# A Comparative Performance Analysis of Approximate String Matching

**Shivani Jain, A.L.N. Rao**

***Abstract— This paper presents a comparative study to evaluate experimental results for approximate string matching algorithms on the basis of edit distance. We compare the algorithms in terms of the number of character comparisons and the running time for molecular data, binary alphabets English alphabets etc.***

***The terms like word processors, web search engine, molecular sequence, DNA sequence analysis and natural language processing have lead to the development of many algorithms in the field of pattern matching in a string. Amongst the various string searching algorithms being used, here the focus is mainly approximate implementation of pattern matching algorithms such as Knuth-Morris-Pratt, Boyer-Moore, Raita, Horspool based on PHP. The comparison between these algorithms is done with the help of Levenshtein distance.***

***It also describes the importance of design of efficient "Approximate Pattern Search Algorithms in molecular database, binary alphabets, English alphabets and so on". This approach is advantageous from all other string-pattern matching algorithms in terms of time complexity. Therefore this procedure improves the efficiency of approximate string matching and gives the near-optimal results.***

***Keywords— Pattern Matching, Edit Distance, Approximate String Searching, Levenshtein Distance***

## I. INTRODUCTION

String matching is one of the main problems in classical string algorithms, with applications to text searching, biological applications, pattern recognition etc. [1]

The problem of finding exact or non-exact occurrences of a pattern *P* in a text *T* over some alphabet is a central problem of combinatorial pattern matching and has a variety of applications in many areas of computer science [2].

An algorithm that returns near-optimal solutions is called an approximation algorithm. The problem of approximate string matching is typically divided into two sub-problems: finding approximate substring matches inside a given string and finding dictionary string that match the pattern approximately.

Approximate String matching- Given a text string T of length n, a pattern string P of length m and a maximal number of errors allowed k, the approximate string matching is to find all text positions where the pattern matches the text up to k errors, where errors can be substituting, deleting, or inserting a character. For instance, if T = "pttapa", P = "patt" and k = 2, the substrings $^{T1,2}$, $^{T1,3}$, $^{T1,4}$ and $^{T5,6}$ are all up to 2 errors with P.[3] Approximate string matching is the challenging problem in Computer Science and requiring a large amount of computational resources.

**Manuscript received October, 2013.**

**Shivani Jain,** IT Mewar University, Chittorgarh, Rajasthan, India

**Dr. A.L.N. Rao,** IT MU, MTU Galgotia Engineering College, Greater Noida, UP India

It has different areas such as computational biology, text processing, pattern recognition and signal processing. For these reasons, fast practical algorithms for approximate string matching are in high demand.

Approximate string matching has a wide variety of applications from DNA matching, to signal processing, to text retrieval [4].

Fig-1: Locate all occurrence of the pattern *P* in a text *T*.

String Matching Example

"you write always research paper, my research paper, for paper, I am writing a paper"paper?

"you write always research paper, my research paper, for paper, I am writing a paper"

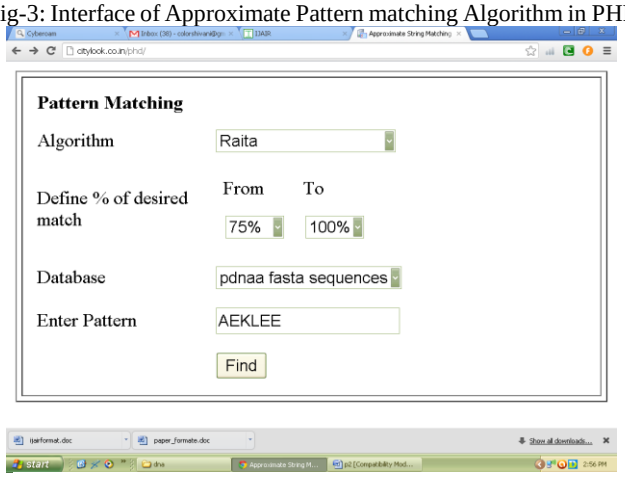
## II. IMPLEMENTATION OF DIFFERENT PATTERN MATCHING ALGORITHMS USING APPROXIMATE METHODOLOGY

