Super-linear Indices for Approximate Dictionary Searching

Leonid Boytsov

leo@boytsov.info

Abstract. We present experimental analysis of approximate search algorithms that involve indexing of deletion neighborhoods. These methods require huge indices whose sizes grow exponentially with respect to the maximum allowable number of errors k. Despite extraordinary space requirements, the super-linear indices are of great interest, because they provide some of the shortest retrieval times.

A straightforward implementation that creates a hash index directly over residual strings (obtained by deletions from dictionary words) is not space efficient. Rather than memorizing complete residual strings, we record only deleted characters and their respective positions. These data are indexed using a perfect hash function computed for a set of residual dictionary strings [2].

We carry out an experimental evaluation of this approach against several well-known benchmarks (including FastSS, which stores residual strings directly [3]). Experiments show that our implementation has a comparable or superior performance to that of the fastest benchmarks. At the same time, our implementation requires 4-8 times less space as compared to FastSS.

Keywords: wildcard neighborhood generation, reduced alphabet neighborhood generation, Mor-Fraenkel method, perfect hashing, FastSS

1 Introduction

Approximate string searching is ubiquitous in information retrieval, spellchecking, computational biology, speech recognition, and security software (e.g., for detection of weak passwords). This problem is twofold: finding the locations of a pattern inside a given text, and finding matching strings in a set, i.e., in a dictionary. In both cases, the pattern needs to match data only approximately. A degree of closeness is determined by a distance function. We restrict our attention to the case of lossless methods which guarantee retrieval of all words within the Levenshtein distance k from the search pattern [15]. This distance function is equal to the minimum number of basic edit operations (insertions, deletions, and substitutions) required to convert one string into another. Note that we are primarily interested in practical aspects of this problem and evaluate only those methods that are capable of tolerating more than one error (i.e., support $k > 1$).

We focus on the methods with super-linear indices, which rely on generation of deletion neighborhoods and/or reduced alphabet neighborhoods. In particular, deletion neighborhoods are memorized in the index. The redundancy in storage allows us to achieve very short retrieval times. It is also noteworthy that these methods involve neither direct computation of the Levenshtein distance nor explicit verification if the distance between the pattern and dictionary words is at most k.

The rest of the paper is organized as follows. Prior art is described in Subsection 1.1. In Subsection 1.2, we introduce notation and formalize the problem. The implemented methods are described in Section 2, which starts with a discussion on the concepts of full and wildcard neighborhood generation. The experiments are presented in Section 3. Section 4 concludes the paper.

1.1 Related Work

Damerau [8] presented misspelling statistics and described one of the first methods of approximate dictionary searching. This method could tolerate only a single error. Levenshtein proposed a string similarity function that is equal to the minimal number of insertions, deletions, and substitutions necessary to make strings equal. A dynamic programming algorithm to efficiently compute Levenshtein distance was independently discovered by several scientists [20]. This classic algorithm has a quadratic complexity and a number of improvements were suggested [18].

To further reduce retrieval time, it is necessary to index the data. There are a lot of indexing techniques for approximate dictionary searching, which rely, among other methods, on generating neighborhoods, indexing of contiguous and gapped string subsequences, as well as on organizing a dictionary in the form of a trie (a prefix tree). Details of the methods for approximate dictionary searching can be found in the surveys on this topic [11,14,19,5].

A common approach to approximate dictionary searching involves generation of a pattern full k-neighborhood: strings obtainable from the pattern by at most k edit operations. Then, elements of the full neighborhood are searched in the dictionary for an exact match. This method is not efficient for large k and/or large alphabets, because the size of the full neighborhood is $O\left(n^k|\Sigma|^k\right)$ (where n and $|\Sigma|$ is the size of the pattern and the alphabet, respectively) [21].

Much shorter retrieval times can be achieved through indexing of residual strings, i.e., strings obtainable by deletions from dictionary words. Along with residual strings, it is necessary to memorize deleted characters and their positions in the original dictionary words. We call these data deletion lists. A special case of this method for $k = 1$ was described by Mor and Fraenkel in 1982 [17]. A generalization of the Mor-Fraenkel method for $k > 1$ was independently proposed by Bocek et al. [3] and Boytsov [5]. Both Bocek et al. and Boytsov suggested to index residual strings and positions of deleted characters directly (via a hash index), which is not space efficient. For $k = 1$, there are methods that have better space requirements. Mihov and Schulz [16] store one-deletion neighborhoods in the form of finite transducers. In the algorithm by Belazzougui [2], all residual

words are enumerated using a minimal perfect hash function. Then, only deleted characters instead of original dictionary strings are memorized. We are not aware of any attempts to utilize compact deletion indices for $k > 1$.

An intermediate approach between the full neighborhood generation and the generation of deletion neighborhoods is a reduced alphabet neighborhood generation. In this approach, the strings over the original alphabet are mapped into strings over a smaller, i.e., reduced, alphabet. An experimental evaluation of the reduced alphabet neighborhood generation was carried out by Boytsov [5]. It is possible to combine the reduced alphabet neighborhood generation with the Mor-Fraenkel method, but we have not seen an implementation of this idea before.

