

White Paper



Binary Pattern Matcher

A New General Purpose Algorithm for De-duplication

Ramesh K. Raghunathan Ph.D.

Binary Pattern Matcher is a new and versatile algorithm formulated for diverse applications including string similarities, online duplicate detection, data mining of large data sets, and plagiarism detection. It provides superior recall and precision characteristics in varied situations as compared to other commonly employed alternatives.

Introduction

Pattern matching remains one of the most actively researched areas of computer science with numerous papers still being published. The literature in this field is vast with thousands of relevant papers. These span numerous categories of matching problems including:

- exact string matching – for instance in a word processing program
- approximate (fuzzy) string matching – for instance in handwriting detection and optical character recognition
- largest common substring
- computational biology – for instance in matching nucleotide sequences with missing fragments
- Information retrieval and querying
- Spam filtering and plagiarism detection
- Signal Processing

The effectiveness of any matching algorithm is measured by comparing the precision and recall of the match results [2, 10]. Precision is defined as the number of relevant matches in the returned results while Recall is defined as the number of relevant matches in the returned results among all relevant matches.

Conceptually, for any given algorithm, as tunable parameters are adjusted to improve the precision of the returned results, recall drops. Conversely if the parameters are tuned to improve the number of relevant matches returned, the precision of the results is low. In general, superior algorithms exhibit better recall and precision characteristics across the entire range as compared to alternatives. This is usually expressed as an FScore, which is the harmonic mean of the Precision and Recall.

The performance of any matching algorithm is judged, as is conventional in the literature, via the “O notation” [1]. Thus, a naive exact string matching algorithm that attempts to match a subject string of n characters against a pattern of m characters would use $O(n \times m)$ operations. In addition, superior algorithms strive to minimize storage requirements to the largest extent possible.

This paper focuses on a new general purpose duplicate detection algorithm developed by the author for use in a variety of situations. Existing commonly employed algorithms, are useful only in specific situations, and are not effective in other contexts. In particular, this new algorithm, provides high recall rates without sacrificing too much precision, and is a natural fit for applications that require such characteristics.

Review of Literature

The four classic exact string matching algorithms are a Naïve brute force approach, the Rabin-Karp algorithm, Finite Automata based matchers, and the Knuth-Morris-Pratt algorithm [1], listed in progressively better performing order.

Numerous string similarity measures have been developed that can compare two similar strings and determine a quantitative measure of the degree of similarity usually expressed in the range [0, 1]. These include:

- Edit distance based metrics like Levenstein, Smith-Waterman, NeedlemanWunch (characterized by the number of inserts, deletes, and updates required to transform one string to another) [3]
- Jaro and Jaro-Winkler metric [3, 5]
- Length based metrics like Chapman Length Deviation and Mean Length [3]
- Qgrams [3, 9] (counts of the number of common substrings of length q shared by the pattern and the subject)
- Token based metrics like Block Distance, Cosine Similarity, and Monge-Elkan [3]
- Phonetic matchers like Metaphone and Soundex [7]

Other algorithms include the BLASTA and FASTA family of algorithms used extensively for DNA analysis and gene sequencing applications [8, 9]. Navarro [4] provides an in-depth tour of approximate string matching in a summary paper. The new Binary Pattern Matching Algorithm, though one-dimensional is conceptually similar to a Generalized Hough Transform [11] with the centroid located at the beginning of the pattern.

Matching Algorithm

The flow chart for the Binary Pattern Matcher algorithm is illustrated in Figure 1. The complete algorithm is described in depth in the Mathematical Details section later. First, the patterns are pre-processed by caching the calculated offset of each binary byte in the pattern. The offset is defined as the zero based index of the byte in the pattern. Once the cache has been created, the initialization phase is complete. Subsequently, during the matching phase, an accumulator is initialized for each subject byte array. Each byte in the subject is tested to see if it is part of the offset cache. If so, an index is calculated by subtracting the cached offset value from the byte's index position in the subject byte array. The accumulator is incremented at this index position. At the end the accumulator maxima are scanned to output match results based on how “close” the maxima are to an exact match.

