which are in use today rely on an online search in the vocabulary of the text, which is quite small compared to the text itself. This takes a few seconds at most. While this may be adequate for single-user environments, it is interesting to improve the search time for multi-user environments. For instance, a Web search engine which receives many requests per second cannot spend a few seconds to traverse the vocabulary.

In this paper we propose to organize the vocabulary as a metric space using the distance function $ed()$, and use a known data structure to index such spaces. We show experimentally that this imposes a reasonable space overhead over the vocabulary, and that the reward is an important reduction in search times (close to half of the best online algorithms). This algorithm may also have other applications where a dictionary of words is searched allowing errors, such as in spelling problems.

This paper is organized as follows. In Section 2 we explain the basic concepts used. In Section 3 we present our metric space technique. In Section 4 we experimentally evaluate our technique. In Section 5 we give our conclusions.

2 Basic Concepts

2.1 Indices for Approximate String Matching

The first indices for approximate string matching appeared in 1992, in two different flavors: *word-oriented* and *sequence-oriented* indices. In the first type, more oriented to natural language text and information retrieval, the index can retrieve every *word* whose edit distance to the pattern is at most k. In the second one, useful also when the text is not natural language, the index will retrieve every *sequence*, without notion of word separation.

We focus on word-oriented indices in this work, where the problem is simpler and hence has been solved quite well. Sequence-retrieving indices are still very immature to be useful for huge text databases (i.e. the indices are very large, are not well-behaved on disk, are very costly to build and update, etc.). It must be clear, however, that word-oriented indices are only capable of retrieving an occurrence that is a sequence of words. For instance, they cannot retrieve `"flower"` with one error from `"flo wer"` or `"many flowers"` from `"manyflowers"`. In many cases the restriction is acceptable, however.

Current word-oriented indices are basically inverted indices: they store the *vocabulary* of the text (i.e. the set of all distinct words in the text) and a list of *occurrences* for each word (i.e. the set of positions where the word appears in the text). Approximate string matching is solved by first running a classical online algorithm on the vocabulary (as if it was a text), thus obtaining the set of words to retrieve. The rest depends on the particular index. Full inverted indices such as Igrep [1] simply make the union of the lists of occurrences of all matching words to obtain the final answer. Block-oriented indices such as Glimpse and variations on it [14, 5] reduce space requirements by making the occurrences point to blocks of text instead of exact positions, and must traverse the candidate text blocks to find the actual answers. In some cases the blocks need not be traversed (e.g. if each block is a Web page and we do not need to mark the occurrences inside the page) and therefore the main cost corresponds to the search in the vocabulary. See Figure 1.

This scheme works well because the vocabulary is very small compared to the text. For instance, in the 1 Gb TREC collection [11] the vocabulary takes no more than 5 Mb. An empirical law known as Heaps Law [12] states that the vocabulary for a text of n words grows as $O(n^\beta)$, where $0 < \beta < 1$. In practice, β is between 0.4 and 0.6 [1]. An online algorithm can search such vocabulary in a few seconds. While improving this

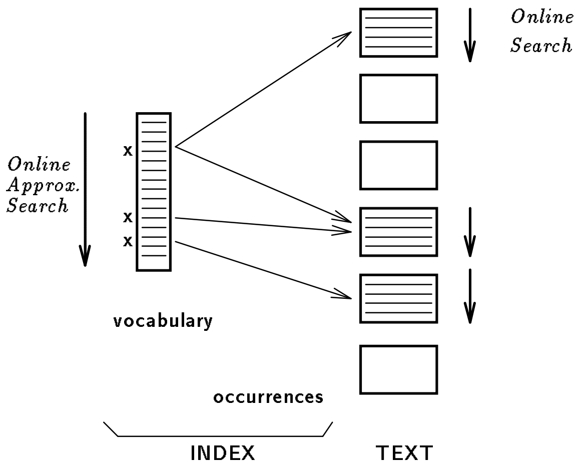

Figure 1. Approximate searching on an inverted index. The online search on the text may or may not be necessary.

may not be necessary in a single-user environment, it is always of interest in a multi-user environment like a Web search engine.

