Fast String Dictionary Lookup with One Error

Timothy Chan[1⋆] and Moshe Lewenstein[2⋆⋆]

[1] University of Waterloo
[2] Bar-Ilan University

Abstract. A set of strings, called a *string dictionary*, is a basic string data structure. The most primitive query, where one seeks the existence of a pattern in the dictionary, is called a *lookup query*. Approximate lookup queries, i.e., to lookup the existence of a pattern with a bounded number of errors, is a fundamental string problem. Several data structures have been proposed to do so efficiently. Almost all solutions consider a single error, as will this result. Lately, Belazzougui and Venturini (CPM 2013) raised the question whether one can construct efficient indexes that support lookup queries with one error in optimal query time, that is, $O(|p|/\omega + occ)$, where p is the query, ω the machine word-size, and occ the number of occurrences.

Specifically, for the problem of one mismatch and constant alphabet size, we obtain optimal query time. For a dictionary of d strings our proposed index uses $O(\omega d \log^{1+\epsilon} d)$ additional bit space (beyond the space required to access the dictionary data, which can be maintained in compressed form). Our results are parameterized for a space-time tradeoff.

We propose more results for the case of lookup queries with one insertion/deletion on dictionaries over a constant sized alphabet. These results are especially effective for large patterns.

1 Introduction

Data mining, information retrieval, web search and database tasks are often variants of string processing. Many of these tasks involve storing a set of strings, also known as a *string dictionary*. These dictionaries may be very large, for example such is the case for search engines, applications in bioinformatics, RDF graphs, and meteorological data. Hence, it is desired to maintain the dictionaries in some succinct format while still allowing quick access to the data at hand. One basic primitive operation necessary on a dictionary is a lookup query. A *lookup* query on a string dictionary is a string for which the answer is yes if it exists in the dictionary, or more generally returns a pointer to the satellite data of that string in the dictionary. Maintaining the dictionaries in a compressed form while allowing lookups has garnered much interest in the last decade.

⋆ Cheriton School of Computer Science, University of Waterloo (tmchan@uwaterloo.ca). The research is supported by an NSERC grant.

⋆⋆ Department of Computer Science, Bar-Ilan University (moshe@cs.biu.ac.il). This research is supported by a BSF grant 2010437 and a GIF grant 1147/2011.

While exact lookups are interesting, often one desires *approximate lookups*. For example, if one queries a search engine and there is one or two typing errors in the query, it is advantageous to find the correct answer nevertheless. In this case, one actually needs to propose all answers that are within the criteria of the number of errors. Clearly, in many applications one desires to find approximate matches and not only matches.

Errors come in different forms. Three of the most common errors are substitutions, insertions, and deletions of characters. Two widely considered distances between strings are based on these errors. The former is *Hamming distance*, which is the minimal number of substitutions necessary to transform one string into another. The latter is *edit distance* [18], which is the minimal number of any combination of the three operations needed to transform one string into another.

Approximate lookups for one error have received a lot of attention, e.g. [3, 4, 7, 8, 24] and along this similar line also text indexing with one error [1], dictionary matching with one error [1, 14, 16] and both with one wildcard [2, 5, 20]. Extensions to k errors, even to 2 errors, is much more difficult. See [10, 9, 12, 17, 21, 23] for results of this form.

In numerous data structure papers over the last decade there has been a separation between the *encoding* model and the *indexing model*. In the encoding model we preprocess our input I (in our case the dictionary of strings) to create a data structure *enc* and queries have to be answered using *enc* only, *without* access to I. In the indexing model, we create an index *idx* and are able to refer to I when answering queries. In the indexing model we measure the *additional space* required. This model difference was already noted in [13]. For more discussion on the modeling differences see [6].

Interestingly, Belazzougui and Venturini [4] proposed a data structure for lookups with edit distance at most one that answers queries in $O(|p| + occ)$ time, where p is the query string and occ is the number of answers. The space required is $2nH_k + o(n) + 2d \log d$, where d is the number of strings in the dictionary, n is the total length of the dictionary and H_k is the k-th order entropy of the concatenated strings of the dictionary. While the model is set as an encoding model result, it actually is an indexing result with $nH_k + o(n)$ dedicated to the compressed dictionary and $nH_k + o(n) + 2d \log d$ additional bits necessary for the data structure.

