STRINGS GALORE

Exploiting prefixes for fun and profit

Aleksander Øhrn

IN3120/IN4120

GENERAL APPROACH

1. Text to be queried in a complex or inefficient way

Clever transformation

2. Text to be efficiently queried using simple prefix matching

Prefix matching

3. OMG, awesome matches!

PERMUTERM INDEXES AND WILDCARDS

$$\text{hello} \rightarrow \left\{ \begin{array}{l} \text{hello\$} \\ \text{ello\$h} \\ \text{llo\$he} \\ \text{lo\$hel} \\ \text{o\$hell} \end{array} \right\}$$

Generate all rotations for all terms to produce the permuterm vocabulary

X	→ X$
X*	→ $X*
*X	→ X$*
X*Y	→ Y$X*
X*Y*Z	→ Z$X* and Y*

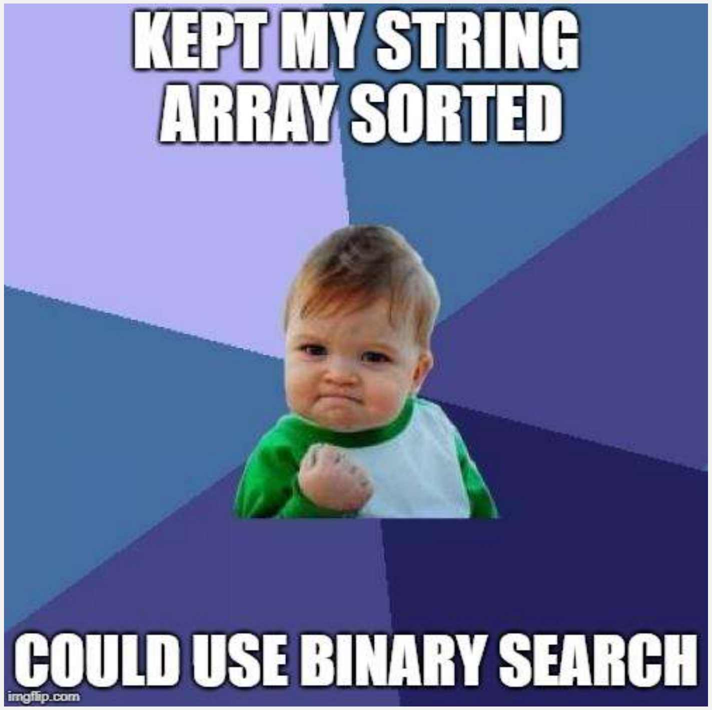
Transform wildcard queries into prefix searches over the permuterm vocabulary

fi*er → er$fi* → {fishmonger, filibuster, …}

DATA STRUCTURES

Some are more “prefix-friendly” than others

SORTED ARRAYS



TRIES

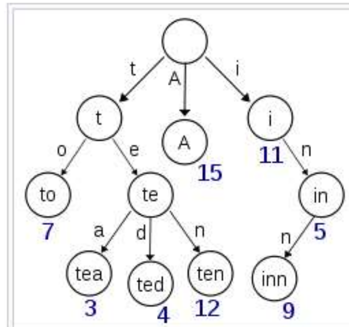
Trie

From Wikipedia, the free encyclopedia

The **lead section of this article may need to be rewritten.** Please discuss this issue on the article's talk page. Use the lead layout guide to ensure the section follows Wikipedia's norms and to be inclusive of all essential details. *(June 2017) (Learn how and when to remove this template message)*

This article is about a tree data structure. For the French commune, see Trie-sur-Baïse.

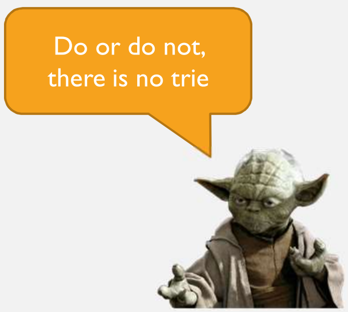
In computer science, a **trie**, also called **digital tree**, **radix tree** or **prefix tree** is a kind of search tree—an ordered tree data structure used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. 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. Keys tend to be associated with leaves, though some inner nodes may correspond to keys of interest. Hence, keys are not necessarily associated with every node. For the space-optimized presentation of prefix tree, see compact prefix tree.



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

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 tree-shaped deterministic finite automaton. Each finite language is generated by a trie automaton, and each trie can be compressed into a deterministic acyclic finite state automaton.

