*International Journal of Computer Applications (0975 – 8887)*
*Volume 120 – No.8, June 2015*

# Approximate String Matching Algorithms: A Brief Survey and Comparison

Syeda Shabnam Hasan
Department of Computer Science and Engineering
Ahsanullah University of Science and Technology
Dhaka-1208, Bangladesh

Fareal Ahmed
Department of Computer Science and Engineering
Ahsanullah University of Science and Technology
Dhaka-1208, Bangladesh

Rosina Surovi Khan
Department of Computer Science and Engineering
Ahsanullah University of Science and Technology
Dhaka-1208, Bangladesh

## ABSTRACT

Many database applications require similarity based retrieval on stored text and/or multimedia objects. This is an area of increasing research interest in the sectors of database, data mining, information retrieval and knowledge discovery. This paper presents a brief survey on the existing approximate string matching algorithms by primarily demonstrating three families of algorithms — the Brute force, the Lipschitz Embeddings and the Ball Partitioning algorithms. While Brute Force performs approximate string matching based on distance measures of the query object from each string stored in the database, Lipschitz Embeddings uses a far more efficient approach which embeds the stored strings in database in vector space so that the distances of embedded strings approximates the actual distances. Ball Partitioning algorithm, much more efficient than Brute force but less efficient than Lipschitz algorithm, performs search in approximate string matching based on distances where queries operate on an arbitrary search hierarchy. The paper compares and makes an analysis of these three algorithms which are suitable for approximate matching of strings stored in database text files, an issue much required in the context of similarity based retrieval of objects. The work can be extended for future work by taking into account a larger number of algorithms suited to approximate string matching for the benefit of a wider scope of comparisons and picking out the most optimal one.

## General Terms

Algorithms for Approximate String Matching.

## Keywords

Approximate String Matching Algorithm, Lipschitz Embeddings Algorithm, Ball Partitioning Algorithm.

## 1. INTRODUCTION

Finding the occurrences of a given query string (pattern) from a possibly very large text is an old and fundamental problem in computer science. It emerges in applications ranging from text processing and music retrieval to bioinformatics. This task, collectively known as string matching, has several different variations. The most natural and simple of these is exact string matching, in which, like the name suggests, one wishes to find only occurrences that are exactly identical to the pattern string. This type of search, however, may not be adequate in all applications if, for example, the pattern string or the text may contain typographical errors. Perhaps the most important applications of this kind arise in the field of bioinformatics, as small variations are fairly common in DNA or protein sequences. The field of approximate string matching, which has been a research subject since the 1960's, answers the problem of small variations by permitting some error between the pattern and its occurrences. Given an error threshold and a metric to measure the distance between two strings, the task of approximate string matching is to find all substrings of the text that are within (a distance of) the error threshold from the pattern.

In this work we concentrate on approximate string matching that uses so called unit-cost edit distance as the metric to measure the distance between two strings. One possible definition of the approximate string matching problem is the following: Given a pattern string $P = p_1p_2...p_m$ and a text string, $T = t_1t_2...t_n$ find a substring $T(i...j) = t_i...t_j$ in $T$, which, of all substrings of $T$, has the smallest edit distance to the pattern $P$. The most common application of approximate matchers until recently has been spell checking. With the availability of large amounts of DNA data, matching of nucleotide sequences has become an important application. Approximate matching is also used to identify pieces of music from small snatches and in spam filtering [1].

This paper presents a brief survey on approximate string matching algorithms (section 2), followed by an elaborate demonstration of three related algorithms — the Brute Force algorithm for approximate string matching, the Lipschitz Embeddings Algorithm and the Ball Partitioning Algorithm and makes a comparison among these three algorithms. A concise overview of edit distance is also discussed (in section 3).