They raised the question in [4] whether one can answer queries in optimal $O(|p|/\omega + occ)$ time while maintaining succinct space. We answer this question affirmatively. For the case of Hamming distance we propose a data structure that requires $O(\omega d \log^{1+\epsilon} d)$ additional bit space (beyond the dictionary which can be maintained in compressed form). For the case of edit distance we can obtain $\delta|p|d$, for arbitrarily small constant $\delta > 0$, additional bit space ($|p|d$ is the size of the open dictionary) and $O((|p|/\log|p|)\log^{\epsilon} d + occ)$ query time. This is an improvement over the times in [4] for $|p| \gg \log d$. However, we do note that the alphabet size in [4] is general, whereas the alphabet size here is constant. Our solution can be generalized to an alphabet of size σ at a cost of a σ factor in time and space.

2 Previous work

Yao and Yao [24] were the first to consider string dictionaries that support lookup queries with one mismatch. The dictionary they suggested had, wlog, all strings of equal size m. The alphabet was binary. They suggested an algorithm in the bit probe model that uses $O(md \log m)$ bits and answers queries in $O(m \log \log d)$ bit probes. Brodal and Gasieniec [7] considered the standard unit-cost RAM model in the same setting, i.e., one mismatch, all strings of length m and a binary alphabet. They proposed a different solution using a trie for the lexicographically ordered strings of $\mathcal{D}$ and a trie for the lexicographically ordered reversed strings of $\mathcal{D}$. The space they used was $O(md)$ words. The query time was $O(m)$. Later, Brodal and Venkatesh [8] considered a perfect-hash solution in the cell-probe model with word-size ω **and** string size $m = \omega$. They proposed a data structure that uses space $O(d \log \omega)$. We elaborate on and generalize their solution in Section 5.

Belazzougui [3] proposed the first $O(|p| + occ)$ time algorithm. The space of the solution is $O(n)$ bits, where n is the total dictionary size. The solution used Karp–Rabin fingerprinting and, hence, runs with high probability. As formerly mentioned, in [4] a result was obtained for dictionary lookups with edit distance one that answers queries in $O(|p| + occ)$ time, and with $2nH_k + o(n) + 2d \log d$, where n is the total length of the dictionary and H_k is the k-th order entropy of the concatenated strings of the dictionary.

3 Outline of Our Results

Our goal is to solve the dictionary matching with one error where the query time is optimal $O(|p|/\omega + occ)$ and the space is succinct. Our method is based on succinct bidirectional indexing structures and range searching data structures, see [19]. The method of this search has been used numerous times before, and was first used to solve a one-error problem in [1]. However, the unique feature in this paper is a succinct code for each string which allows optimal query time while maintaining very efficient space. The encoding of the string is a novel folding of the string which turns out to do the trick. The idea is to take a string s, partition it into equal length substrings, say of length b. Then we do a bitwise exclusive-or among the substrings. This folding of strings reduces the space of the string down to a small size and allows to obtain succinct representations of the strings. The encoding, assisted by the range searching techniques, remains powerful enough to deduce the answers required.

4 Preliminaries

Given a string S, $|S|$ is the length of S. An integer i is a *location* or a *position* in S if $i = 1, \ldots, |S|$. The substring $S[i \ldots j]$ of S, for any two positions $i \leq j$, is the substring of S that begins at index i and ends at index j.

A set of strings is called a *dictionary* and is denoted with $\mathcal{D} = \{S_1, \ldots, S_d\}$. That is the number of strings is d and we denote the total size $n = \sum_{i=1}^{d} |S_i|$. We may safely assume that all strings in the dictionary are of the same size. If this is not the case then $\mathcal{D}$ can be partitioned into $\mathcal{D}_l = \{s \in \mathcal{D} \mid |s| = l\}$. For one substitution one accesses $\mathcal{D}_{|p|}$ and for insertion/deletion one accesses $\mathcal{D}_{|p|+1}$ and $\mathcal{D}_{|p|-1}$.

Let $\mathcal{D} = \{S_1, S_2, \ldots, S_d\}$ be a dictionary of strings. The problem of *String Dictionary with Distance One* is the problem of indexing $\mathcal{D}$ to support *lookup queries at distance* 1, that is, for a lookup query p find all strings in the dictionary within distance 1 of p. The desired distance will be either Hamming distance or edit distance, depending on the problem at hand. We will consider both. The desire will be to maintain the dictionary in some compressed form and to answer the lookup queries of distance 1 quickly.