All described modifications of the Mor-Fraenkel algorithm do not entail computation of the Levenshtein distance or explicit verification if the distance between the pattern and dictionary words is at most k. In the lossy version of the Mor-Fraenkel method descried by Karch et al. [12], deletion indices are used only as a filtering step. Instead of memorizing residual strings, deleted characters, and their positions, Karch et al. propose to keep only identifiers of original dictionary words. Consequently, at the verification step, a list of candidates should be compared directly against a search pattern through computing the Levenshtein distance. Karch et al. combine this approach with pattern partitioning: most dictionary words are divided into halves and each half is indexed separately (this approach was known already in the seventies [13,9]).

There were also attempts to blend partial neighborhood generation with tries. Cole et al. [7] introduced a k-errata tree, where errors are treated by recursively creating insertion, substitution, and deletion subtrees. The k-errata tree has a super-linear index whose size is upper bounded by $O\left(\lambda N + N\frac{(5\log_2 N)^k}{k!}\right)$, where N is the number of dictionary strings. Boytsov [5] conducted an experimental evaluation of this method, which showed that the k-errata tree was impractical for $k > 1$.

1.2 Notation and Problem Formalization

We consider algorithms that operate on strings, i.e., sequences of characters over an ordered finite alphabet Σ ($|\Sigma|$ is the size of the alphabet). The i-th character of the string s is denoted by $s_{[i]}$. The string obtained from s by deletion of the i-th character is denoted by $\Delta_i(s)$. A reverse operation consists in inserting character c into position i, which introduces c *before* the character $s_{[i]}$. We assume that the reader is familiar with notions of a substring as well with the concepts of a prefix, a suffix, and, a q-gram (a substring of the fixed length q).

Several implemented algorithms rely on mapping the original alphabet Σ to a smaller alphabet σ, which is called a reduced alphabet. A projection is done using a hash function $h(c)$, which induces a character-wise projection from the set of strings over the original alphabet Σ to the set of strings over the reduced alphabet σ in a straightforward way.

The similarity between functions u and v is measured via the Levenshtein distance, which is denoted by $\mathrm{ED}(u, v)$. It is equal to the minimum number of basic edits (insertions, deletions, and substitutions) required to convert u into v (and vice versa). Knowledge of algorithms to compute the Levenshtein distance is not required for understanding this paper.

Assume that $W = (s_1, s_2, \ldots, s_N)$ is an ordered set of strings, called dictionary. The search pattern and its length are denoted by p and n, respectively. The maximum allowed edit distance is represented by k. The problem of approximate dictionary searching consists in retrieval of *all* dictionary strings s_i such that $\mathrm{ED}(p, s_i) \leq k$. In the *associative* version of this problem, it is necessary to find all strings s_i within distance k from the pattern as well as data associated with strings s_i (also called satellite data). A string identifier is one well-known example of satellite data.

2 Method Descriptions

2.1 Full, Reduced, and Deletion Neighborhood

A neighborhood generation is a classic search method [10]. The full neighborhood generation entails computation of all strings within the Levenshtein distance k from the search pattern p. These strings comprise a full k-neighborhood. Each element of the full k-neighborhood is searched for in the dictionary exactly. Because the size of the neighborhood is $O\left(n^k|\Sigma|^k\right)$ [21], this algorithm is only practical when neither of the following parameters are large: the size of the alphabet, the maximum allowed Levenshtein distance, and the pattern length.

Consider an example of the string `find`. The full one-neighborhood contains the following strings:

- the original string `find`;
- 3 strings obtained by applying a single deletion;
- 5×26 strings obtained by applying a single insertion;
- 4×25 strings obtained by applying a single substitution.

In total, the one-neighborhood contains 231 unique strings. However, the two- and the three-neighborhood of the string `find` contain about 20K and 1.5M unique strings, respectively.

One approach to compress the full neighborhood is to replace some characters with wildcards. Let us extend the alphabet with a wildcard pseudo-character `?` that matches any alphabet character. Then, the full wildcard one-neighborhood of the word `find` may contain the following strings:

- the original string `find`;
- 3 strings obtained by a single deletion;
- 5 strings obtained by one insertion: `?find`, `f?ind`, `fi?nd`, `fin?d`, `find?`;
- 4 strings obtained by one substitution: `?ind`, `f?nd`, `fi?d`, `fin?`.

The wildcard one-neighborhood comprises only 13 strings as compared to 231 strings of the full one-neighborhood. In general, the size of the wildcard k-neighborhood is smaller than the size of the full k-neighborhood by a factor of $|\Sigma|^k$.

Another approach to compress the neighborhood is to decrease the size of the alphabet. This can be achieved through mapping of the original alphabet Σ to a smaller (reduced) alphabet σ via a hash function $h(c)$. Assume that $\sigma = \{0, 1\}$; $h(c)$ is equal to 0 for English letters from `a` to `m`, and is 1 for letters from `n` to `z`. Note that characters 0 and 1 can be considered as special wildcard characters that represent regular expressions `[a-m]` and `[n-z]`, respectively.

In our example, the reduced string $h($`find`$)$ is equal to `0010`. The full neighborhood of the string `0010` has 14 unique elements, which is much smaller than the full neighborhood of the original string `find`. One should now be convinced that generating a wildcard/reduced alphabet neighborhood entails significant performance improvements, if we can devise algorithms to satisfy wildcard queries efficiently. In the following subsections, we discuss such algorithms.

