# Fast Approximate String Matching in a Dictionary

Ricardo Baeza-Yates  Gonzalo Navarro

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

**Abstract**

A successful technique for approximate searching on large indexed textual databases relies on an on-line search in the vocabulary of the text. This works well because the vocabulary is relatively small (i.e. a few megabytes for gigabytes of text), and therefore the search takes a few seconds at most. While those times are appropriate for single-user environments, they are inappropriate for multi-user setups such as a text database server for the Web. We present a speed-up technique for on-line searching in the vocabulary which needs only a 10% overhead. We also propose to exploit the fact that the problem involves a definition of similarity among words which respects the triangular inequality to structure the vocabulary in such a way that it is not necessary to traverse it completely. We show that the improvement in time is very significative and pays for the extra space needed.

## 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 on-line version of the problem, the pattern can be preprocessed but the text cannot. The classical solution uses dynamic programming and is $O(mn)$ time [27, 23]. Later, a number of algorithms improved this to $O(kn)$ time in the worst case or even less on average, by using cleverly the properties of the dynamic programming matrix (e.g. [13, 17, 32, 11, 36]) or by using an automaton which is used in deterministic or nondeterministic form [35, 4, 21]. Another trend is that of "filtration" algorithms: a fast filter is run over the text quickly discarding uninteresting parts. The interesting parts are later verified with a more expensive algorithm. Examples of filtration approaches are [29, 6]. Some are "sublinear" in the sense that they do not inspect all the text characters, but the on-line problem is $\Omega(n)$ if $m$ is taken as constant.

If the text is large and has to be searched frequently, even the fastest on-line algorithms are not practical, and preprocessing the text becomes necessary. This is especially true for very large text databases, which take gigabytes, while the fastest on-line search algorithms can process a few megabytes per second. We are interested in large text 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.

Although many indexing methods have been developed for exact string matching from a long time ago [34], only a few years ago indexing text for approximate string matching was considered one of the main open problems in this area [35, 2]. The practical indices which are in use today rely on an on-line search in the vocabulary of the text, which is quite small compared to the text itself.

The fastest on-line approximate search algorithms run at 1-4 megabytes per second, and therefore they find the answer in the vocabulary in a few seconds. While this is acceptable for single-user environments, the search time may be excessive in a multi-user environment. For instance, a Web search engine which receives many requests per second cannot spend four seconds to traverse the vocabulary.

We present two proposals in this paper. A first one is a speed-up for the normal on-line traversal which exploits the fact that consecutive strings in a sorted dictionary tend to share a prefix. This speedup costs only 10% extra space.

Our second proposal needs more extra space: organize the vocabulary such as to avoid the complete on-line 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 respects the axioms which make it a metric, and therefore we may apply any data structure to search in metric spaces. This imposes normally a space overhead over the vocabulary, but the reward is an important improvement in search times.

We experimentally compare all the different structures for metric spaces accounting for the search time and space overhead, and compare also the different on-line variations.

This paper is organized as follows. In Section 2 we explain how the current indices for approximate string matching work. In Section 3 we survey the main techniques to search in metric spaces. In Section 4 we explain our setup to speed up the on-line search in the vocabulary, and in Section 5 we explain our method to avoid the on-line traversal. In Section 6 we show experimental results. In Section 7 we give our conclusions and future work directions.

## 2 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, because 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 these 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.

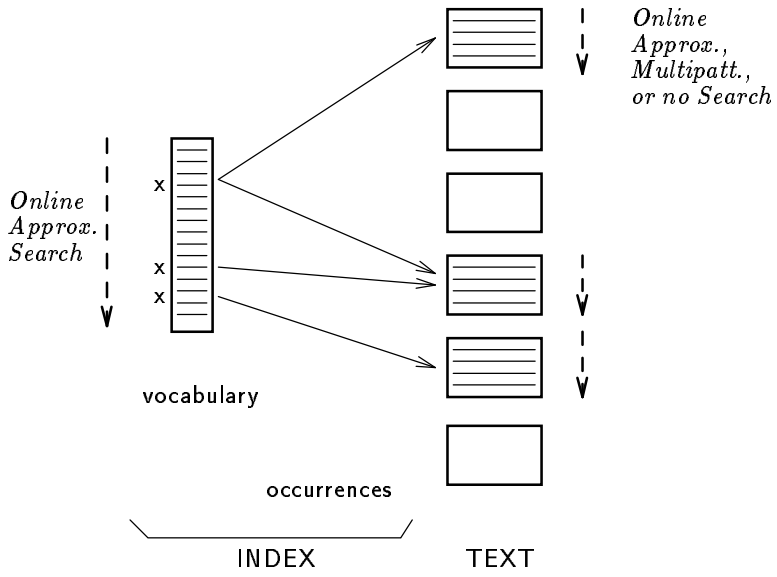


Figure 1: Approximate searching on an inverted index. In the case of full inverted indices the text traversal is not necessary. In the case of block addressing text traversal may or may not be necessary.

