String Extravaganza

INF 3800/INF4800

2013.02.06

“How do you represent big dictionaries in memory?”

“And what are some of the applications?”

Binary Search

Example

```
1  //============================================================ binarySearch
2  /** Binary search of sorted array.  Negative value on search failure.
3   *  The upperbound index is not included in the search.
4   *  This is to be consistent with the way Java in general expresses ranges.
5   *  The performance is O(log N).
6   *  @param sorted Array of sorted values to be searched.
7   *  @param first Index of first element to serach, sorted[first].
8   *  @param upto  Index of last element to search, sorted[upto-1].
9   *  @param key   Value that is being looked for.
10  *  @return      Returns index of the first match, or or -insertion_position
11  *               -1 if key is not in the array. This value can easily be
12  *               transformed into the position to insert it.
13  */
14 public static int binarySearch(int[] sorted, int first, int upto, int key) {
15
16     while (first < upto) {
17         int mid = (first + upto) / 2;  // Compute mid point.
18         if (key < sorted[mid]) {
19             upto = mid;     // repeat search in bottom half.
20         } else if (key > sorted[mid]) {
21             first = mid + 1;  // Repeat search in top half.
22         } else {
23             return mid;     // Found it. return position
24         }
25     }
26     return -(first + 1);    // Failed to find key
27 }
```

Binary Search, cont.

```
1   public static int binarySearch(String[] sorted, String key) {
2       int first = 0;
3       int upto  = sorted.length;
4   
5       while (first < upto) {
6           int mid = (first + upto) / 2;  // Compute mid point.
7           if (key.compareTo(sorted[mid]) < 0) {
8               upto = mid;       // repeat search in bottom half.
9           } else if (key.compareTo(sorted[mid]) > 0) {
10              first = mid + 1;  // Repeat search in top half.
11          } else {
12              return mid;       // Found it. return position
13          }
14      }
15      return -(first + 1);      // Failed to find key
16  }
```

Binary Search, cont.

- Membership checking in $O(log_2(n))$
  - Are the $O(1)$ methods you know of, e.g., hashing techniques, always better?

- What about prefix searches?
  - E.g., "comp*" for {"computation", "computer", …}.
  - Note how many data structures are "prefix friendly"
    - E.g., sorted arrays, trees, tries, state machines.

- Prefix lookups can help solve harder lookup problems
  - Many thornier searches can be cleverly reduced to one or more prefix searches, possibly with some post-processing added.

Suffix Arrays

Suffix arrays:
A new method for on-line string searches

Udi Manber[1]
Gene Myers[2]

Department of Computer Science
University of Arizona
Tucson, AZ 85721

May 1989
Revised August 1991

Abstract

A new and conceptually simple data structure, called a suffix array, for on-line string searches is introduced in this paper. Constructing and querying suffix arrays is reduced to a sort and search paradigm that employs novel algorithms. The main advantage of suffix arrays over suffix trees is that, in practice, they use three to five times less space. From a complexity standpoint, suffix arrays permit on-line string searches of the type, ''Is W a substring of A?'' to be answered in time $O(P + \log N)$, where P is the length of W and N is the length of A, which is competitive with (and in some cases slightly better than) suffix trees. The only drawback is that in those instances where the underlying alphabet is finite and small, suffix trees can be constructed in $O(N)$ time in the worst case, versus $O(N \log N)$ time for suffix arrays. However, we give an augmented algorithm that, regardless of the alphabet size, constructs suffix arrays in $O(N)$ ***expected*** *time, albeit with lesser space efficiency. We believe that suffix arrays will prove to be better in practice than suffix trees for many applications.*

1. Introduction

Finding all instances of a string W in a large text A is an important pattern matching problem. There are many applications in which a fixed text is queried many times. In these cases, it is worthwhile to construct a data structure to allow fast queries. The *Suffix tree* is a data structure that admits efficient on-line string searches. A suffix tree for a text A of length N over an alphabet Σ can be built in $O(N \log |\Sigma|)$ time and $O(N)$ space [Wei73, McC76]. Suffix trees permit on-line string searches of the type, ''Is W a substring of A?'' to be answered in $O(P \log |\Sigma|)$ time, where P is the length of W. We explicitly consider the

[1] Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from AT&T, and by an NSF grant CCR-9002351.

[2] Supported in part by the NIH (grant R01 LM04960-01) , and by an NSF grant CCR-9002351.