This is the discussion of comparison in terms of running time between different pattern matching algorithms by using PHP as the interface. We are also trying to create an interface for approximate string matching pattern in PHP. Now we use only already existing string matching algorithms and Levenshtein distance. [5]

Fig- 2: Computed Edit Distance with error 2

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| | | | s | a | t | t | e | r | n |
| 0 | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | s | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | a | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| 3 | t | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 |
| 4 | e | 4 | 3 | 2 | 1 | 1 | 2 | 3 | 4 |
| 5 | e | 5 | 4 | 3 | 2 | 2 | 1 | 2 | 3 |
| 6 | n | 6 | 5 | 4 | 3 | 3 | 2 | 2 | 2 |

Fig-3: Interface of Approximate Pattern matching Algorithm in PHP




The Fig-3 compare the running time between different already existing pattern matching algorithms using PHP as the interface. This was the initial work, therefore used only already existing string matching algorithms with Levenshtein distance.

The user chooses the desired Exact Matching Algorithm like Boyer Moore or Raita then chooses desired percentage matched. After that the database and then enter the pattern.

The pattern now gets matched with the patterns in the database and their approximate percentage is calculated with help of Levenshtein Distance Algorithm. If the string in database gets exactly matched with the pattern entered by the user that is the output percentage is 100% then the output string goes to the algorithm selected by the user initially and after that running time is calculated. With the help of Levenshtein distance calculate the distance means k-errors between the two strings. In Fig-3 match the pattern with the database records one by one with the help of Levenshtein Distance, and then calculate the percentage of matching. If the program calculates 100% matching then it call the user selected Exact String Matching Algorithm.

Using the Approximation method the users get relevant information after going through the several results produced by approximation method, but in case of exact method users have no choice if the patterns are not exactly matched.

Fig-4: Demonstration of the result with best match percentage and running time in PHP

Algorithm Raita

Desired Percentage 75% to 100%

Database pdnaa fasta sequences

Pattern TTETLHG

**Result**: Highlited pattern using Raita algorithm

And percentage using Levenshtein or Edit distance

| Id | Text | Best Match With Percentage |
|---|---|---|
| 2 | MDKKSARIRRATRARRKLQELGATRLV VHRTPRHIYAQVIAPNGSEVLVAASTVE KAIAEQLKYTGN | 85.71% |
| 3 | MQAIKCVVVGDGAVGKTCLLISYTTNAF PGEYIPTVFDNYSANVMVDGKPVNLGL WDTAGQEDY | 85.70% |
| 5 | MADITLISGSTLGGAEYVAEHLAEKLEE AGFTTETLHGPLLEDLPASGIWLVISSTH GAGDIPDNLSPF | 100% |

Last timestamp= 027192700.1357007090 Macro Sec

First timestamp= 027175800.1357007090 Macro Sec

Time= 0.00016899999999997 Macro Sec

Fig-4 shows the all patterns that are found according to desired percentage match. The highlighted string from the database matches exactly with the pattern entered by the user and therefore its running time is calculated and shown. [6]

For calculating the running time, procedure has taken the system timestamp as reference before the execution of algorithm and again procedure has taken the system timestamp whenever the algorithm is completed. So the difference between the two timestamp is the desired running time.

The user chooses the desired Exact Matching Algorithm like Boyer Moore then he chooses desired percentage matched. After that the database and then he enters the pattern.

Approximate string matching consists in finding all approximate occurrences of a pattern x of length m in a text y of length n. Approximate occurrences of x are segments of y that are close to x according to a specific distance. The distance must be not greater than a given integer k. We consider two distances, the Hamming distance and the Levenshtein distance [7]. These are form of Edit distance.

The pattern now gets matched with the patterns in the database and their approximate percentage is calculated with help of Levenshtein Distance Algorithm. If the string in database gets exactly matched with the pattern entered by the user that is the output percentage is 100% then the output string goes to the algorithm selected by the user initially and after that running time is calculated. With the help of Levenshtein distance we calculate the distance means k-errors between the two strings. In the above interface we match the pattern with the database records one by one and with the help of Levenshtein Distance, we calculate the percentage of matching. If the above program calculates 100% matching then it call the user selected Exact String Matching Algorithm.

By using the Approximation method the users get relevant information after going through the several results produced by approximation method, but in case of exact method users have no choice if the patterns are not exactly matched.