## 2. RELATED WORKS

A number of researchers have presented variations on approximate string matching algorithms. Luis M. S. Russo, Gonzalo Navarro, Arlindo L. Oliveira, and Pedro Morales focused on indexed approximate string matching [2]. They studied approximate string matching algorithms for Lempel-Ziv compressed indexes and for compressed suffix trees/arrays. Lempel-Ziv indexes split the text into a sequence of so-called phrases of varying length. They are efficient to find the (exact) occurrences that lie within phrases, but those that span two or more phrases are costlier. Luis M. S. Russo, Gonzalo Navarro, Arlindo L. Oliveira, and Pedro Morales started by adapting the classical method of partitioning into exact search to self-indexes, and optimized it over a representative of either class of self-index. Then, they showed that a Lempel-Ziv index can be seen as an extension of the classical *q*-samples index and they improved hierarchical verification to extend the matches of pattern pieces to the left or right which largely reduced the accesses to the text, which are expensive in self-indexes. Zheng Liu, James Borneman, Tao Jiang presented a fast algorithm for approximate string matching called FAAST [3]. It aimed at solving a popular variant of the approximate string matching problem, the *k*-mismatch problem, whose objective is to find all occurrences of a short pattern in a long text string with at most *k* mismatches. FAAST generalizes the well-known Tarhio-Ukkonen's *k*-mismatches algorithm. In the Tarhio-Ukkonen algorithm, the shift distance is calculated as the minimum one such that there exists at least one match when aligning the rightmost $k + 1$ text characters in the current alignment with the pattern after a shift. In order to achieve faster matching process, FAAST instead calculates the shift distance as the minimum one such that the rightmost $k + x$ characters of the current aligned text will have at least $x$ matches after the shift. Here, $x$ generally takes a small integer value, e.g., two or three. Theoretically, they proved that FAAST on average skips more characters than the Tarhio-Ukkonen algorithm in a single shift, and makes fewer character comparisons in an entire matching process. Bit-vector algorithm of Myers is one of the most notable recent algorithms in the area of approximate string matching algorithms. The main idea of this algorithm is to parallelize the dynamic programming matrix by using bit-vectors to encode the list of $m$ (arithmetic) differences between successive entries in a column of the dynamic programming matrix. Heikki Hyyro, Kimmo Fredriksson, Gonzalo Navarro explored different ways to increase the bit-parallelism for approximate string matching by modifying the bit-vector algorithm of Myers when the pattern is short or if the maximum number of differences permitted is moderate with respect to the alphabet size [4]. They showed how multiple patterns can be packed in a single computer word so as to search for multiple patterns simultaneously. They showed two ways to do this. The first one permits searching for several patterns simultaneously. The second one boosts the search for a single pattern by processing several text positions simultaneously. William I. Chang and Eugene L. Lawler explored the research area of approximate string matching in sub linear expected time [5]. They defined the approximate substring matching problem and gave efficient algorithms based on their techniques. Special cases include several applications to genetics and molecular biology. For example, even allowing errors, they found long common blocks of the text and pattern (local similarities), or selected from among a set of text fragments — ones that overlap one end of the pattern (sequence assembly). These are common tasks in DNA sequence analysis. Gonzalo Navarro [6] focused on the problem of string matching that allows errors, also called approximate string matching. The general goal is to perform string matching of a pattern in a text where one or both of them have suffered some kind of (undesirable) corruption. An Some examples are recovering the original signals after their transmission over noisy channels, finding DNA subsequences after possible mutations, and text searching where there are typing or spelling errors.

## 3. EDIT DISTANCE AND APPROXIMATE STRING MATCHING

The edit distance $ed(P, T)$ between the strings $P$ and $T$ is defined in general as the minimum cost of any sequence of edit operations that edits $P$ into $T$ or vice versa. Differing in their choices of the allowed set of edit operations and their costs, for example the following types of edit distance have appeared in the literature.

- Levenshtein edit distance
- Damerau edit distance
- Weighted/generalized edit distance
- Hamming distance
- Longest common subsequence

Approximate string matching is closely related to edit distance. It refers to searching for approximate matches of a pattern string $P$ from a usually much longer text string $T$, where edit distance is used as a measure of similarity between $P$ and the substrings of $T$. [7]

The work in this paper concentrates on Levenshtein edit distance in which the allowed edit operations are insertion, deletion or substitution of a single character, and each operation has the cost = 1. This type of edit distance is sometimes called unit-cost edit distance. Levenshtein edit distance is perhaps the most common form of edit distance, and often the term edit distance is assimilated to it [8]. The following algorithm of Levenshtein edit distance fills the (integer) entries in a matrix $m$ whose two dimensions equal the lengths of the two strings ($s1$, $s2$) whose edit distances being computed; the $(i, j)$-th entry of the matrix will hold (after the algorithm is executed) the edit distance between the strings consisting of the first $i$ characters of $s1$ and the first $j$ characters of $s2$. The central dynamic programming step is depicted in Lines 8-10 (fig. 1), where the three quantities whose minimum is taken correspond to substituting a character in $s1$, inserting a character in $s1$ and inserting a character in $s2$ [9].

