A Dictionary for Approximate String Search and Longest Prefix Search

Sreenivas Gollapudi
Ebrary, Inc.
Sreenivas.Gollapudi@ebrary.com

Rina Panigraphy
Stanford University
rinap@cs.stanford.edu

ABSTRACT

In this paper we propose a dictionary data structure for string search with errors where the query string may differ from the expected matching string by a few edits. This data structure can also be used to find the database string with the longest common prefix with few errors. Specifically, with a database of n random strings, each of length of $O(m)$, we show how to perform string search on a query string that differs from its closest match by k edits using a data structure of linear size and query time equal to $\tilde{O}(\log n^{2^{\log n \lfloor \frac{k \log_a 2m}{2m} \rfloor}})$. This means that if $k < \frac{2m}{\log_a 2m}$, then the query time is O(1). This is of significant in practice as there are several applications where k is small relative to m. A simple reduction can be used to obtain similar results for approximate longest prefix search.

1. INTRODUCTION

String matching allowing errors, which is also known as approximate string matching is an integral part of an information retrieval system whose design is based on efficient access of specific information from large collections of data. There is a plethora of work in the area of efficient processing of online queries. The size of today's data collections makes exact string matching very expensive. Approximate string matching allows for skips while "scanning" the database for pattern matches. Another area where approximate string matching can be applied is where the database itself has erroneous data. On of the popular applications for approximate string matching is spell-checking.

In this work, we apply approximate string matching to the following queries -

- `Dictionary queries` - Given a database $\mathcal{D}$ of n terms $t_1, t_2, \ldots, t_n$, find out if a given query q appears in the database.
- `Document queries` - Given a large text $\mathcal{T}$ of terms, find out the offsets of a given phrase or word in the text.
- `Longest common prefix` - Given a dictionary of $\mathcal{D}$ of n terms $t_1, t_2, \ldots, t_n$, find out all terms in the database that are valid prefixes of the given query.

CIKM '06, November 6–11, Arlington, Virginia USA

In all the above queries, we allow for a specified number of errors while computing the matches. The error model, in general, depends on the application. The basic idea is to represent the error as a *distance* between the two strings – the query and the possible match – and return the closest match to the query that minimizes this distance. The *edit distance* is a common metric used to discover a close variant of the input string. For an excellent exposition of approximate string matching, the reader is referred to [15] and references within.

DEFINITION 1. *The* **edit distance** *between two strings S_1 and S_2 is the minimum number of edit operations required to transform S_1 to S_2. The edit operations include insertions, deletions, transpositions, and substitutions.*

Edit distance algorithms can be specialized depending on the application. An interesting special case is the *hamming distance* [12].

DEFINITION 2. *A special case of the edit distance, where the strings are of equal length and the edit operations are limited to substitutions, is the* **hamming distance**. *The hamming distance is also a metric that satisfies triangle inequality.*

The basis for all problems involving string comparisons is the classical pattern matching problem wherein one searches for a pattern p in a given text t. The above defined metrics form the cornerstone of many approximate pattern matching problems.

Application areas - Some of the application areas of approximate string matching are computational biology, signal processing, information retrieval, and pattern recognition.

In computational biology, DNA sequences can be seen large texts over an alphabet of four characters (A, T, G, C). Looking up matches to a query sequence is an important aspect of genomics. For example, an exact match could give information about the position of the gene sequence in the genome. An approximate match could give other information such as number of mutations (edits) in the gene sequence. Studying differences between two DNA sequences forms the basis of many studies in evolutionary biology.

In information retrieval, spell checking is an old but important application of approximate string matching. Looking up substrings in a large text or collection of documents often use approximate string matching to obtain the offsets of the substring or a close match in the large text. Another important application is optical character recognition

wherein the digitized texts often have 7−16% errors. Searching with errors, either in the query string or the database string, uses approximate string matching.

Other areas such as security employ approximate string matching in password checkers wherein the passwords are checked against a list of "low-security" password strings that are deemed vulnerable to cracking by malicious users.

1.1 Related Work

In this section, we present the state of the art in approximate string matching. We broadly classify existing literature on approximate string matching into three categories.

Sketch based schemes - Several studies suggest methods to map strings to bit-vectors in hamming space [2, 16]. The essential idea being that given such an embedding, techniques for approximate search in hamming space can be carried over to strings. The measure of the goodness of an embedding is usually referred to as *distortion*. This is also known to be the error (as a multiplicative factor) introduced in distances while transforming from one metric space to another. Broder [5] suggests a simple approach based on string shingling to convert strings into sets. The idea is to view a document as a set of all its substrings of a certain length. They argue that similar documents are likely to map to similar sets. Bar-Yossef et al [2] extend this technique to produce bit-vectors with theoretical guarantees: they achieve a distortion of $O(d^{3/7})$. Recently, this was improved by Ostrovsky et al [16] to $O(2^{\sqrt{\log d \log\log d}})$ using methods based on recursive shingling. Andoni and Indyk [1] use locality sensitive hashing (LSH) to construct a data structure of size $\tilde{O}(n^{1+1/c})$, where n is the length of the text and c is a distortion factor and $\tilde{O}(n^{1/c} + n^{1+o(1)}M^{1/3})$ where M is the upper bound for the query length.