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 on-line 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 [19, 5] (which reduce space requirements by making the occurrences point to blocks of text instead of exact positions) 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 2 Gb TREC collection [14] the vocabulary takes no more than 2 Mb. An empirical law known as Heaps Law [15] states that the vocabulary for a text of $n$ words grows as $O(n^\beta)$, where $0 < \beta < 1$. In practice, $\beta$ is between 0.4 and 0.6 [1]. The fastest on-line approximate search algorithms run at 1-4 megabytes per second (depending on some parameters of the problem), and therefore they find the answer in the vocabulary in a few seconds. While this is acceptable for single-user environments, the search time may be excessive in a multi-user environment. For instance, a Web search engine which receives many requests per second cannot spend four seconds per query.

## 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 honors the three axioms that makes the set a *metric space*

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

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 consider 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 three typical queries of this kind:

($a$) Retrieve all elements which are within distance $k$ to $q$. This is, $\{x \in S \ / \ d(x,q) \leq k\}$.

($b$) Retrieve the closest elements to $q$ in $S$. This is, $\{x \in S \ / \ \forall y \in S,\ d(x,q) \leq d(y,q)\}$. In some cases we are satisfied with one such element. We can also give a maximum distance $r$ such that if the closest element is at distance more than $r$ we do not want anyone reported.

($c$) Retrieve the $i$ closest elements to $q$ in $S$. This is, retrieve a set $A \subseteq S$ such that $|A| = i$ and $\forall x \in A, y \in S - A, d(x,q) \leq d(y,q)$.

Given a database of $n$ objects, all those queries can be trivially answered by performing $n$ distance evaluations. The goal is to structure the database such that we perform less distance evaluations.

This is applicable to our problem because $ed()$ indeed satisfies the axioms and therefore the search in the vocabulary is an instance of this problem. In our case the database is the set of all the different words of the text, and we are interested in queries of type ($a$). Moreover, our distance is *discrete* (i.e. gives integer answers), which is of importance for the types of data structures which can be applied. We briefly survey the main applicable structures now.

Probably the first general solution to search in metric spaces was presented in [10]. They propose a tree (thereafter called Burkhard-Keller Tree, or bk-tree), which is suitable for discrete distance functions like $ed()$. It is defined as follows: an arbitrary element $a \in S$ is selected as the root, and it has a number of children. 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 $b$), or the tree has a given height $h$.

To answer queries of type $(a)$, where we are given a query $q$ and a distance $k$, 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. Each time we perform a comparison where $d(q,x) \leq k$, we report the element $x$.

In [28], the use of more than one element per node of the tree is proposed. Those elements allow to eliminate more points per level at the cost of doing more distance evaluations. The difference with a tree that uses those points successively downwards the tree is that the query is compared against all the points of the node no matter which the result is.

The advantage of the previous idea is made clear in a further development, called "Fixed-Queries Trees" of fq-trees [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). 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. This is at the expense of somewhat taller trees. They show that their approach is superior to bk-trees. They propose a variant which is 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.

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 present the results in the Appendix. We also present an analysis of fixed-height fq-trees which is new.

An algorithm which is close to all the presented ideas but performs surprisingly better by an order of magnitude is [33]. They select a point $a \in S$ at random and measure $d = d(a,q)$, eliminating all elements $x$ of $S$ which do not satisfy $d - k \leq d(x,s) \leq d + k$. This is repeated until few enough elements remain in the set. Although very similar to bk-trees, the key difference is that the second element to compare against $q$ is selected from the remaining set, instead of from the whole set as in bk-trees. This means that this algorithm is more likely to compare the query against a *centroid* of the remaining set (i.e. an element whose distance distribution against the rest favors smaller values). This is because the distance distribution tends to be very centered (which is bad for all range search algorithms) and the selection of a centroid distributes the distances better.

