Theoretical Computer Science 92 (1992) 107–117
Elsevier

An approximate string-matching algorithm

Jong Yong Kim and John Shawe-Taylor

Department of Computer Science, Royal Holloway and Bedford New College, University of London, Egham, Surrey, TW20 0EX, UK

Abstract

Kim, J.Y. and J. Shawe-Taylor, An approximate string-matching algorithm, Theoretical Computer Science 92 (1992) 107–117.

An approximate string-matching algorithm is described based on earlier attribute-matching algorithms. The algorithm involves building a trie from the text string which takes time $O(N \log_2 N)$, for a text string of length N. Once this data structure has been built any number of approximate searches can be made for pattern strings of length m. The expected complexity analysis is given for the look-up phase of the algorithm based on certain regularity assumptions about the background language. The expected look-up time for each pattern is $O(m \log_2 N)$. The ideas employed in the algorithm have been shown effective in practice before, but have not previously received any theoretical analysis.

1. The motivation for approximate string-matching

In many everyday situations the same information can be stored or queried in various formats (e.g. prefix (suffix) variation, spelling variation, and synonyms etc.). In these cases, we may not be able to retrieve the information using an exact matching algorithm even if it is held by the system. In other situations, the stored or queried information may have been slightly distorted through a noisy communication channel or human typing error, again making exact matching flawed.

Even more extreme are situations where a perfect match is unlikely to occur such as in image and signal processing, pattern recognition, speech interface, and recognizing bird song, etc. In these cases the same information is hardly ever represented in a unique form.

There are many related problems, such as finding the longest common subsequence in a sequence for molecular biology, which also require string algorithms with a tolerance for minor discrepancies. Though we do not consider these problems in this paper it is felt that the techniques will generalise to many different string algorithms.

0304-3975/92/$05.00 © 1992—Elsevier Science Publishers B.V. All rights reserved

The same approach is also applicable to partial information retrieval in a poorly structured information system or in document searching.

2. Review of previous methods

As approximate string-matching methods have been widely used, there is a large and diffuse literature on the various methods [2, 10]. We divide the methods into two groups. First we consider techniques for which a complete complexity analysis has been made. In the second subsection we review a family of related methods which have proved effective in practice but which lack complexity information. The purpose of this paper is to put an algorithm from the second group on a firmer theoretical basis.

2.1. Algorithms with complexity results

- *Dynamic programming method* [13]

Suppose that a pattern string P is arranged along the top and a text string T down the side in two-dimensional array, and $F(i,j)$ is the minimum number of differences (edit distance) between P and T. A function $F(i,j)$ is calculated iteratively using the recurrence relations below.

$$F(0,0)=0$$

$$F(0,j)=0$$

$$F(i,0)=i$$

$$F(i,j)=\min[F(i-1,j)+1, F(i,j-1)+1, F(i-1,j-1)+d(P_i,T_j)],$$

$$\text{where } d(P_i,T_j)=0 \quad \text{if } P_i=T_j,$$

$$=1 \quad \text{otherwise.}$$

To allow reversal errors between neighbours, the following fourth term is added to the minimization;

$$F(i-2,j-2)+d(P_{i-1},T_j)+d(P_i,T_{j-1}).$$

The dynamic programming method takes $O(mn)$ operations sequentially to produce its best matches, where m is the length of the pattern and n the text.

- *Hypercube algorithms* [5]

This work is essentially a parallelization of the dynamic programming technique. To compute edit distance in a matrix, firstly it uses a *divide and conquer* method to decompose the problem into n^2 blocks on a hypercube, and then these blocks are merged pairwise to form a collection of $n^2/2$ blocks. This is continued until eventually leading to 1 block, which are called the *perimeter-pairs shortest path problem.* The

algorithm shows that the string edit problem can be solved in $O(\log^2 n)$ time on a virtual SIMD hypercube of $O(n^3/\log^2 n)$ processors.