This snap shot is the output and shows the all patterns that are found according to desired percentage match. The highlighted string from the database matches exactly with the pattern entered by the user and therefore its running time is calculated and shown.

For calculating the running time we have taken the system timestamp as reference before the execution of algorithm and again we have taken the system timestamp whenever the algorithm is completed. So the difference between the two timestamp is the desired running time.

## III. EXPERIMENTAL METHODOLOGY

In this section we present the testing methodology which used to compare the relative performance of string matching algorithms. The parameters which describe the performances of the algorithms are:

a) The text size,
b) The pattern length and
c) The alphabet size.

It is known that none of the algorithms are optimal or best in all three cases. Therefore, the main goal in our experimental study is to compare the practical performance of the algorithms against the length of the pattern (small and long

patterns) under various alphabets of different sizes (or types of text) i.e. binary alphabet, alphabet of size 8, English alphabet and DNA alphabet, which have different characteristics.

### *3.1 Types of test data*

On the basis of experimental observations the performances of the approximate string matching algorithms depend upon statistical properties of the pattern and the text string from which the test patterns were obtained, experiments were performed on four different types of texts: binary alphabet, alphabet of size 8, English alphabet and DNA alphabet.

#### *a) Binary Alphabet*

The alphabet is Ó={0,1}. The text is consisted of 150,000 characters and was randomly built. For patterns of lengths between 2 and 100 we search 50 of them random built.

#### *b) Alphabet of size 8*

The alphabet is Ó={a, b, c, d, e, f, g, h}. The text is consisted of 150,000 characters and was random built. In addition, for patterns of lengths between 2 and 100 we search 50 of them random built.

#### *c) English Alphabet*

We used a document of English language from a web page. The alphabet is consisted of 70 different characters. The text is consisted of 148,188 characters and we search 50 patterns of each length from 2 to 100 characters were chosen at random from words inside the text.

#### *d) Molecular Sequence (FASTA)*

The FASTA sequence is a DNA and Protein sequence alignment. FASTA takes a given nucleotide or amino-acid sequence and searches a corresponding sequence database by using local sequence alignment to find matches of similar database sequences. The text is consisted of 997,642 characters and we search 50 patterns of each length from 10 to 100 characters. Finally, the text and the patterns is portion of the PDNAA database, as distributed by Hume and Sunday [9].

### *3.2 Measures of Comparison*

For the comparison of the string matching algorithms a number of character comparisons and the practical running time are measured. The counting of the number of character comparisons is the same as that used by Smith [10] that is, computing the number of actually compared characters to the number of passed characters in the text. Since all algorithms are designed to find all occurrences of a pattern in the text in our experiments, the number of passed characters is always n-m+1. The running time is the total time of calling an algorithm to search a pattern in the text including the preprocess time of building the auxiliary arrays. The running time is obtained by calling the PHP function microtime() and it is measured in micro seconds.

As measured the number of character comparisons and the running time of all the algorithms in order to examine the effect of the pattern length, a measurement was taken to analyze the effect of the pattern length in a test series with varying m=2, 4, 6, 8, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100. In case of the PDNAA (Molecular Sequence) alphabet longer patterns were used because this alphabet has biological applications on long patterns. For this reason, in this alphabet we measured the effect of the pattern length in a test series with varying m=2, 4, 6, 8, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100.

In order to decrease random variation, the results of the algorithms are averages of 50 runs with different patterns of each length.

## IV. EXPERIMENTAL RESULTS

In this section, we present the experimental results for the approximate string matching algorithms according to the number of character comparisons and the running time. Finally, the performance of each algorithm was stored in a table and plotted a graph against the length of the pattern for each type of text.

### *4.1 Results for the number of character comparisons*

Figures 5 to 8 and Tables 1 to 4 show the results for the number of character comparisons for a binary alphabet, an alphabet of size 8, an English alphabet and a PDNAA alphabet respectively, against the different pattern lengths. Further, the BMH and KMP algorithm produces approximately the same number of character comparisons for the Binary alphabet.