Though tries are usually keyed by character strings,[not verified in body] they need not be. The same algorithms can 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 any fixed-length binary datum, such as an integer or memory address.



TRIES, CONT.

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

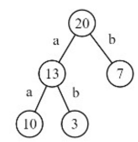
2.1 Tries

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.

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

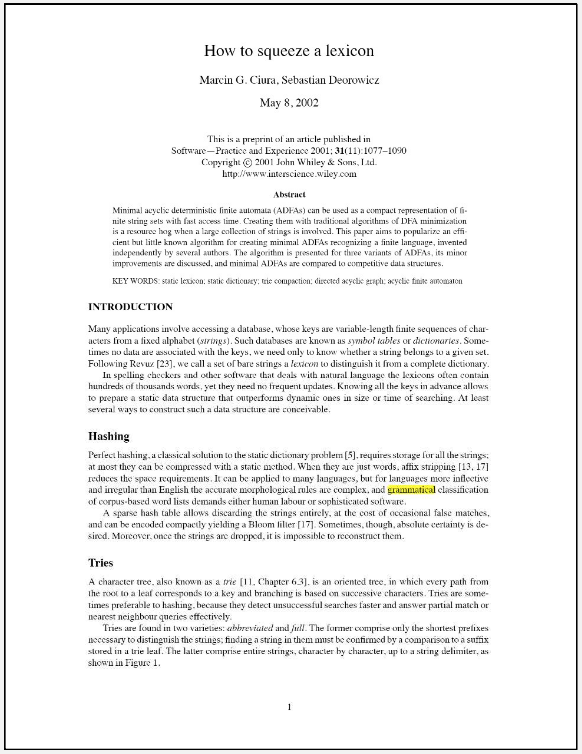
0	13	*offset of root node*
1	10	*node value of* 'aa'
2	0	*size of index to child nodes of* 'aa' in bytes
3	3	*node value of* 'ab'
4	0	*size of index to child nodes of* 'ab' in bytes
5	13	*node value of* 'a'
6	4	*size of index to child nodes of 'a' in bytes*
7	a	*index key for* 'aa' *coming from* 'a'
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.

- Lay stuff out in a contiguous array
- Populate the array by post-order traversal of the trie
- Can be further combined with compression techniques

TRIES, CONT.



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

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.

1

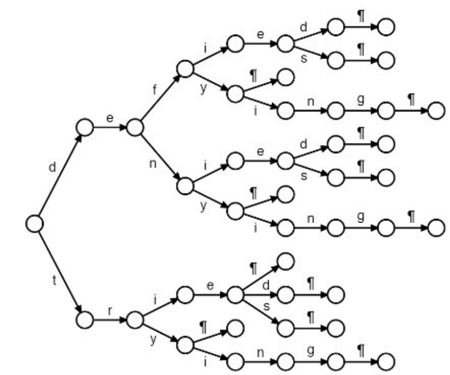


Figure 1: A full trie for a contrived set of strings.

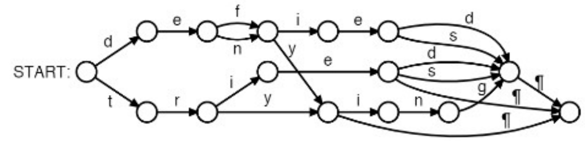


Figure 3: Transition diagram of a simple recognizer.

- Share both prefixes and suffixes!
- Transform tries to more general automata
- Keep track of equivalent states during construction
- Natural language compresses very well

BURROWS-WHEELER TRANSFORM

Make your strings more compressible by
making them easier to run-length encode

BURROWS-WHEELER TRANSFORM

six_mixed_pixies_sift_sixty_pixie_dust_boxes

BWT

BWT^{-1}

texydst_e_ixixixxssmpps_b__e_s_eusfxdiioiiit

tomorrow_and_tomorrow_and_tomorrow$

BWT

BWT^{-1}

w$wwdd__nnoooaatttmmmrrrrrrooo__ooo

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!
- Represent a suffix as an integer offset into the original string
- Sort all suffixes lexicographically
- The sorted list of integers is called the suffix array
- Do a binary search in the suffix array to locate all matching substrings and where they start

SUFFIX ARRAYS, CONT.