2.2 A Generalization of the Mor-Fraenkel Method

In the Mor-Fraenkel method, wildcard queries are answered with a help of deletion indices. Deletion indices store deletion neighborhoods generated at index time. Consider an example, where `find` is a pattern string, `mind` is a dictionary string, and $k = 1$. Strings `find` and `mind` differ by one substitution. Furthermore, the string `?ind` from the wildcard one-neighborhood of the pattern `find` matches the dictionary string `mind`.

To find all dictionary words that differ from `find` only in the first letter, it is sufficient to memorize all strings obtained by deletion of the first character in a special index. At search time, we simply remove the first character of the pattern string p and retrieve all strings from the special index that match the shortened pattern *exactly*. In what follows, we describe a generalization of this idea. Note that Bocek et al. [3] provide an alternative description of the same approach as well as its efficient implementation (FastSS).

The indexing algorithm of the generalized Mor-Fraenkel method iterates over dictionary strings and generates their k-deletion neighborhoods, i.e., all strings obtainable from dictionary strings through k deletions. Consider a residual string $s' = \Delta_{\tau_1}(\Delta_{\tau_2}(\ldots(\Delta_{\tau_l}s)\ldots) = \Delta_{\tau_l+l-1}(\ldots(\Delta_{\tau_2-1}(\Delta_{\tau_1}s))\ldots)$ obtained from a dictionary string s through deleting characters $s_{[\tau_1]}, s_{[\tau_2]}, \ldots, s_{[\tau_l]}$ in positions $\tau_1 \leq \tau_2 \leq \ldots \leq \tau_l$ $(l \leq k)$. For each residual string s', we memorize a triple (s', D^s, C^s), where $C^s = (s_{[\tau_1]}, s_{[\tau_2]}, \ldots, s_{[\tau_l]})$ stands for deleted characters and $D^s = (\tau_1, \tau_2 - 1, \ldots, \tau_l - l + 1)$ represents their positions in the original string s. These triples, called deletion lists, are kept in an index. This index allows us to search for triples by their first elements, i.e., by residual strings.

We note the following:

- Given a triple (s', D^s, C^s), the original string s can be reconstructed by inserting characters C^s_i at positions D^s_i into the string s' in the *decreasing* order of i.

– D^s is a multiset, i.e., a set that may contain repeated elements. A multiset is characterized by its indicator function. The value of the multiset indicator function $\mathbf{1}_A(e)$ is equal to the number of times the element e repeats in A. All multiset operations can be expressed in terms of indicator functions. In particular, the indicator of the intersection is equal to $\min(\mathbf{1}_A(e), \mathbf{1}_B(e))$ and $|A|$ (the cardinality of A) is equal to $\sum_{e \in A} \mathbf{1}_A(e)$.

At search time, we generate the k-deletion neighborhood of the pattern string p. Thus, we obtain pairs (p', D^p), where p' is a residual string obtained from p by deleting characters $p_{[\rho_1]}$, $p_{[\rho_2]}$, $\dots p_{[\rho_m]}$ in positions $\rho_1 < \rho_2 < \ldots < \rho_m$ and $D^p = (\rho_1, \rho_2 - 1, \ldots, \rho_m - m + 1)$ is a multiset that represents ρ_i.

Next, we retrieve all dictionary triples (s', D^s, C^s) (using the exact-search index) that satisfy the conditions:

$$p' = s'$$

$$|D^s| + |D^p| - |D^s \cap D^p| \leq k \tag{1}$$

Finally, dictionary strings are reconstructed from triples satisfying Condition (1).

2.3 A Compact Version of the Mor-Fraenkel Method

Explicit indexing of triples (s', D^s, C^s) – defined in Subsection 2.2 – requires a lot of RAM. A more space efficient version was proposed by Belazzougui for the case $k = 1$. He suggested to enumerate all residual strings s' using a minimal perfect hash function [2]. The minimal perfect hash function $f(s)$ maps m strings to integer values from 1 to m without collisions. During indexing, we convert triples (s', D^s, C^s) into triples $(f(s'), D^s, C^s)$ and index the latter using first elements (i.e., values of the perfect hash function) as keys.

The retrieval algorithm is almost identical to that described in Subsection 2.2. At search time, we compute all pairs (p', D^p), where p' is a residual string obtained from p by deleting up to k characters. Positions of deleted characters are defined by the multiset D^p. Then, we retrieve all dictionary triples $(f(s'), D^s, C^s)$ such that:

$$f(s') = f(p')$$

$$|D^s| + |D^p| - |D^s \cap D^p| \leq k. \tag{2}$$

Note that the first element of the triple is the hash value of the unknown string s'. If $s' = p'$, the triple represents a dictionary string s such that $\mathrm{ED}(p, s) \leq k$. In this case, s can be obtained by inserting C_i^s into p' at positions D_i^s (in the decreasing order of i). However, if $s' \neq p'$, the triple represents a false positive.

Two cases are to be considered. In the first case, the residual pattern string belongs to the set of residual dictionary strings computed during indexing. By the definition of the perfect hash function, $f(s') = f(p')$ implies $s' = p'$. Thus, the retrieved triple represents the dictionary string s such that $\mathrm{ED}(p, s) \leq k$.

The string s can be recovered from the residual pattern string p', the multiset D^s, and the vector C^s as described previously.