2.2 Online Searching

The classical algorithm for approximate string matching [16] is based on dynamic programming and takes $O(mn)$ time. It is a minor modification of an algorithm to compute the edit distance between to words a and b, which costs $O(|a||b|)$. This algorithm is unbeaten in flexibility, since it can be adapted to a number of variations in the distance function (e.g. to allow transpositions, or to give different costs to the operations). There exists no significatively better algorithm to compute the exact edit distance among two random strings, but there are many improvements to the search algorithm allowing k errors. They are orders of magnitude faster than the classical algorihtm, but they are not so flexible and rely on specific properties of the edit distance.

The technique that we study in this paper needs to compute the exact edit distance among strings, and therefore it relies on the classical algorithm. The result is that, although it may perform a few evaluations of the edit distance (say, 5% of the whole vocabulary), it may be slower than an online traversal with a fast algorihtm. It is very important to understand this when an indexing scheme is evaluated, since traversing a small percentage of the vocabulary does not guarantee usefulness in practice. On the other hand, many of the fastest algorithm could not be usable if some extension over the edit distance was desired, while the classical algorithm (and hence our technique) can accomodate many extensions at no extra cost.

On our machine (described later), the fastest on-line approximate search algorithms run at a maximum speed of 25 megabytes per second when searching words (for $k = 1$), and at a minimum of 1 megabyte per second (the dynamic programming algorithm, which is general).

2.3 Searching in General Metric Spaces

The concept of "approximate" searching has applications in a vast number of fields. Some examples are images, fingerprints or audio databases; machine learning; image quantization and compression; text retrieval (for approximate string matching or for document similarity); genetic databases; etc.

All those applications have some common characteristics. There is a universe U of *objects*, and a nonnegative *distance function* $d : U \times U \longrightarrow R^+$ defined among them. This distance satisfies the three axioms that makes the set a *metric space*

$$\begin{array}{rcl} d(x,y) & = & 0 \quad \Leftrightarrow \quad x = y \\ d(x,y) & = & d(y,x) \\ d(x,z) & \leq & d(x,y) + d(y,z) \end{array}$$

where the last one is called the "triangular inequality" and is valid for many reasonable distance functions. The smaller the distance between two objects, the more "similar" they are. This distance is considered expensive to compute (e.g. comparing two fingerprints). We have a finite *database* $S \subseteq U$, which is a subset of the universe of objects and can be preprocessed (to build an index, for instance). Later, given a new object from the universe (a *query* q), we must retrieve all similar elements found in the database. There are different queries depending on the application, but the simplest one is: given a new element q and a maximum distance k, retrieve all the elements in the set which are at distance at most k from q.

This is applicable to our problem because we have a set of elements (the vocabulary) and the distance $ed()$ satisfies the stated axioms. A number of data structures exist to index the vocabulary so that the queries can be answered without inspecting all the elements. Our distance is *discrete* (i.e. gives integer answers), which determines the data structures which can be used. We briefly survey the main applicable structures now.

The first proposed structure is the Burkhard-Keller Tree (or BK-tree) [8], which is defined as follows: an arbitrary element $a \in S$ is selected as the root, whose subtrees are identified by integer values. In the i-th children we recursively build the tree for all elements in S which are at distance i from a. This process can be repeated until there is only one element to process, or there are no more than b elements (and we store a *bucket* of size b).

To answer queries of the form (q, k), we begin at the root and enter into all children i such that $d(a, q) - k \leq i \leq d(a, q) + k$, and proceed recursively (the other branches are discarded using the triangular inequality). If we arrive to a leaf (bucket of size one or more) we compare sequentially all the elements. We report all the elements x found that satisfy $d(q, x) \leq k$.

Another structure is called "Fixed-Queries Tree" or FQ-tree [3]. This tree is basically a BK-tree where all the elements stored in the nodes of the same level are the same (and of course do not necessarily belong to the set stored in the subtree), and the real elements are all in the leaves. The advantage of such construction is that some comparisons are saved between the query and the nodes along the backtracking that occurs in the tree. If we visit many nodes of the same level, we do not need to perform more than one comparison per level. This is at the expense of somewhat taller trees. Another variant is proposed in [3], called "Fixed-Height FQ-trees", where all the leaves are at the same depth h, regardless of the bucket size. This makes some leaves deeper than necessary, which makes sense because we may have already performed the comparison between the query and one intermediate node, therefore eliminating for free the need to compare the leaf. In [17], an intermediate structure between BK-trees and FQ-trees is proposed.