5 The Brodal–Venkatesh Algorithm

The Brodal and Venkatesh [8] algorithm is defined on a dictionary in which all strings are binary and have length exactly ω. However, this can be generalized. We describe this now. We still assume, wlog, that all strings are of the same length.

The proposed scheme is a straightforward solution for the problem based on hashing. Let $Ham(s, x)$ denote the Hamming distance between two equal length strings s and x. Let $H(s) = \{x \in \{0,1\}^{|s|} \mid Ham(s, x) = 1\}$ and let $H = \cup_{s \in \mathcal{D}} H(s)$, i.e., all strings at Hamming distance 1 from a string $s \in \mathcal{D}$. Generate a perfect hash function for $H \cup D$. Queries p, also of the same length as the strings of the dictionary, are read. Applying the hash function on p yields the answers. Recall that the strings are over a binary alphabet. So, by reading ω bits at a time, that is, treating each ω bits as one ω-bit character, the hashing can be implemented on strings with query time of $O(|p|/\omega + occ)$. The space required is the size of the hash table, which is $O(d|p| \log d)$ bit-space[3].

6 Algorithm for Dictionary Lookup with One Mismatch

We are interested in solving the one mismatch case with the same $O(|p|/\omega + occ)$ time. We still consider a binary alphabet, but point out that a general alphabet of size σ is reducible to the binary alphabet with σ blowup. Note that the size of the dictionary is $O(|p|d)$ bits. Hence, the Brodal and Venkatesh [8] algorithm is unsatisfactory as it uses $O(d|p| \log d)$ bits. The dictionary is not even included in this space, but it is not really necessary for their result.

[3] The space attributed to this algorithm in [4] is $O(d|p|^2 \log d)$ bits. However, this is probably because it was assumed that the generated strings, which are of size $O(d|p|^2 \log d)$ bits, need to be maintained. However, this is not the case. It is sufficient to maintain the hash function and not the fully generated strings.

We desire to obtain a result where the additional bit space is strictly sublinear in the size of the dictionary. Our method will use range queries on strings. However, the encoding of the strings to maintain a small data structure is the essence of our algorithm. We now describe the details of the solution.

Define for string $s \in \mathcal{D}$ a point $(x(s), y(s))$ on a 2D $d \times d$ geometric grid. Let $x(s)$ = the rank of s in the lexicographical sort of $\mathcal{D}$ and $y(s)$ = the rank of s^R, s reversed, in the lexicographical sort of $\mathcal{D}$ after reversing all strings.

String encodings: Fix a parameter b (think of b as polylogarithmic and assume, wlog, that b divides s). Divide each of the strings s into b-bit words $s_1, \ldots, s_{|s|/b}$ and let $c(s) = \oplus_{i=1}^{|s|/b} s_i$, where $\oplus$ denotes the bitwise exclusive-or of the s_i's. Note that $c(s)$ itself is a b-bit word. Let $C(s)$ be all b-bit words that have Hamming distance 1 from $c(s)$. Note that $|C(s)| = b$. We think of s as a point $(x(s), y(s))$ in $2D$, assigned multiple colors, one from each member of $C(s)$. See [22] for a string encoding along the same lines.

We are now ready to construct the data structure.

The data structure: We build an orthogonal range reporting structure for each non-empty color class. There are 2^b color classes, but each point is in b color classes. Specifically, consider string $s \in \mathcal{D}$ and $c(s)$ that is associated with it. There are exactly b strings with one bit of $c(s)$ flipped, which is the set $C(s)$. Now, visualize a 3D grid of $d \times d \times 2^b$ where for every string $s \in \mathcal{D}$ we generate grid points $(x(s), y(s), c)$, where $c \in C(s)$. However, we maintain separate orthogonal range reporting structure for each possible c.

Overall there are db points, hence the number of non-empty color classes is bounded by db, but will likely be a lot less. We will use a perfect hash function on these non-empty color classes $\subseteq [2^b]$ so that we can access the, at most, db orthogonal range reporting structures that exist in constant time.

The data structure supports dictionary lookup queries with one error as follows.