In the second case, p' is not a residual dictionary string. Thus, $p' \neq s'$. This case signifies a false positive. As noted by Belazzougui, it can be detected by constructing the string s'' from $\{p', D^s, C^s\}$ as described previously and checking whether the constructed string belongs to the dictionary.

If s'' is a dictionary string, then all the triples satisfying Condition (2) define dictionary strings s such that $\mathrm{ED}(p, s) \leq k$. If the constructed string does not belong to the dictionary, none of the triples satisfying Condition (2) represent a dictionary string s such that $\mathrm{ED}(p, s) \leq k$. Thus, checking whether a reconstructed string s'' belongs to the dictionary has to be done only once for every residual pattern string p'.

To obtain a compact index of deletion lists, Belazzougui recommends to store triples in the increasing order of hash values $f(s')$. For each hash value i, the offset of the first triple $(f(s'), D^s, C^s)$ such that $f(s') = i$ is stored in the offset table $T(i)$. Because the offsets in $T(i)$ is a sequence of non-decreasing integer values, one can efficiently compress $T(i)$. An experimental survey of methods for compact representation of directly addressable ordered sets is given by Brisaboa et al. [6]. In our work, we rely on a simple folklore sampling method, which allows us to compress the offset table $T(i)$ to about 30-50% of its original size.

To conclude this subsection, we note that in our implementation the perfect hash functions are computed using the CMPH library [4].[1] For our data, CMPH fails to generate a perfect hash function when the number of residual strings is large (approximately 100M). To overcome this difficulty, we employ a two-level scheme, where residual strings are divided into shards using a regular hash function. Then, we create a perfect hash function separately for each shard. It is noteworthy, that this approach allows one to construct a perfect hash function for arbitrarily large sets of strings. In addition, dividing the index into shards simplifies updates.

2.4 Reduced Alphabet Neighborhood Generation

The indexing algorithm of the reduced alphabet neighborhood generation employs a hash function $h(c)$ to convert original dictionary strings s_i into their projections $h(s_i)$, which are strings in the reduced alphabet. Then, dictionary strings are organized into buckets based on the values of $h(s_i)$ so that each bucket contains strings with same values of $h(s_i)$. This allows us to efficiently retrieve original strings s_i using their projections $h(s_i)$ as search patterns.

At search time, the pattern p is converted into $r = h(p)$. Then, we create a full k-neighborhood of the reduced pattern r (using characters from the reduced alphabet σ). All dictionary strings s such that $\mathrm{ED}(p, s) \leq k$ are contained in buckets corresponding to strings from the generated neighborhood. This step provides a list of candidate strings. Because the size of the reduced alphabet is

[1] It can be downloaded from `http://cmph.sourceforge.net/`

(much) smaller than that of the original alphabet, computation of the reduced alphabet requires little time.

In the second step, the candidate strings are compared with the original pattern. A naive implementation of the verification step involves computation of the Levenshtein distance. A more efficient approach is to generate an additional wildcard neighborhood, i.e., the wildcard neighborhood of the original pattern.

The generation of two neighborhoods is synchronized in the following manner:

- If we substitute the i-th character of the reduced pattern r, we replace the i-th character of the original pattern p with the wildcard symbol `?`;
- Similarly, if we insert a character at position i of r, we insert the wildcard `?` at position i of p;
- If we delete the i-th character of r, we also delete the i-th character of the original pattern p.

Note that this procedure generates pairs of patterns that have same lengths.

Consider the binary reduced alphabet and the hash function $h(c)$ defined in Subsection 2.1. Assume that the pattern $p =$ `ind` is a misspelled version of the dictionary string `find`. Then, $r = h($`ind`$) =$ `010` and $h($`find`$) =$ `0010`. The reduced-alphabet one-neighborhood of r contains the string `0010`, which is obtained by inserting `0` into the first position of the reduced pattern. The respective element from the "parallel" neighborhood is equal to `?ind`. We use `0010` to identify a bucket that contains the string `find`. Then, the element `?ind` of the second neighborhood is used to compare `ind` with `find`. For this purpose, we treat `?ind` as a simple regular expression where `?` matches any alphabet character. Whenever a dictionary string in the bucket matches p within $k = 1$ errors, it should match such regular expression exactly. This match can be verified efficiently (in time proportional to the length of s) without computing the Levenshtein distance.

2.5 A Hybrid of the Mor-Fraenkel Method and the Reduced Alphabet Neighborhood Generation

The Mor-Fraenkel method can be blended with the reduced alphabet neighborhood generation. The indexing process of this hybrid algorithm starts with creating a reduced alphabet index outlined in Subsection 2.4: The dictionary strings s_i are divided into buckets based on their projections $h(s_i)$ (to the set of reduced alphabet strings). Projections $h(s_i)$ are stored in the form of a dictionary. In the second indexing step, this dictionary is indexed using a compact version of the Mor-Fraenkel method (see Subsection 2.3). One advantage of this approach is that triples $(f(s'), D^s, C^s)$ can be readily compressed. For example, in our experiments we use $|\sigma| = 8$. Thus, each element of C^s can be encoded with 3 bits.

The search algorithm is divided into two steps and involves the parallel generation of two wildcard neighborhoods. This first step of the search algorithm is a modification of the Mor-Fraenkel search method. As described in Subsection

2.2, we create residual patterns $\{p'\}$ by applying up to k deletions. Positions of deleted characters associated with $\{p'\}$ are defined by multisets $\{D^p\}$. In addition, we create residual patterns $\{r'\}$ by deleting characters from the reduced pattern $r = h(p)$ in the same positions as in $\{p\}$.