An analysis of the performance of FQ-trees is presented in [3], which disregarding some complications can be applied to BK-trees as well. We show the results in the Appendix. We also give an analysis of fixed-height FQ-trees which is new.

Some approaches designed for continuous distance functions , e.g. [19, 23, 7, 10], are not covered in this brief review. The reason is that these structures do not use all the information obtained from the comparisons, since this cannot be done in continuous spaces. This is, however, done in discrete spaces and this fact makes the reviewed structures superior to those for continuous spaces, although they would not be directly applicable to the continuous case. We also do not cover algorithms which need $O(n^2)$ space such as [21] because they are impractical for our application.

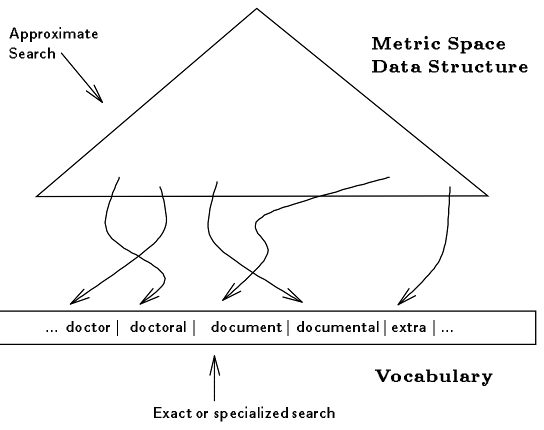
Approximate Search
Metric Space Data Structure
... doctor | doctoral | document | documental | extra | ...
Vocabulary
Exact or specialized search

Figure 2. Proposed data structure.

3 The Vocabulary as a Metric Space

Traversing the whole vocabulary online is like comparing the query against the whole database in a metric space. Our proposal is to organize the vocabulary such as to avoid the complete online traversal. This organization is based on the fact that we want, from a set of words, those which are at edit distance at most k from a given query. The edit distance $ed()$ used satisfies the axioms which make it a metric, in particular a discrete metric.

The proposal is therefore, instead of storing the vocabulary as a sequence of words, organize it as a metric space using one of the available techniques. The distance function to use is $ed()$, which is computed by dynamic programming in time $O(m_1 m_2)$, where m_1 and m_2 are the lengths of the two words to compare. Although this comparison takes more than many efficient algorithms, it will be carried out only a few times to get the answer. On the other hand, the dynamic programming algorithm is very flexible to add new editing operations or changing their cost, while the most efficient online algorithms are not that flexible.

Figure 2 shows our proposed organization. The vocabulary is stored as a contiguous text (with separators among words) where the words are sorted. This allows exact or prefix retrieval by binary search, or another structure can be built onto it. The search structure to allow errors goes on top of that array and allows approximate or exact retrieval.

An important difference between the general assumptions and our case is that the distance function is not so costly to compute as to make negligible all other costs. For instance, the space overhead and non-locality of accesses incurred by the new search structures could eliminate the advantage of comparing the query against less words in the vocabulary. Hence, we do not consider simply the number of comparisons but the complete CPU times of the algorithms, and compare them against the CPU times of the best sequential search algorithms run over the complete vocabulary. Moreover, the efficiency in all cases depends on the number of errors allowed (all the algorithms worsen if more errors are allowed). We have also to consider the extra space incurred because the vocabulary is already large to fit in main memory. Finally, although the asymptotic analysis of the Appendix shows that the number of traversed nodes is sublinear, we must verify how does this behave for the vocabulary sizes which are used in practice.

It is interesting to notice that any structure to search in a metric space can be used for exact searching, since we just search allowing zero errors (i.e. distance zero). Although not as efficient as data structures designed specifically for exact retrieval (such as hashing or binary search), the search times may be so low that the reduced efficiency is not as important as the fact that we do not need an additional structure for exact search (such as a hash table).

4 Experimental Results

We show experimentally the performance obtained with our metric space techniques against online algorithms. We ran our experiments on a Sun UltraSparc-1 of 167 MHz, with 32 Mb of RAM, running Solares 2.5.1.

