Approximate string matching

From Wikipedia, the free encyclopedia

In computer science, **approximate string matching** (often colloquially referred to as **fuzzy string searching**) is the technique of finding strings that match a pattern approximately (rather than exactly). The problem of approximate string matching is typically divided into two sub-problems: finding approximate substring matches inside a given string and finding dictionary strings that match the pattern approximately.

Contents

Overview

The closeness of a match is measured in terms of the number of primitive operations necessary to convert the string into an exact match. This number is called the edit distance between the string and the pattern. The usual primitive operations are:[1]

- insertion: *cot* → *co**a**t*
- deletion: *co**a**t* → *cot*
- substitution: *co**a**t* → *co**s**t*

These three operations may be generalized as forms of substitution by adding a NULL character (here symbolized by *) wherever a character has been deleted or inserted:

- insertion: *co***t* → *co**a**t*
- deletion: *co**a**t* → *co***t*
- substitution: *co**a**t* → *co**s**t*

Some approximate matchers also treat *transposition*, in which the positions of two letters in the string are swapped, to be a primitive operation. Changing *cost* to *cots* is an example of a transposition.[2]

Different approximate matchers impose different constraints. Some matchers use a single global unweighted cost, that is, the total number of primitive operations necessary to convert the match to the pattern. For example, if the pattern is *coil*, *foil* differs by one substitution, *coils* by one insertion, *oil* by one deletion, and *foal* by two substitutions. If all operations count as a single unit of cost and the limit is set to one, *foil*, *coils*, and *oil* will count as matches while *foal* will not.

Other matchers specify the number of operations of each type separately, while still others set a total cost but allow different weights to be assigned to different operations. Some matchers permit separate assignments of limits and weights to individual groups in the pattern.

Problem formulation and algorithms

One possible definition of the approximate string matching problem is the following: Given a pattern string $P = p_1 p_2 ... p_m$ and a text string $T = t_1 t_2 \dots t_n$, find a substring $T_{j',j} = t_{j'} \dots t_j$ in T, which, of all substrings of T, has the smallest edit distance to the pattern P.

A brute-force approach would be to compute the edit distance to P for all substrings of T, and then choose the substring with the minimum distance. However, this algorithm would have the running time $O(n^3 m)$

A better solution, which was proposed by Sellers [3], relies on dynamic programming. It uses an alternative formulation of the problem: for each position j in the text T and each position i in the pattern P, compute the minimum edit distance between the i first characters of the pattern, P_i, and any substring $T_{j',j}$ of T that ends at position j.

For each position j in the text T, and each position i in the pattern P, go through all substrings of T ending at position j, and determine which one of them has the minimal edit distance to the i first characters of the pattern P. Write this minimal distance as $E(i, j)$. After computing $E(i, j)$ for all i and j, we can easily find a solution to the original problem: it is the substring for which $E(m, j)$ is minimal (m being the length of the pattern P.)

Computing $E(m, j)$ is very similar to computing the edit distance between two strings. In fact, we can use the Levenshtein distance computing algorithm for $E(m, j)$, the only difference being that we must initialize the first row with zeros, and save the path of computation, that is, whether we used $E(i - 1, j)$, E(i,$j - 1$) or $E(i - 1, j - 1)$ in computing $E(i, j)$.

In the array containing the $E(x, y)$ values, we then choose the minimal value in the last row, let it be $E(x_2, y_2)$, and follow the path of computation backwards, back to the row number 0. If the field we arrived at was $E(x_1, 0)$, then $T[x_1 + 1] \dots T[y_2]$ is a substring of T with the minimal edit distance to the pattern P.

Computing the $E(x, y)$ array takes $O(mn)$ time with the dynamic programming algorithm, while the backwards-working phase takes $O(n + m)$ time.

On-line versus off-line

Traditionally, approximate string matching algorithms are classified into two categories: on-line and off-line. With on-line algorithms the pattern can be processed before searching but the text cannot. In other words, on-line techniques do searching without an index. Early algorithms for on-line approximate matching were suggested by Wagner and Fisher[4] and by Sellers. [5] Both algorithms are based on dynamic programming but solve different problems. Sellers' algorithm searches approximately for a substring in a text while the algorithm of Wagner and Fisher calculates Levenshtein distance, being appropriate for dictionary fuzzy search only.

On-line searching techniques have been repeatedly improved. Perhaps the most famous improvement is the bitap algorithm (also known as the shift-or and shift-and algorithm), which is very efficient for relatively short pattern strings. The Bitap algorithm is the heart of the Unix searching utility agrep. A review of on-line searching algorithms was done by G. Navarro.[6]

Although very fast on-line techniques exist, their performance on large data is unacceptable. Text preprocessing or indexing makes searching dramatically faster. Today, a variety of indexing algorithms have been presented. Among them are suffix trees[7], metric trees[8] and n-gram methods.[9][10] A detailed survey of indexing

techniques that allows one to find an arbitrary substring in a text is given by Navarro *et al.*[11]. A computational survey of dictionary methods (i.e., methods that permit finding all dictionary words that approximately match a search pattern) is given by Boytsov [12].

