Compressed String Dictionary Look-up with Edit Distance One*

Djamal Belazzougui[1] and Rossano Venturini[2]

[1] LIAFA, Univ. Paris Diderot - Paris 7. dbelaz@liafa.jussieu.fr
[2] Dept. of Computer Science, University of Pisa. rossano@di.unipi.it

Abstract. In this paper we present different solutions for the problem of indexing a dictionary of strings in compressed space. Given a pattern P, the index has to report all the strings in the dictionary having *edit distance* at most one with P. Our first solution is able to solve queries in (almost optimal) $O(|P| + occ)$ time where occ is the number of strings in the dictionary having edit distance at most one with P. The space complexity of this solution is bounded in terms of the k-th order entropy of the indexed dictionary. Our second solution further improves this space complexity at the cost of increasing the query time.

1 Introduction

Modern web search, information retrieval, data base and data mining applications often require solving string processing and searching tasks. Most of such tasks boil down to some basic algorithmic primitives which involve a large dictionary of strings with variable length. The interest in approximate searches over dictionaries of strings is increasing since they appear frequently in many practical scenarios. In Web search, for example, users query the engine with possibly misspelled terms that can be corrected by choosing among the closest terms stored in a trustable dictionary. In data mining and data base applications, instead, an automatically built dictionary may contain noise in the form of misspelled strings. Thus, we may need to resort to approximate searches in order to identify the closest dictionary strings with respect to a (correct) input string.

The *Edit distance* (also known as Levenstein distance) is the most commonly used distance to deal with misspelled strings. The edit distance between two strings is defined as the minimal number of edit operations required to transform the first string into the second string. There are three possible edit operations: deletion of a symbol, insertion of a symbol and substitution of a symbol with another.

* This work has been partially supported by the MADALGO Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation, the French ANR-2010-COSI-004 MAPPI project, the S-CUBE EU-FP7- 215483 project, the ASSETS CIP-ICT-PSP-250527 project, MIUR PRIN MadWeb 2008, and FIRB Linguistica 2006.

The problem *String Dictionary Look-up with Edit Distance One* is defined as follows. Let $\mathcal{D} = \{S_1, S_2, \ldots, S_d\}$ be a set of d strings of total length n drawn from an alphabet Σ of size σ. We want to build a (compressed) index that, given any string $P[1, p]$, reports all the strings in $\mathcal{D}$ having edit distance at most 1 with P. In the following we assume that the strings in $\mathcal{D}$ are all distinct and sorted lexicographically (namely, for any $1 \leq i < d$, $S_i < S_{i+1}$).

In this paper we provide two efficient and compressed solutions for the problem above. The first solution guarantees (almost) optimal query time while requiring compressed space. Namely, we show how to obtain an index of $2nH_k + n \cdot o(\log \sigma) + 2d \log d$ bits, that is able to report all the *occ* strings having edit distance at most 1 with P in time $O(p + occ)$. Here H_k denotes the k-th order entropy of the strings in the dictionary. Interestingly, the time complexity of this solution is independent of alphabet size. This is quite an uncommon result for compressed data structures dealing with texts. The second solution provides possible space/time tradeoffs by using a completely different approach. Its space occupancy, indeed, decreases to $nH_k + n \cdot o(\log \sigma)$ bits. This better space bound is obtained at the cost of increasing the query time to $O(p \log \log \sigma)$.

Interestingly, our first solution can be extended to support an additional operation which has interesting practical applications. We assume that each string S_i in $\mathcal{D}$ has been assigned a score $c(S_i)$. For example, the score could establish the relative importance of any string with respect to the others. It is possible to extend our solution in order to support the extra operation $\mathsf{Top}(P[1, p], k)$ that reports the k highest scored strings in $\mathcal{D}$ having edit distance at most 1 with P. This operation is solved in $O(p + k \log k)$ time.

2 Related work

The literature presents several solutions to the problem of indexing string dictionaries to efficiently search strings with error distance one. In the following we restrict our attention only on results that currently have the best time/space complexities.

The work in [3] proposes two solutions to solve the string dictionary lookup with Hamming distance one[3]. The first solution has $O(p + occ)$ query time and uses $O(\sigma \cdot n \log n)$ bits of space. The main data structure is a trie that indexes strings in $\mathcal{D}$ plus extra strings. An extra string is a string that does not belong to $\mathcal{D}$ but has Hamming distance one with at least a string in the set. Clearly, each root to a leaf path in the trie represents either a string in $\mathcal{D}$ or an extra string. In every leaf representing a string S there is stored the list of indices of strings in $\mathcal{D}$ that have Hamming distance one with S. The query for P is solved by navigating the trie. If a leaf is reached, it reports all the indices stored in the leaf. The major drawback of this solution is represented by its space occupancy for non-constant size alphabet and by its construction time. Indeed, it is unknown how to build this data structure in $O(n\sigma)$ time.