We tested three different structures: BK-trees (BKT), FQ-trees (FQT) and FQ-trees of fixed height (FQH). For the first two we tested buckets of size 1, 10 and 20; while for the last one we tested fixed heights of 5, 10 and 15. As explained before, other structures for metric spaces are not well suited to this case (we verified experimentally this fact). We used the 500,000 words (5 Mb) vocabulary of the English TREC collection (1 Gb). The vocabulary was randomly permuted and separated in 10 incremental subsets of size 50,000 to 500,000.

Our first experiment deals with space and time overhead of the data structures that implement the search in a metric space, and its suitability for exact searching. Figure 3 shows the results. As it can be seen, build times are linear for FQH (exactly h comparisons per element) and slightly superlinear ($O(n \log n)$ in fact, since the height is $O(\log n)$) for BKT and FQT. The overhead to build them is normally below 2 minutes, which is a small percentage (10% at most) of the time normally taken to build an index for a 1 Gb text database.

If we consider extra space, we see that the BKT poses a fixed space overhead, which reaches a maximum of 115% for $b = 1$. This corresponds to the fact that the BKT stores at most one node per element. The space of the FQT is slightly superlinear (the internal nodes are empty) and for this experiment is well above 200% for $b = 1$. Finally, the space of the FQH tends to a constant, although in our case is very large except for $h = 5$[1] (the case $h = 15$ is above 500% and is not shown).

Finally, we show that the work to do for exact searching involves a few distance evaluations (20 or less) with very low growth rate (logarithmic). This shows that the structure can be also used for exact searching. The exception is FQH ($h = 5$), since the FQH is $O(n)$ time for fixed h, and this is noticed especially for small h (it grows linearly from 100 to 1000 and is not shown).

We show in Figure 4 the query performance of the indices to search with one error. As it can be seen, no more than 5-8% of the dictionary is traversed (the percentage is decreasing since the number of comparisons are sublinear except for FQH). The user times correspond quite well to the number of comparisons. We show the percentage of user times using the structures versus the best online algorithm for this case [6] (as implemented in [4]). As it can be seen, for the maximum dictionary size we reach 40% of the online time for the best metric structures. From those structures, we believe that BKT with $b = 1$ is the best choice, since it is faster than all the FQT's (and takes less space) and it is similar to FQH ($h = 15$) and takes much less space. Another alternative which takes less space (close to 70%) is BKT with $b = 10$, while it achieves 60% of the times of online searching.

The result for two errors (not shown) is not so good. This time the metric space algorithms do not improve the online search, despite that the best ones traverse only 17%-25% of the vocabulary. The reason is that the offline algorithms are much more sensitive to the error level than the online ones. This shows that our scheme is only useful to search with one error.

Table 1 shows the results of the least squares fitting over the number of comparisons performed by the different data structures. For $k = 0$ we obtain a good logarithmic approximation, while the bucket size seems to affect the constant rather than the multiplying factor. The exception is the FQH, which is $O(n)$ (and the constant is very close to h as expected).

For $k = 1$, the results confirm the fact that the structures inspect a sublinear number of nodes. Notice that the exponent is smaller for BKT than for FQT, although the last ones have a better constant. The constant, on the other hand, seems to keep unchanged when the bucket size varies (only the exponent is affected). This allows to extrapolate that BKT will continue to improve over FQT for larger data sets[2]. The FQH, on the other hand, shows clearly that it are in fact linear for fixed h (this can be changed if h is taken as a function of n, but we have not done this yet).

The results for $k = 2$ increase the exponent (which will be close to 1 for $k = 3$). The exception is FQH, which increases a lot the constant (its exponent cannot possibly increase). The percentual error is between 15% and 20% in all cases.

The least squares fitting over the real CPU times give similar growth rates, for instance it is $O(n^{0.65})$ for BKT ($b = 1$).

5 Conclusions

We proposed a new method to organize the vocabulary of inverted files in order to support approximate searching on the indexed text collection. Most present methods rely on a sequential search over the vocabulary words using a classical online algorithm. We propose instead to organize the vocabulary as a metric space, taking advantage of the fact that the edit distance that models the approximate search is indeed a metric. This method can also be applied to other problems when a dictionary is searched allowing errors, such as spelling applications.