For each r', we retrieve memorized triples $(f(s'), D^s, C^s)$ such that $f(r') = f(s')$ and $|D^s| + |D^{r'}| - |D^s \cap D^{r'}| \leq k$ (s' is a string in the reduced alphabet). Then, we construct the string s'' by inserting characters C^s into the residual string r' in positions D^s_i in the decreasing order of i. The result is a string r''. We also modify the residual string p', obtained from the non-reduced pattern p. To this end, wildcard character `?` is inserted into positions D^s_i in the decreasing order of i. The result is the pattern p'' that contains zero or more wildcards. The string r'' defines a bucket with candidate strings, which are exhaustively compared with the simple regular expression defined by p''.

2.6 Associativity Consideration

It can be seen that methods defined in Subsections 2.4-2.5 can handle associated data by simply storing it in the buckets (or pointers thereto). Because both methods involve full scanning of the buckets with candidate strings, retrieval of associated data does not have a performance penalty.

However, the compact version of the Mor-Fraenkel method that uses perfect hashing is capable of retrieving only strings themselves. Retrieval of associated data can be supported in two ways. In a more space efficient approach, associated data (or pointers thereto) are stored in the dictionary. Then, for every string generated during the verification step, we have to search the dictionary for an exact match, even though we already know that the generated string must belong to the dictionary. According to our experiments, these additional lookups almost double retrieval time. In a second approach, compressed string identifiers are kept in the deletion lists. This requires an approximately two times larger index, but retrieval time will remain the same.

3 Experiments

3.1 Experimental Setup

Experiments are carried out in a single-thread mode on a laptop with a 2 Ghz Intel CoreDuo Processor, 3 Gb of RAM, and 1 Mb of L2 cache. This laptop is running a 32-bit Linux (kernel version 2.6.x). Time is reported in milliseconds.

We use some of the data sets published by Boytsov [5]: synthetic English dictionaries, frequent words from the ClueWeb09 collection, and DNA sequences extracted from the human genome (of length 11). Each type of the data set has 5 dictionaries with 0.2M, 0.4M, 0.8M, 1.6M, and 3.2M strings. We created new sets of search patterns containing up to 10K, by applying up to k random edits to dictionary strings. These larger test sets allow us to reduce sampling uncertainty in calculating average retrieval times.

All search methods are implemented in C/C++ with correctness of implementation verified by black-box testing. We index a small dictionary and generate a set of strings by applying $i \leq k$ random edits to dictionary words. Then, we check if the algorithm is capable of finding original dictionary words using modified strings as search patterns.

In our experiments, we determine how the following performance characteristics depend on k: the average in-memory retrieval time and the index size. To estimate index memory requirements, we measure the amount of space occupied by a serialized version of the index. For FastSS this method produces a biased estimate: we correct it through multiplying by 2, which is a coefficient empirically determined using the Unix utility `top`.

3.2 Evaluated Methods

We compared the performance of super-linear indices with several methods, in particular with the fastest methods evaluated by Boytsov [5]. We benchmarked the following algorithms:[2]

- FastSS [3], which is straightforward generalization of the Mor-Fraenkel method (see Subsection 2.2). Deletion lists are not compressed.
- A new implementation of the Mor-Fraenkel method with compact indices (see Subsection 2.3). It employs the perfect hash library CMPH [4].[3] We compress deletion lists as follows: for natural language data, each deleted character as well as its position is encoded using 6 bits. For DNA-data, a deleted character together with its position occupy 8 bit.
- The reduced alphabet neighborhood generation implemented by Boytsov [5] (see Subsection 2.4). In the case of ClueWeb09 data, we use $|\Sigma| = 5$ and $|\Sigma| = 3$, otherwise. This method works well only for large and medium alphabets and is not used for the DNA data set.
- The full neighborhood generation (see Subsection 2.1), which is used only for the DNA data set.
- The hybrid of the reduced alphabet neighborhood generation and the Mor-Fraenkel method with $|\Sigma| = 8$ (see Subsection 2.5). Characters and their positions in deletion lists are encoded using 3 and 5 bits, respectively. For a few patterns longer than $31 - k$ character, we resort to the reduced alphabet neighborhood generation.
- The FB-trie proposed by Mihov and Schulz [16] (Boytsov's implementation [5]). It employs a pair of tries: a regular one and a trie built over reversed strings. At search time, the method looks for the original pattern in the regular trie, and for the reversed pattern in the trie built over reversed strings. A first part of either the original or the reversed pattern should match a trie prefix with at most $\lfloor k/2 \rfloor < k$ errors. This is a well-known pattern partitioning approach [13,9].

[2] The source files are available at: `http://boytsov.info/src/`.

