Dictionary Matching and Indexing with Errors and Don't Cares[*]

Richard Cole
Courant Institute
New York University
NY, NY 10012
cole@cs.nyu.edu

Lee-Ad Gottlieb
Courant Institute
New York University
NY, NY 10012
adi@cs.nyu.edu

Moshe Lewenstein
Department of Computer Science
Bar-Ilan University
Ramat Gan 52900
Israel
moshe@cs.biu.ac.il

ABSTRACT

This paper considers various flavors of the following online problem: preprocess a text or collection of strings, so that given a query string p, all matches of p with the text can be reported quickly.

In this paper we consider matches in which a bounded number of mismatches are allowed, or in which a bounded number of "don't care" characters are allowed.

The specific problems we look at are: indexing, in which there is a single text t, and we seek locations where p matches a substring of t; dictionary queries, in which a collection of strings is given upfront, and we seek those strings which match p in their entirety; and dictionary matching, in which a collection of strings is given upfront, and we seek those substrings of a (long) p which match an original string in its entirety. These are all instances of an all-to-all matching problem, for which we provide a single solution.

The performance bounds all have a similar character. For example, for the indexing problem with $n = |t|$ and $m = |p|$, the query time for k substitutions is $O(m + \frac{(c_1 \log n)^k}{k!} + \# \text{ matches})$, with a data structure of size $O(n\frac{(c_2 \log n)^k}{k!})$ and a preprocessing time of $O(n\frac{(c_2 \log n)^k}{k!})$, where $c_1, c_2 > 1$ are constants. The deterministic preprocessing assumes a weakly nonuniform RAM model; this assumption is not needed if randomization is used in the preprocessing.

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: Pattern matching, computations on discrete structures; E.1 [**Data Structures**]: Trees.

*This work was supported in part by NSF grant CCR-0105678.

General Terms

Algorithms.

Keywords

Dictionary query, dictionary matching, suffix trees, wildcards, approximate pattern matching, text indexing.

1. INTRODUCTION.

The internet is awash with an abundance of textual information due to large and growing collections of databases, articles, and books. It is essential to be able to access this information with fast online queries of different sorts and much work has been devoted to doing so efficiently.
In this paper we consider three different query paradigms.

1. Text Indexing: In text indexing one desires to preprocess a text t and to answer where subsequent queries p appear in the text t.
2. Dictionary Queries: Here one is given a dictionary D of strings $p_1, ..., p_d$ and subsequent queries ask whether a given pattern query p appears in the dictionary.
3. Dictionary Matching: In dictionary matching one is given a dictionary D of strings $p_1, ..., p_d$. Subsequent queries provide a query string s and ask for each location in s at which patterns of the dictionary appear.

As we will see, these are all instances of the following all-to-all matching problem: given a preprocessed collection of texts $r_1, r_2, \ldots, r_x$, and a collection of query patterns $q_1, q_2, \ldots, q_y$, find all locations where some q_i matches a substring of some r_j, and all locations where some r_j matches a substring of some q_i.
The classical pattern matching problem is the problem in which one searches for the appearances of a pattern p at locations of a text t. Pattern matching has been generalized to searching with error bounds, e.g. Hamming distance [10, 20, 28], and edit distance [15, 29, 34]. These problems are known as *approximate pattern matching.* Several other pattern matching problems have been considered within the approximate paradigm, e.g. approximate 2-dimensional matching, approximate repeats (in biology).
Fischer and Paterson [19] generalized pattern matching to include don't cares: given a pattern p and a text t either

of which may contain don't cares, denoted ϕ, the goal is to output all locations of t where p matches. (A don't care can match any alphabet character.) They presented an algorithm that runs in time $O(n \log n \log \Sigma)$. Cole and Hariharan [16] improved this to $O(n \log n)$. The don't care paradigm has been extended to several other problems, including 2-dimensional matching and various approximate matching problems [11, 3, 4].

Approximate dictionary querying was introduced by Minsky and Papert [33] in 1969. There are abundant applications for approximate indexing, approximate dictionary queries, and approximate dictionary matching, some of which we shall mention later.

It is very natural to consider the online paradigm in conjunction with the approximate and don't care paradigms. For example, the biological retrieval system of IBM, TEIRESIAS, can answer queries which contain don't cares. (See http://cbcsrv.watson.ibm.com/Tspd.html.)

For the approximate online problems some progress has been made for the case $k = 1$, and for larger k for the Hamming distance measure when some imprecision in the output is acceptable. In particular, there are several solutions for the case where one allows one error, i.e. $k = 1$ [8, 14]. However, even for two errors the naive bounds are still the best currently known. If some imprecision in the output is acceptable, specifically, all matches within distance k and some matches up to distance $k(1 + \epsilon)$, for a fixed constant $\epsilon > 0$, then there are efficient solutions [26, 27], for Hamming distance. (We note that this form of output is called approximate near neighbor, where the approximation refers to the ϵ; we are using approximate to refer to k.) Setting $\epsilon < \frac{1}{k}$ eliminates the imprecision, but the performance is then no better than that of a straightforward approach.

Results allowing don't cares within online queries have been heuristic. We are not aware of any complexity bounds for such problems. Heretofore, it has not been apparent how to approach this problem without paying a large penalty.

The central difficulty in extending the previous solutions, for $k = 1$, is that they are tailored to one-error problems. Most of the methods use a pair of suffix trees, one for the text and the other for its reverse. The idea behind these data structures is to imagine "sitting" on the error and to verify matches to the right and left using the suffix trees. (Other methods use slightly different data structures but suffer from problems of the same flavor.)

In this paper we suggest a new general method for solving this collection of problems. Our approach uses only one suffix tree and treats errors by recursively creating subtrees to handle them. The method entails a non-obvious subtree merging, use of centroid path decompositions and elaborate LCA queries. The primary and most important advantage of our method is that it works for $k \geq 2$. However, it must be noted that it is most effective for constant k. In addition, our method even improves what is known in the case of one error, i.e. for $k = 1$.

For don't cares we consider patterns that contain k don't cares. The methods are similar to those for k mismatches; however, we can tighten the bounds even further here. We can also allow don't cares in the text; then the bounds are similar to those for errors.

Approximate Indexing: The indexing problem arises in textual retrieval systems for static texts and online queries. Suffix trees are the classical data structure used to solve this problem. Suffix trees can be constructed in linear, $O(n)$, time and space [17, 31, 37, 38] for linear-size alphabets. Moreover, subsequent queries p are answered in linear, $O(m + occ)$, time, where occ is the number of answers to the query and $m = |p|$.

For approximate indexing, given a distance k, one preprocesses a specified text t. The goal is to find all locations ℓ of t within distance k of the query p, i.e. for the Hamming distance measure, all locations ℓ such that the length m substring of t beginning at that location can be made equal to p with at most k character substitutions. (The edit distance is analogous, except that a mix of inserts, deletes and substitutions are allowed.) For $k = 1$, two solutions are known [8, 14]. The more efficient method preprocesses the text in time $O(n \log n)$ and answers subsequent queries p in time $O(m \log \log n + occ)$. For small $k \geq 2$, nothing better than the naive solution has been achieved. One possible solution is to traverse a suffix tree. However, while the preprocessing needed to build a suffix tree is cheap, the search is expensive, namely, $O(m^{k+1}|\Sigma|^k + occ)$. Another direct solution, for the Hamming distance measure only, leads to data structures of size approximately $O(n^{k+1})$. This can be slightly improved by using the one-error data structures, which reduce the size to approximately $O(n^k)$. Similar bounds can be achieved by setting $\epsilon = \frac{1}{k+1}$ in the algorithms with imprecision in the output [26, 27], but the direct solution is simpler.