$$\text{banana\$} \rightarrow \left\{ \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix} = \begin{pmatrix} \text{banana\$} \\ \text{anana\$} \\ \text{nana\$} \\ \text{ana\$} \\ \text{na\$} \\ \text{a\$} \\ \text{\$} \end{pmatrix} \right\} \xrightarrow{\text{sort}} \left\{ \begin{bmatrix} 6 \\ 5 \\ 3 \\ 1 \\ 0 \\ 4 \\ 2 \end{bmatrix} = \begin{pmatrix} \text{\$} \\ \text{a\$} \\ \text{ana\$} \\ \text{anana\$} \\ \text{banana\$} \\ \text{na\$} \\ \text{nana\$} \end{pmatrix} \right\}$$

Do a binary search in the suffix array to locate if and where substring matches are found

SUFFIX ARRAYS, CONT.

$$\left\{ \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} \text{lord greystoke\$} \\ \text{lord of the flies\$} \\ \text{lord of the rings\$} \end{pmatrix} \right\} \rightarrow \ldots \xrightarrow{\text{sort}} \left\{ \begin{pmatrix} (1, 12) \\ (0, 5) \\ (0, 0) \\ (1, 0) \\ (2, 0) \\ (1, 5) \\ (2, 5) \\ (2, 12) \\ (1, 8) \\ (2, 8) \end{pmatrix} = \begin{pmatrix} \text{flies\$} \\ \text{greystoke\$} \\ \text{lord greystoke\$} \\ \text{lord of the flies\$} \\ \text{lord of the rings\$} \\ \text{of the flies\$} \\ \text{of the rings\$} \\ \text{rings\$} \\ \text{the flies\$} \\ \text{the rings\$} \end{pmatrix} \right\}$$

The application dictates what we consider to be a searchable suffix, i.e., where matches can begin and end

BURROWS-WHEELER TRANSFORM, CONT.

$$\text{banana\$} \rightarrow \left\{ \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} \text{banana\$} \\ \text{anana\$b} \\ \text{nana\$ba} \\ \text{ana\$ban} \\ \text{na\$bana} \\ \text{a\$banan} \\ \text{\$banana} \end{pmatrix} \right\} \xrightarrow{\text{sort}} \left\{ \begin{pmatrix} 6 \\ 5 \\ 3 \\ 1 \\ 0 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} \text{\$banana} \\ \text{a\$banan} \\ \text{ana\$ban} \\ \text{anana\$b} \\ \text{banana\$} \\ \text{na\$bana} \\ \text{nana\$ba} \end{pmatrix} \right\}$$

$$\text{banana\$} \underset{\text{BWT}^{-1}}{\overset{\text{BWT}}{\rightleftarrows}} \text{annb\$aa}$$

$$\text{BWT}[i] = \begin{cases} S[A[i] - 1] & \text{if } A[i] > 0 \\ \$ & \text{otherwise} \end{cases}$$

TRIES AND EDIT TABLES

Efficiently find all strings in a huge dictionary
that have a small edit distance to a given string

EDIT DISTANCE

- The smallest number of edit operations needed to rewrite one string into another
  - Minimal edit sequence is not unique
- Edit operations:
  - Insert — cat ⟶ cart
  - Delete — cart ⟶ cat
  - Replace — cart ⟶ dart
  - Transpose — watre ⟶ water
- Can be generalized to finding minimal edit costs

Definition [edit]

To express the Damerau–Levenshtein distance between two strings a and b a function $d_{a,b}(i,j)$ is defined, whose value is a distance between an i–symbol prefix (initial substring) of string a and a j–symbol prefix of b.

The function is defined recursively as:

$$d_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if} \min(i,j) = 0, \\ \min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \\ d_{a,b}(i-2,j-2)+1 \end{cases} & \text{if } i,j > 1 \text{ and } a_i = b_{j-1} \text{ and } a_{i-1} = b_j \\ \min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

where $1_{(a_i \neq b_j)}$ is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise.

Each recursive call matches one of the cases covered by the Damerau–Levenshtein distance:

- $d_{a,b}(i-1,j)+1$ corresponds to a deletion (from a to b).
- $d_{a,b}(i,j-1)+1$ corresponds to an insertion (from a to b).
- $d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)}$ corresponds to a match or mismatch, depending on whether the respective symbols are the same.
- $d_{a,b}(i-2,j-2)+1$ corresponds to a transposition between two successive symbols.

The Damerau–Levenshtein distance between a and b is then given by the function value for full strings: $d_{a,b}(|a|,|b|)$ where $i = |a|$ denotes the length of string a and $j = |b|$ is the length of b.