We show in our experiments over a 5 Mb vocabulary of a 1 Gb text, that the best data structure for this task is the Burkhard-Keller tree with no buckets. That structure allows, with almost negligible construction time and reasonable space overhead (100% extra over the space taken by the plain vocabulary), to search close to 5%-8% of the dictionary for one error and 17%-25% for two errors. This cuts down the times of the best online algorithms to 40%-60% for one error, although for two errors the online algorithms (though traversing the whole dictionary) are faster. We have shown experimentally that those trees, as well as the Fixed-Queries trees, perform a sublinear number of comparisons, close to $O(n^{0.6..0.7})$ for 1 error. We also present the first analysis for fixed-height Fixed Queries trees.

Our implementation of the BK-trees is not optimized for space. We estimate that with a careful imple-

[1]It is possible to have h as a function of n, but we cover the reasonable range here by showing three fixed values

[2]It is well known that all the conclusions about metric space data structures depend strongly on the particular space and distance function, so this does not allow a generalization to other cases.

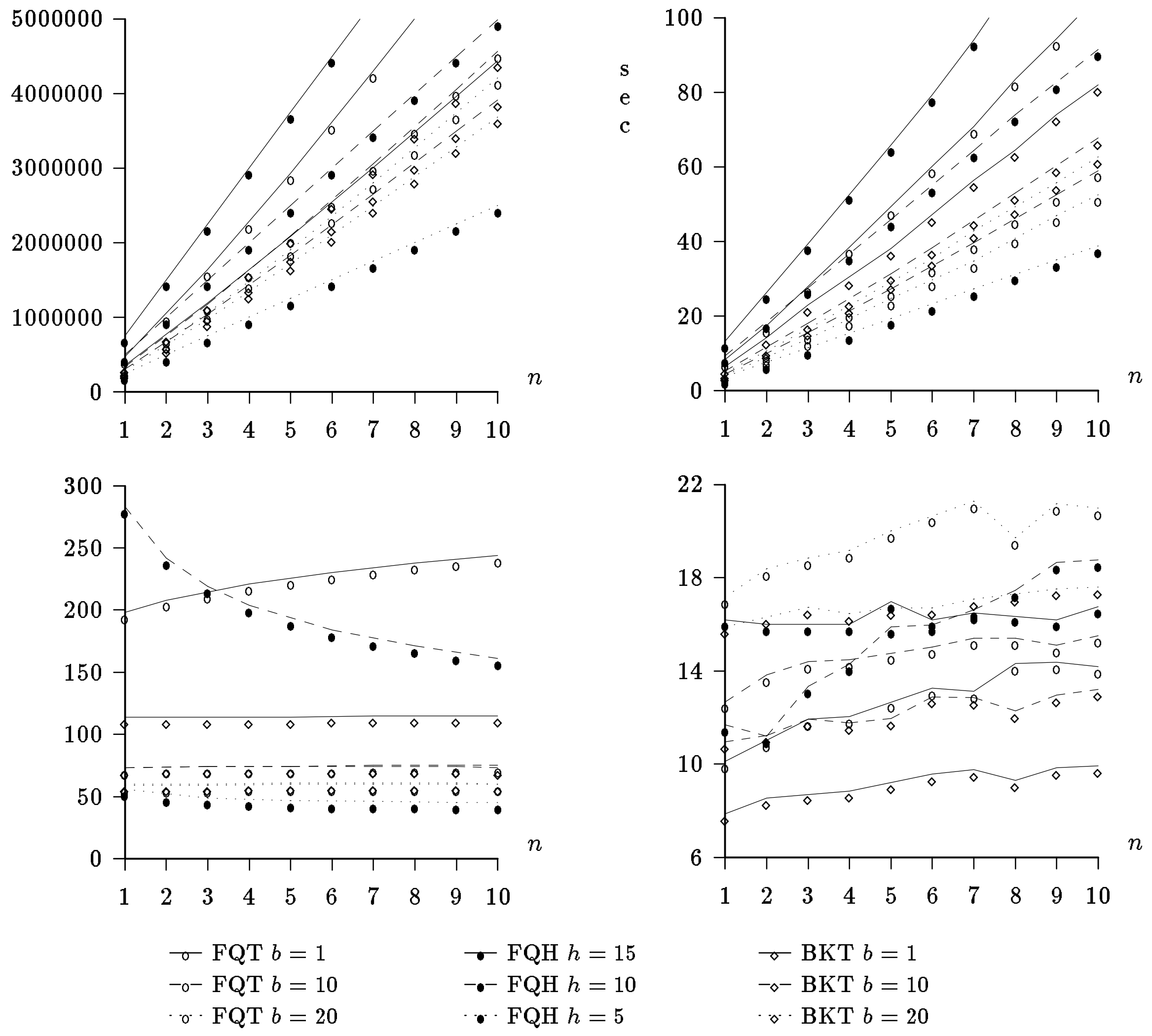

Figure 3. Comparison of the data structures. From top to bottom and left to right, number of distance evaluations and user times to build them, extra space taken over the vocabulary size, and number of distance evaluations for exact search. The x axis is expressed in multiples of 50,000.