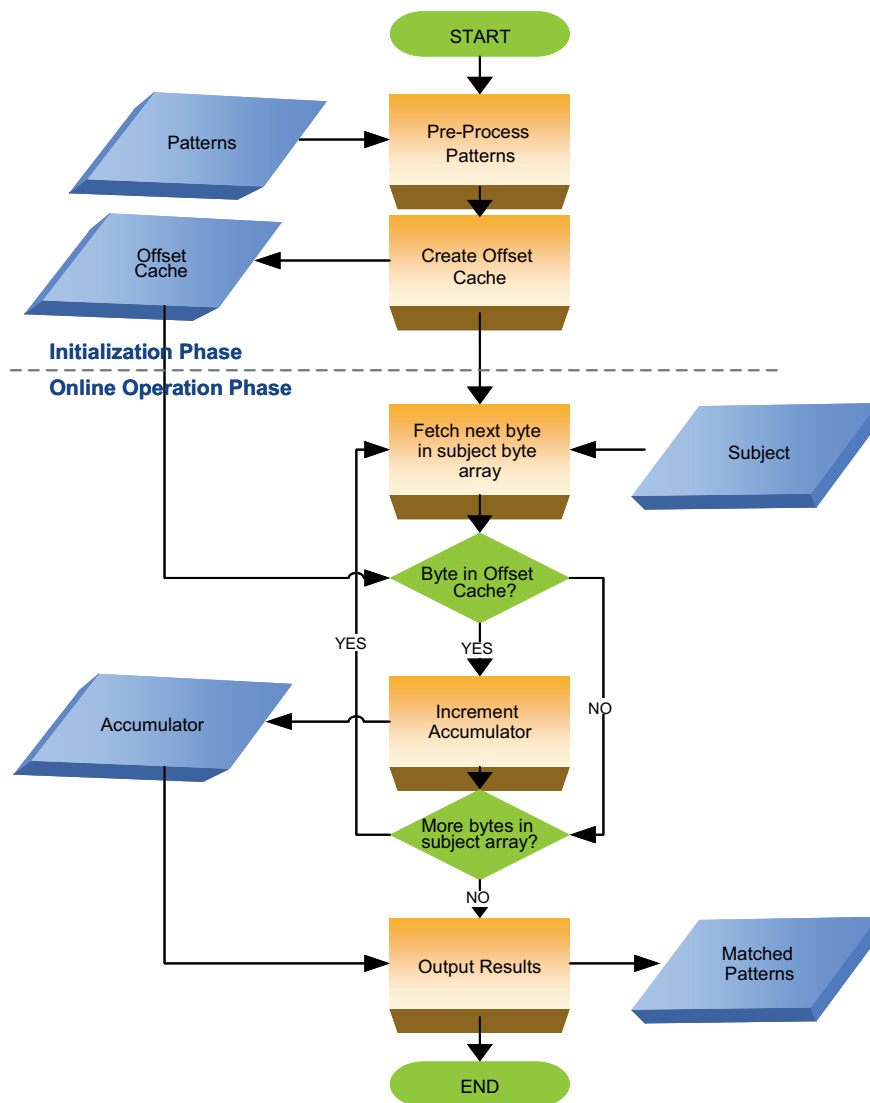


Figure 1: Algorithm Flow

Pattern	I	N	F	O	S	Y	S									
Offsets	0	1	2	3	4	5	6									
Subject	T	E	X	T	I	N	F	O	S	Y	S	T	E	X	T	
Accumulator Contents																
Pass 1 (T)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Pass 2 (E)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Pass 3 (X)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Pass 4 (T)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Pass 5 (I)	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Pass 6 (N)	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0
Pass 7 (F)	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0
Pass 8 (O)	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0
Pass 9 (S)	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0
Pass 10 (Y)	0	0	0	0	6	0	0	0	0	0	0	0	0	0	0	0
Pass 11 (S)	0	0	0	0	7	0	1	0	0	0	0	0	0	0	0	0
Pass 12 (T)	0	0	0	0	7	0	1	0	0	0	0	0	0	0	0	0
Pass 13 (E)	0	0	0	0	7	0	1	0	0	0	0	0	0	0	0	0
Pass 14 (X)	0	0	0	0	7	0	1	0	0	0	0	0	0	0	0	0
Pass 15 (T)	0	0	0	0	7	0	1	0	0	0	0	0	0	0	0	0

Figure 2: Sample Match Illustration

Figure 2 shows the operation of the new algorithm for a simple exact matching exercise. This figure shows the matching algorithm in action for a sample matching scenario given a pattern -- “INFOSYS” and a subject -- “TEXTINFOSYSTEXT”. In the pre-processing stage, the offsets in the pattern were determined as shown in light blue in the figure. Subsequently, for each character in the subject, if the character exists in the offset cache, the accumulator index (determined by subtracting the offset from the character index) value is incremented. The figure shows the accumulator contents in light green after each character in the subject string is processed.