[3] However, they can be easily extended to deal with the more general Edit distance.

The second solution in [3] is slower than the previous one by an additive term $O(\log n)$ (namely, query time is $O(p + \log n + occ)$). The advantage is represented by its space occupancy which is $O(n \log n)$ and, thus, it is better for non-constant size alphabets. The solution resorts to two tries and a balanced search tree. The first trie contains the set of strings $\mathcal{D}$ while the second trie indexes the strings in $\mathcal{D}$ reversed. The query algorithm exploits the following property: if there exists a string S in $\mathcal{D}$ having distance one with $P[1,p]$, it can be factorized as $S = P[1,i] \cdot c \cdot P[i+2,p]$, for some index i and symbol $c \in \Sigma$. This is a key property that has been exploited by almost all the subsequent solutions, including ours. These solutions differ from each other in data structures and algorithms they use to discover all these factorizations. For each string $S[1,s]$ in $\mathcal{D}$, we consider all triplets $(\mathsf{np}_i(S),\ S[i+1],\ \mathsf{ns}_{i+2}(S))$ where $\mathsf{np}_i(S)$ is the identifier of the node corresponding to prefix $S[1,i]$ in the first trie and $\mathsf{ns}_{i+2}(S)$ is the identifier of the node corresponding to $S[i+2,s]$ reversed in the second trie. These triples are inserted in a search tree that is able to report, given a pair of node identifiers u and v, all the triples with u in the first component and v in the third component. The query algorithm works as follows. For any index i, it identifies the nodes $\mathsf{np}_i(P)$ and $\mathsf{ns}_{i+2}(P)$ and uses the search tree by querying for these two nodes. If the triple $(\mathsf{np}_i(P),\ c,\ \mathsf{ns}_{i+2}(P))$ is returned, then the string $S = P[1,i] \cdot c \cdot P[i+2,p]$ is in $\mathcal{D}$ and has distance one from P. Remarkably, this solution can be easily made dynamic: inserting or deleting a string $P[1,p]$ from $\mathcal{D}$ costs $O(p + \log n)$.

The current best solution is the one presented in [1]. This solution follows a similar approach but obtains significantly better time and space complexities. Indeed, this solution achieves $O(p + occ)$ query time by requiring optimal $O(n \log \sigma)$ bits of space. This is obtained by carefully combining compact tries, (minimal) perfect hash functions and Rabin-Karp fingerprinting.

The solution presented in [4] solves the problem for Hamming distance and it deals with binary strings having a fixed length L. The strong limitation on the length of strings allows to achieve optimal $O(L/w)$ query time, where w is the size of a memory word. However, the space usage grows to $O(n \cdot L \lg L)$ bits. Moreover, this solution can only report a single matching string from the dictionary.

Finally, we observe that currently known solutions to solve the more general problem of approximate full-text indexing are not competitive with solutions presented in this paper. Indeed, all known solutions for approximate full-text indexing for edit distance one incur at least a factor $\Omega(\log n)$ in space usage and/or an additive $\Omega(\log n)$ term in query time.

3 Background

In this section we collect a set of algorithmic tools that will be used by our solutions. We report each result together with a brief description. More details can be obtained by consulting the corresponding references.

Compressed strings with fast random access. We will require the availability of a storage scheme for a text T which uses compressed space and is able to decode in $O(1)$ time any symbol of T. To this aim, we use the following result in [9].

Lemma 1. *Given a text $T[1, n]$ drawn from an alphabet of size σ, there exists a compressed data structure that supports the access in constant time of any substring of T of length $O(\log n)$ bits requiring $nH_k(T) + \mathsf{extra}$ bits, where $H_k(T)$ denotes the kth empirical entropy of T and $k = o(\log_\sigma n)$. The extra space depends on the alphabet size σ: $\mathsf{extra} = o(n)$ if $\log \sigma = o(\log n / \log \log n)$, $\mathsf{extra} = n \cdot o(\log \sigma)$ otherwise.*

The scheme can be also used in cases in which T is the concatenation of a set of strings (namely, $T = S_1 \cdot S_2 \cdot \ldots \cdot S_d$). The starting positions of strings in T are stored by resorting to Elias-Fano's representation [7, 8] within $d \log(\frac{n}{d}) + O(d)$ bits. This additional structure allows us to access an arbitrary portion of any string in optimal time.

Rabin-Karp signature. Given a string $S[1, s]$, the Rabin-Karp signature [13] $\mathsf{rk}(S)$ is equal to $\sum_{i=1}^{s} S[i] \cdot t^i \pmod{M}$, where M is a prime number and t is a randomly chosen integer in $[1, M-1]$. Given a set of strings $\mathcal{D}$ of d strings of total length n, it can be obtained an instance $\mathsf{rk}()$ of the Rabin-Karp signature that maps strings in $\mathcal{D}$ to the first $O(M)$ integers without collisions, with M chosen among the first $O(n \cdot d^2)$ integers. It is known that a value of t that guarantees no collisions can be found in expected $O(1)$ attempts (e.g., see the analysis in [6]). The representation of the suitable function requires $O(\log n)$ bits of space.