Structure	$k = 0$	$k = 1$	$k = 2$
BKT $(b = 1)$	$0.87 \ln(n) - 1.52$	$2.25\ n^{0.639}$	$1.91\ n^{0.822}$
BKT $(b = 10)$	$0.96 \ln(n) + 0.39$	$2.21\ n^{0.673}$	$1.52\ n^{0.859}$
BKT $(b = 20)$	$0.69 \ln(n) + 8.36$	$2.16\ n^{0.691}$	$1.42\ n^{0.871}$
FQT $(b = 1)$	$1.91 \ln(n) - 10.84$	$0.36\ n^{0.777}$	$0.54\ n^{0.926}$
FQT $(b = 10)$	$1.17 \ln(n) + 0.26$	$0.50\ n^{0.798}$	$0.63\ n^{0.921}$
FQT $(b = 20)$	$1.73 \ln(n) - 1.58$	$0.49\ n^{0.814}$	$0.69\ n^{0.919}$
FQH $(h = 5)$	$2.3 \times 10^{-3} n + 6.27$	$0.15\ n^{0.998}$	$0.46\ n^{0.992}$
FQH $(h = 10)$	$1.7 \times 10^{-5} n + 10.61$	$0.04\ n^{1.006}$	$0.30\ n^{1.004}$
FQH $(h = 15)$	$1.1 \times 10^{-6} n + 16.02$	$0.02\ n^{0.992}$	$0.26\ n^{0.994}$

Table 1. Least squares fitting for the number of comparisons made by the different data structures.