After the subject has been completely processed, the maxima in the accumulator indicate candidate matches. In the example shown, there is a match of all seven characters, shown in red, in the pattern starting at the fifth character in the subject. All instances of the pattern in the subject are detected in one pass by this method. Note that in the above example and in subsequent discussions we compare strings. This is purely for illustrative purposes and the algorithm as described is fully valid for arbitrary byte arrays of patterns and subjects. Implementations will convert inputs to byte arrays using an appropriate encoding scheme for algorithm execution.

Algorithm Analysis

Consider a pattern string p , of length n , and a subject string s , of length m . The initialization phase requires a full scan of p to determine the offsets ($O(n)$ operations). The match phase requires $O(m)$ operations to update the accumulator. Storage requirements include the pattern cache and the accumulator. Note that the illustrations show an accumulator of the same size as the subject string. In a proper implementation, this can be reduced to the size of the pattern, thus allowing the determination of all matches in a large subject string for an optimally small accumulator size.

Duplicate letters in the pattern result in the accumulator contents being updated at each of the offsets for that character. This can be seen in Figure 2 where the letter s is duplicated in the pattern.

The above illustrations describe exact string matches. The objective in this paper is to develop a versatile algorithm for approximate matching with superior precision and recall characteristics. This is accomplished by introducing the following enhancements to the exact matching process:

- Incorporate a tolerance parameter in the accumulator to allow hits in the vicinity of the offset index. For example a tolerance of 1 will result in 3 indices being updated – the offset index and one index on either side of that index. These tolerances allow inserts, deletions, and substitutions to be robustly handled in the matching process. In tests conducted on large datasets, it was found that the tolerance should be increased as the size of the pattern increases up to a maximum value after which further increases fail to improve the matching characteristics.
- Selecting a threshold criterion to signal a match and scaling this to be a function of the pattern size is also an important parameter to control the matching process. Typically, this is normalized to the maximum number of hits a pattern can produce to result in a number in the range [0,1]. This approach has the added advantage of allowing easy comparison with several string similarity measures which define similar criteria to set the threshold for a match.

Object Model Design for Online Matching

This section describes key parts of the author’s object oriented implementation of the algorithm. Implementations of the *Matcher* interface provide the central matching operations of pre-processing and matching. This key interface also maintains references to other key interfaces – a *MatchContext* interface that provides generalized context information for example match thresholds, an *Accumulator* interface that keeps track of hit counts and internally maintains a reference to a *Tolerance* interface implementation that controls the degree of fuzziness of the hit counts, a *Converter* interface implementation that allows content conversion to and from byte arrays, and an *OffsetCache* data structure that maintains the offsets cached after the pre-processing phase, a *Scrubber* implementation to clean input, and a *Encoder* to transform and convert inputs and outputs. Figure 3 shows a UML static class diagram of these interfaces.