Interestingly, Rabin-Karp signature guarantees that, after a preprocessing phase over a string S, signatures of strings close enough to S can be computed in constant time. This property is formally stated by the following lemma.

Lemma 2. *Given a string $S[1, s]$, for every prefix P of S, $\mathsf{rk}(P)$ can be computed in constant time. Moreover, for every string Q at distance 1 from S, $\mathsf{rk}(Q)$ can be computed in constant time. It is required a preprocessing phase that takes time $O(s)$.*

Minimal Perfect Hash Function. Result in [12] shows how to build a space/time optimal minimal perfect hash function. This result is summarized in the following lemma.

Lemma 3. *Given a subset of $S \subseteq U = 2^w$ of size n, there exists a minimal perfect hash function for S that can be evaluated in $O(1)$ time and requires $n \log e + o(n)$ bits of space.*

Compressed static function. Often we have to represent satellite data associated with the keys in S. Repetitions in these associated values can be exploited in order to reduce space requirements. The following result can be proven by using standard techniques.

Theorem 1. *A function F that assigns values from $[\sigma]$ (with $\sigma = \omega(1)$) to keys in $S = \{x_1, x_2, \ldots, x_n\} \subset U \subseteq 2^w$ can be represented in $nH_0 + n \cdot o(\log \sigma)$ bits such that the evaluation of F requires constant time, where H_0 denotes the empirical entropy of the assigned values $\{F(x_1), F(x_2), \cdots, F(x_n)\}$.*

Proof. We use a minimal perfect hash function $\mathsf{m}()$ to map keys to the first n integers by paying $\log e + o(1)$ bits per key (Lemma 3). We construct a sequence A that has the value associated with key x_j in position $m(x_j)$. Sequence A is represented in compressed form by using schema of Lemma 1.

4 A compressed and fast solution

Our first solution can be seen as a compressed variant of the solution presented in [1]. However, we need to apply significant and non-trivial changes to that solution in order to achieve compressed space and to retain exactly the same (almost optimal) query time. More formally, in this section we prove the following theorem.

Theorem 2. *Given a set of strings $\mathcal{D} = \{S_1, S_2, \ldots, S_d\}$ of d strings of total length n drawn from an alphabet Σ of size σ, there exists an index that, given any pattern $P[1, p]$, reports in $O(p + occ)$ time all the occ strings in $\mathcal{D}$ having edit distance at most one with P. It requires:*

1. *$nH_k + o(n) + 2d \log d$ bits of space for any $k = o(\log n)$, if $\sigma = O(1)$;*
2. *$2nH_k + n \cdot o(\log \sigma) + 2d \log d$ bits of space for a fixed $k = o(\log_\sigma n)$, otherwise.*

At a high level our solution works as follows. Firstly, it identifies a set of $O(p+occ)$ *candidate strings* being a superset of the strings that have edit distance at most one with P. Then, it discards all candidate strings that actually do not belong to $\mathcal{D}$. For the moment, let us assume that establishing whether or not a candidate string belongs to $\mathcal{D}$ costs constant time. Later, we will discuss how to efficiently perform this non-trivial task[4].

Our solution requires identifying strings in $\mathcal{D}$ that share prefixes and suffixes with the query string P. For this aim we resort to two patricia tries $\mathcal{PT}$ and $\mathcal{PT}_r$ that index respectively the strings in $\mathcal{D}$ and the strings in $\mathcal{D}$ written in reversed order. As common, a node in each patricia trie is uniquely identified by the time of its visit in the preorder visit of the tree. The tree structure of each patricia trie is represented in $O(d)$ bits with standard succinct solutions [14]. In order to perform searches on patricia tries, we add data structures to compute the length of longest common prefix (lcp) and longest common suffix (lcp_r) for any pair of strings in $\mathcal{D}$. A standard constant time solution requiring $O(d(1+\log \frac{n}{d}))$ bits of space is obtained by writing lcps between lexicographically consecutive strings (resp. reverse strings) using Elias-Fano's representation [7, 8] and by resorting to Range Minimum Queries (rmq) (see e.g., [11]) on these

[4] Notice that just accessing each symbol of these candidate strings would cost $O(p + p \cdot occ)$ time which is much higher than our claimed complexity.

arrays. Fast percolation of the tries is obtained by augmenting the branching nodes with monotone minimal perfect hash functions as described in [2]. In this way choosing the correct edge to follow from the current node can be done in constant time regardless of the alphabet size. The extra cost in term of space is bounded by $O(d \log\log \sigma)$ bits. The correctness of the steps performed during the search is established by comparing the searched string and labels on the followed edges. This is done by accessing directly to the appropriate portion of strings in $\mathcal{D}$ from their compressed representations. For this aim $\mathcal{D}$ is represented by resorting to the compressed scheme of Lemma 1 that allows constant time access to any symbol of any string in $\mathcal{D}$. The space required by this is bounded by kth order entropy accordingly to Lemma 1. Since the strings do not keep their original order in the trie $\mathcal{PT}_r$, we store a permutation π of $\{1, 2, \dots, d\}$ that keeps track of the original order in $\mathcal{D}$ of each leaf of $\mathcal{PT}_r$. Namely, $\pi(i)$ is the index in $\mathcal{D}$ of the ith lexicographically smaller string in $\mathcal{PT}_r$. Clearly, storing π requires $d \log d + O(d)$ bits.