Trie based schemes - There is a plethora of work that is uses tries (suffix trees) to store the text. The search algorithm is adapted to deal with possible errors using dynamic programming [19, 17]. The main advantage of tries is the search time is independent of the size of the text. Thus, they find many uses in searching large collections. The worst-case query time for a trie is $O(kn)$ while the average case is $O(m)$. The space complexity is linear. On the same token, one of the issues with trie based techniques is that the performance is fundamentally tied to query length. Therefore large queries tend to have a poorer performance compared to sketch based techniques. Cole et al [7] provide theoretical guarantees of $O(\frac{(c_2 \log n)^k \log\log n}{k!} + m + r)$ query time and $O(n\frac{(c_1 \log n)^k}{k!})$ space, where c_1 and c_2 are constants > 1.

Bloom Filter based schemes - Another data structure that has captured attention of research is the Bloom filter [3]. It has been widely used in network applications [4, 8, 18]. It has been used string processing [14, 13]. All of these studies proposed the Bloom filter as a space-efficient data structure to answer membership queries. The data structure offers a succinct representation of a set and allows for membership queries with very small false positives and *no* false negatives, i.e., the structure will never return true when querying for a member that does not exist in the set. Kirscsh and Mitzenmacher [10] propose a generalization of the Bloom filter to answer queries of the form "Is x close to an element of the filter?" where closeness or similarity is defined under the hamming metric. Hamming distance does not recognize similarity of strings that are produced by a few insertions or deletions from a given string.

1.2 Contribution of this study

Our result is a data structure that uses linear space and provably $\tilde{O}(\log n^{2\log n\lfloor \frac{k\log_a 2m}{2m}\rfloor})$ time for a database of random strings. We define the notion of random strings in the next section. We demonstrate the efficiency of our data structure by experiments on a large set of strings.

2. PRELIMINARIES

2.1 k-Approximate String Match

The approximate version of the string matching problem involves allowing upto k errors when computing the nearest match to a given query. We now formally define this problem. Let Σ denote a finite alphabet over which the strings in the database are defined. For clarity, let $a = |\Sigma|$. For any string $s \in \Sigma^*$, we denote the length of the string by $|s|$ and s_i as the i-th character of s, for $i \in [1, |s|]$. A substring of s from character i to character j is denoted by $s_{i..j}$. Given a database text $T \in \Sigma^*$ whose length is n and a query string $Q \in \Sigma^*$ whose length is m, we define the k-approximate string match as:

DEFINITION 3. *Given T, Q, and an error parameter k, return all close variants of Q in T which satisfy $d(Q, T_{i..j}) \leq k$ for $1 \leq i \leq n$ and $j \geq i$, where $d(.,.)$ is the distance function. In this study, we use the edit distance as a measure of string similarity.*

For k-approximate string matching, allowing for some imprecision in the results resulted in the development of efficient algorithms [9, 11]. The imprecision arises from the ϵ parameter in these algorithms which allows for some matches that are within $k(1+\epsilon)$ distance from the query. In this study, we encapsulate approximation by the error parameter k.

In our definition of this problem, we have a dictionary D of d terms, a query Q asks for all terms within a hamming distance k of Q. Thus, for large texts where $n >> m$, T can be decomposed into a set of d strings and stored in a dictionary which can then be queried. If n is the total length of all terms in the dictionary, an earlier result uses space $O(n + d\frac{(c_1 \log x)^k}{k!})$, can be built in time $O(n + d\frac{(c_1 \log d)^k}{k!} + n\log n)$, and has a query time complexity of $O(m + d\frac{(c_2 \log x)^k}{k!}\log\log n + r)$ [7], where constants $c_1, c_2 > 1$.

A different formulation of approximate string matching problem, that often appears in literature, deals with large static texts such as books wherein the goal is to find all locations $\mathcal{I}$ in T such that $d(Q, T_{i..i+m}) \leq k$ for all $i \in \mathcal{I}$. In such cases the entire text can be pre-processed to build an index to support efficient dictionary queries. In fact, using the same process described above, this problem can be reduced to the approximate matching problem by breaking up the text into smaller length shingles and treating each shingle as a term in the dictionary. Known efficient methods preprocess the text in $O(n\log n)$ time and answer queries in time $O(m\log\log n + r)$ time. Another data structure that employs the hamming distance metric uses space $O(n\frac{(c_1 \log n)^k}{k!})$

and takes time $O(n\frac{(c_1 \log n)^k}{k!})$ to build. The query time complexity is $O(\frac{(c_2 \log n)^k \log\log n}{k!} + m + r)$ [7], where constants $c_1, c_2 > 1$ and the number of errors $k \leq \log n$.

In this study, we define a data structure that can be used to solve both the above problems. We describe our data structure in Section 3.