- A prefix of a suffix is an infix
  - Phrase/substring searches!
- Create and sort an array that organizes all suffixes
  - But do it compactly
- Search using binary search
  - Possibly speed things up by considering least common prefixes

Suffix Arrays, cont.

Consider the string

1	2	3	4	5	6	7	8	9	10	11	12
a	b	r	a	c	a	d	a	b	r	a	$

of length 12, that ends with a sentinel letter $, appearing only once, and less (in lexicographical order) than any other letter in the string.

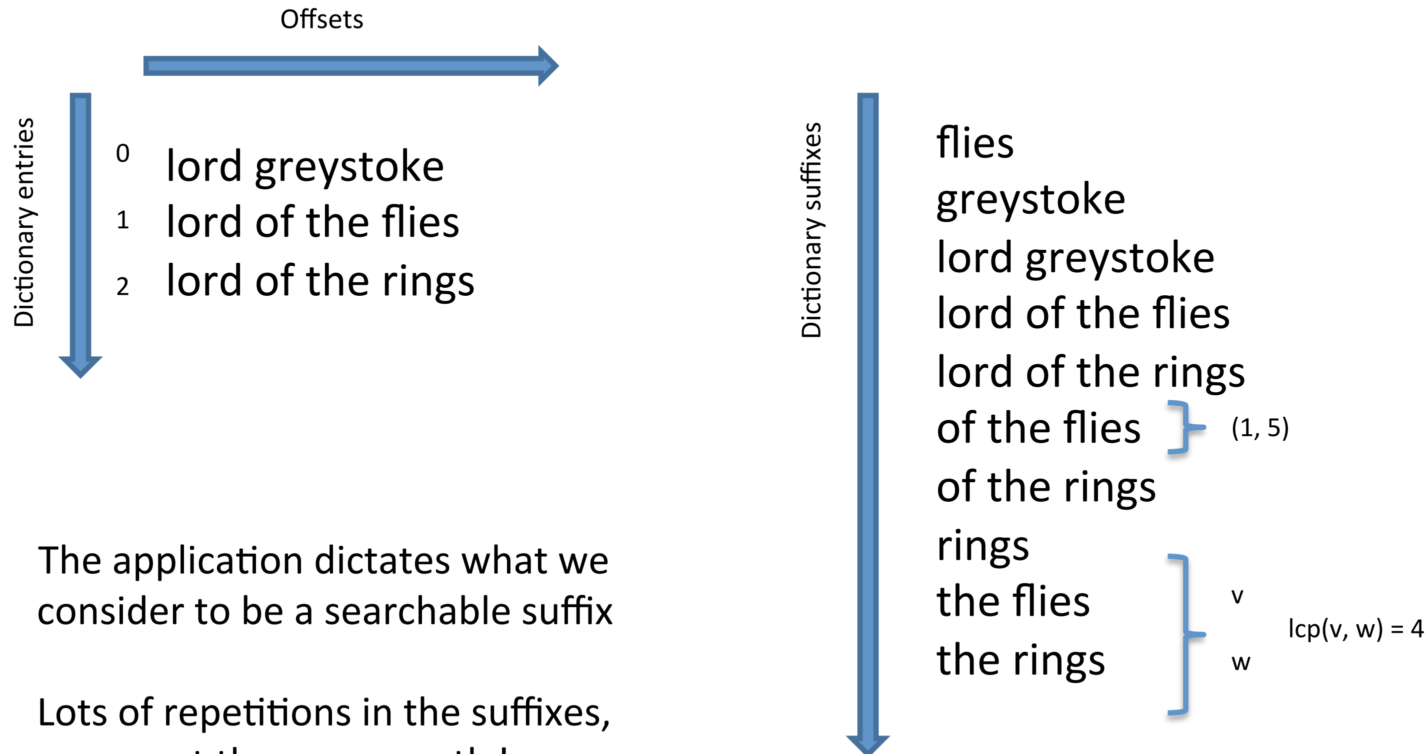
It has twelve suffixes: "abracadabra$", "bracadabra$", "racadabra$", and so on down to "a$" and "$" that can be sorted into lexicographical order to obtain:

index	sorted suffix	lcp
12	$	0
11	a$	0
8	abra$	1
1	abracadabra$	4
4	acadabra$	1
6	adabra$	1
9	bra$	0
2	bracadabra$	3
5	cadabra$	0
7	dabra$	0
10	ra$	0
3	racadabra$	2

If the original string is available, each suffix can be completely specified by the index of its first character. The suffix array is the array of the indices of suffixes sorted in lexicographical order. For the string "abracadabra$", using one-based indexing, the suffix array is {12,11,8,1,4,6,9,2,5,7,10,3}, because the suffix "$" begins at position 12, "a$" begins at position 11, "abra$" begins at position 8, and so forth.