The problem with the algorithm [33] is that it needs $O(n^2)$ space and build time. In this sense it is close to [25]. This is unacceptably high for all by very small databases.

Some approaches designed for continuous distance functions [31, 37, 8, 9, 12, 24] 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. It can, however, be done (and it is done) in discrete spaces and this fact makes the reviewed structures superior to these ones, although they would not be directly applicable in continuous spaces.

```
...  doctor
 6   doctoral
 4   doctrine
 3   document
 8   documental
 0   extra
```

Figure 2: Example of a short section of a vocabulary with prefix information added.

## 4 Speeding Up the On-line Search

Except for filtration algorithms, all the on-line approximate search algorithms traverse the text character by character. They store a *context*, which is the state of the search. For each new character read they modify their context. Whenever their context indicates a match they report it. For instance, if the search is done with a deterministic finite automaton as in [21], the context is simply the current state of the automaton. When run over a vocabulary, their processing is very similar, except because the context is initialized for each new word to process.

If the vocabulary is stored in lexicographical order (which is useful to binary search on it for exact retrieval), each word will share a prefix with the previous word. The larger the vocabulary, the longer the shared prefixes. This property has been used in [22, 7, 20], for instance to compress the vocabulary (since the prefix shared with the previous word needs not be stored). However, direct access is complicated in those compression schemes. Figure 2 shows an example.

We propose to use that property in a different form. We store the complete words, as well as an additional byte which tells the length of the prefix shared with the previous word. The search algorithm will not change except because it will store all the contexts that it traversed from the beginning of the word. That is, it will keep a stack of contexts, and each time a new character is read, the current context is pushed onto the stack before being modifying according to the new character. When the word is finally traversed, we have all the traversed contexts in the stack.

If the next word shares a prefix of length $\ell$ with the word just processed, we do not need to reprocess the first $\ell$ characters. We just take the $\ell$-th context of the stack instead of the initial one and process the string from the $(\ell+1)$-th character on.

This has the additional overhead of storing the contexts instead of just replacing them, which makes the strategy to work better for algorithms where the context is very small. On the other hand, since we always search words (which are rarely longer than 10 letters), we need also algorithms which are especially efficient for short words. Fortunately, both requirements match since the fastest algorithms for short patterns are [4, 21], which use a very small context (we exclude filtration algorithms because the technique is not applicable to them).

The first algorithm simulates using bit-parallelism the behavior of a non-deterministic finite automaton that searches the pattern allowing errors. It uses just one computer word whenever $(m-k)(k+2) \leq w$, where $m$ is the length of the pattern, $k$ is the number of errors and $w$ is the number of bits in the computer word. For instance it can search with $m$ up to 9 in a 32-bit architecture. Although in the original work they show how to use many computer words for longer

patterns, in our case this will occur very infrequently, and when it occurs the pattern will be one or two letters longer. We prefer therefore to prune longer patterns and to verify after a match if there is indeed an occurrence of the complete pattern.

The second algorithm converts the automaton to deterministic form, building only the states which are actually reached in the text traversal. It is shown in [21] to be very efficient on short patterns.

To analyze the expected improvement, we notice that the number of letters that will be effectively traversed by the optimized algorithm is exactly the number of nodes of a trie [16] built over all the words of the vocabulary. This is because, if we consider all the prefixes of all words, we work only once on each *different* prefix. On the other hand, each node of a trie represents a different prefix. The original algorithm, on the other hand, will work on every character of every word.

Not all the characters of all words are present in a trie built from the vocabulary, since once the prefix of a word is unique the trie is not further expanded but the word is stored in a leaf. The parts of the words which are not represented in the trie are worked on in all cases. The difference is in the letters represented in the trie: while the optimized algorithm works once per internal node, the amount of work of the original algorithm is proportional to the *external path length*, which is the sum of the depth of all leaves (i.e. the sum of the lengths of all words, up to where they are represented in the trie).

In [30], some asymptotical statistics are computed on a suffix trie, for large $n$, using a Markovian model (which is quite good for natural language). Statistics for random tries are equivalent to suffix tries over a random text, except for $o(1)$ terms [26]. We take here the simpler case of independent character generation (i.e. a Markovian model with no memory). The only difference in the general case is the constant factor of the results, not the order. The reader is referred to [30] for more details.