[3] `http://cmph.sourceforge.net/`

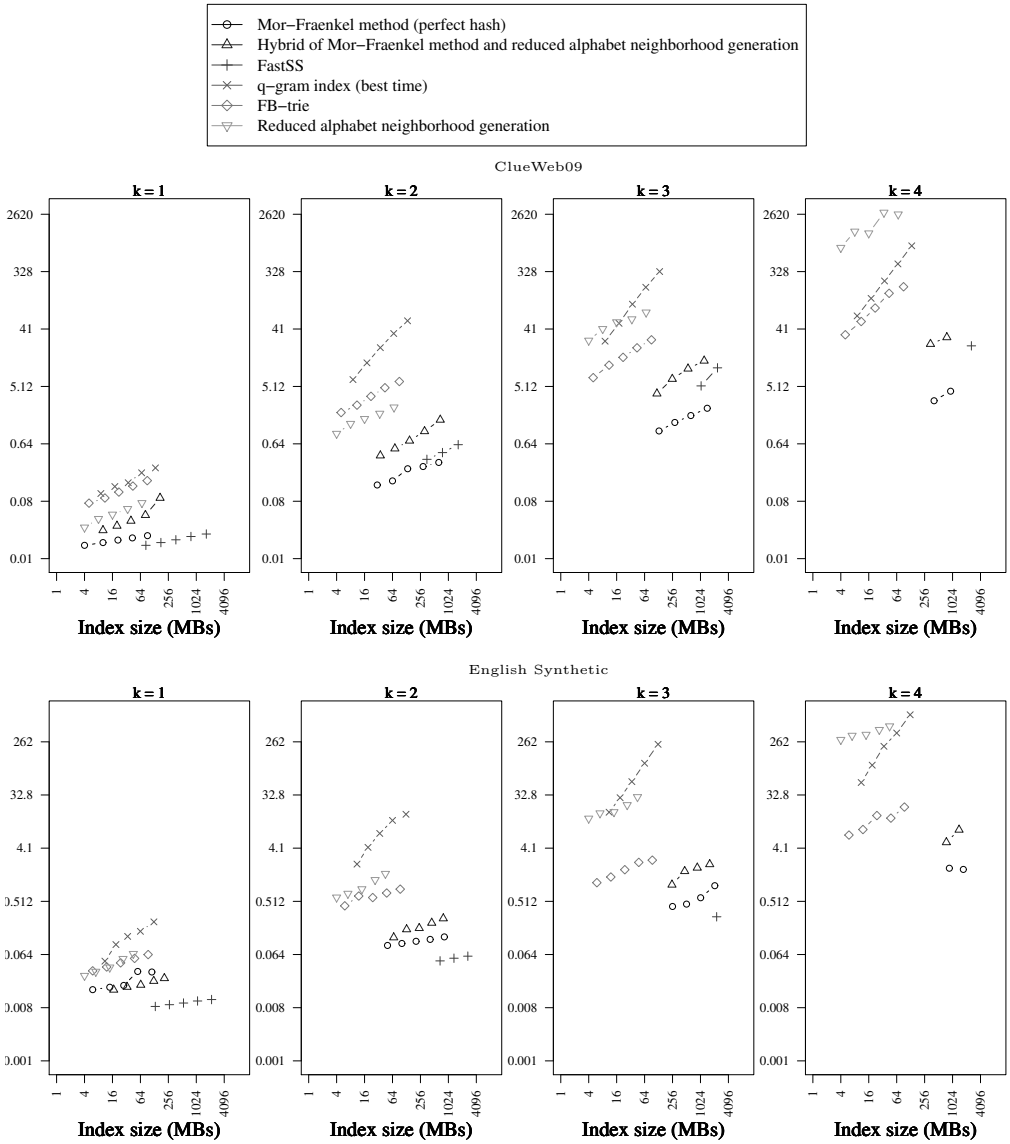

Fig. 1. Relationship between the average retrieval time (in ms) and the index size (log-scale on both axes). Each series of connected dots represents results for dictionaries of at most five sizes: 0.2M, 0.4M, 0.8M, 1.6M, and 3.2M (from left to right).

– Two modifications of q-gram methods. One is implemented by Boytsov [5] and another is provided with the Flamingo package written by Behm et al. [1]. For brevity, we report only the best time achieved by one of the q-gram methods.

3.3 Experimental Results

Figures 1 and 2 shows the relationship between the average retrieval time (in milliseconds) and the index size. Each series of connected dots represents results for a set of dictionaries of increasing size. In most cases, the dots from left to right correspond to the dictionaries of five sizes: 0.2M, 0.4M, 0.8M, 1.6M, and 3.2M. Some connected series contain fewer dots, because larger indices do not fit into RAM. Note that in the case of associative searching, either the average retrieval time or the index size of the Mor-Fraenkel method based on perfect hashing would be twice of that presented in Figures 1-2. The other methods can support associative searching without a penalty in performance or memory requirements (see Subsection 2.6).