**EDIT DISTANCE (*S*1, *S*2)**

```
1.  int  $m[i, j] = 0$
2.  for $i = 1$ to $|S1|$
3.  do $m[i, 0] = i$
4.  for $j = 1$ to $|S2|$
5.  do $m[0, j] = j$
6.  for $i = 1$ to $|S1|$
7.  do for $j = 1$ to $|S2|$
8.  do $m[i, j] =$
9.      min { $m[i-1, j-1]$ + if ($S1[i] = S2[j]$) then 0 else 1,
10.             $m[i-1, j] + 1$,
11.             $m[i, j-1] + 1$ }
12. return  $m[|S1|, |S2|]$
```

**Fig. 1: Levenshtein edit distance algorithm to compute edit distance between the pair of strings *S*1 and *S*2**

## 4. BRUTE FORCE ALGORITHM FOR APPROXIMATE STRING MATCHING

A brute-force approach would be to compute the edit distances to the query object ($q$) from all $N$ substrings of text ($T$) and then choose the substring with the minimum edit distance. The sequential steps are given below.

**Steps:**

1. Read $N$ (say, 50) strings from the text file $T$.
2. Take an input (query object $q$) from the user.
3. Calculate the edit distances to query object $q$ from all $N$ strings of $T$.
4. Find out the minimum edit distance.
5. Output will be the string which has the minimum edit distance.

**Example:**

Let $N = 5$ and $T$ is **{been, bid, moon, sun, star}**.

Suppose, the query object $q$ is '**seen**'

The edit distances to query object $q$ from all $N$ strings of $T$ will be respectively {1, 4, 3, 2, 3}, the minimum of which is 1 and hence the output will be **been**

## 5. LIPSCHITZ EMBEDDINGS ALGORITHM

A Lipschitz embedding is defined in terms of a set $R$ of subsets of $S$ (the set of objects), where $R = \{A_1, A_2, \ldots, A_k\}$. The subsets $A_i$ are termed the reference sets of the embedding. Let $d(o, A)$ be an extension of the distance function $d$ to a subset $A$ of $S$, such that $d(o, A) = min\ \{d(o, x)\}$, where $x$ is an element of $A$. An embedding with respect to $R$ is defined as a mapping $F$ such that $F(o) =( (d(o, A_1), d(o, A_2), \ldots, d(o, A_k))$

To elaborate on how a query is implemented, suppose that we want to find the nearest object to a query object $q$. We first determine the point $F(q)$ corresponding to $q$. Next, we examine the objects in the order of their distances from $F(q)$ in the embedding space. When using a multidimensional index, this can be achieved by using an incremental nearest neighbor algorithm. Suppose that point $F(a)$ corresponding to an object $a$ is the closest point to $F(q)$ at a distance of $\partial(F(a), F(q))$. We compute the distance $d(a, q)$ between the corresponding objects. At this point, we know that any object $x$ with $\partial(F(x), F(q)) > d(a, q)$ cannot be the nearest neighbor of $q$ because, the contractive property then guarantees that $d(x, q) > d(a, q)$. Therefore, $d(a, q)$ now serves as an upper bound on the nearest-neighbor search in the embedding space. We now find the next closest point $F(b)$ corresponding to the object $b$, subject to our distance constraint $d(a, q)$.

If $d(b, q) < d(a, q)$, then $b$ and $d(b, q)$ replace object $a$ and $d(a, q)$ as the current closest object and upper bound distance, respectively; otherwise, $a$ and $d(a, q)$ are retained. This search continues until encountering a point $F(x)$ with the property $\partial(F(x), F(q)) > d(y, q)$, where the $y$ is the current closest object which is now guaranteed to be the actual closest object to $q$. [10]

**Algorithm:**

**Input:** A text file $T$ containing $N$ strings $(O_1, O_2, \ldots, O_N)$ and a query object $q$.

**Output:** Nearest string to $q$ with corresponding edit distance.

**Steps:**

1. Construct a text file $R$ of $k$ strings $(A_1, A_2, \ldots, A_k)$ by choosing randomly from $N$ strings of $T$.
2. Compute $F(q)$, the array of edit distances of the query object $q$ to $k$ strings of $R$, that is, $F(q) = (d(q, A_1), d(q, A_2), \ldots, d(q, A_k))$, where $d(q, A_j)$ is the edit distance between $q$ and $A_j$. Here, $1 \le j \le k$.
3. Compute $F(O_i)$, the array of edit distances of each string $O_i$ in $T$ to the $k$ reference strings in $R$, that is, $F(O_i) = ( (d(O_i, A_1), (d(O_i, A_2), \ldots, d(O_i, A_k))$. Here, $1 \le i \le N$.
4. Calculate $\partial(F(O_j), F(q))$, the distance between each string $O_j$ of $T$ to query string $q$ in the embedding space where $1 \le j \le N$, $1 \le i \le k$, by using the following formula.