- *Parallel algorithm using suffix tree* [13]

The parallel algorithm runs in two steps. Firstly, it concatenates the text and the pattern to one string and builds the suffix tree of this string. Secondly, it finds all occurrences of the pattern in the text with at most k differences using an alternative dynamic programming method and the *lowest common ancestor* problem in the suffix tree. The suffix tree is built in $O(\log n)$ time on n processors, and searching for the *lowest common ancestor* in the tree takes $O(1)$ time for x parallel queries on x processors. The alternative method employs $(n+k+1)$ processors (one per diagonal) and computes the diagonals within at most $(k+1)$ times in parallel. Therefore, the total complexity of the algorithm is $O(\log n + k)$ to match with text size n and pattern size m with k differences.

2.2. *Attribute matching algorithms*

- *Two-index sequential file* [4]

Make two copies of a file, one in which the keys (strings) are in normal alphabetical order and another in which they are alphabetically ordered by their reverse spelling (e.g. *blaze* comes after *blue*). A misspelled string will probably agree up to half or more of its length with an entry in one of these files. This method will be sensitive to the actual distribution of the strings and no theoretical or empirical results concerning its effectiveness are known.

- *Soundex* [1, 6]

The goal is to transform a string into some code that tends to bring together all variants of the same string. For example, in encoding surnames,

- retain the first letter of the name dropping all the vowels.
- assign the following number to the remaining letters:

$$b, f, p, v \rightsquigarrow 1$$

$$c, g, j, k, q, s, x, z \rightsquigarrow 2, \text{ etc.}$$

- if the two letters with the same code were adjacent in the original name, omit all but the first.
- take the first four in the left-hand side from the above result.

An alternative to the above method: each letter is given a number from 3 to 9 as an information weight (the relative frequency which is defaulted to typical English frequency distribution),

$$a, e, i, n, o, s \rightsquigarrow 3$$

$$d, h, l, r, u \rightsquigarrow 4, \text{ etc.}$$

Based on these weights, if a letter is present in both a pattern and a text string, the weight value is added to the likeness value. But each letter of the alphabet contributes

to the likeness value at most once. This method would be most effective for misspelled names that sounded phonetically like the correct spelling under the assumption that all keys are distinct.

- *Fast approximate string-matching* [7]

This method is based on a binary n-gram table to be used for reducing a search space, prefiltering candidates by a given threshold and, finally, a similarity measure based on the Levenshtein metric to give a matched set of strings within the threshold. It is very similar to our method in its overall matching procedures except for the data structures. The table is drastically reduced by overloading (superimposed coding) strings in horizontal and n-grams in vertical. In this way, a significant saving in space is made and this in turn reduces elapsed times by reducing I/O loading. However, no theoretical background is supported. Our analysis indicates why this is such an effective approach.

- *Positional (nonpositional) binary n-gram statistics* [8, 12]
  - *Construction of bit matrices*: For each dictionary word in turn, any n characters could be extracted to form the n-grams. For each n-grams x a "1" is entered in a bit matrix at the address computed by

$$26^n X + 26^{n-1} x_0 + \cdots + x_{n-1}, \qquad (*)$$

    where $X = 0$ for a nonpositional n-gram and is, otherwise, the sequence number. By a random superimposed coding, the storage occupied by the bit matrix can be reduced.
  - *Error detection*: The method tests whether a given word is not in the dictionary by using the stored bit matrix: for each of the n-grams of the word, use (*) to address one bit. If any such bit is 0 then the given input word cannot be in the dictionary.
  - *Error correction*: Let us assume that R of the n-grams detects an error; for the rth such n-gram $N_{i,\ldots,j}$, construct the subset $\beta_r = (i, \ldots, j)$. This implies that an error must have occurred at a position from i to j, or at more than one position. If only a single symbol is in error, then there must be some position i that occurs in every β_r. Thus, we need only the intersection of these subsets $\beta = \bigcap_r^l \beta_r$, $r = 1, \ldots, l$. In n errors, it can be determined by enumerating $\binom{l}{n}$ pairs of positions that could involve n errors and checking whether the positions cover all β_r cases, where l is the number of β_r.