Candidate strings obtained by deleting a symbol. The identification of candidate strings for deletion is an easy task. Indeed, we observe that there are just p possible candidate strings obtainable from $P[1, p]$ by deleting one of its symbol. Thus, we simply consider any string $P[1, i] \cdot P[i + 2, p]$ as a candidate string. However, any of these strings is reported only after having checked that it actually belongs to $\mathcal{D}$. As said above, for the moment we assume that this non-trivial task can be done in $O(1)$ (amortized) time.

Candidate strings obtained by inserting or substituting a symbol. Identifying candidate strings for insertion or substitution of a symbol is an easy task whenever the alphabet has constant size. In this case there are, indeed, $O(\sigma \cdot p) = O(p)$ candidate strings obtained by inserting or substituting any possible symbol of Σ in any position of P. This implies that data structures above suffice for Point 1 in Theorem 2[5]. Identifying insertions and substitutions with a larger alphabet is a much harder task, which requires an additional data structure. Our additional data structure follows the idea presented in [1] which allows us to reduce the number of candidate strings from $O(\sigma \cdot p)$ to $O(p + occ)$. However, our solution is forced to use more sophisticated arguments in order to achieve space bounded in term of kth order entropy. In the following we consider only insertions since substitutions are solved similarly.

Given the set of strings $\mathcal{D}$ and the two patricia tries $\mathcal{PT}$ and $\mathcal{PT}_r$, our first step consists in building a set $\mathcal{T}$ of tuples. For each string S in $\mathcal{D}$ of length s, we consider each of its factorizations of the form $S = S[1, i] \cdot c \cdot S[i + 2, s]$. For each of them, we add to $\mathcal{T}$ the tuple $\langle \mathsf{np}, i, c = S[i + 1], s - (i + 2), \mathsf{ns} \rangle$ where np (resp. ns) is the index of the highest node in $\mathcal{PT}$ (resp. $\mathcal{PT}_r$) prefixed by $S[1, i]$ (resp. $S[i + 2, s]$ reversed). Observe that the cardinality of $\mathcal{T}$ is at most n, since we add at most s tuples for a string S of length s.

[5] Recall that we are still assuming that we can check in $O(1)$ whether a candidate string belongs to $\mathcal{D}$.

The set $\mathcal{T}$ contains enough information to allow the identification of all the candidate strings. For insertion we consider all the factorizations of P having the form $P = P[1,i] \cdot P[i+1,p]$. For each of them, we identify the (highest) nodes np_i and ns_{i+1} in $\mathcal{PT}$ and $\mathcal{PT}_r$ that are prefixed respectively by $P[1,i]$ and $P[i+1,p]$ reversed. Clearly, identifying all these nodes for all the factorizations of P requires $O(p)$ time by resorting to the patricia tries.

The key observation to identify candidate strings is the following: If there exists a tuple $\langle \mathsf{np}_i, i, c, p-i-1, \mathsf{ns}_{i+1} \rangle$ in $\mathcal{T}$, then the string $S = P[1,i] \cdot c \cdot P[i+1,p]$ belongs to $\mathcal{D}$ and, obviously, has distance one from P[6].

Our data structure is built on top of $\mathcal{T}$ and allows us to easily identify the required tuples. We notice that there may exist several tuples of the form $\langle \mathsf{np}, i, \star, \mathsf{ns}, i' \rangle$. These groups of tuples share the same four components np, i, ns and i', and differ just for the symbol c. In order to distinguish them, we arbitrarily rank tuples in the same group and we assign to each of them its position in the ranking. We build a data structure that, given the indexes np and ns of two nodes, two lengths i and i' and rank r, returns the symbol c of the rth tuple of the form $\langle \mathsf{np}, i, \star, \mathsf{ns}, i' \rangle$ in $\mathcal{T}$. The data structure is allowed to return an arbitrary symbol whenever such a tuple does not exist. The use of such a data structure to solve our problem is simple. For each factorization $P[1,i] \cdot P[i+1,p]$ of P, we query the data structure above several times by passing the parameters np_i, i, $p-i-1$, ns_{i+1} and r. The value of r is initially set to 0 and increased by 1 for the subsequent query. After every query, we check if the string $S = P[1,i] \cdot c \cdot P[i+1,p]$ belongs to $\mathcal{D}$, where c is the symbol returned by the data structure. We pass to the next factorization as soon as we discover that either the string S does not belong to $\mathcal{D}$ or symbol c has been already seen for the same factorization. Both these conditions provide the evidence that no tuple $\langle \mathsf{np}_i, i, \star, p-i-1, \mathsf{ns}_{i+1} \rangle$ with rank r or larger can belong to $\mathcal{T}$. It is easy to see that the overall number of queries is $O(p + occ)$.