Suppose our alphabet is composed from a finite or infinite number of symbols, call $q_i$ the probability of the $i$-th symbol, and call $H = \sum_i q_i \log(1/q_i)$ the entropy of the language. Then the external path length is $E_n = (n \ln n)/H = O(n \log n)$, while the number of internal nodes is $S_n = n/H = O(n)$ (this last result is taken from [18], for random tries). Therefore, we work in $n(\ln n - 1)/H$ less characters. Except for the parts of the strings stored in the leaves, we work $O(n)$ instead of $O(n \log n)$. Unfortunately, the part stored in the leaves is important and is proportional to the part stored in the trie in practice.

Finally, notice that our proposal is similar to that of storing a trie with the vocabulary and run the algorithms recursively on the trie to factor out repetitions. However, our technique is faster and has much less memory overhead.

We found experimentally, however, that the technique we are proposing here is of no use against the fastest non-filtering algorithms [4, 21]. This is because the algorithms are extremely efficient and the amount of repetition in the prefixes is not large enough to counter the accesses to the stack of contexts (the stack cannot be put in registers). The extra accesses to the stack eliminate the advantage for the less letters considered.

However, we believe that this idea can still have use for more complex edit distances, where the fastest algorithms cannot be applied and we must resort to the classical $O(mn)$ algorithm. This study is part of our future work.

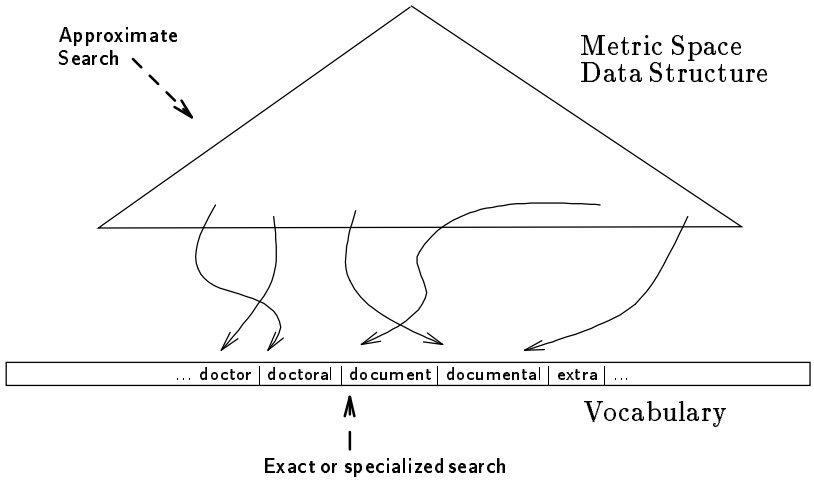


Figure 3: Proposed data structure.

## 5 The Vocabulary as a Metric Space

Traversing the whole vocabulary on-line is like comparing the query against the whole database in a metric space. Our proposal in this section is to organize the vocabulary such as to avoid the complete on-line 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 on-line algorithms are not that flexible.

Figure 3 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). Finally, we have to consider the extra space incurred

because the vocabulary is already large.

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).

# 6 Experimental Results

We show experimentally the performance obtained with our metric space techniques against on-line algorithms. The results are preliminary and must be tested on larger setups. We ran our experiments on a Sun SparcClassic with 16 Mb of RAM, running SunOS 4.1.3.

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 with GNATs and gh-trees). We used a Spanish dictionary composed of more than 80,000 words (which is still modest compared to the 500,000 words of the TREC collection which will be used in future work). The set was randomly permuted and separated in 8 incremental subsets of size 10,000 to 80,000.

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 4 shows the results. As it can be seen, build times are linear for FQH and slightly superlinear ($O(n \log n)$ in fact) for BKT and FQT. The overhead to build them is normally below a minute, which is a small percentage of the time normally taken to build an index for a text database whose vocabulary is of 80,000 words.

If we consider extra space, we see that BKT and FQT pose a fixed space overhead, of 100% or less (with respect to the size of the vocabulary with no more data), with the exception of FQT for $b = 1$ which is 200%. As an index normally has another 100% overhead over the plain vocabulary to store pointers to the index, we can consider that the extra overhead is in fact closer to 50-100%. This is not negligible but acceptable. The FQH indices pose a fixed extra space, whose overhead tends to zero as the vocabulary grows. However, these percentages are quite large for reasonably-sized dictionaries, except for small heights.