The longest common prefix is also shown above as lcp. This value, stored alongside the list of prefix indices, indicates how many characters a particular suffix has in common with the suffix directly above it, starting at the beginning of both suffixes. The lcp is useful in making some string operations more efficient. For example, it can be used to avoid comparing characters that are already known to be the same when searching through the list of suffixes. The fact that the minimum lcp value belonging to a consecutive set of sorted suffixes gives the longest common prefix among all of those suffixes can also be useful.

Suffix Arrays, cont.



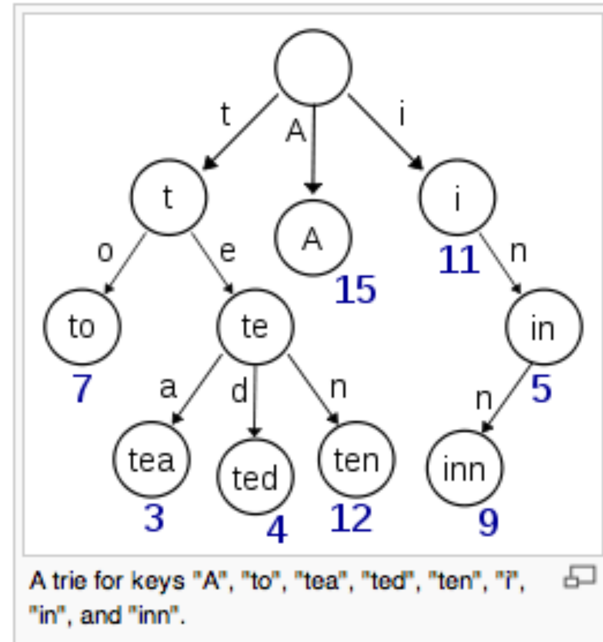
- The application dictates what we consider to be a searchable suffix

  Lots of repetitions in the suffixes, represent them compactly!

- Exploiting *lcp*(*v*, *w*) is useful if the substrings we search for are long
- Links to the Burrows-Wheeler transform

Tries

In computer science, a **trie**, or **prefix tree**, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest.



A trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn".

The term trie comes from re**trie**val. Following the etymology, the inventor, Edward Fredkin, pronounces it /ˈtriː/ "tree".[1][2] However, it is pronounced /ˈtraɪ/ "try" by other authors.[1][2][3]

In the example shown, keys are listed in the nodes and values below them. Each complete English word has an arbitrary integer value associated with it. A trie can be seen as a deterministic finite automaton, although the symbol on each edge is often implicit in the order of the branches.

It is not necessary for keys to be explicitly stored in nodes. (In the figure, words are shown only to illustrate how the trie works.)

Though it is most common, tries need not be keyed by character strings. The same algorithms can easily be adapted to serve similar functions of ordered lists of any construct, e.g., permutations on a list of digits or shapes. In particular, a **bitwise trie** is keyed on the individual bits making up a short, fixed size of bits such as an integer number or pointer to memory.

Tries, cont.

Programming Techniques — Glenn Manacher, Editor

Efficient String Matching: An Aid to Bibliographic Search

Alfred V. Aho and Margaret J. Corasick
Bell Laboratories

This paper describes a simple, efficient algorithm to locate all occurrences of any of a finite number of keywords in a string of text. The algorithm consists of constructing a finite state pattern matching machine from the keywords and then using the pattern matching machine to process the text string in a single pass. Construction of the pattern matching machine takes time proportional to the sum of the lengths of the keywords. The number of state transitions made by the pattern matching machine in processing the text string is independent of the number of keywords. The algorithm has been used to improve the speed of a library bibliographic search program by a factor of 5 to 10.

Keywords and Phrases: keywords and phrases, string pattern matching, bibliographic search, information retrieval, text-editing, finite state machines, computational complexity.

CR Categories: 3.74, 3.71, 5.22, 5.25

Copyright © 1975, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that ACM's copyright notice is given and that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Authors' present addresses: A. V. Aho, Bell Laboratories, Murray Hill, N.J. 07974. M. J. Corasick, The Mitre Corporation, Bedford, Mass. 01730.

1. Introduction

In many information retrieval and text-editing applications it is necessary to be able to locate quickly some or all occurrences of user-specified patterns of words and phrases in text. This paper describes a simple, efficient algorithm to locate all occurrences of any of a finite number of keywords and phrases in an arbitrary text string.