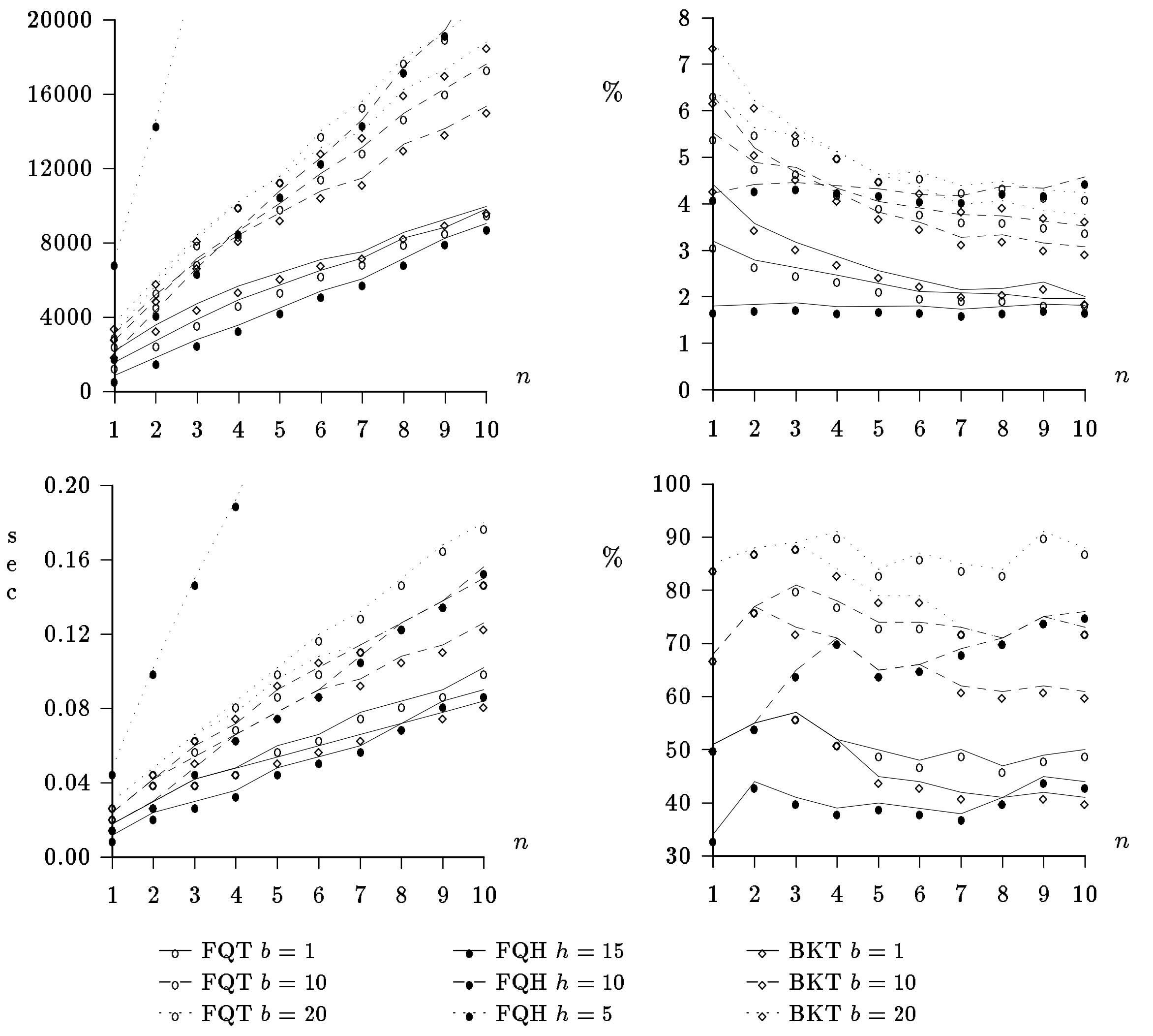

Figure 4. Search allowing one error. The first row shows the number of comparisons (on the left, absolute number, on the right, percentage over the whole dictionary). The second row shows user times (on the left, seconds, on the right, percentage over the best online algorithm). The x axis is expressed in multiples of 50,000.

mentation the overhead can be reduced from 100% to 65%. This overhead is quite reasonable in most cases. We also leave for future work putting h as a function of n for fixed-height Fixed-Queries trees, so that they also show their sublinear behavior that we have analytically predicted in this paper.

References

[1] M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In *Proc. WSP'97*, pages 2–20. Carleton University Press, 1997.

[2] R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, Sep 1992.

[3] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. CPM'94*, LNCS 807, pages 198–212, 1994.

[4] R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 1–23, 1996.

[5] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. In *Proc. ACM CIKM'97*, pages 1–8, 1997.

[6] R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. In *Proc. CPM'92*, pages 185–192, 1992. LNCS 644.

[7] S. Brin. Near neighbor search in large metric spaces. In *Proc. VLDB'95*, pages 574–584, 1995.

[8] W. Burkhard and R. Keller. Some approaches to best-match file searching. *CACM*, 16(4):230–236, 1973.

[9] W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, pages 172–181, 1992. LNCS 644.

[10] C. Faloutsos and K. Lin. Fastmap: a fast algorithm for indexing, data mining and visualization of traditional and multimedia datasets. *ACM SIGMOD Record*, 24(2):163–174, 1995.

[11] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. TREC-3*, pages 1–19, 1995. NIST Special Publication 500-207.

[12] J. Heaps. *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, 1978.

[13] G. Landau and U. Vishkin. Fast string matching with k differences. *J. of Computer and Systems Science*, 37:63–78, 1988.

[14] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. Technical Report 93-34, Dept. of CS, Univ. of Arizona, Oct 1993.

[15] G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic progamming. In *Proc. CPM'98*. Springer-Verlag, 1998. To appear.

[16] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.

[17] M. Shapiro. The choice of reference points in best-match file searching. *CACM*, 20(5):339–343, 1977.

[18] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. ESA'95*, 1995. LNCS 979.

[19] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

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

[21] E. Vidal. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[22] S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, 1992.

[23] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. ACM-SIAM SODA'93*, pages 311–321, 1993.