We are now ready to present a data structure to index $\mathcal{T}$ as described above that requires $O(1)$ time per query and uses entropy bounded space.[7] The first possible compressed solution consists in appropriately defining a function $F()$ which is then represented by using solution in Theorem 1. For any tuple $\langle \mathsf{np}, i, c, \mathsf{ns}, i' \rangle$ having rank r in $\mathcal{T}$, we set $F(\mathsf{np}, i, \mathsf{ns}, i', r)$ equal to c. Queries above are solved by appropriately evaluating function $F()$. Accordingly to Theorem 1, each query is solved in constant time. As far as space occupancy is concerned, we observe that $F()$ is defined for at most n values and that any symbol of any string in $\mathcal{D}$ is assigned at most once. Thus, by combining these considerations with Theorem 1, it follows that the representation of $F()$ requires at most $nH_0 + n \cdot o(\log \sigma)$ bits. A boost of this space complexity to nH_k is obtained by defining several functions F, one for each possible context of length k. Here $k = o(\log_\sigma n)$ is an arbitrary but fixed parameter. The function $F_{\mathsf{cntxt}}()$ is defined only for tuples $\langle \mathsf{np}, i, c, \mathsf{ns}, i' \rangle$ where the symbol c is preceded by the

[6] Observe that similar considerations hold also for substitutions with the difference that we skip ith symbol in factorizations of the form $P = P[1,i-1] \cdot P[i] \cdot P[i+1,p]$.

[7] We remark that the set of tuples $\mathcal{T}$ is just conceptual and not explicitly stored.

context cntxt in the string that induced the tuple. By summing up the cost of storing the representations of these functions, we have that the space occupancy is bounded by $nH_k + n \cdot o(\log \sigma)$ bits for the fixed $k = o(\log_\sigma n)$. Notice that splitting F in several functions is not an issue for our aim. In the algorithm above, indeed, we can query the correct function since we are always aware of the correct context.

Checking candidate strings. It is left to explain how to establish, in constant time, whether a candidate string belongs to $\mathcal{D}$. Observe that any candidate string has the form $S = P[1,i] \cdot P[i+2,p]$ in case of deletion, $S = P[1,i] \cdot c \cdot P[i+1,p]$ in case of insertion, or $S = P[1,i] \cdot c \cdot P[i+2,p]$ in case of substitution, for some symbol c and index i. The main issue behind this task is given by the fact that strings may not fit in a constant number of memory words. Thus, we cannot manage them directly in constant time. For this aim Rabin-Karp function $\mathsf{rk}()$ is used to create small sketches of the strings in $\mathcal{D}$ that fit in $O(1)$ memory words and that uniquely identify each string. Observe that the signatures assigned by function $\mathsf{rk}()$ are values smaller than M and, thus, each of them fits in $O(1)$ words of memory.

Once we have these perfect signatures, we use a minimal perfect hash function to connect each signature to the corresponding string in $\mathcal{D}$. Let $\mathcal{D}_{\mathsf{rk}}$ be the set of signatures assigned by $\mathsf{rk}()$ to strings in $\mathcal{D}$ (i.e., $\mathcal{D}_{\mathsf{rk}} = \{\mathsf{rk}(S) \mid S \in \mathcal{D}\}$). We construct a minimal perfect hash function mph that maps signatures in $\mathcal{D}_{\mathsf{rk}}$ to the first n integers. Lemma 3 guarantees $O(1)$ evaluation time by requiring $O(d)$ bits of space. As satellite data, the entry for the string S stores in $\log d + O(1)$ bits the index of the leaf in $\mathcal{PT}_r$ that corresponds to S reversed. Clearly, if S belongs to $\mathcal{D}$, $\mathsf{mph}(\mathsf{rk}(S))$ gives us in constant time the index of S reversed in $\mathcal{PT}_r$ while $\pi(\mathsf{mph}(\mathsf{rk}(S)))$ reports the index of S in $\mathcal{PT}$. It is worth noticing that the result of these operations are meaningless whenever S does not belong to $\mathcal{D}$.

The check of candidate strings requires a preprocessing phase shared among all the candidate strings. Firstly, we compute in $O(p)$ the Rabin-Karp signatures of all prefixes and suffixes of P. In this way, the signature of any candidate string S can be computed in constant time by appropriately combining two of those signatures (Lemma 2). Then, we identify a leaf pleaf in $\mathcal{PT}$ that shares the longest common prefix with P. Similarly, we identify a leaf sleaf in $\mathcal{PT}_r$ having the longest common prefix with P reversed. Given the properties of patricia tries and our succinct representation, identifying these two leaves cost $O(p)$ time.