Finally, we show that the work to do for exact searching involves a few distance evaluations (16 or less) with very low growth rate (logarithmic). This shows that the structure can be also used for exact search. The exception is FQH ($h = 5$), since these structures are $O(n)$ time for fixed $h$, and this is noticed especially for small $h$.

We show in Figure 5 the query performance of the indices to search with one error. As it can be seen, no more than 10% 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 it can be seen, for the maximum dictionary size we reach 40% of the online time for many metric structures (this percentage will improve for BKT and FQT in larger dictionaries). From those structures, we believe that FQT and BKT with $b = 1$ are the best choices, since they are sublinear and they have a reasonable overhead (in contrast to FQH). For

larger dictionaries, FQH with $h = 10$ could also be a good choice.

Figure 6 shows the result with two errors. This time the online algorithm selected was [4] and the metric space algorithms do not improve the online search. The reason is that the offline algorithms are much more sensitive to the error level than the online algorithm used. This shows that our scheme is only useful to search with one error.

We also tested the search for the nearest neighbor, and the results are very similar to a search with $k$ equal to the distance to that nearest neighbor.

# 7 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).

We show in our preliminary experiments that the best data structures for this task are Burkhard-Keller trees or Fixed-Queries trees, using no buckets. Those structures allow, with almost negligible construction time and reasonable space overhead (50%-100% extra over typical space taken by the vocabulary, which is already very small), to search close to 5% of the dictionary for one error and 25% for two errors. This cuts down the times of the best online algorithms to 40% for one error, although for two errors the online algorithms (though traversing the whole dictionary) are faster. For larger dictionaries, Fixed-Height fq-trees of height 10 could also be a good choice.

We determined also that other structures not aimed to discrete spaces are not well suited for this task, being their performance very inferior to the ones we presented. We also determined that a proposed idea to improve online search algorithms on a sorted vocabulary by skipping common prefixes, although theoretically appealing, is of no interest in practice. A study of this idea for the case of a more expensive algorithm (e.g. to compute a more complex distance function) is of interest, however.

Future work also involves repeating all the experiments on a larger machine and on a larger vocabulary, to obtain figures adequate to very large text databases. The dictionary used had 80,000 words, which corresponds to a text of less than 100 megabytes. The 2 gigabytes TREC collection has a vocabulary of 500,000 words and we plan to use that vocabulary.

# References


[1] M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In *Proc. 4th South American Workshop on String Processing, WSP'97*, 1997. Valparaíso, Chile. To appear.

[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. CIKM'97*, 1997. Las Vegas, Nevada, Nov 11-15. To appear.

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

[7] E. Barbosa and N. Ziviani. From partial to full inverted lists for text searching. In R. Baeza-Yates and U. Manber, editors, *Proc. of the Second South American Workshop on String Processing (WSP'95)*, pages 1–10, April 1995.

[8] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. Manuscript.

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

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

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

[12] 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.

[13] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. of Computing*, 19(6):989–999, 1990.

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

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

[16] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

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

[18] H. Mahmoud. *Evolution of Random Search Trees.* John Wiley & Sons, 1991.

[19] 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.

[20] G. Navarro. An optimal index for pat arrays. In *Proc. WSP'96*, pages 214–227, 1996. `ftp://-ftp.dcc.uchile.cl/pub/users/gnavarro/wsp96.1.ps.gz`.

[21] G. Navarro. A partial deterministic automaton for approximate string matching. In *Proc. of WSP'97*, pages 112–124. Carleton University Press, 1997. `ftp://ftp.dcc.uchile.cl/pub/-users/gnavarro/wsp97.2.ps.gz`.

[22] G. Navarro, J. Kitajima, B. Ribeiro, and N. Ziviani. Distributed generation of suffix arrays. In *Proc. CPM'97*, LNCS 1264, pages 102–115, 1997. `ftp//ftp.dcc.uchile.cl/pub/users/-gnavarro/cpm97.ps.gz`.

[23] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. of Molecular Biology*, 48:444–453, 1970.

[24] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. Technical Report CUCS-030-95, Dept. of Computer Science, Columbia University, NY, October 1995.

[25] D. Sasha and T. Wang. New techniques for best-match retrieval. *ACM TOIS*, 8(2):140–158, 1990.

[26] R. Sedgewick and P. Flajolet. *Analysis of Algorithms*. Addison-Wesley, 1996.

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

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

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

[30] Wojciech Szpankowski. Probabilistic analysis of generalized suffix trees. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proceedings of CPM'92 Third Annual Symposium on Combinatorial Pattern Matching*, pages 1–14, Tucson, Arizona, April 1992. Springer-Verlag. LNCS 644.