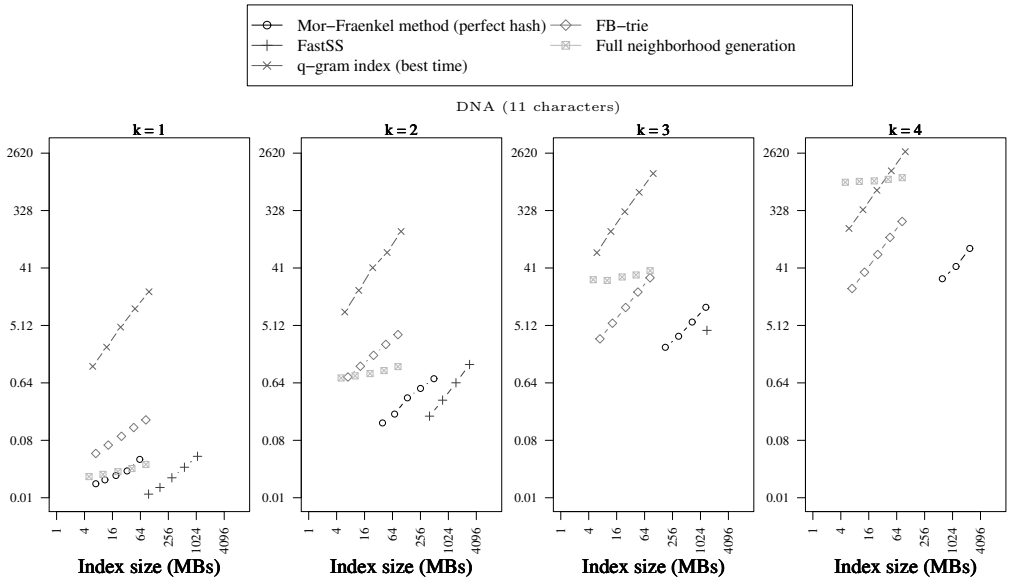

Fig. 2. Relationship between the average retrieval time and the index size (log-scale on both axes). Each series of connected dots represents results for dictionaries of at most five sizes: 0.2M, 0.4M, 0.8M, 1.6M, and 3.2M (from left to right).

One can immediately see that all three modifications of the Mor-Fraenkel method (which includes FastSS) are more efficient than other methods in almost all cases. In particular, they are:

– about two orders of magnitude faster than q-gram based methods, which are often considered as good benchmarks;
– up to an order of magnitude faster than the FB-trie.

This efficiency comes at the price of huge indices and long indexing times (up to one hour for the variant based on perfect hashing). Consider the panel in Figure 1 corresponding to the case of $k = 1$. The second dot in the FastSS series represents the index for the second largest dictionary (0.4M strings). It has the size 100MB, which is larger than a q-gram index built for the dictionary with 3.2M strings. However, the index size of the Mor-Fraenkel method based on perfect hashing is only 8MB, or about 1/10 of the FastSS index. Taking into account that about 25% reduction is achieved through lightweight compression of deletion lists (both characters and positions occupy 6 bits each), we obtain that the use of perfect hashing alone lead to about 8-fold reduction in index sizes as compared to straightforward memorization of deletion neighborhoods. For larger k the difference is approximately 4-fold.

One can also see that the hybrid of the Mor-Fraenkel method and the reduced alphabet neighborhood generation is not a very practical method. Even though the hybrid method significantly improves over the reduced alphabet neighborhood generation (especially for larger k), it is up to an order of magnitude slower than the Mor-Fraenkel method based on perfect hashing (see $k = 4$, ClueWeb09 data). In that, the hybrid method has equivalent space requirements to those of the Mor-Fraenkel method.

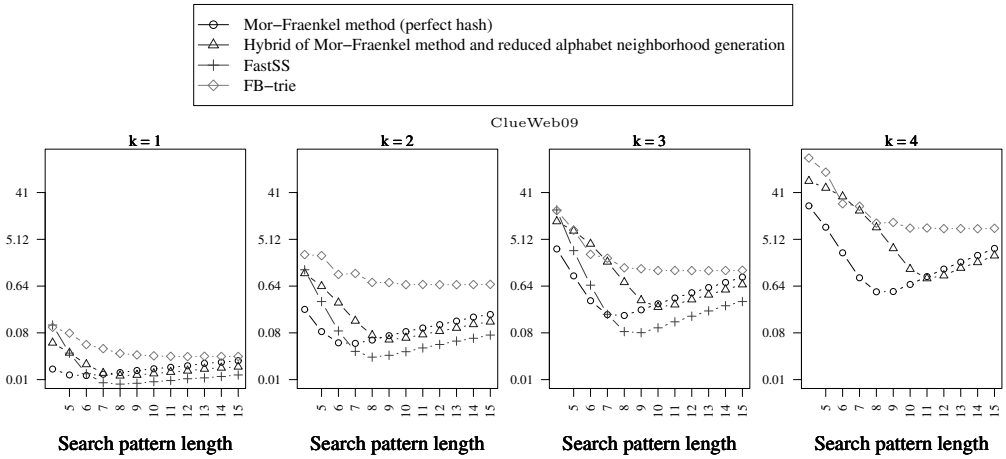

Fig. 3. Relationship between the average retrieval time (in ms) and pattern length (log-scale on time axis)

Consider the case of DNA data. For small dictionaries, the variants of Mor-Fraenkel methods are among the fastest algorithms. As the number of dictionary strings grows, performance of these methods deteriorates. For $k \leq 2$, it becomes

equivalent to that of full neighborhood generation. We believe that this fact can be explained by a density effect (see Section C.3.3 in the paper by Boytsov [5]). In our case, the number of unique 11-character DNA sequences is about 4M. The largest dictionary with 3.2M entries contains most of them and, thus, is very dense. Consequently, the algorithm has a low filtering efficiency. Note that for $k \geq 3$ and DNA data, the Mor-Fraenkel method outperforms the full neighborhood generation, but it has the equivalent performance to that of the FB-trie.