2.2 k-Approximate Longest Prefix Match

The k-approximate longest prefix match is an approximation to the longest common prefix problem. Here, we allow upto k errors when matching the longest prefix common to both database term(s) and the query. Under the same conditions defined in Section 2.1. we define k-approximate longest prefix match as -

DEFINITION 4. *Given T, Q, and an error parameter k, return longest common prefix of Q and T which satisfy the relation* $\arg\max_i d(Q_{1..1.1^i}, T_{1..1.1^i}) \leq k$, *where $d(.,.)$ is the distance function. Again, we use the edit distance as the similarity measure between two strings.*

Basically, we compute the approximate match of all valid prefixes of the input query with the prefixes of the database string. The valid prefixes are pre-computed and stored in different hash tables, one table for each prefix length. To reduce the overhead of storing prefixes of all possible lengths, we store only prefixes of lengths that are powers of 1.1. The approximate longest common prefix is the largest index j such that $d(Q_{1..1.1^j}, T_{1..1.1^j}) \leq k$. The choice of the power is dependent on the allowable error in the closest match computation. If the error is small, the power is small. For example, with a value of 1.1, the algorithm truncates no more than 10% of the characters in a string.

3. THE DICTIONARY DATA STRUCTURE

In this section, we define the data structure for the k-approximate string match. We will show how this data structure can be extended to solve a natural relaxed version of the k-approximate longest prefix match problem.

First, if the query is of length d, we can assume that all strings are of length $d \pm k$. Any solution to such a problem can be easily extended to database strings of arbitrary length by simply partitioning strings by rounding their lengths to a nearest multiple of $2k$ and searching in at most two partitions. We will detail this later in this section. First, we present the data structure for the specific case of k-approximate string match for equal length database strings and queries.

3.1 Special case of k-approximate string match

To simplify the analysis, we consider all strings to be approximately of the same length. Later we show that it is simple to extend the analysis to strings of varying lengths. Let all database strings and the query string to be of length $d \pm k$. The key step in this algorithm is to map database strings to bit-vectors of length approximately $2d$. The mapping uses the well-known embedding of edit distance metric to the hamming metric [5, 11]. The mapping defined by Broder in [5] involves a simple computation using document shingles that is easy to implement in practice. Bar Yossef et. al. [2] showed how such a shingling method can be used to achieve an embedding to hamming space with distortion at most $O(d^{3/7})$. Ostrovsky and Rabani [16], on the other hand, give a lower distortion of $2^{O(\sqrt{\log n \log\log n})}$ but their algorithms are complex.

Mapping from strings to bit vectors - The mapping from strings of length approximately d is composed of two steps. The first step in the mapping scheme involves reducing the string to a set of canonical sequence of tokens. Each token is a subsequence of length w of the input string. The length of a token or shingle is $\approx \log_a d$ where a is the length of the alphabet. This length is chosen such that $a^w \geq 2d$, the length of the sketch. The second step involves mapping the shingle set to bit-vector of size αd, where $\alpha \approx 2$. The length of the bit-vector is derived from standard Bloom filter technique [3, 6]. It is obtained by computing the optimal case wherein the number of *false positives* are minimized with respect to the number of hash functions, n_h, in the filter. For a more detailed description of Bloom filters, the reader is referred to [4]. The number of bits to store d keys in a Bloom filter with n_h hash functions is $\frac{n_h d}{\ln 2}$. In this case, the number of hash functions is 1. Thus $\alpha = \frac{d}{\ln 2}$.

LEMMA 1. *Assuming a string s has y distinct shingles, for a bit-vector of length $x = \frac{y}{\ln 2}$, the* map(s) *takes a random value $\in \{0,1\}^x$.*

PROOF. Assuming there are y distinct shingles, we set the size of the bit-vector x n such a way that the bit-vector takes a random value $\in \{0,1\}^x$. For this the probability of turning on a bit equal to $(1 - \frac{1}{x})^y$ must be 1/2 giving $x = \frac{y}{\ln 2}$. □

Note that for a random string, with high probability, the number of distinct shingles is concentrated close to $d - w$ for $w > \log_a d + \Omega(1)$.

Finally, we sample $b \approx \log n$ bits from the bit-vector to generate the mapping of the given string to hamming vectors of size $\approx \log n$ bits. We define map(S) as the sketch of string S resulting from the mapping scheme.

LEMMA 2. *For two strings S_1, S_2, $E[H(map(S_1), map(S_2))] \leq \frac{ED(S_1,S_2)w}{d} b$, where $H(.,.)$ is the hamming distance and $ED(.,.)$ is the edit distance between the two given strings.*

PROOF. Let $e = ED(S_1, S_2)$. Each edit results in w shingles being different. So the sets differ in at most ew elements. Since the mapping from sets to bit vectors preserves the set differences in expectation, we have the proof. □

Clearly, a lower value of w is better. Too small a value of w will restrict the set size to the size of the alphabet making the bit vector sparse. Our objective is to ensure that a random string of length d results in a random bit vector of length $\frac{d}{\ln 2}$. To this end, w needs to be set in such a way where all the shingles are nearly distinct for a random string. For an alphabet of size a, setting $w = \log_a d + c$ will ensure this.