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

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

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

[34] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, New York, 1994.

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

[36] S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

[37] 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 $\beta_k$ 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 $\beta_k$ 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

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

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. While normal fq-trees or bk-trees are $O(n)$ size, fixed-height fq-trees can in principle be superlinear. In fact, we could not reach the optimal $h_k$ in our experiments because of space limitations.

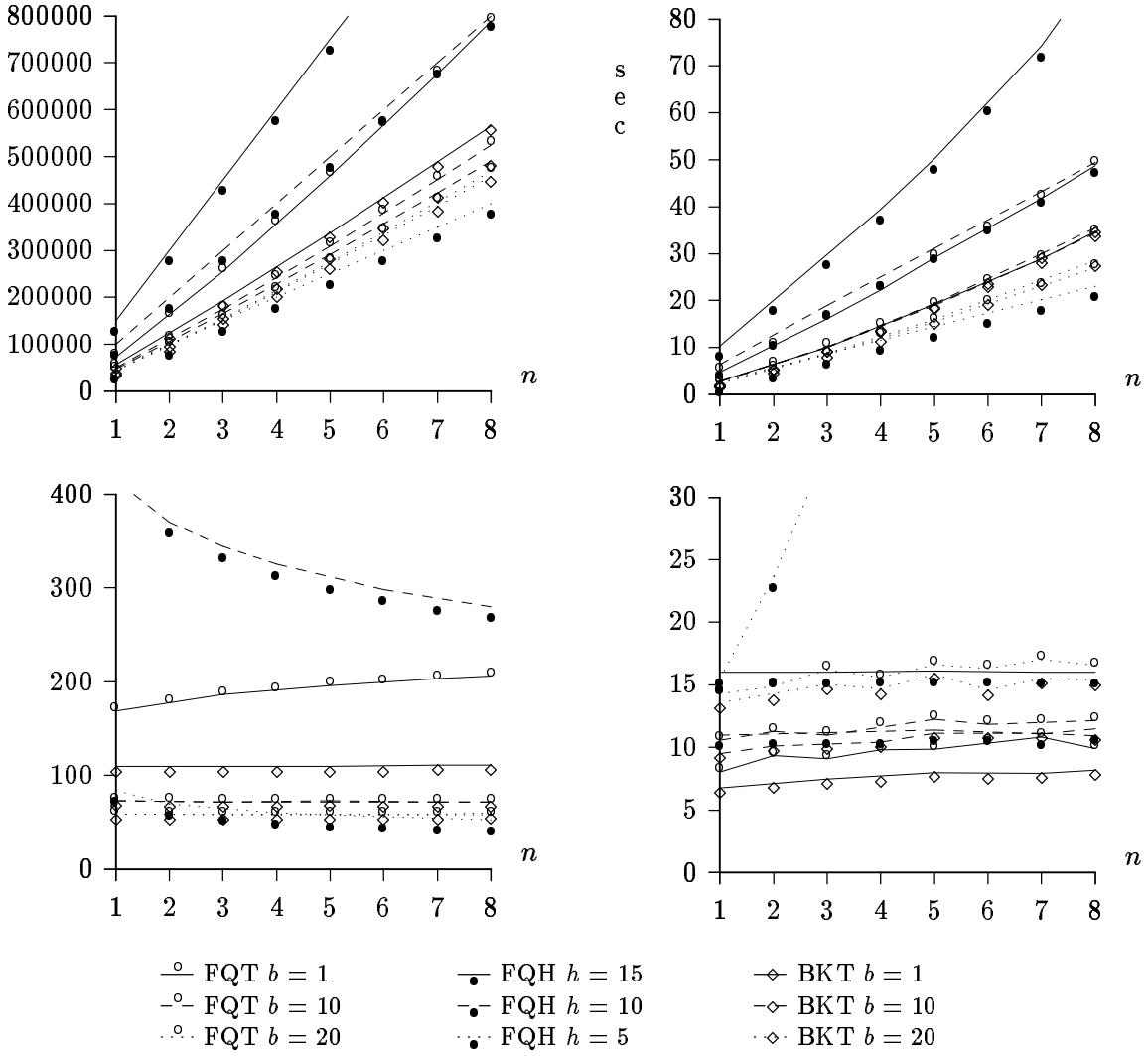


Figure 4: Comparison of the data structures. From top to bottom and left to right, number of distance evaluations to build the structures, user times to build the structures, extra space taken by the structures as a percentage of the size of the vocabulary (FQH $h = 15$ is close to 700) and number of distance evaluations for exact search (FQH $h = 5$ grows linearly). The $x$ axis are the number of words in the dictionary in multiples of 10,000.