$$\partial\left(F\left(O_j\right), F(q)\right) = \sum_{i=1}^{k}\left(\left(\frac{d\left(O_j, A_i\right) - d\left(q, A_i\right)}{k^{1/p}}\right)^p\right)^{\frac{1}{p}}$$

5. Find the minimum value among $\partial(F(O_i), F(q))$, for $1 \le i \le N$. If the minimum is $\partial(F(O_m), F(q))$, then find $d(O_m, q)$.
6. For $i = 1$ to $N$ do

```
if $\partial(F(O_i), F(q)) > d(O_m, q)$ then
    $edit\_distance = d(O_m, q)$ and $nearest\_string = O_m$
else if $d(O_i, q) < d(O_m, q)$ then
    $edit\_distance = d(O_i, q)$ and $nearest\_string = O_i$
```

7. Show the *edit_distance* and *nearest_string* to the user.

## 6. BALL PARTITIONING ALGORITHM

In this method, we pick a pivot element $p$ randomly from $S$ containing, say, 50 strings/objects. The $p$ is termed as a vantage point. The algorithm computes the median $r$ of the distances of the other objects to the pivot $p$, and then divides the remaining objects into roughly equal sized subsets $S1$ and $S2$ as follows.

$$S1 = \left\{\ o \in S \setminus \{p\}\ \middle|\ d(p,\ o) < r\ \right\} \text{ and}$$

$$S2 = \left\{\ o \in S \setminus \{p\}\ \middle|\ d(p,\ o) \ge r\ \right\}$$

Thus, the objects in $S1$ are *inside* the ball of radius $r$ around $p$, while the objects in $S2$ are *outside* this ball. Applying this rule recursively leads to a binary tree, called the 'vantage point tree' (or, vp-tree) where the pivot objects are stored in each internal node, with the left and right sub trees corresponding to the subsets inside and outside the corresponding ball, respectively. In the leaf nodes of the vp-tree, we would store one or more objects, depending on the desired capacity.

**Pivot Selection:** Pivot is chosen randomly in this algorithm and in the vp-tree, the ball radius is always chosen as the median so that the two subsets are roughly equal in size.

**Search:** Visit left child if and only if $max\{d(q, p) - r, 0\} \le \varepsilon$ and the right child if and only if $max\{r - d(q, p), 0\} \le \varepsilon$

**Definitions of symbols:**

Radius of pivot: $r$
Query object: $q$
Pivot object: $p$
Edit distance of pivot and query object: $\varepsilon$
Another notation of edit distance of pivot and query: $d\ \{q, p\}$

The ball partitioning algorithm is divided into the following two parts.

**a) CREATING VP-TREE:**

1. Read $N$ (say, 50) strings from a text file.
2. Select a pivot ($p$) randomly from $N$ strings.
3. Compute edit-distance ($d$) of the pivot from the other remaining strings.
4. Values of edit-distances (from pivot to others) will be kept in an array.

5. Sort the values of the array and find out the median value using the formula of median.

6. The median is the radius (*r*) of pivot. A flag value will be increased.

7. Take this pivot and radius as the input of VP-tree's first node.

8. Similarly, repeat the above steps 2 – 6 and find out pivot for each step. If the pivot's radius, say $r'$, is less than its parent node's radius $r$ and $r' \geq 0$, then put it on left node and if $r' \geq r$ then put it on right node. In step 2, each time $N$ decreases by 1.

9. This process will continue until each internal node ends at leaf with a child or we can keep a cluster of strings at each leaf.