Data structure - Lemma 2 implies the following data structure. Map all strings to bit-vectors of length b. Construct a table storing each bit vector along with its original string. To search a given query Q, we compute its map, map(Q), and search it in the table. Although, it is not necessary that map(Q) coincides with the map of its nearest neighbor, lemma 2 implies that they differ in bits $\frac{kwb}{d}$ in expectation. So, if the entry corresponding to map(Q) does not contain a nearby string, we search in all buckets close

to map(Q), i.e., all bit-vectors that differ from map(Q) in at most $\frac{kwb}{d}$ bits.

Instead of storing the entire string in the table, we can store again a B-bit vector sketch of the string. The sketch is computed using the same mapping scheme. Instead of sampling b bits, we sample $b+B$ bits. We then use b bits as a key to lookup the table and the B bits for comparing the strings. Let us note that having a sketch of size B produces a collision probability of $\frac{1}{2^B}$. Thus, choosing $B = 8$ sets the probability of collision to 1/256. Note that we can also compute the length of B as a function of the number of strings, n, in each range, say, $\log n$. In this study, we set the value of B to a constant. A comparison between the query and a potential match is set to be close if they differ in at most $\frac{kwc}{d}$ bits as implied by lemma 2. The space complexity of the data structure is clearly linear.

Query time complexity - Since we search in all entries that differ from map(Q) in at most $\frac{kwb}{d}$ bits, the number of entries searched is $\sum_{j \leq i} \binom{b}{j}$ where $1 \leq i < \frac{kwb}{d}$. The sum is less than $b^{\frac{kwb}{d}}$. In particular, if $kwb < d$, the query time is $\tilde{O}(1)$.

3.2 High Probability Bounds

For a tighter and more precise analysis, we assume the strings are random. The above discussion observed that $\frac{kwb}{d}$ is expected number of bits on which map(Q) and the map of its nearest neighbor differed. It is possible that tha actual difference is greater than this. But we show that the probability of it being much greater is small. Formally, the probability that it differs by more than twice the above difference is at most 1/2 by Markov's inequality. So, by using $O(\log n)$ independent instances of the data structure, the algorithm can be made to succeed with high probability.

Also, in the query time complexity computation, we assumed that the number of entries in a bucket is constant. Again, this may not be true with a small probability. Since each string (assuming the $d-w$ shingles are nearly distinct) maps to a random bit-vector, the expected number of entries per bucket is less than 1. Again, by Markov's inequality, with probability 1/2, the bucket containing the nearest neighbor has at most two entries. So, if we limit our search to only two entries per buckets, then again, with $O(\log n)$ instances of the data structure, we succeed with high probability. We summarize with the following theorem.

THEOREM 1. *Assuming strings are random, that is the $d-w$ shingles are near distinct, our k-approximate string match can be implemented using a near linear size data structure and query time at most $O(\log n^{2 \log n \lfloor \frac{k \log_a 2m}{2m} \rfloor})$ with high probability.*

4. ALGORITHMS FOR APPROXIMATE STRING MATCHING

We now summarize the algorithms for approximate string matching. The input to the algorithm is a set of strings, $\mathcal{S}$, extracted from a large text or collections of documents, the size of a shingle, w, in a string, and the size of sketch, B, of each string. The strings are assumed to be fairly random and non-repetitive, that is, no string generates a k-gram more than once. Informally, our algorithm computes a sketch for each string and uses the sketch as a key in a hash table. The length of the sketch generated depends on the length of the string. Specifically, we group strings in intervals of size $2k$ where k is the maximum allowable error in finding closest matches. We use index(s) to denote this grouping operation, where $s \in \mathcal{S}$. In this study, we use the hamming distance to represent the error between a potential match in the hash table and the query string. During querying, we similarly compute the sketch of a query and use the sketch to lookup the appropriate hash table for a match. The hash table to search is indexed by i such that the length of the query is in the range $[2ki, 2k(i+1))$. The number of ranges is dependent on the maximum length of a string in the database. Without loss of generality, let us assume the number of ranges to be N_b. Each range is represented by a hash table, $\mathcal{H}_i$, that is indexed by keys of length $\log N_i + c$, where N_i are the number of strings in the range. During the pre-processing step, we partition the database strings into appropriate ranges and keep a count of strings in each range. We store the counts in an array count.

We now present the algorithms for constructing the dictionary data structure.