The approach should be familiar to those acquainted with finite automata. The algorithm consists of two parts. In the first part we construct from the set of keywords a finite state pattern matching machine; in the second part we apply the text string as input to the pattern matching machine. The machine signals whenever it has found a match for a keyword.

Using finite state machines in pattern matching applications is not new [4, 8, 17], but their use seems to be frequently shunned by programmers. Part of the reason for this reluctance on the part of programmers may be due to the complexity of programming the conventional algorithms for constructing finite automata from regular expressions [3, 10, 15], particularly if state minimization techniques are needed [2, 14]. This paper shows that an efficient finite state pattern matching machine can be constructed quickly and simply from a restricted class of regular expressions, namely those consisting of finite sets of keywords. Our approach combines the ideas in the Knuth-Morris-Pratt algorithm [13] with those of finite state machines.

Perhaps the most interesting aspect of this paper is the amount of improvement the finite state algorithm gives over more conventional approaches. We used the finite state pattern matching algorithm in a library bibliographic search program. The purpose of the program is to allow a bibliographer to find in a citation index all titles satisfying some Boolean function of keywords and phrases. The search program was first implemented with a straightforward string matching algorithm. Replacing this algorithm with the finite state approach resulted in a program whose running time was a fifth to a tenth of the original program on typical inputs.

2. A Pattern Matching Machine

This section describes a finite state string pattern matching machine that locates keywords in a text string. The next section describes the algorithms to construct such a machine from a given finite set of keywords.

In this paper a *string* is simply a finite sequence of symbols. Let $K = \{y_1, y_2, \ldots, y_k\}$ be a finite set of strings which we shall call *keywords* and let x be an arbitrary string which we shall call the *text string*. Our problem is to locate and identify all substrings of x which are keywords in K. Substrings may overlap with one another.

A pattern matching machine for K is a program which takes as input the text string x and produces as output the locations in x at which keywords of K appear as substrings. The pattern matching machine consists of a set of *states*. Each state is represented by a number. The machine processes the text string x by successively reading the symbols in x, making state transitions and occa-

- Do a trie-walk to find all dictionary occurrences contained in given text fragment
  - Scales linearly with the length of the text fragment
  - The size of the dictionary "doesn't matter"!
- The application dictates constraints on where matches can begin and end
  - Should usually coincide with token boundaries in an NLP setting

Tightly Packed Tries

Tightly Packed Tries: How to Fit Large Models into Memory, and Make them Load Fast, Too

Ulrich Germann
University of Toronto and
National Research Council Canada
germann@cs.toronto.edu

Eric Joanis
National Research Council Canada
Eric.Joanis@cnrc-nrc.gc.ca

Samuel Larkin
National Research Council Canada
Samuel.Larkin@cnrc-nrc.gc.ca

Abstract

We present *Tightly Packed Tries* (TPTs), a compact implementation of read-only, compressed trie structures with fast on-demand paging and short load times.

We demonstrate the benefits of TPTs for storing n-gram back-off language models and phrase tables for statistical machine translation. Encoded as TPTs, these databases require less space than flat text file representations of the same data compressed with the *gzip* utility. At the same time, they can be mapped into memory quickly and be searched directly in time linear in the length of the key, without the need to decompress the entire file. The overhead for local decompression during search is marginal.

1 Introduction

The amount of data available for data-driven Natural Language Processing (NLP) continues to grow. For some languages, language models (LM) are now being trained on many billions of words, and parallel corpora available for building statistical machine translation (SMT) systems can run into tens of millions of sentence pairs. This wealth of data allows the construction of bigger, more comprehensive models, often without changes to the fundamental model design, for example by simply increasing the n-gram size in language modeling or the phrase length in phrase tables for SMT.

The large sizes of the resulting models pose an engineering challenge. They are often too large to fit entirely in main memory. What is the best way to organize these models so that we can swap information in and out of memory as needed, and as quickly as possible?

This paper presents *Tightly Packed Tries* (TPTs), a compact and fast-loading implementation of read-only trie structures for NLP databases that store information associated with token sequences, such as language models, n-gram count databases, and phrase tables for SMT.

In the following section, we first recapitulate some basic data structures and encoding techniques that are the foundations of TPTs. We then lay out the organization of TPTs. Section 3 discusses compression of node values (i.e., the information associated with each key). Related work is discussed in Section 4. In Section 5, we report empirical results from run-time tests of TPTs in comparison to other implementations. Section 6 concludes the paper.