Our solution: The data structure for substitutions uses space $O(n\frac{(c_1 \log n)^k}{k!})$, takes time $O(n\frac{(c_1 \log n)^k}{k!})$ to build, and answers queries in time $O(\frac{(c_2 \log n)^k \log \log n}{k!} + m + occ)$, where $c_1, c_2 > 1$ are constants, and $k \leq \log n$. For edit distance, the constants c_1 and c_2 are larger, and the query time becomes $O(\frac{(c_2 \log n)^k \log \log n}{k!} + m + 3^k \cdot occ)$. Note that even for $k = 1$ our query time improves on the previous results. More importantly, this yields an efficient solution for $k \geq 2$. However, it must be pointed out that this result is most effective for constant k.

For indexing with k don't cares in the pattern, we can preprocess t in $O(n \log^k n + n \log \Sigma)$ time to build a size $O(n \log^k n)$ data structure which answers queries in time $O(2^k \log \log n + m + occ)$.

Approximate Dictionary Queries: Minsky and Papert [33] originally formulated the problem as follows. Given a dictionary D of x[1] binary strings, a k-query p asks for all strings in D within Hamming distance k of p. This problem in its more modern version is known as the *nearest neighbor* problem, with Hamming distance used as the desired measure. Applications for this problem appear in Geographic Information Systems (GIS), Computer Aided Design (CAD), computational biology, decision support, and pattern recognition [36]. Password security is an application that uses very small constant k [30]. We let n denote the sum of the lengths of the strings in D and $m = |p|$.

For the case of imprecise outputs, progress has been made in obtaining efficient data structures, as with the approximate indexing problem. Otherwise, once again the only progress has been for the case of $k = 1$. Here, Yao and Yao [40] presented the first results, namely $O(m \log \log x)$ query time with a data structure of size $O(n \log m)$. Brodal and Gąsieniec [12] improved this to a size $O(n)$ structure with $O(m)$ query time. Brodal and Venkatesh [13] improved this

[1]conventionally, d is used to denote $|D|$.

further, for the cell-probe model with word size $\Theta(m)$, giving a solution requiring $O(\frac{n}{m}\log m)$ space and $O(1)$ query time. However, all these results are for $k = 1$.

Our solution: We present results that carry over to a larger number of errors. Namely, for k substitutions our data structure uses space $O(n + x\frac{(c_1 \log x)^k}{k!})$, is built in time $O(n + x\frac{(c_1 \log x)^k}{k!} + n\log n)$, and has a query time of $O(m + \frac{(c_2 \log x)^k}{k!} \cdot \log\log n + occ)$, where $c_1, c_2 > 1$ are constants. The bounds for edit distance have larger constants c_1 and c_2. Randomization reduces the $n \log n$ term in the preprocessing to n.

Approximate Dictionary Matching: The dictionary matching problem and several variants of the problem have been well studied [2, 5, 6, 7, 25]. We are interested in the approximate version of the problem. There are many applications to this very natural problem; see, for example, an application in text filtering [18]. Here, once again, there are several algorithms for the case where $k = 1$ [8, 14, 18]. The best solution for this problem has query time $O(m \log\log n + occ)$ [14], where m is the query text size, and $n = \sum_{i=1}^{x} |r_i|$ is the dictionary size, where r_i are the patterns in the dictionary; the data structure uses space $O(n \log n)$ and can be built in time $O(n \log n)$.

Our solution: We achieve the same bounds as in the approximate dictionary query problem.

Remark. For both the dictionary problems the $\log\log n$ term can be reduced to $O(\log\log x)$.

Our constructions all require Σ to be ordered. Further, we are using a RAM model with length w words, $w \geq \log n$. In the deterministic preprocessing, if $w > \Theta(\log n)$, we need a fixed number of precomputed constants (as specified in [23]). This is the weakly nonuniform model mentioned earlier.

Paper organization: Section 2 covers definitions and other preliminaries. The heart of the paper, Section 3, provides the solution for each match relation in turn: wildcards in the pattern, wildcards in the text and wildcards in both, substitution or Hamming distance, and edit distance. In each case, we begin with the solution for $k = 1$ and then extend it to general k with a simple recursion. We then provide two technical sections which contain the methods for merging compressed tries and for constructing the LCP data structure (defined below).

2. PRELIMINARIES AND DEFINITIONS

We are concerned with strings over a finite alphabet Σ. A *prefix-free* collection of strings has the property that no string in the collection is a prefix of another. Given a not necessarily prefix-free collection $\{s_1, \cdots, s_k\}$ we make it prefix-free by appending a new character $\$ \notin \Sigma$ to each string. As usual t denotes a text and p a pattern. $n = |t|$ and $m = |p|$.

A *trie* is a tree that stores a prefix-free collection of strings as follows. Each edge in the trie is labelled by a character. Each leaf is associated with a unique string in the collection. Further, the concatenated labels of the edges on the path from the root to a leaf ℓ form the string s_ℓ associated with that leaf. Finally, the labels on edges exiting a node are all distinct. Sometimes, to avoid heavy handed terminology we will refer to a string u, where u is a leaf; s_u, the string associated with u, will be intended. Likewise a string s_u may be called a leaf; and u will be intended.

In a *compressed trie* the labels on the edges are strings of one or more characters. A compressed trie for a collection of strings is obtained from the corresponding (uncompressed) trie by replacing maximal paths of one-child nodes with a single edge.

A *location* in a trie is the name of a node in the trie. A location in a compressed trie is the node or (imaginary) point on an edge corresponding to a node in the corresponding uncompressed trie. It can be specified as a pointer to an edge together with an index into that edge's label.

The *suffix tree* for a string t is the compressed trie storing all suffixes of $t\$$. As is well known, such a suffix tree can be built in linear time [17, 31, 37, 38], as can the suffix tree for a collection of prefix-free strings. We will also need to add the suffixes of a query string p to the suffix tree which can be done readily in $O(p \log \Sigma)$ time. If a hash function has been built for Σ (or rather that portion of Σ used in the text), which can be done in deterministic time $O(\min\{n, \Sigma\} \log n)$ [23] (the weak nonuniformity arises here) or randomized time $O(n)$, then p and its suffixes can be added in $O(p)$ time.

Often, data structures for constant time LCA queries are used with suffix trees, as will be the case here. An LCA query $lca(u, v)$ is given two nodes u, v in a tree T and reports the lowest common ancestor w of u and v in T. Data structures for answering LCA queries in $O(1)$ time can be built in linear time [24, 35]. These data structures also allow the reporting in $O(1)$ time of the edges exiting w on the paths to u and v. In addition, they yield the length of the longest common prefix of the suffixes s_u and s_v associated with u and v, again in $O(1)$ time.