Algorithm 1 APPROXSTRINGINSERT($\mathcal{S}, w, B$)

```
1: for s ∈ S do
2:    count[index(s)] + +
3: end for
4: for i = 1 to N_b do
5:    N_i ← count[i]
6: end for
7: for all s ∈ S do
8:    d ← |s|
9:    i ← index(s)
10:   produce a shingle set S of size d − w
11:   bitvector(s) ← hash(s) {2d number of bits}
12:   key(s) ← sample b (= log N_i + c) bits
13:   value(s) ← sample B bits
14:   add entry ⟨key(s), value(s)⟩ in H_i
15: end for
```

REMARK 4.1. *The query algorithm takes in a query q, the shingle width w, and the allowable error k in finding a match. If the computed key for the query does not exist in the appropriate hash table, we flip bits in the key for $\sum_i \binom{b}{i}$, $1 \leq i \leq \frac{kwb}{d}$, times. That is, we want to lookup all keys that differ from the query key in at most i bits. One way to do this is to try all the $x = \sum_{j \leq i} \binom{b}{j}$ tries. A simpler implementation is to try random i-bit flip where each bit is flipped with probability 1/2 x times. A coupon collection argument would imply that doing such random flips $x \log x$ times covers all combinations with high probability. In reality, with kx trials, the probability of missing it is at most e^{-k}.*

5. APPROXIMATE LONGEST PREFIX MATCH

We use the data structure for approximate string matching, described in the previous section, to compute a k-approximate longest prefix match between a given string and the database strings. Instead of storing all possible prefixes for a given a string, we keep a sketch of a string's prefixes by truncating a prefix to its nearest power of 1.1. Thus, a prefix of length $2d$ is rounded to length $1.1^{\lfloor \log_{1.1} 2d \rfloor}$ by truncating

Algorithm 2 ApproxStringQuery(q, k, w)

```
1: d ← |q|
2: i ← index(q)
3: produce a shingle set S of size d − w
4: bitvector(q) ← hash(q)
5: key(q) ← sample b (= log N_i + c) bits
6: value(q) ← sample B bits
7: sketch ← H_i(key(q)).
8: if H(value(q), sketch) ≤ k then
9:   return value
10: else
11:   for i = 1 to kwb/d do
12:     ntrials ← C(b,i) log C(b,i)
13:     for j = 1 to ntrials do
14:       flip upto i bits in all Σ_{j<=i} C(b,j) ways {see remark 4.1}
15:       sketch ← H_i(key(q)).
16:       if H(value(q), sketch) ≤ k then
17:         return value
18:       end if
19:       revert the flipped bits
20:     end for
21:   end for
22: end if
23: return failure
```

the extra characters. A dictionary is then created for each prefix length and all prefixes of the same length are stored in the appropriate dictionary. To find the approximate longest common prefix, we compute the prefixes of the query Q and starting with the longest prefix, we find the closest match for each prefix using the approximate string matching algorithm. Once we find a match, we stop.

Thus, memory required to compute the approximate longest common prefix is $O(n \log_{1.1} d)$. The running time is longer by a factor of $O(\log_{1.1} d)$. One of the disadvantages of this approach is that most of the k allowed edits are in the trailing characters that can be truncated if the length of database string and/or the query are not powers of 1.1. However, the number of truncated characters is never more than 10% of the prefix length. Thus, given a prefix of length $2d$ bits, the truncated length is no less than $2d - \delta$, where $\delta = 2d - 1.1^{\lfloor \log_{1.1} 2d \rfloor}$ in the number of truncated characters. Assuming random strings and uniform selection of the $\log n$ bits that form the map(.) of the given prefix, we loose no more than $k\frac{\delta}{2d}$ edits to truncation. In fact, our experiments show that the information loss is minimal in reality.

6. EXPERIMENTS

In this section, we illustrate the efficiency of the proposed dictionary data structure in approximate string matching. We test the algorithms with different values of parameters – key size variable c, size of string sketch B, error k, length of queries d, number of strings in the database n – and their effect on query times, accuracy of a match, number of hash table lookups, and memory.

6.1 Implementation details

We set the maximum length of a string to be indexed in the dictionary to be 128 characters. This limits the number of index ranges to 20. We assume all strings are at least of length 6 characters. Furthermore, we group all strings 10 characters and less into one index range. This range can be handled separately by small string approximation algorithms such as those based on dynamic programming. For the rest of this section, we do not deal with small strings. The shingle size is set to 3. The size of a bit-vector to map $d-3$ shingles of a string of length d is set to $\frac{6i+10}{\ln 2}$, where i is the index number. The hash(s) function is a simple function for hashing strings based on Horner's rule.

In the querying phase, to make sure we have enough number of bits to flip in case of not finding a search key in the hash table, we set the number of flips to $\frac{1.5kwb}{d} + 1$. The number of trials for each set of flips is set to $2\binom{b}{j}$ where j is the number of bits to flip. We note that at no time we had to flip more than three bits to find an approximate match. The strings to be inserted in the dictionary were randomly generated from a set of 115000 unique words by concatenating a randomly choosing a specified number of words. We set this number to 5. The number of strings was also parametrized and set to a million for our experiments. Setting the number of words to 5 generated strings of lengths varying from 17 to 74 with a mean length of 42.

Note that using small number of lookups (~ 2), the probability of finding a nearest neighbor in a bucket is 1/2. In order to increase the success probability to say 90%, one way would be to increase c and hence the number of lookups into the table. The expected number of lookups is dependent on the length of the key b as $\sum_i \binom{b}{i}$. This implies a large number of lookups (exponential in c). Another approach would be to work on multiple copies of the data structure, upto $\log n$ copies in fact. However, in reality we could do with lot lesser number of copies. For a 90% success probability – $(1 - 0.5^4)$ – we would need 4 copies of the data structure.