Query: Given a pattern p, divide it into b-bit words $p_1, \ldots, p_{|p|/b}$ and let $c(p) = \oplus_{i=1}^{|p|/b} p_i$. For each i, we want to search for all s in $\mathcal{D}$ such that:

1. s has prefix $p_1 \ldots p_{i-1}$ and
2. s has suffix $p_{i+1} \ldots p_{|p|/b}$ and
3. s_i and p_i have Hamming distance 1.

Property (1) is equivalent to having $x(s)$ lie in the interval of the lexicographical sort of $\mathcal{D}$ that is associated with the prefix $p_1 \ldots p_{i-1}$, and (2) is equivalent to having $y(s)$ lie in the interval of the lexicographical sort of the reversed strings of $\mathcal{D}$ that is associated with the suffix $p_{i+1} \ldots p_{|p|/b}$.

To implement (1) and (2) one needs to find the above-described intervals. This can be done using bidirectional tries, as has been done in some of the previous results.

Specifically, divide s into ω-bit words $s_1, \ldots, s_{|s|/\omega}$. The b discussed previously will be a multiple of ω. The current partition of words into $|s|/\omega$ is our choice for the compacted tries construction, whereas the partition of words into $|s|/b$ words will be for the range searching structure.

Construct a compacted trie T of all lexicographically sorted dictionary strings, treating each as a string over alphabet $= [2^\omega]$. To allow constant time traversal from each node in the trie we generate a hash on all first (ω-length) characters emanating from a node. That is if edge e is labeled in the compacted trie with $l(e)$ then for the set $\{(v, a, u) \mid e = (v, u) \in T, l(e) = ax\}$ we generate a perfect hash function h where, in constant time, we can access u from $h(v, a)$. When, traversing with the pattern p, which we also partition into $p_1, \ldots, p_{|p|/\omega}$, we only evaluate the appropriate (ω-length) character of p with the first (ω-length) character on the edge. Once we reach a leaf, or cannot traverse further in the trie - in which case we choose an arbitrary descendant leaf, we use the dictionary string s represented by the leaf to evaluate how far p matches in the trie, by comparing p and s, in comparisons of ω-length characters using the compressed text. We construct a symmetric compacted trie T^R over the reversed strings of the dictionary.

Once we know the path know where p matches in T we traverse this path to compute the boundaries of the range searches described above. That is, after every b binary characters or, in other words, after every b/ω ω-length characters, we need the range of the array of lexicographically ordered strings described by this node. At each such node, we maintain two indices to describe the range. This is the information needed for the range queries. In T we traverse the path from top to bottom and in T^R we traverse from bottom to top.

Now, assuming that (1) and (2) are true, we can show an appropriate condition for (3) to hold.

Lemma 1. *Assume that s has prefix $p_1 \ldots p_{i-1}$ and s has suffix $p_{i+1} \ldots p_{|p|/b}$. Then $c(s)$ and $c(p)$ have Hamming distance 1, i.e., $c(p) \in C(s)$, iff s_i and p_i have Hamming distance 1.*

Proof. Since s has prefix $p_1 \ldots p_{i-1}$ and suffix $p_{i+1} \ldots p_{d/b}$ it follows that $\forall j \neq i$ *and* $\forall l : \; s[jb + l] = p[jb + l]$. Hence, $s[ib + l] = p[ib + l]$ iff for the l-th bit $c(s)_l = c(p)_l$. So we can conclude that $c(s)$ and $c(p)$ have Hamming distance 1 iff s_i and p_i have Hamming distance 1. □

It follows from the lemma that it is sufficient to verify whether $c(p) \in C(s)$ for all dictionary strings s that have prefix $p_1 \ldots p_{i-1}$ and suffix $p_{i+1} \ldots p_{d/b}$. This translates into a single orthogonal 4-sided range reporting query with the 2 ranges defined by prefix $p_1 \ldots p_{i-1}$ and suffix $p_{i+1} \ldots p_{d/b}$ found using the bidirectional tries. The query is asked in the orthogonal range searching structure associated with the color class of $c(p)$. We access this range searching structure, in constant time, with the above-described hash function. Once we have accessed the correct data structure it is a straightforward range query.

6.1 Time and Space

We note that the time and space have interdependencies which we will shortly address. These are affected by the way the dictionary text is saved, by the implementation of the range searching data structures and by the choice of our parameter b. We first explain the space and time and then offer a couple of possible choices of parameters.