The check for the single candidate string $S = P[1,i] \cdot c \cdot P[i+1,p]$ obtained by inserting symbol c in $(i+1)$th position is done as follows [8]. We compute in constant time the values $k = \pi(\mathsf{mph}(\mathsf{rk}(S)))$ and $k' = \mathsf{mph}(\mathsf{rk}(S))$. Then, we have to check that the candidate string S is equal to the string S_k in $\mathcal{D}$. Instead of comparing S and S_k symbol by symbol, we exploit the fact that S and S_k coincide if and only if the following three conditions are satisfied:

- $\mathsf{lcp}(k, \mathsf{pleaf})$ is at least i;

[8] Checks for other types of errors are done in a similar way.

– $\mathsf{lcp}_r(k', \mathsf{sleaf})$ is at least $p - i$;
– $(i+1)$th symbol of S_k is equal to c.

Clearly, these three conditions are checkable in constant time. The $O(p)$ preprocessing time is amortized over the $O(p + occ)$ candidate strings.

Finding Top-k strings. As we mentioned in the Introduction, our solution could be extended to support an additional operation which has interesting practical applications. Assume that each string S_i in $\mathcal{D}$ has assigned a score $c(S_i)$. For example, the score could establish the relative importance of any string with respect to the others. It is possible to extend our solution in order to support the extra operation $\mathsf{Top}(P[1,p], k)$ that reports the k highest scored strings in $\mathcal{D}$ having edit distance at most 1 with P. This operation is solved in $O(p + k \log k)$ time. We assume that values $c()$ are available for free. Notice that we can easily avoid this assumption by storing in $d \log d + O(d)$ bits the ranking of strings in $\mathcal{D}$ induced by $c()$.

We first present a simpler $O((p + k) \log k)$ time algorithm which is, then, modified in order to achieve the claimed time complexity. We said above that an arbitrary rank is assigned to tuples in $\mathcal{T}$ belonging the same group (namely, tuples of the form $\langle \mathsf{np}, i, \star, \mathsf{ns}, i' \rangle$ that differ just for the symbol $\star$). Instead, this algorithm requires that the assigned ranks respecting the order induced by $c()$. Namely, lower ranks are assigned to tuples corresponding to strings with higher values of $c()$. The searching algorithm is similar to the previous one. The main difference is in the order in which the factorizations of $P[1,p]$ are processed. The algorithm works in steps and keeps a heap. The role of the heap is that of keeping track of the top-k candidate strings seen so far. Each factorization is initially considered *active* and becomes *inactive* later in the execution. Once a factorization becomes inactive, it is no longer taken into consideration. Each factorization has also associated a score which is initially set to $+\infty$. At each step, we process the active factorization with the largest score. We query function $F()$ with the correct value of r for the current factorization. Let S be the candidate string identified by resorting to $F()$. If S does not belong to $\mathcal{D}$, the current factorization becomes inactive and we continue with the next factorization. Otherwise, we insert S into the heap with its score $c(S)$ and we decrease the score associated to the current factorization to $c(S)$. At each step we also check the number of string into the heap. If they are $k + 1$, we remove the string with the lowest score and we declare inactive the factorization that introduced that string.

Notice that, apart from the first k steps, in each step a factorization becomes inactive. Since there are $O(p)$ factorizations, our algorithm performs at most $O(p + k)$ insertions into a heap containing at most k strings. Thus, the claimed time complexity easily follows. The improvement is obtained by observing that most of the time (i.e., $O(p \log k)$) is spent in inserting the first string of each factorization into the heap. This is no longer necessary if we use the following strategy. We first collect the first string of each factorization together with its score and we apply the classical linear time selection algorithm to identify the k-th smallest score. This step costs $O(p)$ time. We immediately declare inactive the $p-k$ factorizations whose strings have a smaller score. We insert the remaining k strings into the heap and we use the previous algorithm to complete the task. The latter step costs now $O(k \log k)$ time, since we have just k active factorizations.

5 A more compressed solution

The factor 2 multiplying the H_k term in space bound of Theorem 2 may be annoying in some scenario. In this section we provide a solution which is able to overcome this limitation at the cost of (slightly) increasing the query time. Formally, we prove the following theorem.