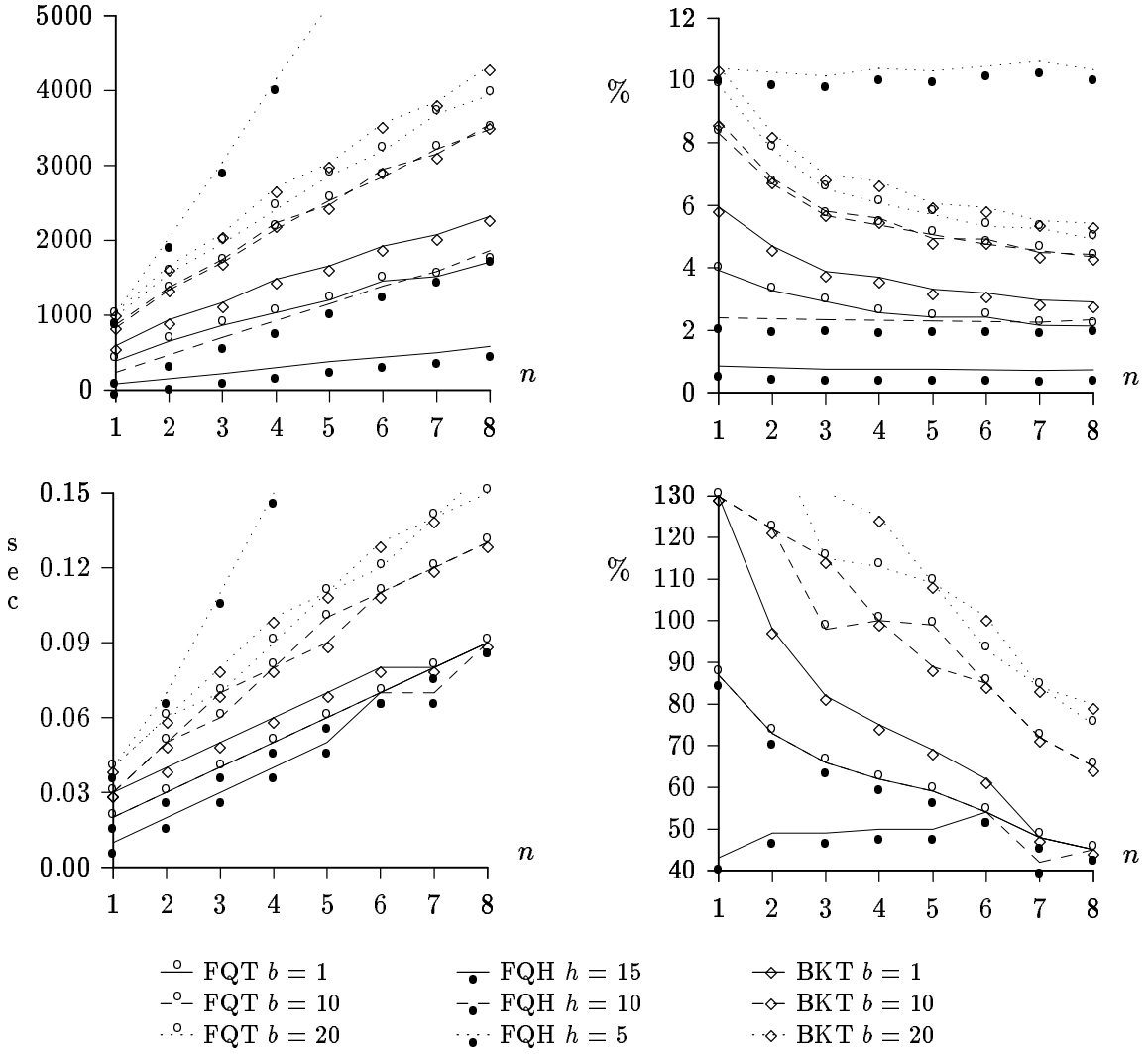


Figure 5: Comparison allowing one error. The first row shows the number of comparisons (on the left, absolute number, on the right, percentage over the whole dictionary). The second row shows user times for the queries (on the left, seconds, on the right, percentage over the best online algorithms). The $x$ axis are the number of words in the dictionary in multiples of 10,000.