Space: Since we only save the skeleton of T, and T^R, their size, including the data saved on the edges and nodes, is $O(d)$ words, or $O(d \log d)$ bits. The perfect hash function table for the data on the edges is also of size $O(d \log d)$ bits.

Hence, the two main space factors are the dictionary size and the range searching data structures. The dictionary itself is only accessed to read substrings. Hence, it can be saved either in open format, in which case the space used will be $|p|d$ bits of space for the dictionary or it can be saved in accessible compressed format. This allows one to analyze the results in either the encoding or indexing model. We give a parameter allowing the user to insert the data structure of their choice. One possible data structure is the following:

Lemma 2. [15] *Given a text $\mathcal{T}$ of length t over constant-sized alphabet there exists a compressed data structure that supports the access in constant time of any substring of $\mathcal{T}$ of length $O(\log t)$ bits requiring $tH_k(T) + o(t)$, where $H_k(\mathcal{T})$ denotes the kth empirical entropy of $\mathcal{T}$ and $k = o(\log t)$.*

Finally the space required by the range searching data structures is dependent on the implementation used which affects the query time as well. To summarize the space: we maintain the dictionary itself in $|DS(\mathcal{D})|$ bit space, where $DS(T)$ is the data structure of choice for maintaining T. The additional space required is $O(d \log d + bS_{\text{rs}}(d))$ bits of space, where $S_{\text{rs}}(d)$ is the number of bits required by the implementation of the range searching data structure on d values.

Query time: First we read the pattern in $O(|p|/\omega)$ time. We also generate the encoding $c(p)$ in this time. Finally, we find all the ranges within the bidirectional tries within $O(|p|/\omega)$ time. This is true because we make one pass on the trie for each pattern using the hash functions on the nodes and access the dictionary text in parallel which we assume can be done in $O(1)$ for each machine word. While walking down in the tree ω bits at a time, we stop on the nodes which are at depths of multiples of b (bits), that is, after every b/ω characters (of ω-bits each). There we learn the ranges by the data stored in the trie. Hence, all the above is done in $O(|p|/\omega)$ time.

The next phase is the query on the orthogonal data structure. We need to access the orthogonal range data structure maintaining the data for color class $c(p)$. This is accessed by hashing $c(p)$. This is done in $O(|p|/\omega)$ time whilst generating $c(p)$. Finally the query itself is a tradeoff based on the implementation of the range searching data structure in use. Let us denote with $Q_{\text{rs}}(d) + occ\, Q'_{\text{rs}}(d)$ the query cost of the $2D$ orthogonal range reporting of our choice. Hence, the overall query time is:

$$O(|p|/\omega + (|p|/b)Q_{\mathrm{rs}}(d) + occ\ Q'_{\mathrm{rs}}(d)).$$

The current best results on 2D orthogonal range reporting in the word RAM model are due to Chan, Larsen, and Pătraşcu [11]. One possible choice is $S_{\mathrm{rs}}(d) = O(d)$, $Q_{\mathrm{rs}}(d) = O(\log^{\epsilon} d)$, and $Q'_{\mathrm{rs}}(d) = O(1)$. In this case we have a solution for lookups with 1 mismatch in the binary alphabet setting which requires $O(bd \log d)$ additional bits of space and $O(|p|/\omega + (|p|/b)\log^{\epsilon} d + occ)$ query time. Set $b = \omega \log^{\epsilon} d$, and we have optimal query time with $O(d\omega \log^{1+\epsilon} d)$-bit space. We note that if the string lengths $= |p|$ are $\leq \omega \log^{\epsilon} d$ then one can use the data structure of [8].

Another option is as follows. Assume that the additional bits of space is bounded by $c(bd \log d)$, for some constant c. We can set $b = \delta|p|/c \log d$ for arbitrary small constant δ and get $\delta|p|d$ bits and $O((1/\delta)\log^{1+\epsilon} d + |p|/\omega + occ)$ query time. This answers the open question of Belazzougui and Venturini [4] in the uncompressed setting for sufficiently large $|p| \gg \omega \log^{1+\epsilon} d$.

Another range searching alternative [11] has $S_{\mathrm{rs}}(d) = O(d \log\log d)$ and $Q_{\mathrm{rs}}(d) = Q'_{\mathrm{rs}}(d) = O(\log\log d)$. This gives $O(|p|d)$ bits and $O(\log d \log^2 \log d + |p|/w + occ \log\log d)$ time.