EDIT TABLES

N^2W^2		
	NW	N
	W	

$= f(N, W, NW, N^2W^2)$

		E	L	E	P	H	A	N	T
	0	1	2	3	4	5	6	7	8
R	1	1	2	3	4	5	6	7	8
E	2	1	2	2	3	4	5	6	7
L	3	2	1	2	3	4	5	6	7
E	4	3	2	1	2	3	4	5	6
V	5	4	3	2	2	3	4	5	6
A	6	5	4	3	3	3	3	4	5
N	7	6	5	4	4	4	4	3	4
T	8	7	6	5	5	5	5	4	3

- Fill table using the Damerau-Levenshtein update rule
- Start at the NW-most corner
- Answer is in the SE-most corner

EDIT TABLES, CONT.

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 $ed(P, T_{h..j}) \leq k$ for some $h \leq j$.

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.

$$
\begin{aligned}
TC' &\leftarrow PM_{T_j} \;|\; ((((\sim TC)\ \&\ PM_{T_j}) << 1)\ \&\ PM_{T_{j-1}})\\
D0' &\leftarrow (((TC'\ \&\ VP) + VP) \wedge VP) \;|\; TC' \;|\; VN\\
HP' &\leftarrow VN \;|\; \sim(D0' \;|\; VP)\\
HN' &\leftarrow VP\ \&\ D0'\\
VP' &\leftarrow (HN' << 1) \;|\; \sim(D0' \;|\; (HP' << 1))\\
VN' &\leftarrow (HP' << 1)\ \&\ D0'
\end{aligned}
$$

- Assuming unit edit costs we can represent an edit table using four bit vectors
  - Encoding vertical/horizontal positive/negative differences between table cells
- Speedups proportional to the machine word size!
  - Extensions exist that allow multiple strings to be packed into the same machine word

EDIT TABLES, CONT.

- Edit distance computations against a large set of strings?
  - Strings that share a prefix also share columns in the edit table!

- If we also cap the maximum allowed edit distance to a small number?
  - We can search efficiently!

…
E L E M E N T S
E L E V A T E D
E L E V A T O R

		E	L	E	P	H	A	N	T
	0	1	2	3	4	5	6	7	8
R	1	1	2	3	4	5	6	7	8
E	2	1	2	2	3	4	5	6	7
L	3	2	1	2	3	4	5	6	7
E	4	3	2	1	2	3	4	5	6
V	5	4	3	2	2	3	4	5	6
A	6	5	4	3	3	3	3	4	5
N	7	6	5	4	4	4	4	3	4
T	8	7	6	5	5	5	5	4	3

TRIES AND EDIT TABLES

Tries for Approximate String Matching

H. Shang and T.H. Merrett*

September 8, 1995

Abstract

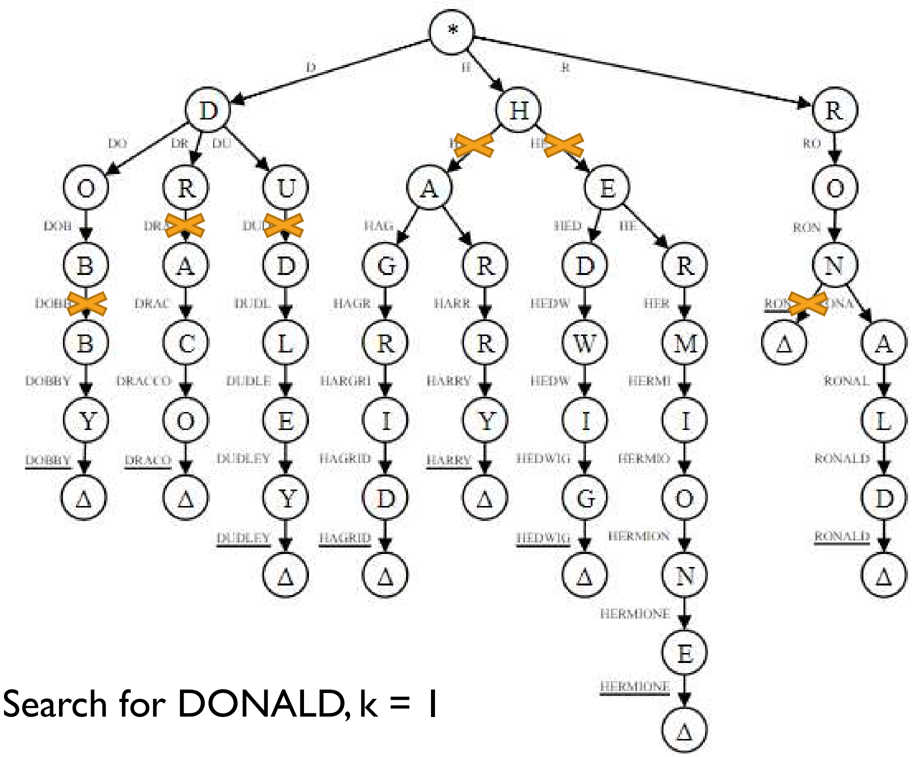
Tries offer text searches with costs which are independent of the size of the document being searched, and so are important for large documents requiring spelling checkers), case insensitivity, and limited approximate regular secondary storage. Approximate searches, in which the search pattern differs from the document by k substitutions, transpositions, insertions or deletions, have hitherto been carried 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