Applications

The most common application of approximate matchers until recently has been spell checking.[13] With the availability of large amounts of DNA data, matching of nucleotide sequences has become an important application.[14] Approximate matching is also used to identify pieces of music from small snatches and in spam filtering.[15]

See also

- String metric
- Locality-sensitive hashing
- Needleman–Wunsch algorithm
- Smith–Waterman algorithm
- Levenshtein distance
- Concept Search
- Approximate matching with addition of regular expressions ability
- Regular expressions for non-fuzzy (exact) matching
- Metaphone
- Soundex
- Agrep
- Plagiarism detection

External Links

- An extension of Ukkonen's enhanced dynamic programming ASM algorithm (http://berghel.net/publications/asm/asm.php), Hal Berghel, University of Arkansas, and David Roach, Acxiom Corporation
- Flamingo Project (http://flamingo.ics.uci.edu)
- Efficient Similarity Query Processing Project (http://www.cse.unsw.edu.au/~weiw/project/simjoin.html) with recent advances in approximate string matching based on an edit distance threshold.
- Fuzzy Matcher (http://www.zhihuita.org/service/fuzzy_matcher) an online approximate string matching service
- StringMetric project (http://rockymadden.com/stringmetric/) a Scala library of string metrics and phonetic algorithms
- Natural project (https://github.com/NaturalNode/natural) a JavaScript natural language processing library which includes implementations of popular string metrics

References

- ^ Baeza-Yates R, Navarro G (June 1996). "A faster algorithm for approximate string matching". In Dan Hirchsberg, Gene Myers. *Combinatorial Pattern Matching (CPM'96), LNCS 1075*. Irvine, CA. pp. 1–23.
- ^ Baeza-Yates R, Navarro G. "Fast Approximate String Matching in a Dictionary" (http://reference.kfupm.edu.sa/content/f/a/fast_approximate_string_matching_in_a_di_105948.pdf). *Proc. SPIRE'98*. IEEE CS Press. pp. 14–22.

- ^ Boytsov, Leonid (2011). "Indexing methods for approximate dictionary searching: Comparative analysis". *Jea Acm* **16** (1): 1–91. doi:10.1145/1963190.1963191 (http://dx.doi.org/10.1145%2F1963190.1963191).
- ^ Cormen, Thomas; Leiserson, Rivest (2001). *Introduction to Algorithms* (2nd ed.). MIT Press. pp. 364–7. ISBN 0-262-03293-7.
- ^ Galil, Zvi; Apostolico, Alberto (1997). *Pattern matching algorithms*. Oxford [Oxfordshire]: Oxford University Press. ISBN 0-19-511367-5.
- ^ Gusfield, Dan (1997). *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge, UK: Cambridge University Press. ISBN 0-521-58519-8.
- ^ Myers G (May 1999). "A fast bit-vector algorithm for approximate string matching based on dynamic programming". *Journal of the ACM* **46** (3): 395–415. doi:10.1145/316542.316550 (http://dx.doi.org/10.1145%2F316542.316550).
- ^ Navarro, Gonzalo (2001). "A guided tour to approximate string matching". *ACM Computing Surveys* **33** (1): 31–88. doi:10.1145/375360.375365 (http://dx.doi.org/10.1145%2F375360.375365). CiteSeerX: 10.1.1.96.7225 (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.96.7225).
- ^ Navarro, Gonzalo, Ricardo Baeza-Yates, E. Sutinen and J. Tarhio (2001). "Indexing Methods for Approximate String Matching" (http://www.dcc.uchile.cl/~gnavarro/ps/deb01.pdf). *IEEE Data Engineering Bulletin* **24** (4): 19–27.
- ^ Sellers, Peter H. (1980). "The Theory and Computation of Evolutionary Distances: Pattern Recognition". *Journal of Algorithms* **1** (4): 359–73. doi:10.1016/0196-6774(80)90016-4 (http://dx.doi.org/10.1016%2F0196-6774%2880%2990016-4).
- ^ Skiena, Steve (1998). *Algorithm Design Manual* (1st ed.). Springer. ISBN 978-0-387-94860-7.
- ^ Ukkonen E (1985). "Algorithms for approximate string matching". *Information and Control* **64**: 100–18. doi:10.1016/S0019-9958(85)80046-2 (http://dx.doi.org/10.1016%2FS0019-9958%2885%2980046-2).
- ^ Wagner R, Fischer M (1974). "The string-to-string correction problem" (http://portal.acm.org/citation.cfm?id=321811). *Journal of the ACM* **21**: 168–73. doi:10.1145/321796.321811 (http://dx.doi.org/10.1145%2F321796.321811).
- ^ J. Zobel, P. Dart. Finding approximate matches in large lexicons. Software-Practice & Experience 25(3), pp 331–345, 1995.