Theorem 3. *Given a set of strings $\mathcal{D} = \{S_1, S_2, \ldots, S_d\}$ of d strings of total length n drawn from an alphabet Σ of size σ, there exists an index requiring $nH_k + n \cdot o(\log \sigma)$ bits of space for a fixed $k = o(\log_\sigma n)$ that, given any pattern $P[1, p]$ reports all the occ strings in $\mathcal{D}$ having edit distance at most one with P in:*

1. *$O(p(min(p, \log_\sigma n \log\log n)) + occ)$ worst-case time when $\sigma = \log^c n$ for some constant c.*
2. *$O(p \log\log \sigma + occ)$ worst-case time when $\sigma = \omega(\log^c n)$ for any constant c.*

This solution uses a completely different approach with respect to our previous one. Indeed, it resorts to a collection of compressed permuterm indexes [10] built on the dictionary $\mathcal{D}$. More precisely, we divide strings in $\mathcal{D}$ based on their length. Let $\mathcal{D}_\ell$ denote the set of strings in $\mathcal{D}$ of length ℓ. A compressed permuterm index R_ℓ is built for each set $\mathcal{D}_\ell$.[9]

In [10] it is shown how design a Burrows-Wheeler Transform [5] (BWT) based index for a dictionary of strings. Among the other types of queries, the index solves efficiently the so-called PrefixSuffix query that, given a prefix P and a suffix S, identifies all the strings in the dictionary having P as prefix and S as suffix. In our solution we are interested in this type of query which is solved by using a slightly different variant of the compressed permuterm index. The main difference is the sorting strategy used to obtain the underlying Burrows-Wheeler Transform (BWT) [5]. In [10] a text is obtained by concatenating the strings of the dictionary by using, as separator, a special symbol # not appearing in Σ. Then, all the suffixes of this text are sorted lexicographically to obtain the rows of the Burrows-Wheeler matrix. In our variant we first append the symbol # at the end of each string, then we construct the BWT matrix by sorting lexicographically all the cyclic rotations of the strings in the set. This different way to proceed guarantees us that symbols in any row belong to the same string.

[9] We notice that the number of distinct lengths and, thus, compressed permuterm indexes is $O(\sqrt{n})$.

This fact turns out to be useful below when we will define parent and depth operations. The searching algorithm presented in [10] does not change[10].

Given a pattern $P[1,p]$, we query only three compressed permuterm indexes: R_{p-1} for deletions, R_p for substitutions and R_{p+1} for insertions. In the following we will only describe the solution for insertion. Deletion and substitution are solved in a similar way. The basic idea behind our searching algorithm is the following. For each cyclic rotation $P_i = P[i,p]\#P[1,i-1]$ of P, we use the compressed permuterm index R_{p+1} in order to identify the range of rows of Burrows-Wheeler Transform [5] which are prefixed by P_i, if any (see [10] and references therein for more details). We observe that having that range $[r,l]$ suffices for identifying the strings in $\mathcal{D}$ obtained by inserting a symbol in ith position on P. These symbols are, indeed, the ones contained in $\mathsf{BWT}_{p+1}[l,r]$, where BWT_{p+1} is the Burrows-Wheeler Transform of set $\mathcal{D}_{p+1}$. However, we cannot compute all these ranges in a naïve way (i.e., searching each P_i separately), since it would cost at least p^2 time.

A faster solution requires to augment the compressed permuterm index with a data structure that supports the two operations: parent and depth. Consider the conceptual compact trie build on top of rows of BWT_{p+1}. Given a range $[l,r]$, let u be the node of the above trie corresponding to range $[l,r]$. The two operations are defined as follows:

1. $\mathsf{parent}(u)$ returns the range $[l',r']$ corresponding to the parent of the node u;
2. $\mathsf{depth}(u)$ returns the length of the locus of node u.

Using the solution presented in [15], we are able to support both these operations in $O(\log_\sigma \hat{n} \log\log \hat{n})$ time by requiring $O(\hat{n}\frac{\log\sigma}{\log\log\hat{n}})$ bits of additional space, where $\hat{n}$ is the total size of the indexed dictionary.

Our solution works in two phases. In the first phase, it identifies the range of rows of BWT_{p+1} sharing the longest common prefix with $P_0 = \#P[1,p]$. This is done by using the following strategy. We search P_0 backwards. At any step i, we keep the following invariant: $[l_i,r_i]$ is the largest range of rows of BWT_{p+1} which are prefixed by the longest prefix of $P_0[p-i,p+1]$. We also keep a counter ℓ that tells us the length of this prefix. Notice that it may happen that a backwards step from $[l_i,r_i]$ with the next symbol $P[p-i-1]$ returns an empty range. In this case, we repeatedly enlarge the range $[l_i,r_i]$ via parent operations until the subsequent backwards step is successful. The value of ℓ is kept updated by increasing it by one after every successful backwards step or by setting it equal to the value returned by depth after every call to parent.

Similarly, the second phase matches suffixes of P backwards. The main difference is given by the fact that the starting range $[l_1,r_1]$ and value of ℓ are the ones computed in the previous phase. In each step, we claim that we have identified the range of rows prefixed by some $P_i = P[i,p]\#P[1,i-1]$, for the appropriate i, as soon as the value of ℓ reaches $p+1$. The overall time complexity of these

[10] Actually, the different construction of the Burrows-Wheeler Transform defined here was already implicitly in use in [10] and simulated at query time by means of function jump2end (see [10] for more details).

two phases is $O(p \log_\sigma \hat{n} \log\log \hat{n})$, since we have at most $2p$ calls to parent and depth.