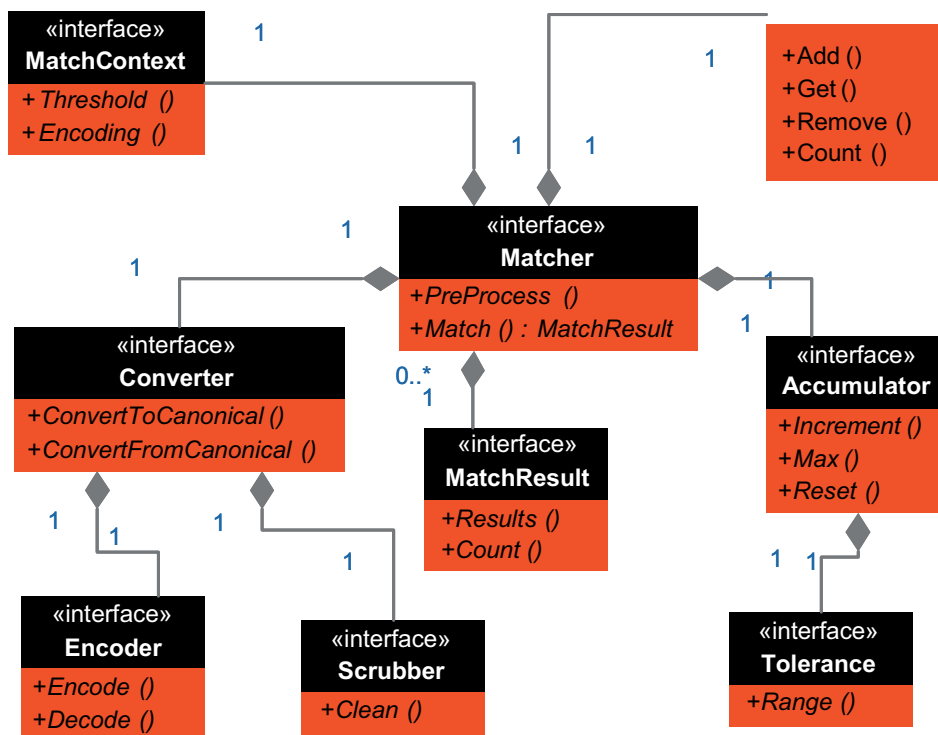


Figure 3: Static Class Diagram of Key Interfaces

Results

This algorithm has been tested in a variety of ways – duplicate detection in large databases, text plagiarism, name and address retrieval, and data mining. As part of one such test, a large database of financial institution names was collected. This constitutes a large subject population of around 80,000 names, to test the matching characteristics of the algorithm, and to compare it to other common alternatives.

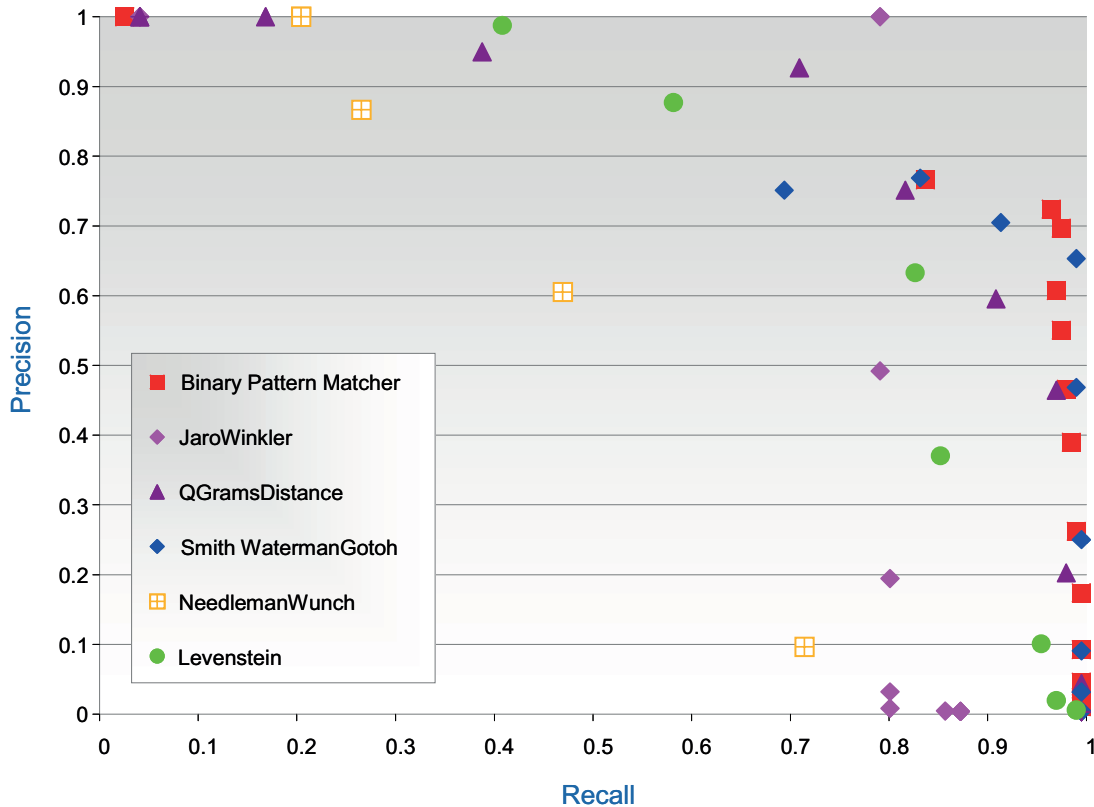


Figure 4: Sample Matching Results

Figure 4 shows the matching results for one test pattern against this subject population. The test population of 80,000 contains about 200 matches relevant to this chosen test pattern. The graph in Figure 4 shows the results of the effectiveness of the Binary Pattern Matcher algorithm in comparison to some of the common alternatives including Jaro-Winkler, Qgrams, Smith-Waterman-Gotoh, NeedlemanWunch, and Levenstein.