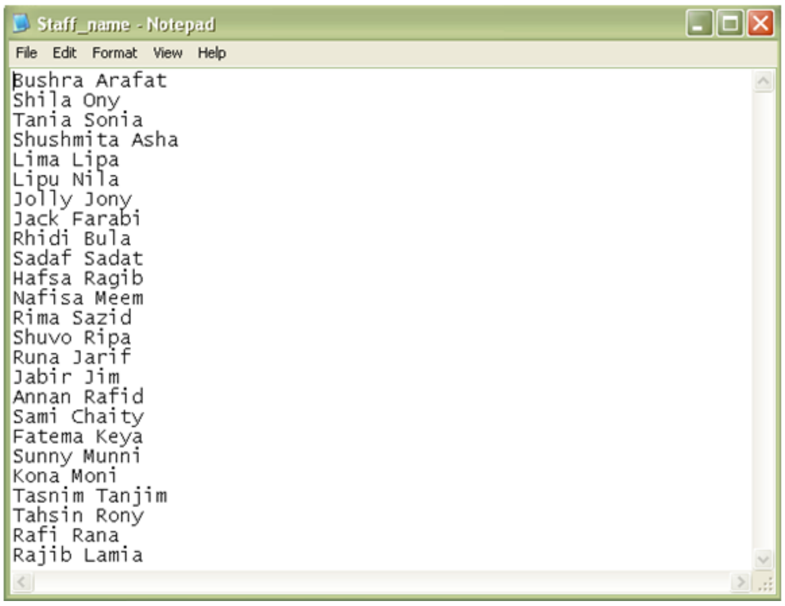
**b) SEARCHING PHASE:**

1. Take an input of query object ($q$).
2. After a value is assigned in the VP-tree (pivot and radius), find the edit distance of pivot and the query object. And if $max\ \{d\ (q, \text{p}) - r,\ 0\} \leq E$, then visit the left node; otherwise don't. Also, if $max\ \{r - d\ (q, \text{p}),\ 0\} \leq E$, then visit the right node; otherwise don't. Here, $E$ is the edit-distance between the pivot and query object.
3. Keep the edit-distance and the pivot in an array of structure.
4. Repeat the above steps 1 – 3 of the searching phase until we get the child of each node.
5. Now find the minimum edit distance of all edit distances, and the corresponding pivot which will be the nearest neighbor of the query object.

## 7. ANALYSIS

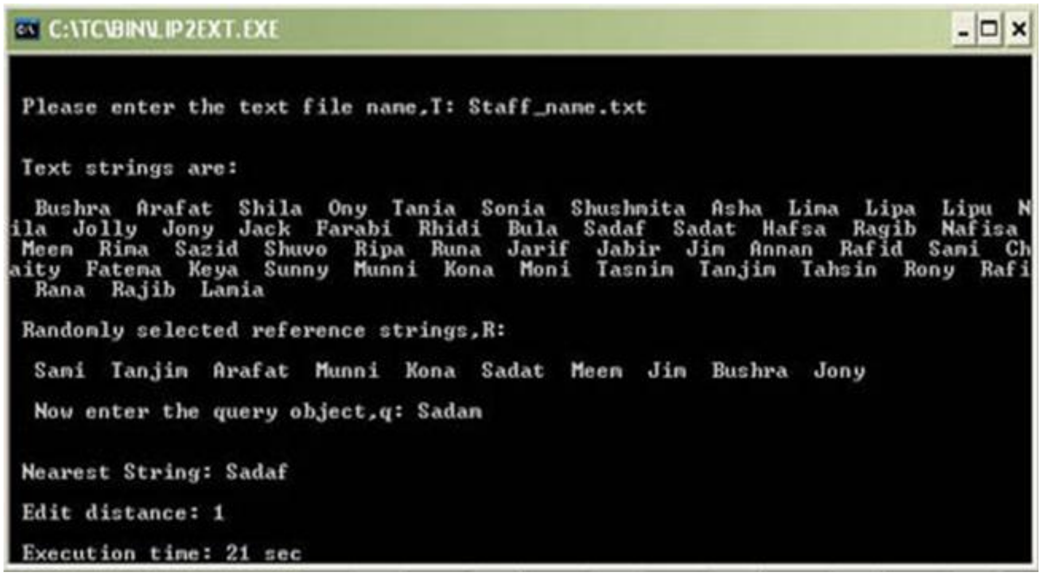
Lipschitz Embedding Algorithm uses a straight forward procedure to match an approximate string to the query string. It creates reference strings and uses mathematical formula to find out minimum edit distance and the corresponding nearest string to the query string. On the other hand, the Ball Partitioning Algorithm creates a VP-tree and visits the left and right nodes of the tree to find minimum edit distance and the corresponding string which is nearest to the query string. The execution time increases with the increase of visited nodes. The Brute Force Algorithm for the approximate string matching uses the most inefficient way to find out nearest string. It calculates edit distances of all the text strings from the query string and then checks all the distances to find out the minimum distance. In Lipschitz Embeddings and Ball Partitioning Algorithms, this checking is not applicable, because both of them follow some different mathematical procedures to find the expected results. As a result, among the three approximate string matching algorithms, the fastest is the Lipschitz Embeddings Algorithm, then the Ball Partitioning Algorithm and the slowest is the Brute Force Algorithm. In order to evaluate the practical performances of these three approximate string matching algorithms, we have implemented them using the programming language C, following a homogeneous procedure, using the same input text file having 50 strings to make the comparisons significant. Here, the input text file, named as 'Staff_name.txt', has been extracted from a data-table of a standard database management system.