This method is limited to only replacement errors such as might occur in an optical character recognition system (OCR). If there is an error in more than one position, it is difficult to fix the error positions. But the computational time of the method is independent of the word and dictionary size. The article shows the error detection, error correction rate and density of the binary matrix on various sets of positional (nonpositional) n-grams. Especially, positional *tri*-grams show how the error, reject, and correction rates relate to the size of the word set.

The purpose of this paper is to introduce an approximate string-matching algorithm adapted from [7] above and to give an expected complexity analysis for the

look-up phase of the algorithm. This is estimated to be $O(m \log N)$ for a suitable choice of parameters for text size N and pattern size m.

3. Concept of an n-gram

An n-gram is an n character contiguous substring of a given string. Hence, for a string $w_1 w_2 \ldots w_N$ with N characters there are at most $N-n+1$ distinct n-grams, $w_j w_{j+1} \ldots w_{j+n-1}$, for $j=1,\ldots,N-n+1$ (Fig. 1).

Further to this traditional definition of an n-gram, it can be helpful to group n-grams together which are likely to have arisen as a result of simple errors, such as swapping characters. So, for example, in the case when $n=3$ the tri-grams may be identified by the set of characters they contain, completely ignoring the ordering information. It should be mentioned that we are not fixing the value of n at this stage. Indeed, in the analysis it becomes necessary to choose n according to the size of the database considered, in order to obtain the optimal bounds on the complexity of the look-up procedure.

One of the key assumptions we will make in the analysis is that the number $P(n)$ of n-gram groups grows exponentially in n. This will be considered in the context of a background language, where there is assumed to be an underlying probability $p_i > 0$ for each n-gram group g_i, where $g_1, \ldots, g_{P(n)}$ are the $P(n)$ n-gram groups which do

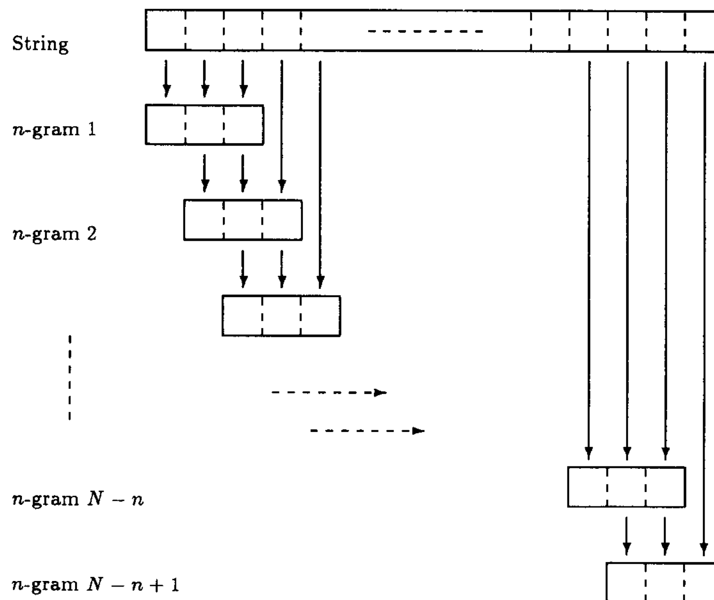

Fig. 1. Diagram of the n-gram structure ($n=3$).

occur. The other assumption which will be made about the background language is that the probabilities p_i are bounded by some constant K times $1/P(n)$ independently of n. This is saying that as we choose larger and larger n-grams, no particular string or set of strings becomes excessively dominant.

4. Data structure

In the algorithm, we build two kinds of data structures which store the sections of words and n-grams they contain. The first is a division of the dictionary into sections. Each section will contain the same number l characters, though this number is not specified at this stage. There are M sections and $N=Ml$ characters in all. If the boundaries between sections are not clearly defined it may be necessary to overlap sections, but in order to simplify the analysis, this will not be considered.

The second structure is the "n-gram tree". This is a trie whose nodes consist of an array of pointers indexed by the alphabet. To find the leaf node corresponding to a particular n-gram, we simply follow the pointers for each of the characters of the n-gram in turn. The leaf node will itself contain a pointer to the list of sections for the group containing the given n-gram.