The BMH requires more character comparisons for small size alphabet (i.e. the binary or the genome alphabet). Based on the empirical results, it is clear that for patterns of length greater than 20, the number of character comparisons is approximately 2, twice the number required by the KMP and BMH algorithms for the binary alphabet. For the PDNAA alphabet case the Raita requires on average 1, 40 character comparisons. This occurs because when the small size alphabet is used it leads to many exact pattern matches in the text and as a result the number of character comparisons tends to be greater than 1. However, when a larger alphabet is used this phenomenon is alleviated according to Figures 7 and 8.

The numbers of character comparisons of the algorithms (such as BM, BMH and Raita) have on average much suitable character comparisons. Furthermore, it must be noted that the number of character comparisons of the BMH and the KMP algorithms is significantly higher when the binary alphabet is used than with any other type of text. It should also be observed that for all those algorithms the number of character comparisons decreases significantly as the pattern length increases. Thus the empirical results support theoretical evidence that the BMH and the Raita algorithms are providing efficient running time in the number of character comparisons.

The number of character comparisons decreases more slowly as the pattern length increases because for long patterns the probability is higher that the character just fetched occurs somewhere in the pattern, and therefore the distance the pattern can be moved forward (if a mismatch occurs) is shortened.

Moreover, it is noticed that the character comparisons of all BMH algorithms are very close to one another results and tend to stabilize to a certain performance measure except for the binary alphabet.

In all cases, it can be seen that the BMH algorithms and Raita have better results. More specifically, the BMG algorithms and the Raita algorithm are much more efficient in terms of number of character comparisons than the remaining

algorithms for small and long patterns respectively.

### *4.2 Results for the Running Time*

Figures 5 to 8 and Tables 1 to 4 show the results for the practical running time for a binary alphabet, an alphabet of size 8, an English alphabet and a PDNAA (molecular sequence ) alphabet respectively, against the pattern length.

We observe that in the case of binary alphabet Raita algorithm requires much more time than any other algorithms. In the case of alphabet of size 8 BM requires much more time whereas in the case of English alphabet and Molecular sequence KMP requires much more time than any of the other evaluated algorithms. This observation agrees with the expected behavior that the computation of the hash values is computationally expensive in terms of machine cycles and so increases the running time of the algorithm. Therefore, this algorithm isn't recommended for text applications.

Analyzing on the basis of empirical results, it is clear that in maximum cases the KMP algorithm is relatively slower than the BM algorithm for almost all pattern lengths with the exception of the binary alphabet.

This behavior support theoretical evidence that the KMP algorithm isn't better than the BM algorithm on the average case. Further, it can also be seen that in all cases the BM and KMP algorithms are slightly slower than the BM and BMH algorithms.

The running time of the KMP and BMH algorithms decreases significantly as the pattern length increases. Moreover, it should be noticed that the BM algorithms produce similar running times i.e. very close to each other in all cases with the exception of the binary alphabet.

In addition, for long patterns the difference between the running times of BM and BMH algorithms are very close to each other, Whereas that of KMP increases in all cases with the exception of the English alphabet. The best performance is analyzed for Raita on the basis of running time.

The most appropriate algorithm identified amongst all is BMH and Raita for small patterns. And Raita algorithm identified the most appropriate for all small to lengthy patterns. The latter observation is valid in all cases with the exception of the binary alphabet.

Finally, it can be seen that in the majority of cases the Raita algorithm has a faster running time than the BM and the BMH algorithms for long patterns. Further, the BMH algorithms have better running times for small patterns except for the binary alphabet.

| M | KMP | BM | BMH | RAITA |
|---|---|---|---|---|
| 2 | 0.001883 | 0.00027 | 0.001251 | 0.001982 |
| 4 | 0.000382 | 0.000171 | 0.000584 | 0.000584 |
| 6 | 0.000353 | 0.000149 | 0.000238 | 0.000524 |
| 8 | 0.000156 | 0.000144 | 0.000257 | 0.00096 |
| 10 | 0.000263 | 0.000154 | 0.000294 | 0.000588 |
| 20 | 0.000233 | 0.000164 | 0.000291 | 0.001083 |
| 30 | 0.000264 | 0.000218 | 0.000296 | 0.000731 |
| 40 | 0.000261 | 0.000359 | 0.000297 | 0.000826 |
| 50 | 0.000266 | 0.000267 | 0.000325 | 0.001122 |
| 60 | 0.000312 | 0.000345 | 0.000337 | 0.00049 |
| 70 | 0.000317 | 0.000386 | 0.000342 | 0.000725 |
| 80 | 0.000338 | 0.000419 | 0.000344 | 0.000454 |
| 90 | 0.000334 | 0.000581 | 0.000537 | 0.000397 |
| 100 | 0.000345 | 0.000514 | 0.000339 | 0.000276 |
| Average | 0.000408 | 0.000296 | 0.000409 | 0.000767 |