7 Dictionary Lookup with Edit Distance One

We would like to extend the previous idea of using the xor function to the case of one insertion or deletion. However, an insertion or deletion can skew the entire b-bit encoding and make the dictionary string encodings and the pattern encodings incompatible. Hence, we do something slightly different. We still maintain the idea of the xor encoding, and we generate range queries for them, but we do it differently.

For every $u \in \{0,1\}^{b+1}$ and $v \in \{0,1\}^b$, define the subset:
$\mathcal{D}(u,v) = \{s \in \mathcal{D} : c(s) \oplus v$ can be obtained by deleting 1 character from $u\}$

Now build a $2D$ orthogonal range searching structure for $\{(x(s), y(s)) \mid s \in \mathcal{D}(u,v)\}$, where $x(s)$ and $y(s)$ are defined as before.

Note that each $s \in \mathcal{D}$ belongs to $O(b2^b)$ $D(u,v)$'s (because there are 2^b choices for v and $O(b)$ ways to insert 1 character to $c(s) \oplus v$). So, space blows up by a factor $O(b2^b)$.

7.1 Query algorithm for one character deletion from p

Once again we use a trie for the lexicographically sorted strings of $\mathcal{D}$ and a trie for the sorted reversed strings of $\mathcal{D}$. We traverse both similarly to the one mismatch case.

At the i-th iteration, write p as $\alpha_i p_i \beta_i$ with $|\alpha_i| = bi, |p_i| = b+1, |\beta_i| = b(|s|/b - i - 1)$. We are looking for all $s \in \mathcal{D}$ such that

1. s has prefix α_i and

2. s has suffix β_i and
3. s_i can be obtained by deleting 1 character from p_i.

We make a claim here that is appropriate for the case of one deletion from the pattern.

Lemma 3. *Given that s has prefix α_i and suffix β_i, s_i can be obtained by deleting 1 character from p_i $\iff$ $s \in \mathcal{D}(p_i, c(\alpha_i) \oplus c(\beta_i))$.*

Proof. Given that s has prefix α_i and suffix β_i, we have $c(s) = s_i \oplus c(\alpha_i) \oplus c(\beta_i)$ which directly implies that $s_i = c(s) \oplus c(\alpha_i) \oplus c(\beta_i)$. Set $u = p_i$ and $v = c(\alpha_i) \oplus c(\beta_i)$. Hence, $s_i (= c(s) \oplus v)$ can be obtained by deleting 1 character from $p_i (= u)$ is equivalent by definition to $s \in \mathcal{D}(u, v) = \mathcal{D}(p_i, c(\alpha_i) \oplus c(\beta_i))$. □

We use the same technique on the trie and reverse trie as for the one mismatch case. That is, we have a range in the trie of the dictionary that is appropriate to α_i and a range defined by β_i in the trie of reversed strings. During the traversal of the query we compute $c(\alpha_i) \oplus c(\beta_i)$ at every stage. Now we need to access the orthogonal range reporting structure that is $\mathcal{D}(p_i, c(\alpha_i) \oplus c(\beta_i))$ which is accessible in constant time by a hash based on u and v to $\mathcal{D}(u, v)$, which in our case is p_i and $c(\alpha_i) \oplus c(\beta_i)$ to $\mathcal{D}(p_i, c(\alpha_i) \oplus c(\beta_i))$. Once in the right orthogonal range reporting data structure we ask a 4-sided query based on the ranges we found.

Time and space analysis By following an analysis similar to the mismatch case, we can conclude the following.

The space and time analysis is: $O(b2^b d \log d)$ additional bits of space (over the compressed or uncompressed dictionary) and $O(|p|/\omega + (|p|/b) \log^{\epsilon} d + occ)$ query time.

We can set $b = \log(\delta|p|/\log d \log(\delta|p|))$ for an arbitrary constant $\delta > 0$ and get $\delta|p|d$ bits and $O(((\delta|p|)/\log|p|) \log^{\epsilon} d + occ)$ query time for $|p| \gg \log d$, which is a speedup when $|p|$ is large.

Alternatively, we can get $O((|p|/\log|p|) \log\log d + occ \log\log d)$.