5. Building the trie

Given a value for n and l, we can build the tree by simply passing through the dictionary once. At each stage we will be processing at most n n-grams and we will need pointers into the trie for each of them. As the next character is taken each of the pointers must move down the tree by using the appropriate pointer from its current node. If necessary, a new node will be created. The work taken will be constant for each individual pointer, and so will be proportional to n overall. When an n-gram is completed we must install the number of the current section into the list for the n-grams group. We will assume that we can determine the group of an n-gram when its leaf node is first created using a method which takes $O(n)$ time. Note that by working through the file and always inserting the section numbers at the beginning of the list, we ensure that the numbers are in reverse order without duplication.

The overall time to perform the construction of the trie is, therefore, $O(Nn)$, where N is the number of characters in the dictionary.

6. Searches for approximate strings

In order to look for a pattern string, we again follow the paths through the tree corresponding to its n-grams and the lists of sections corresponding to these tree nodes are merged adding the total numbers of occurrences for sections containing one of the n-grams.

Any section which contains more of the n-grams than a user-defined threshold is passed to the second stage for exact matching. Since the set of n-grams of a string corrupted through insertion, deletion or reversal errors is slightly altered, the algorithm will include searching sections containing only a proportion of the full set of n-grams. The method ensures, however, that any section containing the string or a slightly corrupted version will necessarily be searched, so that no string that is a sufficiently good match will be overlooked. It is expected that the number of sections to be searched will be, typically, quite small. The complexity analysis estimates this number of sections.

In summary, the overall algorithm is divided into three main parts as follows.

- *Searching for the lists of sections which contain any of the n-grams contained in the sought string*

The n-gram tree is accessed to find the needed lists, which are kept separate so that the number of n-grams which the individual sections contain can be calculated during the next phase.

- *Netting sections to remove sections under the threshold*

Here the lists of sections corresponding to the individual n-grams are collated. A new list of section numbers is created being those sections which occur in more than a given fraction of the original lists. The key observation concerning the new list is that any section which contains the string or a slightly corrupted version of it must be contained in this list. This is because such a section will contain most of the sought n-grams and so will occur in more than the required fraction of the individual n-gram lists.

- *Approximate string-matching between the pattern string and the text strings in the sections*

Each of the sections found at the previous stage has to be examined to see if the string or some slight corruption of it occurs in it. Clearly, it is possible that sections could have all of the sought n-grams and yet not contain the required string. It is, therefore, necessary to check for the string using some standard approximate string-matching technique. In the complexity analysis this is assumed to take time proportional to the product of the length of the sought string and the length of the section.

7. Complexity analysis

We have already assessed the complexity of building the trie as $O(Nn)$ in the previous sections. This clearly depends on the choice of n. We begin this section by considering the expected complexity of the look-up procedure. This analysis will indicate an appropriate choice for n, hence indicating the actual complexity of building the trie.

Each section of the dictionary has $(l-n+1)$ n-grams. The n-grams will be partitioned into sets of related strings, which could have arisen from each other as a result of typographical or other errors. The choice of these partitions will be crucial to the

error-correcting performance of the algorithm, but does not concern the complexity analysis in so far as the conditions outlined below are satisfied.

As described above, let there be $P(n)$ n-gram partitions, $g_1, \ldots, g_{P(n)}$ with corresponding probabilities (relative frequencies) $p_1, \ldots, p_{P(n)}$. We denote this distribution by D. We make the following further assumptions about these probabilities:

- $p_i \leqslant K/P(n)$ for some constant K.
- $P(n)$ grows exponentially with n, i.e. $P(n) \geqslant \alpha^n$ for some $\alpha > 1$.

These assumptions relate to the choice of partitioning and the properties of the underlying language. In the case of natural language the second can be intuitively justified by considering the L sentences of a particular length k. To make sentences of length $2k$, we can combine any two of the L sentences of length k. Hence, there are more than L^2 sentences of length $2k$. This is consistent with exponential growth in the number of n-grams.