2 Fundamental data structures and encoding techniques

2.1 Tries

Tries (Fredkin, 1960), also known as *prefix trees*, are a well-established data structure for compactly storing sets of strings that have common prefixes. Each string is represented by a single node in a tree structure with labeled arcs so that the sequence of arc labels from the root node to the respective node "spells out" the token sequence in question. If we augment the trie nodes with additional information, tries can be used as indexing structures for databases that rely on token sequences as search keys. For the remainder of this paper, we will refer to such additional

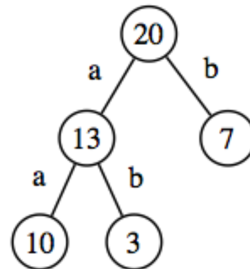
Proceedings of the NAACL HLT Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing, pages 31–39,
Boulder, Colorado, June 2009. ©2009 Association for Computational Linguistics

- Lay stuff out in a single contiguous byte array
  - Facilitates a compact representation
  - Enables memory mapping
- Populate the array by traversing the trie in post-order
  - Logically, at least
- Can be further combined with compression techniques
  - E.g., various variable length encodings

Tightly Packed Tries, cont.

total count	20
a	13
aa	10
ab	3
b	7

(a) Count table



(b) Trie representation

field	**32-bit**	**64-bit**
index entry: token ID	4	4
index entry: pointer	4	8
start of index (pointer)	4	8
overhead of index structure } node value	x	y
total (in bytes)	$12 + x$	$20 + y$

(c) Memory footprint per node in an implementation using memory pointers

0	13	*offset of root node*
1	10	*node value of* ‘aa’
2	0	*size of index to child nodes of* ‘aa’ in bytes
3	3	*node value of* ‘ab’
4	0	*size of index to child nodes of* ‘ab’ in bytes
5	13	*node value of* ‘a’
6	4	*size of index to child nodes of ‘a’ in bytes*
7	a	*index key for* ‘aa’ *coming from* ‘a’
8	4	*relative offset of node* ‘aa’ $(5 - 4 = 1)$
9	b	*index key for* ‘ab’ *coming from* ‘a’
10	2	*relative offset of node* ‘ab’ $(5 - 2 = 3)$
11	7	*node value of* ‘b’
12	0	*size of index to child nodes of* ‘b’ in bytes
13	20	*root node value*
14	4	*size of index to child nodes of root in bytes*
15	a	*index key for* ‘a’ *coming from root*
16	8	*relative offset of node* ‘a’ $(13 - 8 = 5)$
17	b	*index key for* ‘b’ *coming from root*
18	2	*relative offset of node* ‘b’ $(13 - 2 = 11)$

(d) Trie representation in a contiguous byte array. In practice, each field may vary in length.

Figure 1: A count table (a) stored in a trie structure (b) and the trie's sequential representation in a file (d). As the size of the count table increases, the trie-based storage becomes more efficient, provided that the keys have common prefixes. (c) shows the memory footprint per trie node when the trie is implemented as a mutable structure using direct memory pointers.

Sharing Prefixes *and* Suffixes

How to squeeze a lexicon

Marcin G. Ciura, Sebastian Deorowicz

May 8, 2002

This is a preprint of an article published in
Software—Practice and Experience 2001; **31**(11):1077–1090
Copyright © 2001 John Whiley & Sons, Ltd.
http://www.interscience.wiley.com

Abstract

Minimal acyclic deterministic finite automata (ADFAs) can be used as a compact representation of finite string sets with fast access time. Creating them with traditional algorithms of DFA minimization is a resource hog when a large collection of strings is involved. This paper aims to popularize an efficient but little known algorithm for creating minimal ADFAs recognizing a finite language, invented independently by several authors. The algorithm is presented for three variants of ADFAs, its minor improvements are discussed, and minimal ADFAs are compared to competitive data structures.

KEY WORDS: static lexicon; static dictionary; trie compaction; directed acyclic graph; acyclic finite automaton

INTRODUCTION

Many applications involve accessing a database, whose keys are variable-length finite sequences of characters from a fixed alphabet (*strings*). Such databases are known as *symbol tables* or *dictionaries*. Sometimes no data are associated with the keys, we need only to know whether a string belongs to a given set. Following Revuz [23], we call a set of bare strings a *lexicon* to distinguish it from a complete dictionary.