**Table 1**: Running times for a binary alphabet

| M | KMP | BM | BMH | RAITA |
|---|---|---|---|---|
| 2 | 0.000365 | 0.000626 | 0.000458 | 0.000274 |
| 4 | 0.000175 | 0.000337 | 0.000173 | 0.000141 |
| 6 | 0.00017 | 0.000176 | 0.000152 | 0.000134 |
| 8 | 0.000274 | 0.000275 | 0.000141 | 0.000117 |
| 10 | 0.000268 | 0.000259 | 0.000139 | 0.000113 |
| 20 | 0.000233 | 0.000208 | 0.000174 | 0.000164 |
| 30 | 0.000206 | 0.000215 | 0.000188 | 0.000185 |
| 40 | 0.000222 | 0.000214 | 0.000204 | 0.000183 |
| 50 | 0.00024 | 0.000265 | 0.000218 | 0.000197 |
| 60 | 0.000334 | 0.000464 | 0.000328 | 0.000229 |
| 70 | 0.000271 | 0.00034 | 0.000274 | 0.000249 |
| 80 | 0.000283 | 0.000286 | 0.000279 | 0.000234 |
| 90 | 0.000287 | 0.000385 | 0.000302 | 0.000321 |
| 100 | 0.0003 | 0.000443 | 0.000301 | 0.000261 |
| Average | 0.000259 | 0.00032093 | 0.000238 | 0.0002 |

**Table 2:** Running times for an alphabet of size 8

| M | KMP | BM | BMH | RAITA |
|---|---|---|---|---|
| 2 | 0.00093 | 0.000398 | 0.000913 | 0.000878 |
| 4 | 0.000459 | 0.000371 | 0.00036 | 0.000311 |
| 6 | 0.000462 | 0.000381 | 0.000338 | 0.0003 |
| 8 | 0.000449 | 0.000337 | 0.00031 | 0.000278 |
| 10 | 0.000517 | 0.000275 | 0.000571 | 0.000328 |
| 20 | 0.000378 | 0.000242 | 0.000236 | 0.000219 |
| 30 | 0.000422 | 0.000292 | 0.000262 | 0.000237 |
| 40 | 0.000553 | 0.000336 | 0.000292 | 0.000254 |
| 50 | 0.00043 | 0.000329 | 0.000283 | 0.000248 |
| 60 | 0.000616 | 0.000555 | 0.000513 | 0.0004 |
| 70 | 0.000745 | 0.000608 | 0.000579 | 0.000577 |
| 80 | 0.001059 | 0.000737 | 0.000502 | 0.000599 |
| 90 | 0.00076 | 0.000672 | 0.000557 | 0.000705 |
| 100 | 0.000967 | 0.0011 | 0.00081 | 0.000689 |
| Average | 0.000625 | 0.00047379 | 0.000466 | 0.00043 |

**Table 3:** Running times for an English alphabet

**Table 4**: Running times for a PDNAA FASTA Sequence Alphabet

| M | KMP | BM | BMH | RAITA |
|---|---|---|---|---|
| 2 | 0.000629 | 0.000386 | 0.000827 | 0.000405 |
| 4 | 0.00051 | 0.00035 | 0.000442 | 0.000209 |
| 6 | 0.000604 | 0.000566 | 0.000233 | 0.000193 |
| 8 | 0.000493 | 0.000603 | 0.000378 | 0.000244 |
| 10 | 0.000328 | 0.000562 | 0.000229 | 0.000208 |
| 20 | 0.000267 | 0.000383 | 0.000394 | 0.000204 |
| 30 | 0.000375 | 0.000351 | 0.000249 | 0.000334 |
| 40 | 0.000596 | 0.00027 | 0.000408 | 0.000241 |
| 50 | 0.000399 | 0.000444 | 0.000276 | 0.000268 |
| 60 | 0.00074 | 0.00034 | 0.000498 | 0.000442 |
| 70 | 0.000596 | 0.000375 | 0.000557 | 0.000312 |
| 80 | 0.000493 | 0.000718 | 0.000575 | 0.000448 |
| 90 | 0.000537 | 0.000756 | 0.000576 | 0.000541 |
| 100 | 0.000458 | 0.000472 | 0.000361 | 0.000553 |
| Average | 0.000501786 | 0.000469714 | 0.000428786 | 0.000329 |