We will also need a data structure for answering the following query on an n-node compressed trie in $O(\log\log n)$ time. Given a leaf u and a distance h, report the location at which the prefix of s_u of length $|s_u| - h$ ends, i.e. the location distance h above u, where edges are deemed to have length equal to their labels. We call this the measured ancestor structure. Again, such data structures can be built in linear time [9].

Our construction uses *centroid paths* and *centroid path decompositions*. For our setting, we define the *centroid path* of a tree T to be the path starting at T's root, which at each node v on the path branches to v's "largest" child, with ties broken arbitrarily; the size of a node is simply the number of leaves in the subtree rooted at that node. In a centroid path decomposition, we decompose each off-path subtree of the centroid path recursively.

The weight of a node on a centroid path is defined to be the number of leaves in its off-path subtrees.

In our applications, for node v on a centroid path with off-path child u, it is convenient to include edge (v, u) in the off-path subtree T_u incorporating node u. We will also say that T_u *hangs from* node v.

The following property of a centroid path decomposition of a tree is well known.

PROPERTY 1. *Let T be an n-node tree with a centroid path decomposition. Let v be a node of T. The path from the root of T to v traverses at most $\log n$ centroid paths.*

In the uncompressed trie, the one-character edge label of the edge on C leaving ℓ will sometimes be called the *next character* after location ℓ on centroid path C. The same terminology is used with respect to the corresponding location in the corresponding compressed trie. Likewise, the *first character* in a subtree U rooted at node v is simply the next character after v.

Finally, we need a new data structure, which we call the LCP data structure, but before defining it, we specify the query problem formally.

2.1 Problem Specification

The matching problem is parameterized by an error bound k and a match relation R (one of wildcards in the pattern, wildcards in both the text and the pattern, Hamming distance, edit distance).

The matching problem has the following form.

Input: text strings $r_1, r_2, \cdots, r_x$.

Query: pattern strings $q_1, q_2, \cdots, q_y$.

Output: Category 1 - for each q_j, all i such that a prefix of r_i matches q_j, and

Category 2 - for each r_i, all j such that a prefix of q_j matches r_i.

If q_j matches r_i this part of the output is in both categories.

In text indexing, the r_i's consist of all the suffixes of an input text t. A query consists of a single pattern. Thus $x = |t|$ and $y = 1$. The output is filtered so as to include only Category 1 locations.

In the dictionary query problem, the r_i's consist of the strings $p_1, \cdots, p_d$ forming the dictionary, and again a query consists of a single pattern. Thus $x = d$ and $y = 1$. The output is filtered so as to include only those locations in both categories.

In dictionary matching, the r_i's again consist of the strings $p_1, \cdots, p_d$. The query patterns consist of all the suffixes of an input string s. Thus $x = d$ and $y = |s|$. The output is filtered so as to include only Category 2 locations.

Below, to simplify, we only describe how to find Category 1 locations. Only cosmetic changes are needed to find Category 2 locations, or locations in both categories.

The complexity of the data structure used for our algorithm depends on the cost of building a suffix tree S for $r_1, r_2, \cdots, r_x$ and for adding $q_1, q_2, \cdots, q_y$ to S. To this end, we introduce the following notation. Let $n = |S|$ and let m be the size of a suffix tree for $q_1, q_2, \cdots, q_y$. Further, suppose the input is provided so that S can be computed in $O(n)$ time and so that the suffixes $q_1, q_2, \cdots, q_y$ can be added to S in $O(m \log n)$ time. That this matters can be seen for example in the indexing problem, where $r_1, r_2, \cdots, r_x$ are the suffixes of the input string t; given t, S can be built in $O(t) = O(n)$ time rather than $O(\sum_{i=1}^{x} |r_i|) = O(n^2)$ time. Then the size of the data structure is $O(n + \frac{c^k x \log^k x}{k!})$ where c is a suitable constant which depends on the match relation, and it can be built in deterministic time $O(n \log n + \frac{c^k x \log^{k+1} x}{k!})$ and in randomized time $O(n + \frac{c^k x \log^k x}{k!})$. The query time is $O(m + \frac{y b^k \log^k x \log\log n}{k!} + occ)$ where b is a suitable constant which depends on the match relation. (Actually when the match relation is wildcards in the pattern, the query time is smaller: it is $O(m + y 2^k \log\log n + occ)$; and when the match relation is edit distance, the term occ is replaced by $3^k \cdot occ$ for category 1 matches.)

Note that in text indexing, $n = |t|$. Likewise, in dictionary matching $m = |s|$.

When dealing with wildcards in building the suffix tree, wildcards in the text are regarded simply as another distinct character. However, for a pattern $q_i = q_{i0} \phi q_{i1} \phi \cdots \phi q_{ik}$, where ϕ denotes a wildcard, instead of adding q_i to the suffix tree, each of $q_{i0}, q_{i1}, \cdots, q_{ik}$ are added to the suffix tree. This does not affect the complexities stated above.

2.2 The LCP Data Structure

We need to introduce a new data structure, the *longest common prefix* data structure, or *LCP* for short. This structure comes in two forms: the *rooted* LCP and the *unrooted* LCP, depending on the form of the queries, as described next.

The LCP structure stores a set of strings $s_1, s_2, \ldots, s_z$. In our application, $s_1, s_2, \ldots, s_z$ are some of the suffixes of the input strings $r_1, r_2, \ldots, r_x$. The LCP structure is built on top of the compressed trie T for $s_1, s_2, \ldots, s_z$.

A query consists of a pattern p. In a rooted query, the task is to report the location ℓ in T corresponding to the longest common prefix of p and $s_1, \cdots, s_z$. This is the location reached if one follows the path from the root labelled by p until one can go no further. We also call this the location at which p *diverges from* T. In an unrooted query, the query consists of an input location h in addition to pattern p. The query is simply a rooted query on the subtrie rooted at h. Suppose that it outputs location ℓ. We call ℓ the location at which p diverges from T when starting at h.

The rooted LCP takes space $O(z)$ and the unrooted LCP takes space $O(z \log z)$. Given the compressed trie T, the time to build the data structures is $O(z)$ and $O(z \log z)$, respectively, assuming that the suffix tree S for strings $r_1, r_2, \ldots, r_x$ is available. (The weak nonuniformity is used here too.)

We explain later for which collections of suffixes rooted and unrooted LCP structures are built.

The query patterns too will be somewhat limited. Each matching task concerns a collection of patterns $q_1, q_2, \cdots, q_y$. The queries to the rooted or unrooted LCPs are all suffixes of $q_1, q_2, \cdots, q_y$. By preprocessing $q_1, q_2, \cdots, q_y$ once they are known, each query can then be answered in $O(\log\log n)$ time. This further round of preprocessing takes time $O(\sum_{i=1}^{y} |q_i|)$, and occurs once for each matching query. This preprocessing consists of adding each of the strings $q_1, q_2, \cdots, q_y$ to S, so as to determine for each suffix s of each q_i an index that lies between the indices for the two strings in S straddling s in lexicographic order. (This can be kept integral by using only even indices for strings in S.)

Thus, the picture in general is that the text $(r_1, r_2, \ldots, r_x)$ is preprocessed once and for all; Then, on receipt of a matching query, the pattern $(q_1, q_2, \ldots, q_y)$ is also preprocessed. Finally, the matching query itself is performed.