Results show that the JaroWinkler approach is skewed heavily towards the beginning of a string, and this fails to retrieve any matches where the pattern is embedded in the interior of a subject string. The NeedlemanWunch and Levenstein methods provides the worst recall and precision characteristics of all the methods tested. QgramsDistance and SmithWatermanGotoh provide better results, but the Binary Pattern Matcher shows uniformly superior results across the entire range of possible application scenarios. For high recall requirements (95 % or better) coupled with moderate to high precision requirements (70 % or better) the Binary Pattern Matcher algorithm should be the algorithm of choice.

The new algorithm has the added advantage of dealing seamlessly with unequally sized patterns and subjects and detecting multiple matches in the subject in one pass.

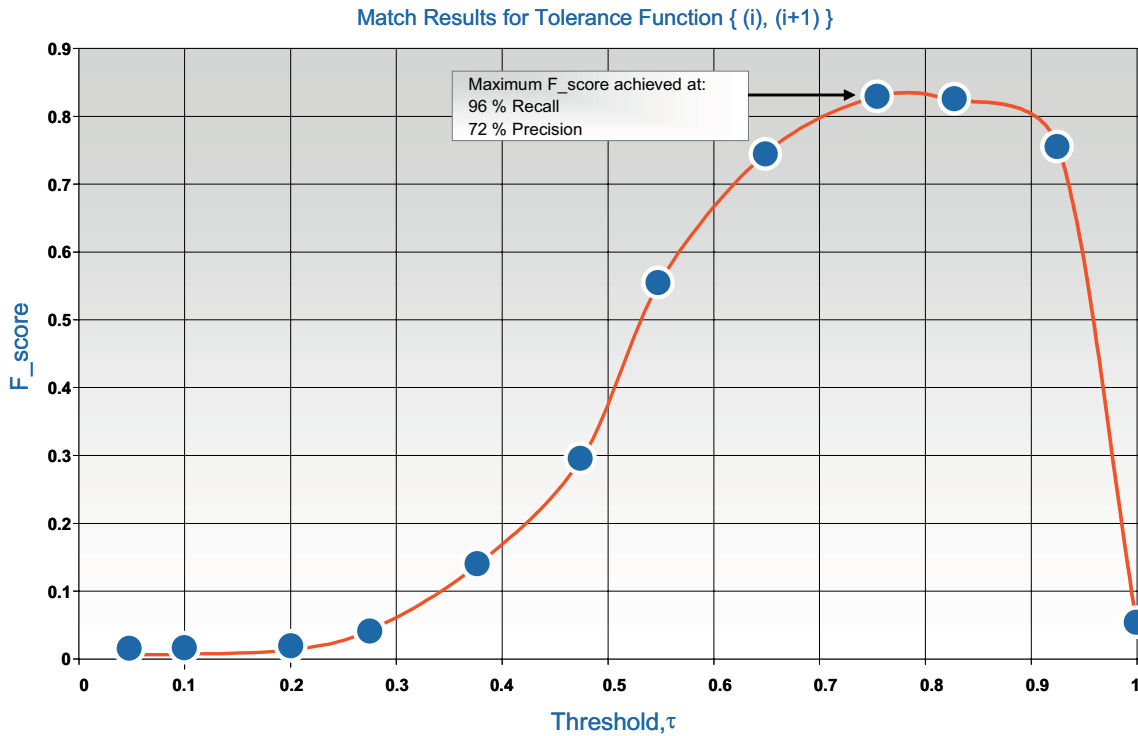


Figure 5: Sample results for a Tolerance Function

Figure 5 shows the Binary Pattern Matching Algorithm behavior as a function of the chosen threshold for a single choice of a tolerance function. The best overall performance occurs at a threshold level of 0.75 which corresponds to a very high recall (96 %) coupled to a high precision (72 %). Setting the threshold to a low value e.g. 0.5, results in a loss of matching effectiveness, as precision plummets. Conversely, setting the threshold to a high value e.g. 1.0, results in a similar loss of matching effectiveness, as recall drops sharply.