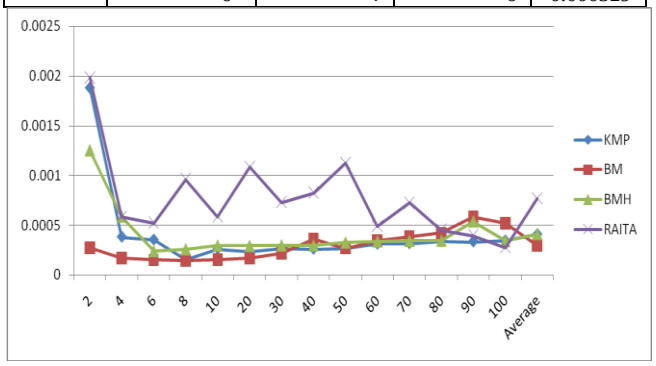



Fig 5: Binary Alphabet

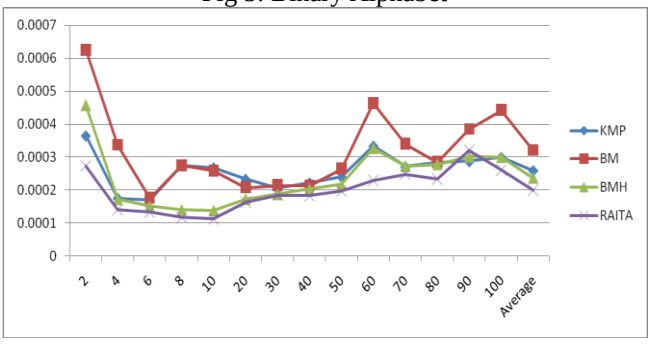



Fig 6: Alphabet of size 8

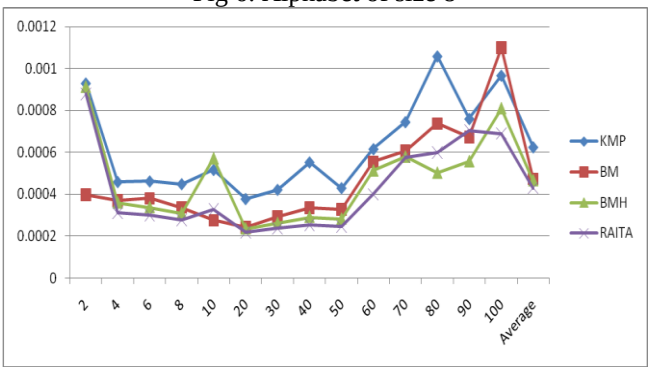



Fig 7: English Alphabet

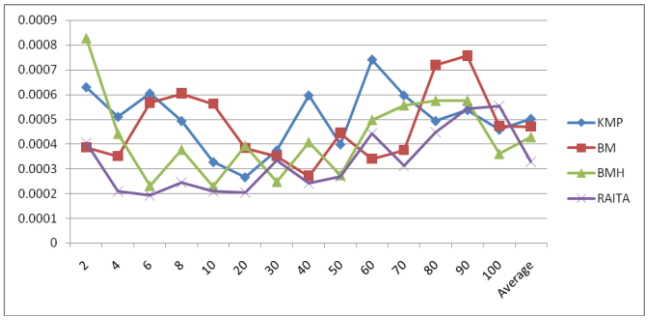



Fig 8: PDNAA FASTA Sequence Alphabet

## V. CONCLUSIONS

This paper describes the concept of approximate string matching algorithms or searching algorithms to find places where one or several patterns are found within a larger text approximately or exact. We focus on carrying out a comparative study of the already existing exact string matching and approximate string matching algorithms with their running time and implementing it in an approximate way by using Levenshtein distance. With this approach users at least get some results and it may happen that the results are proved relevant for them.