3. THE K-ERRATA TRIE

We describe in turn the following data structures:

1. for up to k wildcards in the pattern,
2. for up to k wildcards in the pattern or text,
3. for up to k substitution errors,
4. for up to k substitutions, insertions or deletions.

In each case, the data structure provides an efficient implementation of the following naive strategy for matching a query string q. Given a compressed trie T for the collection of strings to be searched, for example the suffix tree of a text string, the algorithm simply follows each possible path P starting at the root of T until k wildcard errors are encountered on P in matching the given pattern q. (In case (4) multiple possible combinations of substitutions/insertions/deletions may have to be considered on each path.) For an alphabet Σ, in problem (1) this yields a search time of $O(q\Sigma^k)$, in problems (2) and (3) of $O(q^{k+1}\Sigma^k)$, and in problem (4) of $O(q^{k+1}\Sigma^k k)$. One factor of q can be reduced to an $O(\log\log n)$ factor by using the LCP data structure. We show how to achieve polylogarithmic search time for fixed k, at the cost of a polylogarithmic increase in the size of the data structure.

3.1 Wildcards in the Pattern

We begin by describing the solution for $k = 1$.
The expense in the naive algorithm comes from the fact that if a wildcard in the pattern corresponds to the first character following a node v in T, then all the up to Σ subtrees rooted at v have to be searched. We show how to reduce this to two searches.
Our solution is readily understood by considering a centroid path decomposition of T. Let C be one of the resulting centroid paths and let v be a node on C. We form a new *wildcard subtree* at v comprising the merge of all off-path subtrees of v, but with the first character of each subtree being replaced by a new symbol $\phi \notin \Sigma$.
We start by building the suffix tree S for $r_1, r_2, \ldots, r_x$ in time $O(n)$. Then, from S, we extract the compressed trie T for $r_1, r_2, \ldots, r_x$. Next, we build a rooted LCP structure for each of the wildcard subtrees minus their first character, and an unrooted LCP structure for T.
Given a query string $q = p_0\phi p_1$, where ϕ denotes a wildcard, the search proceeds as follows.

1. Add p_0 and p_1 to S.

2. Perform an LCP query for p_0 from the root of T. If this location ℓ is at depth $|p_0|$ the search can continue.

3. Advance one character along the centroid path edge at ℓ to location ℓ' (effectively reading the wildcard in q), and perform an (unrooted) LCP query for p_1 from ℓ'. If this yields a location ℓ'' at depth $|q|$, then all the descendant leaves of ℓ'' correspond to matches of q.

4. If ℓ is a node, then Step 3 is also performed on the wildcard subtree at ℓ, except that the LCP query is rooted.

Lemma 1. *The above search runs in time $O(q+\log\log n+ occ)$.*

Lemma 2. *The above data structure uses space $O(n + x\log x)$.*

Proof. Let u be a leaf in T. We argue that u (or rather leaves corresponding to strings obtained by replacing one character in s_u by ϕ) belongs to at most $\log x$ wildcard trees. For consider traversing the path $v_j, \cdots, v_1$ from u to the root of T. u belongs to the wildcard tree hanging from v_i exactly if v_i and v_{i+1} are on distinct centroid paths. But there are at most $\log x$ such nodes. $\square$
To build a k-errata data structure we begin by building the data structure for one wildcard and then recursively processing each wildcard subtree to handle $k-1$ (additional) wildcards.
Let $q = p_0\phi p_1\phi p_2\phi\cdots\phi p_k$ be a query pattern. Now each of $p_0, p_1, p_2, \cdots, p_k$ has to be added to the suffix tree S.
In the search p_0 is sought as before. Then, in steps 2 and 3, $p_1\phi p_2\phi\cdots\phi p_k$ is sought recursively. It is readily seen that:

Lemma 3. *The above data structure supports searches in time $O(q + 2^k \log\log n + occ)$.*

We show:

Lemma 4. *The above data structure has size $O(n+x\frac{(k+\log x)^k}{k!})$.*

Proof. Consider a structure on a collection R of x input strings. Consider the compressed trie T for R. Let C be the centroid path starting at the root of R. The wildcard trees hanging from C have size at most $x/2$ and total size x. Consider the centroid paths hanging from C; the wildcard trees hanging from these centroid paths each have size at most $x/4$ and total size x; at the next level down the wildcard trees each have size at most $x/8$ and total size x, and so forth.
Let $S_k(x)$ denote the space taken by the structure for k wildcards on a collection of x strings and inductively suppose that $S_k(x)/x$ is a non-decreasing function of x. Then:
$S_k(x) \le x + x\frac{S_{k-1}(x/2)}{x/2} + x\frac{S_{k-1}(x/4)}{x/4} + \cdots x\frac{S_{k-1}(1)}{1}, \quad k \ge 1$
$S_0(x) = x$
It is not hard to verify by induction that

$$S_k(x) \le x+x\frac{\log x}{1!}+\cdots+x\frac{(\log x)(1+\log x)\cdots(k-1+\log x)}{k!}$$

For $\sum_{i=1}^{\ell} \frac{i(i+1)\cdots(i+j-1)}{j!} = \frac{\ell(\ell+1)\cdots(\ell+j)}{(j+1)!}$. $\square$
It then follows readily that:

Lemma 5. *The above data structure can be built in deterministic time $O(n\log n + x\frac{(k+\log x)^k}{k!})$ and in randomized time $O(n + x\frac{(k+\log x)^k}{k!})$.*

3.2 Wildcards in the Text and Pattern

Again, we start with the case $k = 1$. For simplicity, we start with the case in which the wildcards occur only in the text. Suppose we build a trie for the text in which wildcards are treated as a new character, α say. We call each off-path subtree whose first character is α an α-wildcard subtree.
Consider a search. It proceeds as before, but if it encounters an α on the current centroid path, it continues the search from the next location on the edge, just as in Step 3 in the wildcards in the pattern problem. Additionally, suppose the search path leaves the current centroid path C at node v; then all α-wildcard subtrees hanging from v or its ancestors on C need to be searched. Potentially, many α-wildcard subtrees need to be searched. To keep the search costs bounded we will form groups of subtrees.
The goal is to ensure that only $O(\log x)$ group trees need to be searched. Each group tree will be derived from a collection of suffixes of the original set R of input strings. Let $T_1, T_2, \cdots, T_h$ be the α-wildcard subtrees hanging from centroid path C, in top to bottom order. The obvious solution is to form a $\log x$-level hierarchy of group trees. The bottom level comprises the individual trees, the next level pairs of adjacent trees (i.e. $T_1 \cup T_2, T_3 \cup T_4$, etc.), the next level foursomes of adjacent trees, and so forth. (However, as we see later, the meaning of a merge is not completely obvious.) Then on any centroid path C, in order to search trees $T_1, T_2, \cdots, T_g$, we may need to search up to $\log x$ group trees. As up to $\log x$ distinct centroid paths may be traversed in tracing the pattern q without error, this may entail the search of $\theta(\log^2 x)$ group trees.
In fact, by using suitable weights in organizing the hierarchy, the total number of group trees that need to be searched

can be kept to $O(\log x)$. Specifically, α-wildcard subtree T_f, hanging off node v, is given a weight equal to the number of strings stored in the off-path subtrees hanging from v (i.e. their number of leaves). In particular, suppose that trees $T_1, T_2, \cdots, T_h$ have weights $w_1, w_2, \cdots, w_h$ and let $W = \sum_{i=1}^{h} w_i$. The hierarchy of groups is organized so that T_f belongs to $O(\log \frac{W}{w_f})$ groups, in analogy with weighted search trees.