Snapshots of the results and execution times of the three algorithms, applied on the Staff_name.txt file, are presented below, in the ascending order of their execution times (i.e., from the fastest to the slowest).



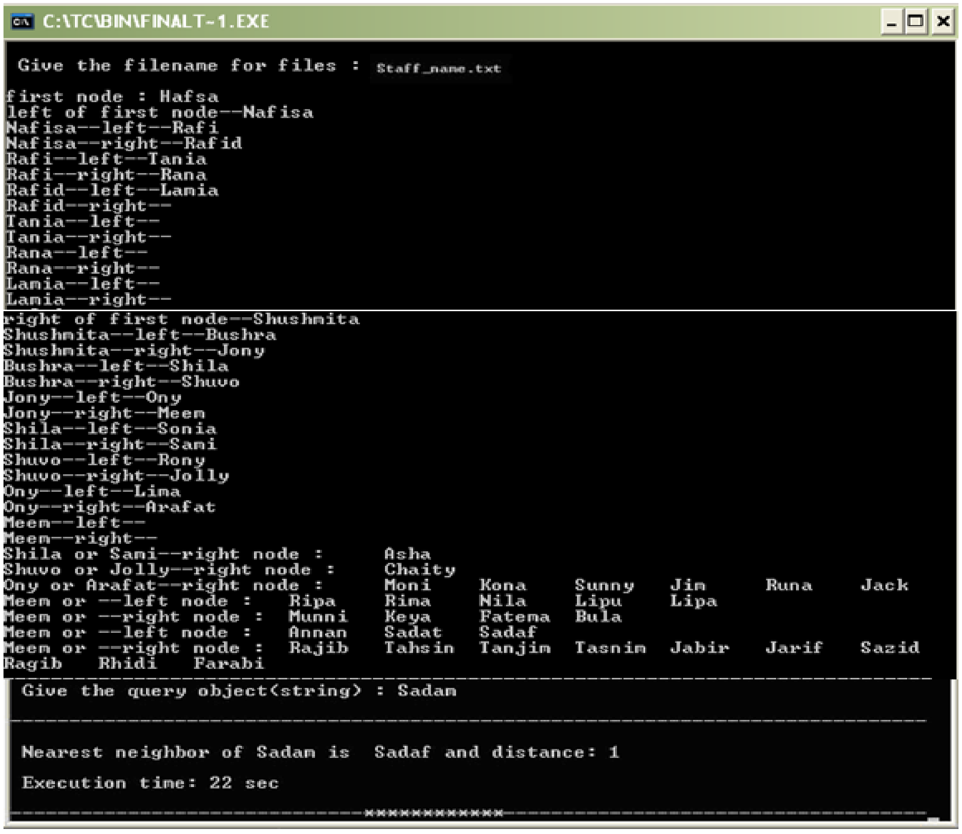
```
Bushra Arafat
Shila Ony
Tania Sonia
Shushmita Asha
Lima Lipa
Lipu Nila
Jolly Jony
Jack Farabi
Rhidi Bula
Sadaf Sadat
Hafsa Ragib
Nafisa Meem
Rima Sazid
Shuvo Ripa
Runa Jarif
Jabir Jim
Annan Rafid
Sami Chaity
Fatema Keya
Sunny Munni
Kona Moni
Tasnim Tanjim
Tahsin Rony
Rafi Rana
Rajib Lamia
```