This paper described the concept of approximate string matching algorithms. String matching or searching algorithms try to find places where one or several patterns are found within a larger text. This approach focused on approximate string matching algorithms based on edit distance. The purpose is implementing an approximate way using Levenshtein distance. With this approach users are able to arrive at certain predictable results. This approach is advantageous from all other approximate string-pattern matching algorithms which have been inferred to after the analysis.

The conclusion of this paper fall into two main categories: general conclusions regarding the algorithms and their testing procedures, and conclusions relating to the performance of specific algorithms.

The inference drawn out from the general conclusion says that testing the algorithms on four different types of text (binary alphabet, alphabet of size 8, English alphabet and DNA alphabet) indicates that varying parameters such as the pattern length and the alphabet size can produce different performances.

The specific performance conclusions are: It should be noticed that the absolute shapes of the lines on the performance graph are not conclusive. Information can only be derived from the relative positions of the curves for each algorithm at each pattern length. This is because the patterns were chosen at random and obviously the running time is related to how far into the text the pattern occurs. The running times for all the eleven algorithms can be compared at each pattern length because the same type of text and set of patterns were used with each algorithm.

If you plan on direct searching with simple text, the linear algorithm is a proper choice because it produces relatively good running time results despite its striking simplicity. In addition, the BM algorithm has no special memory requirements and needs no preprocessing or complex coding and thus can be surprisingly fast.

Despite its theoretical elegance, the KMP algorithm provides no significant speedup advantage over the BM

algorithm in practice unless the pattern has highly repetitive sub patterns. However the KMP algorithm may be a good choice when the alphabet size is near the text size or when dealing with the binary alphabet.

As far as the variations of the BMH approach we can make the following remarks: Based on empirical results, it is clear that the Raita algorithm is proved to be much faster algorithm in practice than the rest BM, KMP, suffix automata and bit-parallelism algorithms for large alphabets and short patterns.

Therefore it is typically suited for search in the English alphabet. In addition, we must also note that the main disadvantage of BMH, KMP and Raita algorithms is the preprocessing time and the space required, which depends on the alphabet size and/or the pattern size.

It should be noted that for long patterns the running time of the Raita increases because of the preprocessing phase, the time for which is equal to the time for the searching phase. Thus, the Raita algorithm is efficient in theory and practice for small alphabets and long patterns. Therefore, Raita algorithm is a good choice to be used for PDNAA applications (Molecular Subsequences).

## REFERENCES


1) Efficient Matching of Biological Sequences Allowing for Non-overlapping Inversions 2011, Volume 6661/2011, 364-375, DOI: 10.1007/978-3-642-21458-5_31
2) Badoiu M. et al. 2004, "Fast Approximate Pattern Matching with Few Indels via Embeddings," in Proceedings of 15th Annual ACM-SIAM Symposium on Discrete Algorithms, Louisiana, pp. 651-652, 2004.
3) Dan Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, New York, 1997.
4) Navarro, Gonzalo. A Guided Tour to Approximate String Matching. ACM
5) Shivani jain, et al. 2012, " A Relative Study of Pattern Matching Algorithms", Journal of Computing Technologies ISSN 2278 – 3814
6) Shivani jain et al. 2012 "Different Pattern Matching Algorithms with Molecular Sequence in PHP", IJAIR ISSN: 2278-7844
7) Pattern matching and text compression algorithms, Maxime Crochemore 1 Thierry Lecroq2 http://www-igm.univ-mlv.fr/~lecroq/articles/lir9511.pdf
8) Shivani jain et al. 2013 "Multi-Threaded Approximate Pattern Matching Based on Edit Distance", IJAIR Vol. 2 Issue 2 ISSN: 2278-7844
9) Hume A., Sunday D., Fast string searching, Software-Practice and Experience, Vol. 21, No. 11, pp. 1221-1248, 1991.
10) Smith P., Experiments with a very fast substring search algorithm, Software-Practice and Experience, Vol. 21, No. 10, pp. 1065-1074, 1991.