To understand the construction, it is helpful to imagine the subtree T_f occupying an interval I_f of length w_f, with the intervals arranged contiguously in index order on the interval $(0, W]$. The construction proceeds top-down. At the top there is a single group for the interval $(0, W]$. A group for the interval $(u, x]$ will have three subgroups; one will contain a single tree, the tree whose interval includes point $\frac{u+x}{2}$, and covers interval $(v, w]$; the other two subgroups cover intervals $(u, v]$ and $(w, x]$, respectively. Clearly, an interval at depth i has size at most $\frac{W}{2^{i-1}}$, and hence a subtree of size w_f belongs to at most $1+\lceil \log \frac{W}{w_f} \rceil$ groups.

Recall we assumed that the path traced by pattern q leaves the centroid path C at node v. This causes α-wildcard trees $T_1, T_2, \cdots, T_g$ to be searched, where T_g is hanging from node v. The handling of $T_1, \cdots, T_{g-1}$ and T_g differ, as we explain next.

In order to form groups involving subsets of $T_1, \cdots, T_{g-1}$, we need the group tree to match the characters in the pattern originally aligned with a wildcard in some α-wildcard tree. To this end, for each $T_f, 1 \le f < g$, its first character is changed to equal the next character on the centroid path C from which it hangs. The group trees are formed from C and these altered wildcard trees T'_f. The rooted LCP queries are made on the group trees whose disjoint union forms $T'_1 \cup T'_2 \cup \cdots \cup T'_{g-1}$.

However, the search in T_g is different. As the path traced by the query pattern leaves C at node v, q does not match the next character labelling C. Instead, here T_g is maintained as is, and the search continues in T_g, skipping the next character in q and the first character in T_g.

We note that in principle two rooted LCP structures on each T_f are needed. First, an LCP structure is needed for the modified tree T'_f because it will occur in a singleton group, which may need to be searched. Second, an LCP structure is needed for tree T_f minus its first character. But the latter structure can be used for the first problem by starting the search after skipping the first characters in T'_f and the LCP query pattern.

To allow the right group trees to be searched, the path P_q traversed by q has to be identified to the following extent: for each centroid path C_i overlapping P_q the vertex v_i at which P_j leaves C_i has to be found. This is most readily done following the LCP query, which reports the location ℓ at which query pattern q diverges from T, by tracing back from location ℓ to the root of T, taking $O(1)$ time to go from node v_i to the root of centroid path C_i and then to its parent node v_{i-1}, for each of the $O(\log x)$ centroid paths ancestral to ℓ. This takes $O(\log x)$ time.

We describe later, in Section 4, how the group trees are built. There, we analyze the search time and the space used by the data structure.

LEMMA 6. *The search takes time $O(\log x \log \log x + occ)$.*

PROOF. Consider the path traced in S by a wildcard-free pattern p and the wildcard trees hanging from this path. It is convenient to view the search as generating recursive 1-errata search problems as it descends the group hierarchy or switches paths. Consider a size x 1-errata search problem. We argue that in reaching a size at most $\frac{x}{2}$ 1-errata problem, it generates at most 4 LCP searches on appropriate subgroups. This can occur if the first search generates a recursive 1-errata subproblem on the middle group (of size almost x). The next recursive 1-errata problem will have size at most $\frac{x}{2}$, however. Any other choice results in only 2 LCP searches. Thus, overall, there are $O(\log n)$ LCP searches. □

LEMMA 7. *The data structure uses space $O(x \log x)$.*

PROOF. Consider the group wildcard trees to which a string s belongs. As these are all associated with the first wildcard in s, they are all associated with one centroid path and consequently there are at most $\log x + 1$ of them. □

To handle wildcards in both the pattern and text, we overlay the above two constructions. More specifically, for each node v, the off-path subtrees hanging from v, with the exception of the α-wildcard tree, are merged to form a wildcard tree at v as is done in the wildcards in the pattern data structure. The only change in the search arises if a wildcard in the pattern is encountered. If this wildcard is read immediately after a node v is met then LCP queries must be performed on both the α-wildcard and wildcard trees hanging from v. The asymptotic complexity of the solution for $k = 1$ is unaffected, however.

The k-errata structure is built as before by recursion on the α-wildcard and wildcard trees.

LEMMA 8. *For $k \le \log n$, the above k-errata structure uses space $O(\frac{x \log x (1+\log x) \cdots (k-1+\log x)}{k!})$.*

PROOF. First, we analyze the structure that handles wildcards in the text alone.

Consider an arbitrary string $s \in T$ and consider the $(k-1)-$ α-wildcard group trees in which s is placed. This series of α-wildcard group trees, from largest to smallest, at least halves in size from tree to tree, except possibly at the bottommost level, which is formed by the singleton group.

Let $S_k(m)$ denote the space used by a k-errata tree on m strings. Let $\tilde{S}_k(m) = S_k(m)/m$. We assume inductively that $\tilde{S}_k(m)$ is a non-decreasing function of m. Then the space used by $(k-1)$-errata structures on the non-singleton group trees is at most:

$$x[\tilde{S}_{k-1}(x/2) + \tilde{S}_{k-1}(x/4) + \cdots + \tilde{S}_{k-1}(1)].$$

While the space used by $(k-1)$-errata structures for the singleton group trees is at most:

$$\sum_{i=1}^{z} x_i \tilde{S}_{k-1}(x_i), \text{ where } \sum_{i=1}^{z} x_i = x,$$

for the sets of strings stored in the α-wildcard trees are disjoint.

But $\sum_{i=1}^{z} x_i \tilde{S}_{k-1}(x_i) \le x \tilde{S}_{k-1}(x)$, thus the space used by the k-errata structure $S_k(x)$ is bounded by:

$$S_k(x) \le x[\tilde{S}_{k-1}(x) + \tilde{S}_{k-1}(x/2) + \cdots + \tilde{S}_{k-1}(1)] + x.$$

This yields the stated bound as in Lemma 4.