6.2 Results

In the first set of experiments we analyze the time and space requirements of our data structure. We compare the memory taken by our data structure to a compressed trie. We insert a unique list of 1000000 strings with an average length of 42 characters. The compressed Trie took about 272M of memory to store all the phrases while the dictionary took 199M - about 36.7% reduction. This memory size was obtained with $c = 3$ bits and $B = 20$ bits. Figure 1 shows the running times for querying the dictionary for different values of the error parameter. Clearly, the running time increases with the number of errors in the query string. As Figure 1 shows, the algorithm performs very well in terms of running time as it takes only 0.22 ms to find an approximate match for a query averaging 42 characters in length. The response times shown in Figure 1 were averaged over 10000 queries on a dictionary containing 1000000 strings.

Analogous to the running time is the number of lookups a query takes to return an approximate match. Our simulation results show that the number of lookups for approximate string search is small. For a database of 1000000 strings of length about 42 characters each, successful queries with at most 4 errors can be performed in under 15 memory lookups. At the same time the success rate of finding a "correct" approximate match for the input query with at most 4 errors is 48.2%. This success rate can be increased significantly by using multiple copies of the data structure and running the query independently on each of the copies. For example, by running the query on 4 copies of the data

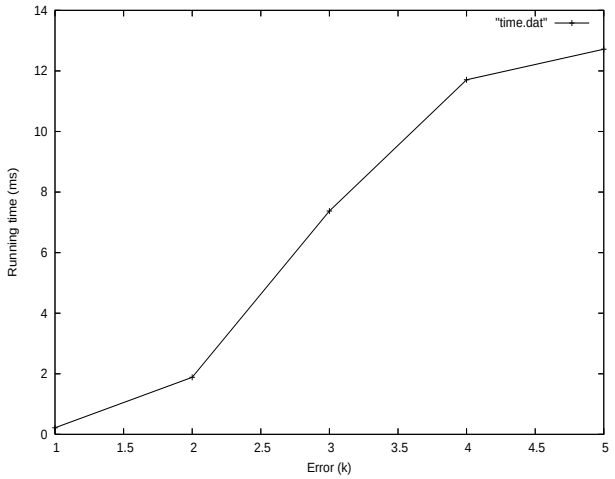

Figure 1: Running time vs Error, k

structure, we saw the success rate increase to 86.0%. In the following analysis, we highlight our observations in more detail.

An important metric to measure is the number of lookups as a function of the error. We vary the error from 1 to 5 for the query strings for different values of B. We set $c = 3$ for this run. Figure 2 shows that the number of lookups increases with the error for $B = 30$ and 40. This is because the number of flips in the query algorithm is increasing. Again, the number of strings in the database is 1000000. Furthermore, the number of trials are proportional to the parameter B. Figure 2 illustrates this behavior. The parameter B was varied from 30 to 40. In general, for a fixes error parameter k, the number of lookups increase as we increase the parameter B. In this set of experiments, we set the number of copies of the data structure to 1.

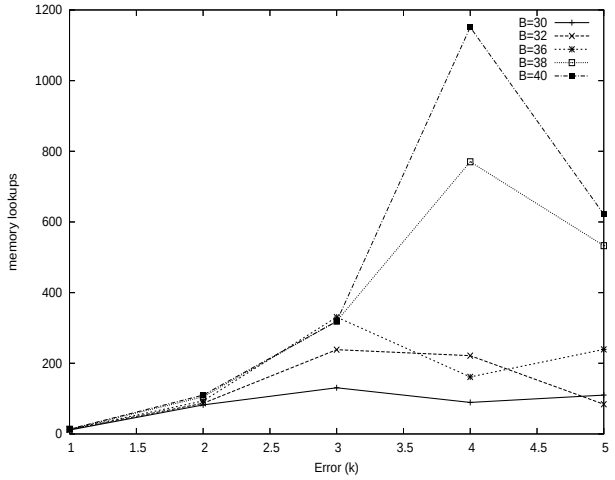

Figure 2: Lookup vs k

In another sets of experiments we illustrate the number of lookups performed as a function of the parameters c and the length of query strings d. The number of lookups was averaged over a 10000 queries. Figure 3 illustrates the fact the number of lookups is exponential in c. We fix $B = 30$ and $k = 2$. The number of copies of the data structure is