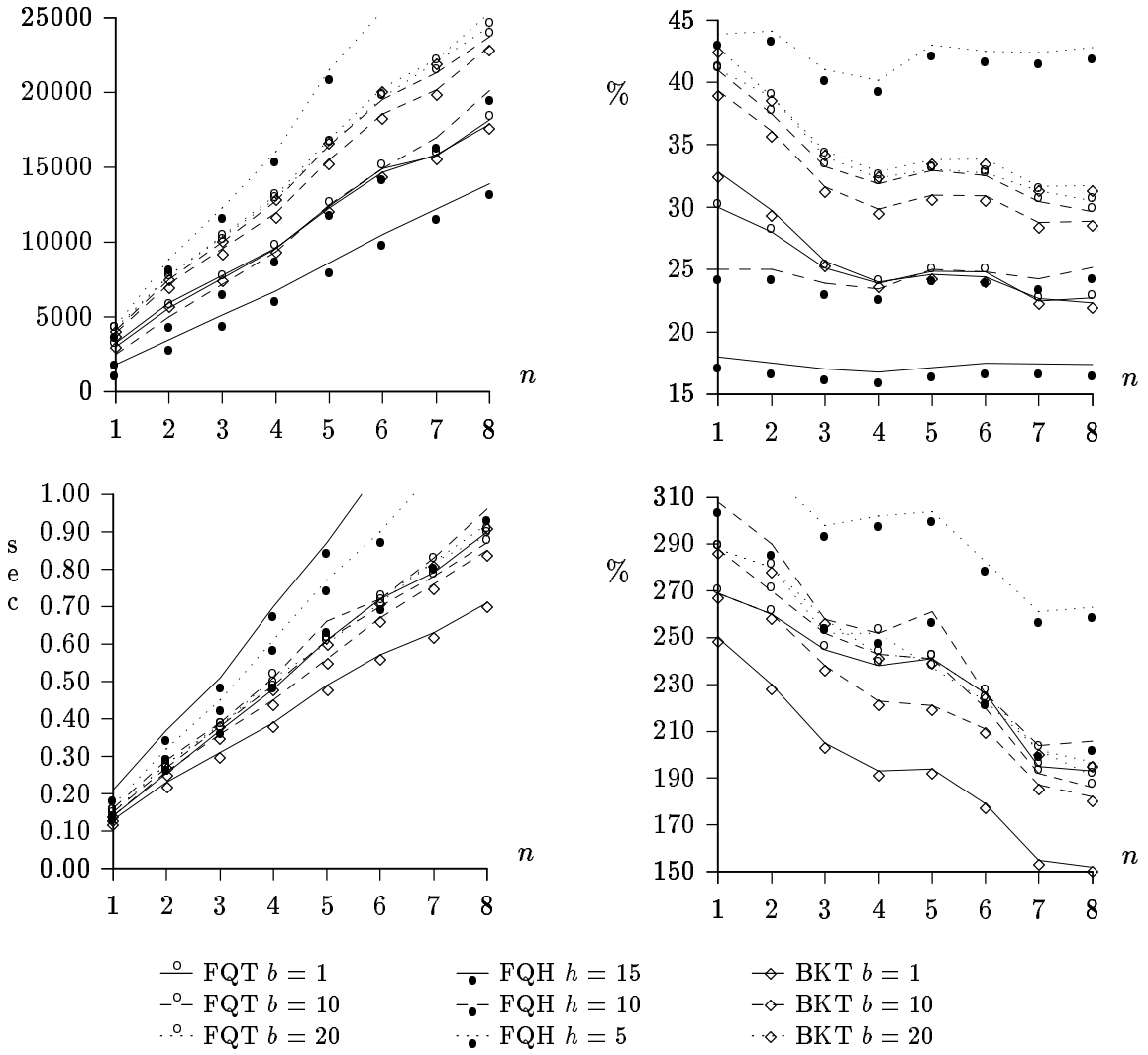


Figure 6: Comparison allowing two errors. The first row shows the number of comparisons (on the left, absolute number, on the right, percentage over the whole dictionary). The second row shows user times for the queries (on the left, seconds, on the right, percentage over the best online algorithms). The $x$ axis are the number of words in the dictionary in multiples of 10,000.