**Fig 2: Snapshot of the input file *Staff_name.txt***



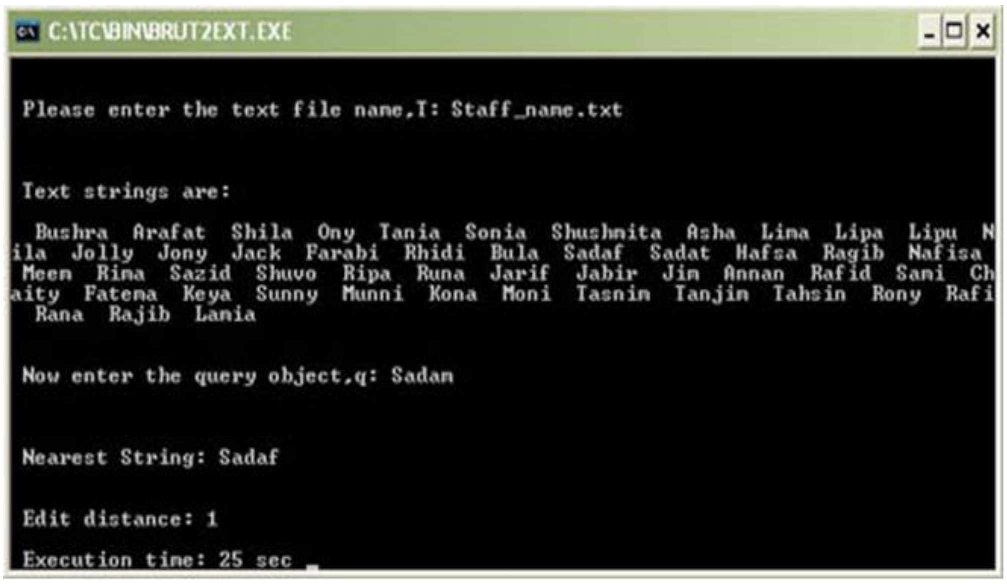
```
C:\TC\BIN\LIP2EXT.EXE

Please enter the text file name,T: Staff_name.txt

Text strings are:

 Bushra  Arafat  Shila  Ony  Tania  Sonia  Shushmita  Asha  Lima  Lipa  Lipu  N
ila  Jolly  Jony  Jack  Farabi  Rhidi  Bula  Sadaf  Sadat  Hafsa  Ragib  Nafisa
Meem  Rima  Sazid  Shuvo  Ripa  Runa  Jarif  Jabir  Jim  Annan  Rafid  Sami  Ch
aity  Fatema  Keya  Sunny  Munni  Kona  Moni  Tasnim  Tanjim  Tahsin  Rony  Rafi
 Rana  Rajib  Lamia

Randomly selected reference strings,R:

 Sami  Tanjim  Arafat  Munni  Kona  Sadat  Meem  Jim  Bushra  Jony

 Now enter the query object,q: Sadan

Nearest String: Sadaf

Edit distance: 1

Execution time: 21 sec
```

**Fig 3: Snapshot of the execution time (i.e., 21 sec.) and results of the Lipschitz Embeddings Algorithm**



```
C:\TC\BIN\FINALT-1.EXE
 Give the filename for files :  Staff_name.txt
first node : Hafsa
left of first node--Nafisa
Nafisa--left--Rafi
Nafisa--right--Rafid
Rafi--left--Tania
Rafi--right--Rana
Rafid--left--Lamia
Rafid--right--
Tania--left--
Tania--right--
Rana--left--
Rana--right--
Lamia--left--
Lamia--right--
right of first node--Shushmita
Shushmita--left--Bushra
Shushmita--right--Jony
Bushra--left--Shila
Bushra--right--Shuvo
Jony--left--Ony
Jony--right--Meem
Shila--left--Sonia
Shila--right--Sami
Shuvo--left--Rony
Shuvo--right--Jolly
Ony--left--Lima
Ony--right--Arafat
Meem--left--
Meem--right--
Shila or Sami--right node :     Asha
Shuvo or Jolly--right node :    Chaity
Ony or Arafat--right node :     Moni    Kona    Sunny   Jim     Runa    Jack
Meem or --left node :   Ripa    Rima    Nila    Lipu    Lipa
Meem or --right node :  Munni   Keya    Fatema  Bula
Meem or --left node :   Annan   Sadat   Sadaf
Meem or --right node :  Rajib   Tahsin  Tanjim  Tasnim  Jabir   Jarif   Sazid
Ragib   Rhidi   Farabi
-------------------------------------------------------------------------------
 Give the query object(string) : Sadam
 ------------------------------------------------------------------------------

 Nearest neighbor of Sadam is  Sadaf and distance: 1

 Execution time: 22 sec

-----------------------------*************-------------------------------------
```