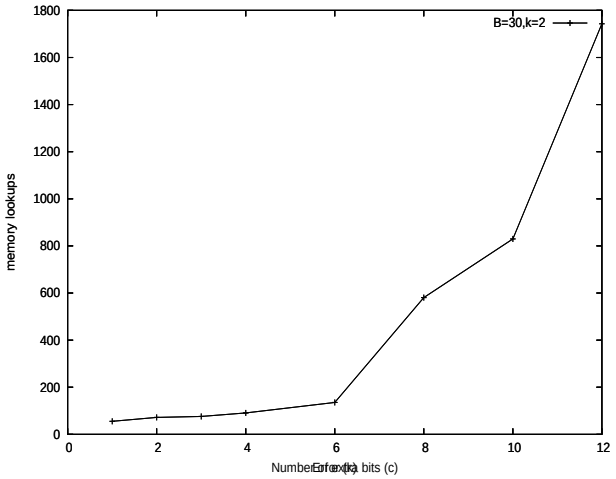

Figure 3: Lookups vs. c

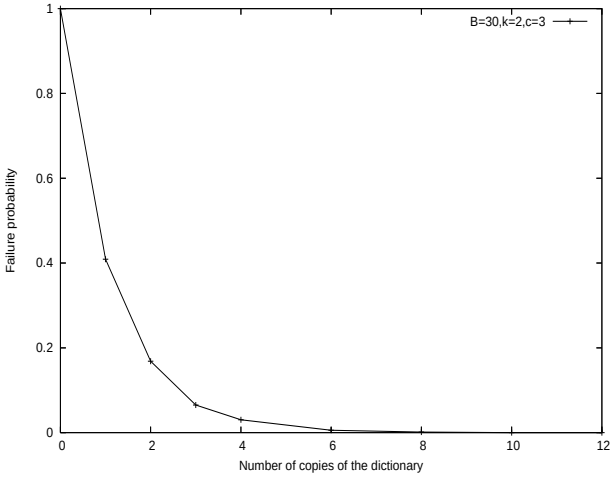

Figure 4: Failure probability vs. Number of copies

set to 1.

Clearly, as Figure 3 shows, the number of lookups increases from nearly 1 to very large ($\approx$ 1800) for $c = 12$. This motivates us to consider the alternative of using copies of the data structure and using small values of c to achieve the desired success rate.

Figure 4 shows the affect the number of copies of the data structure has on the query accuracy which is expected as the query outcome for each data structure is independent. The failure probability drops exponentially with the number of copies. We fix $B = 30$ and $c = 3$. From the figure, we can see that using just 3 copies will give us an error probability of around 10%.

The parameters c and B are very important to the performance of our data structure. Figures 5 and 6 show the failure probability as a function of these parameters. The number of copies is set to 1. The error value was changed from 1 to 6. In the first experiment $B = 30$ while in the second experiment $c = 3$. Figure 5 shows that for any fixed value of c, the failure probability decreases with the error parameter. For any fixed k, the failure probability decreases with increase in c and this is more pronounced for larger

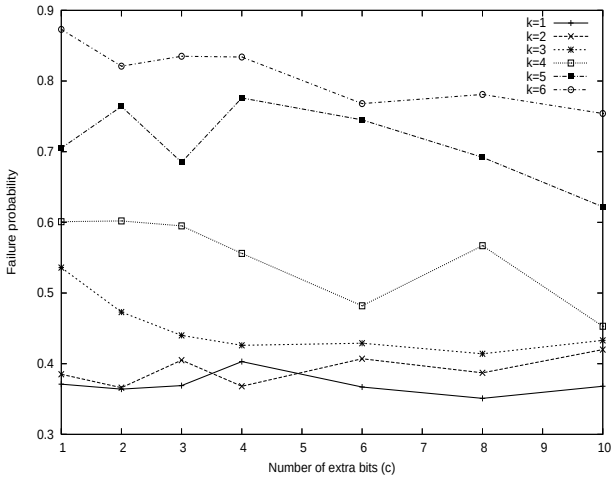

Figure 5: Failure probability vs. c

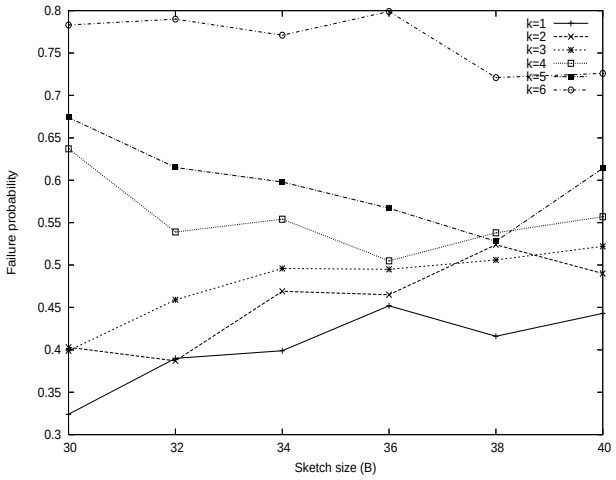

Figure 6: Failure probability vs. B

values of k. For small errors, the number of extra bits used in the sketch do not have have the same impact as they do for large errors. This has a consequence on how much the failure probability varies for different values of c.

Figure 6 shows a similar correlation between the sketch size B and the failure probability as one found between the number of extra bits and the failure probability. In fact, both B and c contribute toward the total number of bits used to represent a key in our dictionary. Hence it is not surprising we observe similar behavior w.r.t failure probability when one of the parameters is changed while keeping the other fixed.