In spelling checkers and other software that deals with natural language the lexicons often contain hundreds of thousands words, yet they need no frequent updates. Knowing all the keys in advance allows to prepare a static data structure that outperforms dynamic ones in size or time of searching. At least several ways to construct such a data structure are conceivable.

Hashing

Perfect hashing, a classical solution to the static dictionary problem [5], requires storage for all the strings; at most they can be compressed with a static method. When they are just words, affix stripping [13, 17] reduces the space requirements. It can be applied to many languages, but for languages more inflective and irregular than English the accurate morphological rules are complex, and grammatical classification of corpus-based word lists demands either human labour or sophisticated software.

A sparse hash table allows discarding the strings entirely, at the cost of occasional false matches, and can be encoded compactly yielding a Bloom filter [17]. Sometimes, though, absolute certainty is desired. Moreover, once the strings are dropped, it is impossible to reconstruct them.

Tries

A character tree, also known as a *trie* [11, Chapter 6.3], is an oriented tree, in which every path from the root to a leaf corresponds to a key and branching is based on successive characters. Tries are sometimes preferable to hashing, because they detect unsuccessful searches faster and answer partial match or nearest neighbour queries effectively.

Tries are found in two varieties: *abbreviated* and *full*. The former comprise only the shortest prefixes necessary to distinguish the strings; finding a string in them must be confirmed by a comparison to a suffix stored in a trie leaf. The latter comprise entire strings, character by character, up to a string delimiter, as shown in Figure 1.

- From tries toward more general automata
  - Natural language compresses very well!
- Keep track of equivalent states during construction
  - Assuming static dictionaries
- Can be very compactly represented
  - Previously mentioned packing techniques apply

Sharing Prefixes *and* Suffixes, cont.

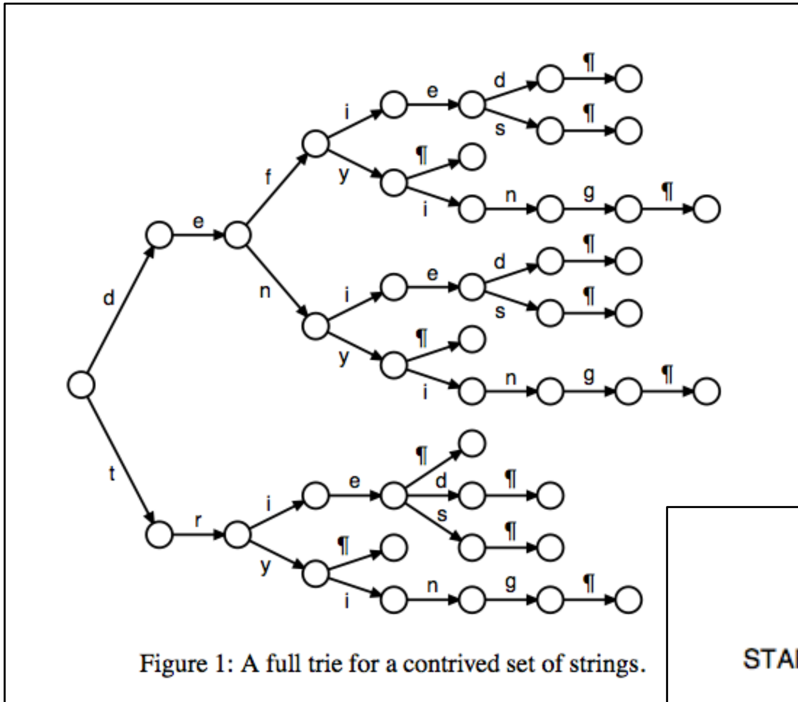


Figure 1: A full trie for a contrived set of strings.

01 $s_0[0] \leftarrow$ '¶'; $i \leftarrow 0$; *larval_state*$[0] \leftarrow \emptyset$;
02 **while not** *eof* **do**
03 read next string into $s_1[0 \,..\, q-1]$, and set q to its length;
04 $s_1[q] \leftarrow$ '¶'; $p \leftarrow 0$;
05 **while** $s_0[p] = s_1[p]$ **do** $p \leftarrow p+1$; **end while**; $\{p \leq q\}$
06 **while** $i > p$ **do** *new_state* $\leftarrow$ *make_state*(*larval_state*$[i]$);
07 $i \leftarrow i-1$; *larval_state*$[i] \leftarrow$ *larval_state*$[i] \cup \{\langle s_0[i],$ *new_state*$\rangle\}$;
08 **end while**; $\{i \leq p \leq q\}$
09 **while** $i \leq q$ **do** $s_0[i] \leftarrow s_1[i]$;
10 $i \leftarrow i+1$; *larval_state*$[i] \leftarrow \emptyset$;
11 **end while**; $\{i = q+1\}$
12 **end while**;
13 **while** $i > 0$ **do** *new_state* $\leftarrow$ *make_state*(*larval_state*$[i]$);
14 $i \leftarrow i-1$; *larval_state*$[i] \leftarrow$ *larval_state*$[i] \cup \{\langle s_0[i],$ *new_state*$\rangle\}$;
15 **end while**;
16 *start_state* $\leftarrow$ *make_state*(*larval_state*$[0]$);

Figure 4: The algorithm for creating a simple recognizer.