To accommodate wildcards in the pattern, wildcard trees are needed as well as α-wildcard trees. But the sets of strings in the wildcard and α-wildcard trees are disjoint, so the same recurrence equation, which held for each scenario separately, continues to apply. $\square$

LEMMA 9. *The search time in the k-errata structure for x strings is $O(4^k \log x(1+\log x)\cdots(k-1+\log x)/k! \cdot \log\log n + occ)$.*

PROOF. It is convenient to view the k-errata search as it descends the group hierarchy or switches centroid paths as generating a recursive k-errata search. The $O(\log\log n)$ cost of the one LCP query is charged to the size 1 k-errata problem at the base of the recursion. The $O(\log x)$ cost of tracing the centroid paths is charged at a rate of $O(1)$ to each of the k-errata recursive calls.
There can be at most 4 recursive $(k-1)$-errata searches, using the same argument as in the proof of Lemma 6.
Thus letting $T_k(x)$ denote the search time for a k-errata search on a data structure for x strings, we have:

$$\begin{array}{rcll} T_k(x) & \leq & T_k(x/2) + 4T_{k-1}(x) + 1, & k \geq 1, \quad x > 1 \\ T_k(1) & = & \log\log n + T_{k-1}(1) & k \geq 1 \\ T_0(x) & = & \log\log n & \end{array}$$

It is a straightforward induction to verify that:

$$\begin{array}{rcl} T_k(x) & \leq & \{4^k \log x(1+\log x)\cdots(k-1+\log x)/k! \\ & + & 4^{k-1} 2\log x(1+\log x)\cdots(k-2+\log x)/(k-1)! \\ & + & \cdots \\ & + & 4k\log x \\ & + & (k+1)\}\log\log x \\ & + & 4^{k-1}\log x(1+\log x)\cdots(k-1+\log x)/k! \\ & + & 4^{k-2}\log x(1+\log x)\cdots(k-2+\log x)/(k-1)! \\ & + & \cdots \\ & + & \log x \end{array}$$

$\square$

3.3 Hamming or Substitution Distance

We begin with the solution for the case $k = 1$. It is similar to the wildcard in text data structure.
Consider a centroid path C in the compressed trie T. For each node v on C, for each off-path subtree T_{va}, where a denotes the first character in T_{va}, a substitution tree T^s_{va} is formed. T^s_{va} is identical to T_{va}, except that the first character is replaced by a new symbol ψ. The substitution trees T^s_{va} hanging from v are grouped in a manner analogous to the grouping of α-wildcard trees to form a hierarchy of group trees associated with v.
In addition, substitution trees T^s_v are formed. T^s_v comprises the merge of all the substitution trees T^s_{va}, but with the ψ changed to be the next character on centroid path C. The trees T^s_v are also grouped in the same way as the α-wildcard trees.
The search method is similar to that used for wildcards in the text. An LCP query for p is performed on the compressed trie T. Suppose p traces out a path through centroid paths $C_1, C_2, \cdots, C_j$, leaving centroid path C_i at location ℓ_i, $1 \leq i \leq j$. Note that for $i < j$, ℓ_i is a node, which we rename v_i, while for $i = j$ it need not be. Then the substitution trees T^s_u hanging from nodes u above ℓ_i on C_i are searched by means of LCP queries on $O(\log x)$ groups, exactly as in the wildcard in the text data structure. In addition, at node v_i, the substitution trees $T^s_{v_i a}$, apart from the substitution tree containing centroid path C_{i+1}, are also searched by means of LCP queries on group trees; as we will see, this entails searching a further $O(\log x)$ group trees. Finally, for each C_i, an unrooted LCP query is performed starting one character below location ℓ_i.

LEMMA 10. *For $k \leq \log n$, the above k-errata structure uses space $O(3^k n \log n(1+\log n)\cdots(k-1+\log n)/k!)$.*

PROOF. Consider an arbitrary leaf $u \in T$ and consider the $(k-1)$-substitution group trees in which u is placed. Each successive group tree, from top to bottom, is decreasing in size, and in fact at least halves in size, except when the group tree corresponds to a middle interval, which occurs at most twice per centroid path ancestral to u.
Let $S_k(x)$ denote the space used by a k-errata tree on x strings. Let $\tilde{S}_k(x) = S_k(x)/x$. We assume inductively that $\tilde{S}_k(x)$ is a non-decreasing function of x. Then:
$S_k(x) \leq 3x[\tilde{S}_{k-1}(x) + \tilde{S}_{k-1}(x/2) + \cdots + \tilde{S}_{k-1}(1)] + x$ for $k \geq 1$ and $S_0(x) = x$.
It is readily verified that this has solution:

$$\begin{array}{rcl} S_n(x) & \leq & x[3^k \log x(1+\log x)\cdots(k-1+\log x)/k! \\ & + & 3^{k-1}\log x(1+\log x)\cdots(k-2+\log x)/(k-1)! \\ & + & \cdots \\ & + & 1] \end{array}$$

$\square$

LEMMA 11. *The search time in the above k-errata data structure is $O(6^k \log x(1+\log x)\cdots(k-1+\log x)/k! + occ)$.*

PROOF. Let $T_k(x)$ be the search time on the k-errata structure for x strings. We will derive:

$$\begin{array}{rcll} T_k(x) & \leq & T_k(x/2) + 6T_{k-1}(x) + 1 & k \geq 1, \ x > 1 \\ T_k(1) & \leq & \log\log n + T_{k-1}(1) & k \geq 1 \\ T_0(x) & = & \log\log n & \end{array}$$

which yields the lemma, as before.
To help verify the above recursive equation we introduce some more notation to specify different types of recursive subproblems. Let $CPG_k(x)$ denote the running time for searching a k-errata group tree on x strings for a collection of two or more nodes on a centroid path, and let $CPS_k(x)$ denote the same running time for a group associated with a single node; also, let $OPG_k(x)$ denote the corresponding running time for a group associated with two or more off-path subtrees hanging from a common node on a centroid path, and $OPS_k(x)$ be the running time associated with a single off-path subtree. Then:

$$\begin{array}{rcl} OPS_k(x) & \leq & max\{CPG_k(x/2) + 2OPS_{k-1}(x), CPS_k(x/2) \\ & & +2OPS_{k-1}(x)\} + 1 \\ CPG_k(x) & \leq & max\{CPG_k(x/2) + 2OPS_{k-1}(x), CPS_k(x) \\ & & +2OPS_{k-1}(x)\} \\ CPS_k(x) & \leq & max\{OPG_k(x/2) + 2OPS_{k-1}(x), OPS_k(x) \\ & & +2OPS_{k-1}(x)\} \\ OPG_k(x) & \leq & max\{OPG_k(x/2) + 2OPS_{k-1}(x), OPS_k(x) \\ & & +2OPS_{k-1}(x)\} \end{array}$$

The result now follows from:

$$T_k(x) = max\{CPG_k(x), CPS_k(x), OPG_k(x), OPS_k(x)\}.$$

$\square$

3.4 Edit Distance

Again, we start with the case $k = 1$.

Our approach is to create deletion and insertion subtrees that act in the same way as substitution trees. Consider an off-path subtree U hanging from node v on a centroid path of compressed trie T. The insertion tree U_I, attached to node v, is formed by adding a new first character $\beta \notin \Sigma$ to U, preceding the correct first character for U. Consider an attempted match with query pattern q that seeks to follow the path into U_I. It must take a substitution on character β, which adds one to the number of mismatches. While this mismatch can now be processed as a substitution, it is in fact an insertion in the text.

The deletion tree U_D is formed by replacing the first character of U with β. U_D is attached to the location v_{up} one character above v, if need be by creating a new node at that location. The effect is to replace the deletion of the first character of U with a substitution with respect to the previous character in q. However, we need to ensure that this substitution is allowed only if q matches the character labelling edge (v_{up}, v). To this end, U_D will be included in the group trees for the centroid path including v, but not in the group trees for the trees hanging from node v_{up}.

There is one significant change to note regarding the search. At the location ℓ at which query pattern q mismatches in the LCP search, all three possible edits are explored along the centroid path containing ℓ; i.e. a deletion or substitution of the next character after ℓ, and an insertion immediately following ℓ.