**Fig 4: Snapshot of the execution time (i.e., 22 sec.) and results of the Ball Partitioning Algorithm**



```
C:\TC\BIN\BRUT2EXT.EXE

Please enter the text file name,T: Staff_name.txt

Text strings are:
  Bushra  Arafat  Shila  Ony  Tania  Sonia  Shushmita  Asha  Lina  Lipa  Lipu  N
ila  Jolly  Jony  Jack  Farabi  Rhidi  Bula  Sadaf  Sadat  Hafsa  Ragib  Nafisa
 Meen  Rima  Sazid  Shuvo  Ripa  Runa  Jarif  Jabir  Jim  Annan  Rafid  Sani  Ch
aity  Fatema  Keya  Sunny  Munni  Kona  Moni  Tasnim  Tanjim  Tahsin  Rony  Rafi
  Rana  Rajib  Lamia

Now enter the query object,q: Sadam

Nearest String: Sadaf

Edit distance: 1

Execution time: 25 sec _
```

**Fig 5: Snapshot of the execution time (i.e., 25 sec.) and results of the Brute Force Algorithm**

## 8. CONCLUSION AND FUTURE WORK

In this paper we have presented some existing approximate string matching algorithms, explored their characteristics and implemented them in the C programming language. As we have found, their execution times are different. The Lipschitz Embeddings Algorithm is faster than the Ball Partitioning and Brute Force Algorithms. On the other hand Ball Partitioning Method needs too much memory than the others. We have used real life data and converted database tables into text files and applied the approximate string matching algorithms on these input text files.

This research work may be extended along several possible directions. For example, there are many other advanced approximate string matching algorithms having different characteristics and usefulness which can be used for wider comparisons. Besides, we have worked on text files having 50 strings each, which can be expanded by including more strings — say, 1000 strings or more. Also, in Ball Partitioning Method, clusters of strings may be represented at the leaves for the sake of saving the storage space. Furthermore, these methods can be applied for similarity search which plays a leading role in the modern multimedia databases and for many other database applications involving complex objects.

## 9. REFERENCES

[1] Approximate String Matching. Available Online source: http://en.wikipedia.org/wiki/Approximate_string_matching. Last accessed: July 2011.

[2] Luis M. S. Russo, Gonzalo Navarro, Arlindo L. Oliveira and Pedro Morales, "Approximate string matching with compressed indexes", Algorithms, Volume 2, Iss ue 3, Pages 1105–1136, September 2009.

[3] Zheng Liu, James Borneman and Tao Jiang, "A fast algorithm for approximate string matching on gene sequences", in 16th Annual Symposium on Combinatorial Pattern Matching, LNCS, Springer-Verlag, pages 79–90, June 2005.

[4] Heikki Hyyro, Kimmo Fredriksson and Gonzalo Navarro, "Increased bit -parallelism for approximate and multiple string matching", Journal of Experimental Algorithmics (JEA), Volume 10, article 2.6, December 2005.

[5] William I. Chang and Eugene L. Lawler, "Approximate string matching in sublinear expected time", 31st Annual Symposium on Foundations of Computer Science (FOCS 1990), vol.1, pages 116–124, October 1990.

[6] Gonzalo Navarro, "A guided tour to approximate string matching", Journal of ACM Computing Surveys, (CSUR), Vol. 33, No. 1, pages 31–88, March 2001.

[7] Heikki Hyyro, "Practical methods for approximate string matching", Acta Electronica Universitatis Tamperensis, 2003.

[8] V. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals", Soviet Physics Doklady, 10 (8), pages 707–710, 1966.

[9] Christopher Manning, Prabhakar Raghavan and Hinrich Schutze, "Introduction to Information Retrieval", Cambridge University Press, 2008.

[10] Gísli R. Hjaltason and Hanan Samet, "Properties of embedding methods for similarity searching in metric space," IEEE transactions on pattern analysis and machine intelligence, Vol. 25, No. 5, pages 530–549, May 2003

[11] Gísli R. Hjaltason and Hanan Samet, "Index-driven similarity search in metric spaces", ACM Transactions on Database Systems, Vol. 28, No. 4, pages 517–580, December 2003.