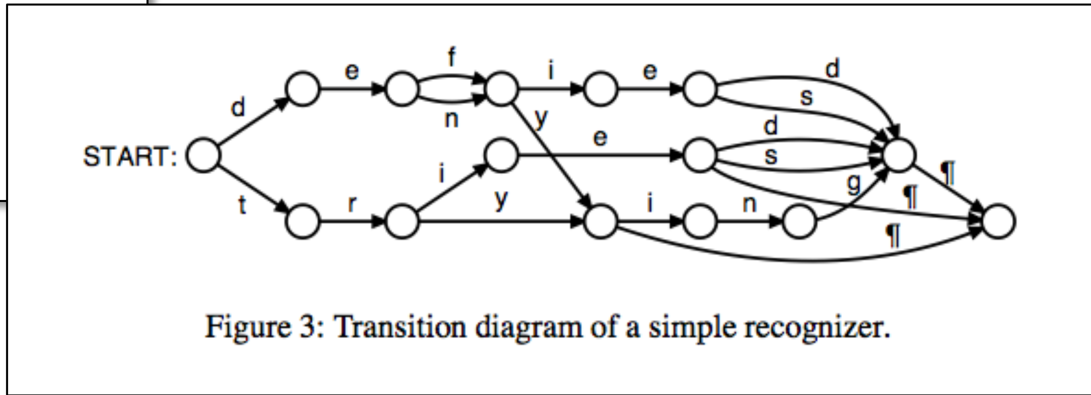


Figure 3: Transition diagram of a simple recognizer.

“How do you determine if two strings are syntactically close?”

“And how do you compute edit distance efficiently against a large dictionary?”

Edit Distance

- Given two strings *s* and *t*, the minimum number of operations to convert one to the other
- Operations are typically character-level
  - Insert, Delete, Replace
  - Transpose
- Generally found by dynamic programming

Edit Distance, cont.

$$D = \begin{cases} 0 & i = j = 0 \\ \infty & i < 0 \text{ or } j < 0 \\ \min \begin{pmatrix} D(P_i, W_{j-1}) + 1 \\ D(P_{i-1}, W_j) + 1 \\ D(P_{i-1}, W_{j-1}) + S_{ij} \\ D(P_{i-2}, W_{j-2}) + R_{ij} \end{pmatrix} & \text{else} \end{cases}$$

where $p_i = w_j = \phi$ when $i, j \leq 0$ (the null character), and

$$S_{ij} = \begin{cases} 0 & p_i = w_j \\ 1 & \text{else} \end{cases}, \quad R_{ij} = \begin{cases} 1 & p_{i-1} = w_j \wedge p_i = w_{j-1} \\ \infty & \text{else} \end{cases}.$$

Edit Tables

		i	n	f	o	r	m	a	t	i	o	n
	0	1	2	3	4	5	6	7	8	9	10	11
i	1	0	1	2	3	4	5	6	7	8	9	10
n	2	1	0	1	2	3	4	5	6	7	8	9
f	3	2	1	0	1	2	3	4	5	6	7	8
r	4	3	2	1	1	1	2	3	4	5	6	7
m	5	4	3	2	2	2	1	2	3	4	5	6
a	6	5	4	3	3	3	2	1	2	3	4	5
t	7	6	5	4	4	4	3	2	1	2	3	4
i	8	7	6	5	5	5	4	3	2	1	2	3
i	9	8	7	6	6	6	5	4	3	2	2	3
o	10	9	8	7	6	7	6	5	4	3	2	3
n	11	10	9	8	7	7	7	6	5	4	3	2

- Start at (1, 1), answer at $(|s|, |t|)$
  - Usually, but not necessarily, computed column by column
- We might get away with computing only part of a column
  - Ukkonen's cutoff
- Costs don't have to be integers
  - But with unit edit costs the table has some special properties
  - Costs can take statistics, keyboard layout etc into account

Edit Distance and Dictionaries

Tries for Approximate String Matching

H. Shang and T.H. Merrett*

September 8, 1995

Abstract

Tries offer text searches with costs which are independent of the size of the document being searched, and so are important for large documents requiring spelling checkers), case insensitivity, and lim ited approximate regular secondary storage. Approximate searches, in which the search pattern differs from the document by k substitu- tions, transpositions, insertions or deletions, have hitherto been car ried out only at costs linear in the size of the document. We present a trie-based method whose cost is independent of document size.