LEMMA 12. *For $k \leq \log x$, the above k-errata structure uses space $O(5^k x(k+1+\log x)(k+2+\log x)\cdots(2k+\log x)/k!)$.*

PROOF. Omitted.

LEMMA 13. *The search time in the above k-errata data structure is $O(6^k \log\log n(k+\log x)(k+1+\log x)(k+2+\log x)\cdots(2k-1+\log x)/k! + 3^k \cdot occ)$.*

PROOF. As in previous proofs, we identify the worst sequence of k-errata problems on which to recurse. We start with a weight $2w$ problem for the combined insert and substitute trees for path C. The next problem, a non-middle group tree for path C, has weight at most $\frac{3}{2}w$. Its middle group, of weight at most w, is the next recursive problem. (This corresponds to the merged insert and substitute trees hanging from a node v.) the next problem is of weight almost w and corresponds to the merged insert and substitute trees for one centroid path C' hanging from v. This generates 6 recursive problems on $(k-1)$-errata trees each of weight at most $\frac{3}{2}w < 2w$, giving the recurrence equation

$$\begin{array}{lcll} T_k(x) & \leq & 6T_k(x/2) + 6T_{k-1}(2x) + 1 & k \geq 1, \quad x > 1 \\ T_k(1) & \leq & \log\log n + 3T_{k-1}(1) & k \geq 1 \\ T_0(x) & = & \log\log n & \end{array}$$

where $T_k(x)$ denotes the search time on a k-errata structure for x strings. The $3^k \cdot occ$ term reflects the possibility that each match could be reported up to 3^k times, once for each possible pattern of k edits. □

4. GROUP TREES CONSTRUCTION

We describe the solution for $k = 1$, first. The group trees are formed top-down. First, all the (non-group) errata trees associated with a centroid path C are merged together to form the group tree at the top of the hierarchy for C. Then the smaller group trees are formed level by level by unmerging. To form the top level group tree, for each errata tree T the following path is identified: the path T_C formed by the continuation in the errata tree of the centroid path C. It takes $O(\log\log n)$ time to identify where this path leaves the errata tree using a rooted LCP query on the errata tree, and traversing up from this point identifies the path T_C in the errata tree, in time proportional to the number of edges on this path.

The subtrees in the errata tree hanging from T_C are separated, and reattached to the centroid path C, possibly with the introduction of additional nodes on C. The top level group tree is completed by merging, at each node v on the centroid path, the errata tree subtrees now hanging from v. The resulting tree rooted at the root of C is the top level group tree for C.

Note that the trees being merged are all compressed tries for collections of suffixes of the input strings. Using LCA queries on S, the suffix tree for the input strings, the merge of two such trees T_a and T_b can be done in time $|T_a|+|T_b|$. To keep the cost adequately bounded, we perform the merges so that a tree T hanging from node v is involved in $O(\log \frac{W}{w})$ merges, where W is the total weight of the subtrees hanging from node v, and w is the weight of the errata tree $\widetilde{T}$ from which T was formed.

A merge of T_a and T_b proceeds via a straightforward algorithm. The leaves of T_a and T_b are traversed left to right with the next leaf in lexicographic order being added to the combined tree. The height of the insertion point for node v is determined by the query lca(s_u,s_v) on S, where u is the last inserted node. The rightmost path in the current new tree is traversed upward from u to find the possibly new node at which the path to v diverges. It is not hard to argue that this traversal takes $O(|T_a| + |T_b|)$ time.

The telescoping argument used to bound the total size of the group error trees also shows that over all the centroid paths the time to build the top level group tree for each path is $O(x \log x)$. It remains to form the other group trees. We describe how to form T_c and T_d from T, where T is the merge of T_c and T_d. We simply label the leaves of T associated with suffixes having leaves in T_c. Then a straightforward traversal of T yields T_c. T_d is built in the same way. As the total size of the group trees is $O(x \log x)$, this takes time $O(x \log x)$. We have shown:

LEMMA 14. *The group trees for the one-errata tree can be built in $O(x \log x)$ time.*

The generalization to larger k is immediate, yielding:

LEMMA 15. *The group trees for the k-errata tree can be built in time linear in the size of the k-errata tree.*

5. THE LCP DATA STRUCTURE

Recall that S is the suffix tree for the input strings $r_1, \ldots, r_x$. Let T be the compressed trie for a collection of suffixes of $r_1, \ldots, r_x$. We explain how to build the rooted and unrooted LCP structures for T.

We assume that the following information has already been computed for S. First, its leaves have been numbered in lexicographic order. Second, an LCA structure on S has been built, plus a measured ancestor structure. This can all be done in $O(S)$ time in any event.

For T, first the LCA and measured ancestor structures are built in $O(T)$ time. Then its leaves are put in an array in lexicographic order, recording, for each leaf, its index (or numbering) in S. Finally, a Y-fast trie [39] is built over this array, in time $O(T \log n)$. (See [23] for how to do the hashing deterministically.) This will be used to support $O(\log\log n)$ time searches over the leaf indices. In fact, by building a Y-fast trie over every $\log n$-th item and binary search trees over the intermediate items, one can achieve a preprocessing of $O(T)$ time, while maintaining the $O(\log\log n)$ query time.

To build the unrooted LCP structure, a centroid path decomposition is computed, and for each subtree hanging from a centroid path the rooted LCP structure is formed.

When a pattern p is received, for each suffix p' of p, its longest prefix in common with any of the suffixes of the input strings, called the longest common prefix of p' and S, is found as follows. p is processed as if S were being augmented with the suffixes of p. Let $\tilde{r}$ be the location at which p' branches away from S. $\tilde{r}$ immediately yields the longest common prefix of p' and S.

We also determine the two suffixes in S straddling p' in lexicographic order. At least one of these suffixes has prefix equal to the longest common prefix of p' and S; this suffix, breaking ties arbitrarily, is called the highest overlap string in S for p'. This computation is readily done if we have previously recorded, with each node in S, its leftmost and rightmost descendant leaves.

Suppose the highest overlap string for p' has index g; then p' is given index $g + \frac{1}{2}$ (all indices can be doubled to keep them integral, if desired). Clearly, the preprocessing of p takes time $O(p)$.

Thus the cost of building the LCP for T is given by:

Lemma 16. *The rooted LCP can be built in time $O(T)$ and the unrooted LCP in time $O(T \log T)$.*

A rooted LCP query (p', T) proceeds as follows.

1. Find the predecessor sp and successor su of p' in T. This can be done in time $O(\log\log n)$ by searching for the index of p' in the Y-fast trie for T. This takes time $O(\log\log n)$.

2. Compute $v = lca(\ell_{sp}, \ell_{su})$ in T, where ℓ_x is the leaf in T corresponding to suffix x. Also report the edges exiting v on the paths to ℓ_{sp} and ℓ_{su}. This takes $O(1)$ time using the LCA data structure for T.
   Step 2 is useful because the sought location r' is one of: v, a location on the path from v to ℓ_{sp}, or a location on the path from v to ℓ_{su}.