Appendix. Analysis of Fixed-Height FQ-trees

We call p_i the probability that two random elements from U are at distance i. Hence, $\sum_{i\geq 0} p_i = 1$, and $p_{-i} = 0$ for $i > 0$. In [3] the FQ-trees are analyzed under the simplifying assumption that the p_i distribution does not change when we enter into a subtree (their analysis is later experimentally verified). They show that the number of distance evaluations done to search an element with tolerance k (in our application, allowing k errors) on an FQ-tree of bucket size b is

$$P_k(n) = O(n^\alpha)$$

where $0 < \alpha < 1$ is the solution of

$$\sum_{i\geq 0} \gamma_i(k) p_i^\alpha = 1$$

where $\gamma_i(k) = \sum_{j=i-k}^{i+k} p_j$. This P_k result is the sum of the comparisons done per level of the tree (a logarithmic term) plus those done at the leaves of the tree, which are $O(n^\alpha)$.

The CPU cost depends also on the number of traversed nodes $N_k(n)$, which is also shown to be $O(n^\alpha)$ (the constant is different). Finally, the number of distance evaluations for an exact search is $O(b + \log n)$.

Under the same simplifying assumption the analysis applies to BK-trees too. The main difference is that the number of comparisons is for this case the same as the number of nodes traversed plus the number of leaf elements compared, which also adds up $O(n^\alpha)$ (although the constant is higher). The distribution of the tree is different but this difference is overriden by the simplifying assumptions anyway.

We analyze now FQ-trees of fixed height. The analysis is simpler than for FQ-trees. Let $F_k^h(n)$ be the number of elements not yet filtered by a proximity search of distance up to k after applying h fixed queries. Then, the expected number of comparisons for a proximity query is

$$P_k^h(n) = h + F_k^h(n)$$

Let β_k be the probability of not filtering an element when doing the proximity search at distance k. If an element is at distance i to a query, it is not filtered with probability $\sum_{j=i-k}^{i+k} p_j$. The element is at distance i with probability p_i, so

$$\beta_k = \sum_{i \geq 0} p_i \sum_{j=i-k}^{i+k} p_j$$

Note that β_k converges to 1 when k increases. So, the expected number of elements not filtered between two consecutive levels are related by $F_k^h(n) = \beta_k F_k^{h-1}(n)$. Clearly, $F_k^0 = n$, so $F_k^h(n) = \beta_k^h n$. Because $F_k^h(n)$ decreases when h grows, the optimal h is obtained when $P_k^h(n) \leq P_k^{h+1}(n)$. That is, when

$$h + \beta_k^h n \leq h + 1 + \beta_k^{h+1} n$$

Solving, we obtain the optimal h for a given k

$$h_k = \frac{\log(n(1-\beta_k))}{\log(1/\beta_k)}$$

Replacing this h in $P_k^h(n)$ we get

$$P_k(n) = \frac{\log(n(1-\beta_k))}{\log(1/\beta_k)} + \frac{1}{1-\beta_k}$$

That is, $P_k(n)$ is logarithmic for the optimal h_k (and linear for a fixed h). This is asymptotically better than the $O(n^\alpha)$ results for FQ-trees and BK-trees. Nevertheless, the constant factor in the log term grows exponentially with k, so this is good for small to medium k.

To obtain this logarithmic behavior, the fixed height must increase as the number of elements grows (i.e. $h_k = O(\log n)$). Unfortunately the optimal height is dependent on the search tolerance k. However, the logarithmic cost can be maintained even for non-optimal h provided we use $h = \Theta(\delta \log n)$, where $\delta \geq 1/\log 1/\beta_k$ (i.e. we overestimate the optimal height).

On the other hand, the number of nodes visited is bigger than in FQ-trees. In fact, using a recurrence similar to the one for FQ-trees, it is possible to show that the number of nodes visited is $O(h_k n^\alpha)$ for $\alpha < 1$ which could easily be larger than n even for small k. So, these trees are good when the cost of comparing two elements is very high, like comparing two genetic sequences, polygons or graphs.

A related problem is the size of the data structure, which can be superlinear. In fact, it is possible that the optimal h cannot be used in many applications because of space limitations (for instance, we could hardly reach the limit $h = 15$ in this work).

Another problem is that the variance of the number of elements filtered per level is large (increases with every level), so we may need more queries in practice to achieve the desired filtering.

To decrease the number of nodes visited, we may compress paths of degree 1 by using the same idea of Patricia trees. We can store in every node which fixed query (or how many we have to skip) we have to use in that node. Still, we cannot compress all the nodes if we want to filter that element. Another idea, instead of fixing the height, is fixing the probability of filtering in every path recursively.