7.2 Query algorithm for inserting one character to p

The case for insertion to p is symmetrical to the deletion case. Hence, we only give the changed definition of $\mathcal{D}(u, v)$.

For every $u \in \{0,1\}^{b-1}$ and $v \in \{0,1\}^b$, redefine the subset $\mathcal{D}(u, v) = \{s \in \mathcal{D} \mid c(s) \oplus v$ can be obtained by inserting 1 character to $u\}$.

References

1. A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Text indexing and dictionary matching with one error. *Journal of Algorithms*, 37(2):309–325, 2000.

2. A. Amir, A. Levy, E. Porat, and B. R. Shalom. Dictionary matching with one gap. In *Combinatorial Pattern Matching - 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16-18, 2014. Proceedings*, pages 11–20, 2014.
3. D. Belazzougui. Faster and space-optimal edit distance "1" dictionary. In *Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, Lille, France, June 22-24, 2009, Proceedings*, pages 154–167, 2009.
4. D. Belazzougui and R. Venturini. Compressed string dictionary look-up with edit distance one. In *Proc. of the Symposium on Combinatorial Pattern Matching (CPM)*, pages 280–292, 2012.
5. P. Bille, I. L. Gørtz, H. W. Vildhøj, and S. Vind. String indexing for patterns with wildcards. *Theory Comput. Syst.*, 55(1):41–60, 2014.
6. G. S. Brodal, P. Davoodi, and S. S. Rao. On space efficient two dimensional range minimum data structures. *Algorithmica*, 63(4):815–830, 2012.
7. G. S. Brodal and L. Gasieniec. Approximate dictionary queries. In *Combinatorial Pattern Matching, 7th Annual Symposium, CPM 96, Laguna Beach, California, USA, June 10-12, 1996, Proceedings*, pages 65–74, 1996.
8. G. S. Brodal and S. Venkatesh. Improved bounds for dictionary look-up with one error. *Inf. Process. Lett.*, 75(1-2):57–59, 2000.
9. H. Chan, T. W. Lam, W. Sung, S. Tam, and S. Wong. Compressed indexes for approximate string matching. *Algorithmica*, 58(2):263–281, 2010.
10. H.-L. Chan, T.-W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong. A linear size index for approximate pattern matching. *Journal of Discrete Algorithms*, 9(4):358–364, 2011.
11. T. M. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*, pages 1–10, 2011.
12. R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. of Symposium on Theory of Computing (STOC)*, pages 91–100, 2004.
13. E. D. Demaine and A. López-Ortiz. A linear lower bound on index size for text retrieval. *Journal of Algorithms*, 48(1):2–15, 2003.
14. P. Ferragina, S. Muthukrishnan, and M. de Berg. Multi-method dispatching: A geometric approach with applications to string matching problems. In *Proc. of Symposium on Theory of Computing (STOC)*, pages 483–491, 1999.
15. P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theor. Comput. Sci.*, 372(1):115–121, 2007.
16. W.-K. Hon, T.-H. Ku, R. Shah, S. V. Thankachan, and J. S. Vitter. Compressed dictionary matching with one error. In *Data Compression Conference (DCC), 2011*, pages 113–122, 2011.
17. T.-W. Lam, W.-K. Sung, and S.-S. Wong. Improved approximate string matching using compressed suffix data structures. *Algorithmica*, 51(3):298–314, 2008.
18. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.
19. M. Lewenstein. Orthogonal range searching for text indexing. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 267–302. 2013.
20. M. Lewenstein, J. I. Munro, Y. Nekrich, and S. V. Thankachan. Document retrieval with one wildcard. In *Mathematical Foundations of Computer Science 2014*, pages 529–540. 2014.
21. M. Lewenstein, Y. Nekrich, and J. S. Vitter. Space-efficient string indexing for wildcard pattern matching. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, pages 506–517, 2014.

22. A. Policriti and N. Prezza. Hashing and indexing: Succinct datastructures and smoothed analysis. In *Algorithms and Computation - 25th International Symposium, ISAAC 2014*, pages 157–168, 2014.
23. D. Tsur. Fast index for approximate string matching. *Journal of Discrete Algorithms*, 8(4):339–345, 2010.
24. A. C. Yao and F. F. Yao. Dictionary look-up with one error. *J. Algorithms*, 25(1):194–202, 1997.