An interesting observation suggesting the use of multiple copies of the data structure instead of larger sketch sizes is the fact that with each independent run, the number of lookups, and hence the query times, decrease significantly as most of the work seems to be done in the earlier runs. Note that we stop querying the dictionary in the following runs once we have a successful hit for a query. At the same time, the success rate increases significantly asn the number of trials are increased. Figure 7 illustrates the above observations as the trials progress. In this run, the number of errors was

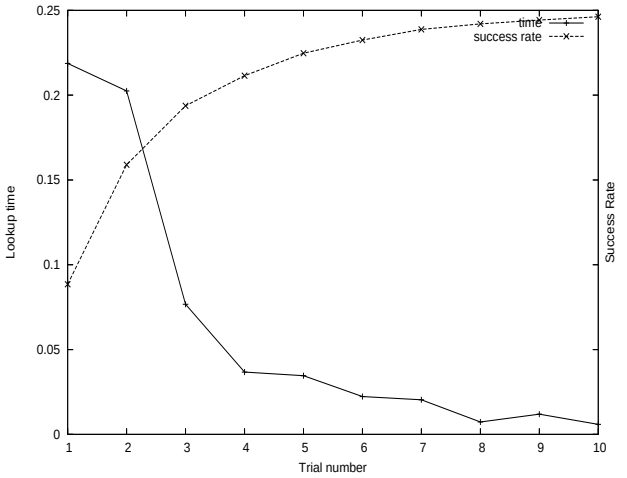

Figure 7: Effect of the number of trials on the success rate and lookup time

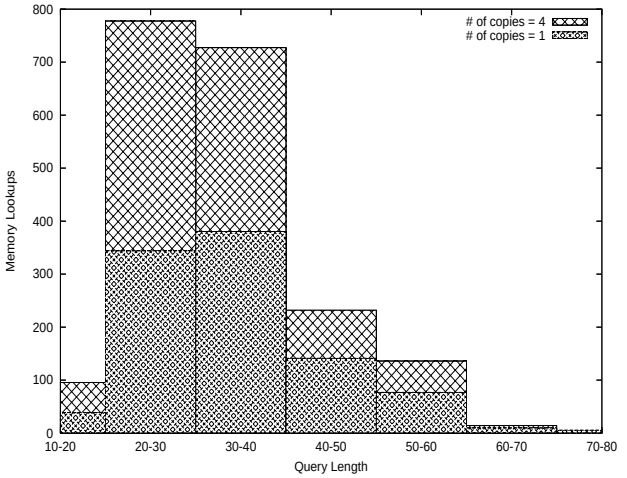

Figure 8: Effect of the query length on the number of lookups

set to 3, $B = 20$, and $c = 3$. The times were averaged over 10000 queries. Observe that the response time falls significantly by round 5 while at the same time the success rate increases to almost 92%.

Finally we studied the effect of the query length on the number of lookups. Intuitively, it seems that the number of lookups should increase with the query length. This is, however, not the case. Initially, as the query length increases, so do the number of lookups. For queries of larger lengths, as our analysis in Section 3 suggests, we need fewer lookups to find a match. Figure 8 illustrates this behavior for different number of copies of the data structure.

7. CONCLUSIONS

We proposed a dictionary data structure for string search with errors where the query string may differ from the expected matching string by a few edits. This data structure can also be used to find the database string with the longest common prefix with few errors. For few errors, our methods produce very few lookups, especially if the query strings are

large.

8. REFERENCES

[1] Alexandr Andoni and Piotr Indyk. Efficient algorithms for substring near neighbor problem. In *Proceedings of the 17th annual ACM-SIAM symposium on Discrete algorithm*, pages 1203–1212, 2006.

[2] Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. Approximating edit distance efficiently. In *Proc. 45th Symposium on Foundations of Computer Science (FOCS 2004)*, pages 550–559, 2004.

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey, 2002.

[5] Andrei Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of SEQUENCES SEQS: Sequences '91*, 1998.

[6] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[7] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the 36th annual ACM symposium on Theory of computing (STOC)*, pages 91–100, 2004.

[8] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, 2003.

[9] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. of 30th ACM Symposium on Theory of Computing (STOC)*, pages 604–613, 1998.

[10] Adam Kirsch and Michael Mitzenmacher. Distance-sensitive bloom filters. In *Proceedings of ALENEX 06*, 2006.

[11] Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proc. of 30th ACM Symposium on Theory of Computing (STOC)*, pages 614–623, 1998.

[12] Gad M. Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[13] M. D. McIllroy. Development of a spelling list. *IEEE Transactions on Communications*, 30(1):91–99, 1982.

[14] James K. Mullin and Daniel J. Margoliash. A tale of three spelling checkers. *Software – Practice and Experience*, 20(6):625–630, 1990.

[15] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[16] Rafail Ostrovsky and Yuval Rabani. Low distortion embeddings for edit distance. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 218–224, 2005.

[17] H. Shang and T. H. Merrettal. Tries for approximate string matching. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):540–547, 1996.

[18] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[19] E. Ukkonnen. Finding approximate pattern in strings. *Journal of Algorithms*, 6:132–137, 1985.