We let P^* denote the following quantity:

$$P^* = \sum_{i=1}^{P(n)} p_i^2,$$

where we suppress the dependency on n in the notation. This implies

$$\begin{aligned} P^* &= \sum_{i=1}^{P(n)} p_i^2 \leqslant \frac{K}{P(n)} \sum_{i=1}^{P(n)} p_i \\ &= \frac{K}{P(n)} \leqslant \frac{K}{\alpha^n} \\ &\leqslant K\beta^n, \end{aligned}$$

where $\beta = 1/\alpha < 1$.

With a sought string of length m, we have $(m+1-n)$ n-grams. Since for each n-gram we simply make n moves down the trie, the expected complexity of accessing the n-gram tree is

$$O(m(n+E)), \tag{1}$$

where E is the expected number of sections in which an n-gram will occur. It is assumed that the section numbers are stored in a linked list, so that access time is bounded by the number of entries. As the list is accessed the number of occurrences in the individual sections can also be calculated by keeping the sections in lists indexed by the current number of the n-grams they contain.

By the definition of probability as relative frequency, the expected number of occurrences of this n-gram in the $N = lM$ characters of the dictionary is lMp_i. The

number E can be bounded above by the expected number of occurrences of an n-gram chosen randomly according to the distribution D. Hence,

$$\begin{aligned} E &\leqslant \sum_{i=1}^{P(n)} p_i(lp_iM) \\ &\leqslant lM \sum_{i=1}^{P(n)} p_i^2 \\ &\leqslant NP^*. \end{aligned} \tag{2}$$

Finally, we must search any sections for which the threshold is surpassed. Let S be the expected number of such sections. Then this will take

$$\mathrm{O}(Slm). \tag{3}$$

Suppose the threshold proportion is θ, then given $(m+1-n)$ n-grams we require to estimate the probability that $\theta(m+1-n)$ of these occur in a section. By the above the probability that a randomly chosen n-gram occurs in a section in P^*l. Hence, the probabilities that more than θm n-grams occur independently in a section which does not contain the sought string or a corruption of it is

$$\begin{aligned} S/M &\leqslant \sum_{i=\theta m'}^{m'} \binom{m'}{i} (lP^*)^i (1-lP^*)^{m'-i} \\ &\leqslant (lP^*)^{\theta m'} \sum_{i=\theta m'}^{m'} \binom{m'}{i} \\ &\leqslant (lP^*)^{\theta m'} \sum_{i=0}^{m'(1-\theta)} \binom{m'}{i} \\ &\leqslant (lP^*)^{\theta m'} \left(\frac{em'}{m'(1-\theta)}\right)^{m'(1-\theta)} \\ &\leqslant \left((lP^*)^{\theta} \left(\frac{e}{1-\theta}\right)^{1-\theta}\right)^{m'} \\ &\leqslant r^{m'}, \end{aligned}$$

where

$$r=(lP^*)^{\theta}\left(\frac{e}{1-\theta}\right)^{1-\theta} \quad \text{and} \quad m'=m+1-n.$$

Hence, the expected number of sections where the threshold is exceeded is

$$S \leqslant Mr^{m'}. \tag{4}$$

We can, therefore, estimate the expected time-searching sections using (3) and (4),

$$\mathrm{O}(Slm) \leqslant \mathrm{O}(Nmr^{m}). \tag{5}$$

From (1), (2) and (5), the expected access time is, therefore,

$$O(m(n+NP^*+Nr^m)), \tag{6}$$

where the individual parts are derived as follows:

- mn: time accessing the n-gram tree,
- mNP^*: time accessing the lists of sections and collating them,
- mNr^m: time for the search for the string within the sections exceeding the threshold.