Mathematical Details

Define:

- Σ as a finite alphabet for a specified encoding;
- S as a subject array of length n whose elements belong to Σ ;
- P as a pattern array of length m whose elements belong to Σ ;
- O(.) as a many valued offset function of size $\leq m$ mapping each instance of Σ in P to a set of integers;
- $T_m(\cdot)$ as a many valued tolerance function that varies by pattern length m and maps any integer k to a set of integers K;
- τ as a threshold defined in [0, 1];
- A as an (accumulator) array of size n.

The Binary Pattern Matching algorithm can be stated formally as:

1. Given S, m, P, n, Σ , O, A, T_m and τ
2. For each $i = 0$ to $m-1$, update O (a lookup hashtable) by adding i to a (linked) list keyed by $P[i]$
3. For each $i = 0$ to $n-1$, increment A at the non-negative indices defined by:

$$T_m(i - O(S[i])) \mid O(S[i]) \neq null$$

4. Return tf A (and equivalently S) where

$$\frac{A[i]}{m} \geq \tau$$

Sample Tolerance Functions:

- $T_4(i) = \{ (i-1), (i), (i+1) \}$
- $T_{10}(i) = \{ (i+5), (i-4), (i-3), (i-2), (i-1), (i), (i+1), (i+2), (i+3), (i+4), (i+5) \}$
- $T_6(i) = \{ (i-2), (i-1), (i) \}$

Note that setting $T_m(i) = \{ (i) \}$ and $\tau = 1$ corresponds to requiring an exact match.

Definition of some terms from information retrieval theory:

- U as the set of all relevant (“correct”) matches;
- V as the set of retrieved match results;
- Precision, $P = \frac{U \cap V}{V}$
- Recall, $R = \frac{U \cap V}{U}$
- Harmonic Mean of P and R, $F_score = \frac{R \times P}{\left(\frac{R + P}{2}\right)}$

Conclusions

The Binary Pattern Matching algorithm provides a convenient, fast, and easy to implement approach for general purpose duplicate detection and matching. This is useful for a variety of scenarios ranging from matching short string fields like addresses and name fields to arbitrary patterns in large binary subjects. The high recall with only a moderate loss in precision is well suited to for matching applications that can benefit from such behavior. Implementations can range from online matching for cached patterns to large scale database level de-duping via batch operations.

References

1. Introduction to Algorithms, Cormen, Leiserson, Rivest, and Stein (Second Edition, 2001, MIT Press).
2. Data Mining Concepts and Techniques, Han and Kamber (Second Edition, 2006 Elsevier).
3. SimMetrics Library, <http://sourceforge.net/projects/simmetrics>, UK Sheffield University, GNU General Public License.
4. A Guided Tour to Approximate String Matching, Gonzalo Navarro (ACM Computing Surveys, Vol. 33, No. 1, March 2001).
5. Overview of Record Linkage and Current Research Directions, William Winkler (Research Report Series #2006-2, Statistical Research Division, U.S. Census Bureau).
6. Algorithms on strings, trees, and sequences: computer science and computational biology (1997, Cambridge University Press).
7. Apache Commons Codec Project, <http://commons.apaches.org/codecs> (2008).
8. Identification of Common Molecular Subsequences, Smith and Waterman (Journal of Molecular Biology, 1981, 147, 195-197).
9. Ngram Statistics Package, <http://ngram.sourceforge.net> (v1.09 2008).
10. Artificial Intelligence: A Modern Approach (Second Edition 2003, Prentice Hall).
11. Use of the Hough Transformation to Detect Lines and Curves in Pictures (Comm. ACM, Vol. 15, pp 11-15, 1972).

About the Author

Ramesh K. Raghunathan is a Senior Technical Architect with Infosys. He has extensive software consulting and architecture experience in numerous industry verticals and currently focuses on Solution Architecture and Enhanced Delivery (SEED) in the Energy, Utilities, and Services (EUS) vertical.

He has a B-Tech (Hons.) degree in Engineering from IIT Kharagpur, a Master of Science degree from Clarkson University, and a Ph.D. from the State University of New York at Buffalo.



For more information, contact askus@infosys.com

About Infosys

Many of the world's most successful organizations rely on Infosys to deliver measurable business value. Infosys provides business consulting, technology, engineering and outsourcing services to help clients in over 30 countries build tomorrow's enterprise.

For more information about Infosys (NASDAQ:INFY), visit www.infosys.com.