*H. Shang and T.H. Merrett are at the School of Computer Science, McGill University, Montréal, Québec, Canada H3A 2A7, Email: {shang, tim}@cs.mcgill.ca

- Given *s*, find the closest *t* in a large dictionary
  - Organize the dictionary entries in a trie
  - Possibly also partition the entries by length
  - Assumes small edit distance, e.g., k={1, 2, 3}
- The trie defines a search space
  - We want to prune the search space early
  - Each step in the search involves computing a column in an edit table
  - All strings below a node share the same prefix, and hence also the same columns in the edit table
  - We can prune away a branch when the edit distance exceeds a given threshold

Edit Tables and Bit-Parallelism

Bit-Parallel Approximate String Matching Algorithms with Transposition

Heikki Hyyrö *

Department of Computer and Information Sciences
University of Tampere, Finland.
Heikki.Hyyro@cs.uta.fi

Abstract. Using bit-parallelism has resulted in fast and practical algorithms for approximate string matching under the Levenshtein edit distance, which permits a single edit operation to insert, delete or substitute a character. Depending on the parameters of the search, currently the fastest non-filtering algorithms in practice are the $O(kn\lceil m/w \rceil)$ algorithm of Wu & Manber, the $O(\lceil km/w \rceil n)$ algorithm of Baeza-Yates & Navarro, and the $O(\lceil m/w \rceil n)$ algorithm of Myers, where m is the pattern length, n is the text length, k is the error threshold and w is the computer word size. In this paper we discuss a uniform way of modifying each of these algorithms to permit also a fourth type of edit operation: transposing two adjacent characters in the pattern. This type of edit distance is also known as the Damerau edit distance. In the end we also present an experimental comparison of the resulting algorithms.

1 Introduction

Approximate string matching is a classic problem in computer science, with applications for example in spelling correction, bioinformatics and signal processing. It has been actively studied since the sixties [8]. Approximate string matching refers in general to the task of searching for substrings of a text that are within a predefined edit distance threshold from a given pattern. Let $T_{1..n}$ be a text of length n and $P_{1..m}$ a pattern of length m. In addition let $ed(A,B)$ denote the edit distance between the strings A and B, and k be the maximum allowed distance. Using this notation, the task of approximate string matching is to find from the text all indices j for which $ed(P, T_{h..j}) \le k$ for some $h \le j$.

Perhaps the most common form of edit distance is the Levenshtein edit distance [6], which is defined as the minimum number of single-character insertions, deletions and substitutions (Fig. 1a) needed in order to make A and B equal. Another common form of edit distance is the Damerau edit distance [2], which is in principle an extension of the Levenshtein distance by permitting also the operation of transposing two adjacent characters (Fig. 1b). The Damerau edit

* Supported by the Academy of Finland and Tampere Graduate School in Information Science and Engineering.

- Represent the edit table as a set of horizontal and vertical bit vectors
  - Assuming unit edit costs
- Edit table computations become fancy bit masking and shifting operations
  - Allows a constant speed-up proportional to the machine's word size

Edit Distance, cont.

- Together with *n*-gram matching
  1. Find the *m* best *n*-gram matches
  2. Rerank matches using edit distance, possibly considering word permutations

3-gram matching:
"nowember" yields {"november", "december"}

Edit distance:
Makes us select "november"

- Together with phonetic hashing
  1. Preprocess the dictionary to hold *(h(t), {(t, v)})* instead of *(t, v)*
  2. Look up *h(s)* using a very low edit threshold (possibly 0)
  3. Rerank matches using edit distance between *s* and *t*

Example choices of *h*:
Soundex, Double Metaphone

Double Metaphone:
{"carlisle", "karlysle", …} yields "krll"

Spellchecking and Context

- Spellchecking word by word only gets us so far
  - "untied airlines"
  - "blackmonitor", "micro soft"
- Some candidates are more likely than others
  - Score candidates using real-world frequency information
- When shouldn't we spellcheck queries?