We now assume a choice of $\theta=0.5$ and make the following choices for the size n of the n-grams and size l of the sections into which the text is divided:

$$n=(\log_2 N-\log_2\log_2 N)/\log_2\alpha,$$

$$l=P^{*2/m-1}/2e.$$

In order to make these choices we would need to know α though this can be estimated by simply building the tree. For the optimal choice the number of n-gram groups should be $N/\log_2 N$. This is because

$$\begin{aligned}\log_2\beta^n&=\log_2\frac{1}{\alpha^n}\\&=-n\log_2\alpha\\&=-\frac{(\log_2 N-\log_2\log_2 N)}{\log_2\alpha}\log_2\alpha\\&=\log_2\frac{\log_2 N}{N},\end{aligned}$$

giving $\alpha^n=N/\log_2 N$. This also means that $P^*\leqslant(K\log_2 N)/N$.

To choose l we should also know P^* and the size m of the look-up strings. However, we may safely take

$$l=\frac{P^{*2/n}}{2eP^*}.$$

This is a slightly smaller size of l. In this case,

$$r^m=P^{*m/n}\leqslant P^*,$$

since we must always use look-up strings which are longer than a single n-gram. Combining all of the above, we obtain the following complexity bound from (6) above:

$$\begin{aligned}O(m(n+NP^*+Nr^m))&\leqslant O\left(m\left(\frac{\log_2 N}{\log_2\alpha}+2NP^*\right)\right)\\&\leqslant O\left(m\left(\frac{\log_2 N}{\log_2\alpha}+2K\log_2 N\right)\right)\\&=O(m\log_2 N).\end{aligned}$$

8. Conclusion

This paper has considered an algorithm which uses n-grams to estimate which sections in a large text may contain a pattern string. These sections can then be searched by standard approximate string-matching algorithms for the particular pattern. This algorithm has been shown to perform very fast in practice [7], a fact which a standard worst-case analysis would be unable to explain. This paper has taken the novel approach of attempting an expected complexity analysis under certain reasonable statistical assumptions about the language from which the text string has been drawn. Using these assumptions a very impressive bound has been obtained on the complexity, which reflects the performance observed in experiments.

A similar kind of analysis is performed on a substring search algorithm in a paper by Shawe-Taylor [11]. This paper also uses n-gram techniques. They appear to be a very promising approach to substring problems, though analysis of their performance will necessarily rely on statistical assumptions.

References

[1] L. Davidson, Retrieval of misspelled names in an airlines passenger record system, *Comm. ACM* **5** (1962) 169–171.

[2] G.R. Dowling and P. Hall, Approximate string matching, *ACM Comput. Surveys* **12** (4) (1980) 381–402.

[3] O.H. Ibarra, T.C. Pong and S.M. Sohn, String Processing on the Hypercube, *IEEE Trans. Acoust. Speech Signal Process.* **38** (1) (1990) 160–164.

[4] D.E. Knuth, *Sorting and Searching* (Addison-Wesley, Reading, MA, 1973).

[5] G.H. Landau and U. Vishkin, Fast parallel and serial approximate string-matching, *J. Algorithms* **10** (1989) 157–169.

[6] A.B. Michael, Automatic correction to misspelled names: a fourth generation language approach, *Comm. ACM* (1987) 224–228.

[7] O. Owolabi and D.R. McGregor, Fast approximate string-matching, *Software-Practice and Experience* **18** (4) (1988) 387–393.

[8] E.M. Riseman and A.R. Hanson, A contextual postprocessing system for error correction using binary n-grams, *IEEE Trans. Comput.* **5** (1974) 480–493.

[9] D. Sankoff and J.B. Kruskal, *Time Warps, String Edit, and Macromolecules: The Theory and Practice of Sequence Comparison* (Addison-Wesley, Reading, MA, 1983).

[10] S.N. Srihari, ed., Computer text recognition and error correction, Tutorial, IEEE Computer Society Press, 1984.

[11] John Shawe-Taylor, Fast string matching in a stationary ergodic source, Tech. Report RHBNC, University of London CSD-TR-633, 1990.

[12] J.R. Ullmann, A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words, *Comput. J.* **20** (2) (1977) 141–147.

[13] R.A. Wagner and M.J. Fischer, The string-to-string correction problem, *J. ACM* **21** (1974) 168–178.