The discussion above provides a proof of Point 1 of Theorem 3. Observe that the time complexity of the above solution is dominated by the time spent in performing parent and depth operations.

Point 2 of Theorem 3 is obtained by showing that, for sufficiently large alphabet (i.e., $\sigma = \omega(\log^c n)$, for some constant c), faster implementations of these operations (i.e., $O(\log\log\sigma)$ time) are possible. We can, indeed, improve the time complexity of the solution above if we are allowed to use more space. More precisely, using more space, we can improve the time of parent (for all cases) and depth (for the case of large depths):

1. The operation parent can be supported in constant time using $O(n)$ additional bits of space. This is feasible by using the Sadakane's compressed suffix tree [16].
2. The operation depth can be supported in constant time using $O(n \log t)$ bits of space when the string depth is at least $p - t$ for some parameter t. For this aim, we can just store a table Δ which stores $\log(t+1)$ bits per node. These bits will store a special value whenever the depth is less than $p - t$, otherwise, we store the difference between the depth and p.

Now that we have a constant time parent operation, the depth operation remains as the only bottleneck for achieving faster query time. Assume that the compressed suffix tree supports the depth operation in time t. We first notice that a given range obtained after $t' < t$ backwards steps can correspond to a P_i if and only if the depth of the node obtained after the last parent operation was precisely $p - t'$. This condition can be checked directly by probing the table Δ. If this is not the case, we adopt a lazy strategy. Instead of computing a depth after each parent, we safely wait until we performed t backwards steps after the last parent operation. The $O(t)$ time required by depth is amortized on the cost of these (at least) t backwards steps. Point 2 of Theorem 3 is proven by setting $t = O(\log_\sigma n \log\log n)$ and by observing that the backwards steps become the dominant cost (i.e., $O(p \log\log\sigma)$).

6 Conclusion

In this paper we described two different compressed solutions for the look-up with Edit distance one in a dictionary of strings. The first solution requires $2H_k(S) + n \cdot o(\log\sigma) + 2d \log d$ bits of space for a fixed $k = o(\log_\sigma n)$. It is able to solve queries in (almost optimal) $O(|P| + occ)$ time where occ is the number of strings in the dictionary having edit distance at most one with the query pattern P. The second solution further improves this space complexity which is reduced to $nH_k + n \cdot o(\log\sigma)$ bits. However, the time complexity grows to $O(|P| \log_\sigma n \log\log n + occ)$ or $O(|P| \log\log\sigma + occ)$ depending on the alphabet size. Interestingly enough, the two solutions solve the problem at hand by resorting to two different approaches: the former is based on (perfect) hashing while the latter is based on the compressed permuterm index.

An interesting open problem asks to design an index that obtains simultaneously the time complexity of the former solution and the space complexity of the second one. Furthermore, it is still open the question regarding the possibility of designing a solution that solves the problem in $O(|P| \cdot \log \sigma / w + occ)$ time, where w is the size of a word machine. At the moment, there does not exist any solution achieving such a time complexity, even non compressed one.

Finally, building efficient dictionaries for edit distance d larger than 1 is still an open problem. However, the approaches we used in our two solutions are not easily extendible to efficiently solve query for higher edit distance. Indeed, we could just solve a query in $O(\sigma^{d-1}|P|^d + occ)$ time for edit distance d by resorting to the standard dynamic programming approach.

References

1. Djamal Belazzougui. Faster and space-optimal edit distance "1" dictionary. In *CPM*, pages 154–167, 2009.
2. Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. In *ESA*, pages 748–759, 2011.
3. Gerth Stølting Brodal and Leszek Gąsieniec. Approximate dictionary queries. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching*, volume 1075 of *Lecture Notes in Computer Science*, pages 65–74. Springer Verlag, Berlin, 1996.
4. Gerth Stølting Brodal and Venkatesh Srinivasan. Improved bounds for dictionary look-up with one error. *Information Processing Letters*, 75(1-2):57–59, 2000.
5. Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
6. Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable (extended abstract). In *ICALP*, pages 235–246, 1992.
7. Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21:246–260, April 1974.
8. Robert Mario Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, Project MAC*, 1971.
9. Paolo Ferragina and Rossano Venturini. A simple storage scheme for strings achieving entropy bounds. *Theor. Comput. Sci.*, 372(1):115–121, 2007.
10. Paolo Ferragina and Rossano Venturini. The compressed permuterm index. *ACM Transactions on Algorithms*, 7(1):10, 2010.
11. Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.
12. Torben Hagerup and Torsten Tholey. Efficient minimal perfect hashing in nearly minimal space. In *STACS*, pages 317–326, 2001.
13. Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
14. J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.
15. Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):53, 2011.
16. Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.