3. This step finds the length h of the longest common prefix of p' and T.

   Let ssp' denote the highest overlap string in S for p', and suppose this overlap has length h'. Let h_p denote the length of the longest common prefix of ssp' and sp, and h_u the length of the longest common prefix of ssp' and su. Then $h = \min\{h', \max\{h_p, h_u\}\}$. h' could be precomputed as part of the preprocessing of p, and h_p, h_u can be found using, respectively, the LCA queries $\text{lca}(ssp', sp)$ and $\text{lca}(ssp', du)$ on S.

4. Find the location in T at distance $|sp| - h$ above leaf ℓ_{sp}. This can be done in time $O(\log\log n)$ thanks to the preprocessing of T.

An unrooted LCP query (p', T, r) proceeds in two steps.

1. Consider the centroid path C of T going through r. Determine h, the length of the longest common prefix of p' and the portion σ of C starting at r. Indexing into C yields location ρ on C at distance h from r.

   To this end, determine the length h_p of the longest common prefix of σ and sp', where sp' is the highest overlap string in T for p'. This can be found by means of a query $\text{lca}(\sigma, sp')$ on S. As before, we let h' denote the length of the longest common prefix of sp' and p'. Then $h = \min\{h', h_p\}$.

2. If ρ is a node there may be one off-path subtree of C at ρ in which the search for p can continue (the subtree whose first character matches the next character of p'). The search on this subtree is performed by a rooted LCP query.

We have shown:

Lemma 17. *An LCP query takes time $O(\log\log n)$.*

Remark. For the dictionary problems, in the suffix tree S, only the suffixes that occur in an LCP structure need be numbered. There are $O(c^k x(k + \log x)^k)$ such strings where $c > 1$ is a constant, and consequently the term $\log\log n$ in the complexity (based on n consecutively numbered strings) can be reduced to $O(\log\log x)$ (at least for k =polylog(x)).

6. OPEN QUESTION

Can the 3^k term multiplying the # matches term in the query time for the edit problem be reduced, ideally to $O(1)$?

7. ACKNOWLEDGMENTS

We thank a referee for multiple helpful comments, and in particular for asking whether an all-to-all problem could be solved.

8. REFERENCES

[1] K. Abrahamson. Generalized string matching. *SIAM J. Computing*, 16(6):1039–1051, 1987.

[2] A.V. Aho and M.J. Corasick. Efficient string matching. *Comm. ACM*, 18(6):333-340, 1975.

[3] T. Akutsu. Approximate string matching with don't care characters. *Information Processing Letters*, 55:235-239, 1995.

[4] T. Akutsu. Approximate string matching with variable length don't care characters. *IEICE Trans. Information and Systems*, E79-D:1353-1354, 1996.

[5] A. Amir and M. Farach. Adaptive dictionary matching. *Proc. of the Symposium on Foundations of Computer Science*, 1991, 760-766.

[6] A. Amir, M. Farach, R. Giancarlo, Z. Galil and K. Park. Dynamic dictionary matching. *J. of Computer and System Sciences*, 49(2):208-222, 1994.

[7] A. Amir, M. Farach, R.M. Idury, J.A. La Poutré, and A.A Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995.

[8] A. Amir, D. Keselman, G.M. Landau, N. Lewenstein, M. Lewenstein, and M. Rodeh. Text Indexing and dictionary matching with one error. *J. of Algorithms*, 37(2): 309-325, 2000.

[9] A. Amir, G. Landau, M. Lewenstein and D. Sokol. Dynamic pattern, static text matching. Submitted for journal publication; preliminary version, *Proc. of Workshop on Algorithms and Data Structures*, 2003, 340-352.

[10] A. Amir, M. Lewenstein and E. Porat. Faster algorithms for string matching with k mismatches. *Proc. of the Symposium on Discrete Algorithms*, 2000, 794-803.

[11] A. Amir, M. Lewenstein and E. Porat. Approximate subset matching with don't cares. *Proc. of the Symposium on Discrete Algorithms*, 2001, 279-288.

[12] G. S. Brodal and L. Gasieniec. Approximate dictionary queries. *Proc. of the Symposium on Combinatorial Pattern Matching*, 1996, 65-74. (LNCS 1075).

[13] G. S. Brodal and S. Venkatesh. Improved bounds for dictionary lookup with one error. *Information Processing Letters*, 75(1-2):57-59 (2000).

[14] A.L. Buchsbaum, M.T. Goodrich, J. Westbrook. Range searching over tree cross products. *Proc. of the European Symposium on Algorithms*, 2000, 120-131.

[15] R. Cole and R. Hariharan. Approximate string matching: A faster simpler algorithm. *Proc. of the Symposium on Discrete Algorithms*, 1998, 463-472.

[16] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. *Proc. of the Symposium on Theory of Computing*, 2002, 592-601.

[17] M. Farach. Optimal suffix tree construction with large alphabets. *Proc. of the Symposium on Foundations of Computer Science*, 1997, 137–143.

[18] P. Ferragina, S. Muthukrishnan, and M. deBerg. Multi-method dispatching: a geometric approach with applications to string matching. *Proc. of the Symposium on the Theory of Computing*, 1999, 483-491.

[19] M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.

[20] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4), 52-54, 1986.

[21] D. Greene, M. Parnas and F. Yao. Multi-index hashing for information retrieval. *Proc. of the Symposium on the Foundations of Computer Science*, 1994, 722-731.

[22] R. Grossi, J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *Proc. of the Symposium on Theory of Computing,* 2000, 397-406.

[23] T. Hagerup, P. B. Miltersen, R. Pagh. Deterministic dictionaries. *J. of Algorithms*, 41(1):69-85, 2001.

[24] D. Harel, R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338-55, 1984.

[25] R.M. Idury and A.A Schäffer. Dynamic dictionary matching with failure functions. *Proc. of the 3rd Annual Symposium on Combinatorial Pattern Matching*, 1992, 273-284.

[26] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. *Proc. of the Symposium on Theory of Computing*, 1998, 604–613.

[27] E. Kushilevitz, R. Ostrovsky, Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *SIAM Journal on Computing*, 30(2):457–474, 2000.

[28] G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43, 239-249, 1986.

[29] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. of Algorithms*, 10(2):157-169, 1989.

[30] U. Manber and S. Wu. An algorithm for approximate membership checking with applications to password security. *Information Processing Letters,* 50:92-197, 1994.

[31] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23:262-272, 1976.

[32] K. Mehlhorn. *Data Structures and Efficient Algorithms, Vol. 1: Sorting and Searching, pp. 177-178.* Springer Verlag, EATCS Monographs, 1984.

[33] M. Minsky and S. Papert. *Perceptrons.* MIT Press, Cambridge, Mass., 1969.

[34] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. *Proc. of the Symposium on Foundations of Computer Science*, 1996, 320-328.

[35] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplifications and parallelization. *SIAM Journal on Computing*, 17(6):1253-62, 1988.

[36] S. Shekhar, S. Chawla, S. Ravada, A. Fetterer, X. Liu, C.T. Lu. Spatial databases - accomplishments and research needs. *TKDE*, 11(1):45-55 (1999).

[37] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[38] P. Weiner. Linear pattern matching algorithm. *Proc. of the Symposium on Switching and Automata Theory*, 1973, 1-11.

[39] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(n)$. *Information Processing Letters*, 17(2):81-84, 1983.

[40] A. C.-C. Yao and F. F. Yao. Dictionary lookup with one error. *J. of Algorithms*, 25(1):194–202, 1997.