We conducted an additional experiment to study the relationship between the average retrieval time and the pattern length. To this end, we use the smallest ClueWeb09 dictionary (0.2M strings) and patterns with length from 4 to 15. According to Figure 3, the average retrieval time of all Mor-Fraenkel methods first decreases until $n \approx 8$. Afterwards, it increases monotonically. The FB-trie utilizes pattern partitioning, which is essentially filtering by word halves. The longer is the pattern, the better is filtering efficiency. Consequently, the average retrieval time of the FB-trie decreases monotonically with n. For long patterns, $k = 1$, and $k = 3$, performance of Mor-Fraenkel methods is equivalent to that of the FB-trie. Thus, Mor-Fraenkel methods would be most useful for short and medium-size patterns.

4 Conclusions

Mor-Fraenkel methods have tremendous space requirements: The index size grows exponentially with k. This also applies to FastSS, which belongs to the family of Mor-Fraenkel methods. We have empirically confirmed that space requirements can be 4-8 times lower if perfect hashing is employed (the idea proposed by Belazzougui [2]). Given that typical servers are now equipped with 8-32 Gb of memory, this method is applicable to natural language dictionaries containing several million entries. At the same time, the efficiency of the Mor-Fraenkel method based on perfect hashing is similar to that of the straightforward implementation of the Mor-Fraenkel method, which indexes deletion neighborhoods directly. Both the straightforward and perfect-hash implementations outperform our fastest benchmarks in most cases.

Mor-Fraenkel methods work best for small and medium patterns (at most 10 characters). For longer search strings one should employ a pattern partitioning strategy similar to the one used by Karch et al. [12]. However, it remains to be determined which pattern partitioning strategy would be most efficient. Another open question is whether (and to what extent) one can improve performance of a trie-based method through precomputing wildcard neighborhoods at index time. One such algorithm was proposed by Cole et al. [7], but we are unaware of any space efficient implementation of this (or similar) method.

Acknowledgments. I am very grateful to my wife Anna for editorial assistance.

References

1. Behm, A., Vernica, R., Alsubaiee, S., Ji, S., Lu, J., Jin, L., Lu, Y., Li, C.: UCI Flamingo Package 4.0 (2010)
2. Belazzougui, D.: Faster and space-optimal edit distance 1 dictionary. In Kucherov, G., Ukkonen, E., eds.: Combinatorial Pattern Matching. Volume 5577 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2009) 154–167
3. Bocek, T., Hunt, E., Stiller, B.: Fast similarity search in large dictionaries (2007) Technical report No. ifi-2007.02, Department of Informatics (IFI), University of Zurich.
4. Botelho, F.C.: Near-Optimal Space Perfect Hashing Algorithms. PhD thesis, Graduate Program in Computer Science, Federal University of Minas Gerais, Brazil (2008)
5. Boytsov, L.: Indexing methods for approximate dictionary searching: Comparative analysis. J. Exp. Algorithmics **16** (May 2011) 1.1:1.1–1.1:1.91
6. Brisaboa, N., Ladra, S., Navarro, G.: Directly addressable variable-length codes. In: Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE). LNCS 5721, Springer (2009) 122–130
7. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: STOC '04: Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing, ACM (2004) 91–100
8. Damerau, F.: A technique for computer detection and correction of spelling errors. Communications of the ACM **7**(3) (1964) 171–176
9. Doster, W.: Contextual postprocessing system for cooperation with a multiple-choice character-recognition system. IEEE Trans. Comput. **26** (November 1977) 1090–1101
10. Gorin, R.E.: SPELL: Spelling check and correction program. (1971) Online documentation: Describes operation of PDP-10 SPELL program. `http://pdp-10.trailing-edge.com/decuslib10-03/01/43,50270/spell.doc.html`. Accessed 28 May 2012.
11. Hall, P., Dowling, G.: Approximate string matching. ACM Computing Surveys **12**(4) (1980) 381–402
12. Karch, D., Luxen, D., Sanders, P.: Improved fast similarity search in dictionaries. CoRR **abs/1008.1191** (2010)
13. Knuth, D.: The Art of Computer Programming. Sorting and Searching. 1st edn. Volume 3. Addison-Wesley (1973)
14. Kukich, K.: Technique for automatically correcting words in text. ACM Computing Surveys **24**(2) (1992) 377–439
15. Levenshtein, V.: Binary codes capable of correcting deletions, insertions, and reversals. Doklady Akademii Nauk SSSR, **163**(4) (1965) 845–848
16. Mihov, S., Schulz, K.U.: Fast approximate string search in large dictionaries. Computational Linguistics, **30**(4) (2004) 451–477
17. Mor, M., Fraenkel, A.S.: A hash code method for detecting and correcting spelling errors. Communications of the ACM **25**(12) (1982) 935–938
18. Navarro, G.: A guided tour to approximate string matching. ACM Computing Surveys **33**(1) (2001) 31–88
19. Owolabi, O.: Dictionary organizations for efficient similarity retrieval. Journal of Systems and Software **34**(2) (1996) 127–132
20. Sankoff, D.: The early introduction of dynamic programming into computational biology. Bioinformatics **16**(1) (2000) 41–47

21. Ukkonen, E.: Approximate string matching over suffix trees. In: Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching, number 684 in Lecture Notes in Computer Science, Springer-Verlag (1993) 228–242