100

- Efficient one-to-many matching
  - For reasonable edit distances
- Spellchecking
  - Can be combined with, e.g., phonetic hashing
- Query completion
  - Exact or approximate

TRIES AND EDIT TABLES, CONT.



Search for DONALD, k = 1

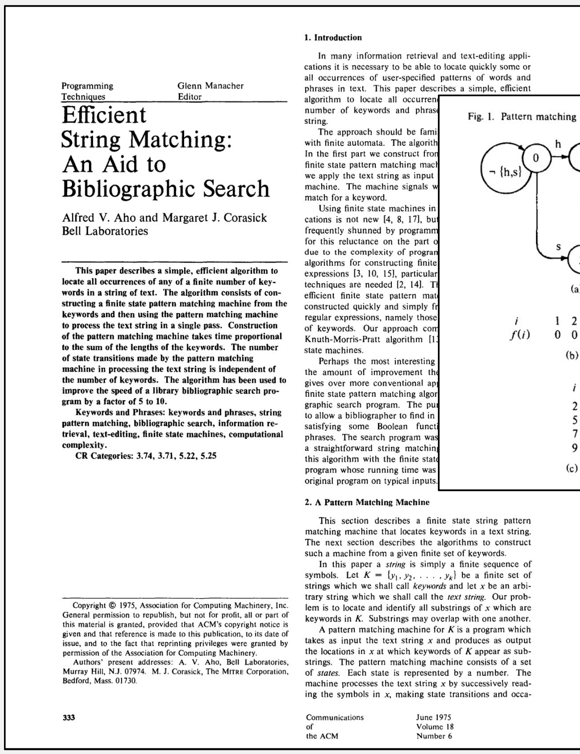
- Organize your dictionary in a trie
  - Traverse it keeping track of the depth
- Reuse parts of the edit table!
  - Only one column to update
  - Abort column updates early
- Prune the search space early!
  - Abort traversal of branches when edit threshold is exceeded
  - Further pruning tricks possible

THE AHO-CORASICK ALGORITHM

Efficiently find all strings in a given buffer that also appear in a huge dictionary

AHO-CORASICK

- A trie with some extra edges
- Search for all strings simultaneously
  - Dictionary size doesn't matter!



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 occurren number of keywords and phras string.

The approach should be fami with finite automata. The algorith In the first part we construct fron finite state pattern matching mach we apply the text string as input machine. The machine signals w match for a keyword.

Using finite state machines in cations is not new [4, 8, 17], but frequently shunned by programm for this reluctance on the part o due to the complexity of program algorithms for constructing finite expressions [3, 10, 15], particular techniques are needed [2, 14]. Th efficient finite state pattern mat constructed quickly and simply fr regular expressions, namely those of keywords. Our approach com Knuth-Morris-Pratt algorithm [1 state machines.

Perhaps the most interesting the amount of improvement the gives over more conventional ap finite state pattern matching algor graphic search program. The pu to allow a bibliographer to find in satisfying some Boolean functi phrases. The search program was a straightforward string matchin this algorithm with the finite state program whose running time was original program on typical inputs.

2. A Pattern Matching Machine

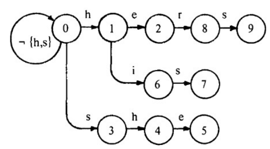
This section describes a finite state string pattern matching machine that locates keywords in a text string. The next section describes the algorithms to construct such a machine from a given finite set of keywords.

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-

333 — Communications of the ACM, June 1975, Volume 18, Number 6

Fig. 1. Pattern matching machine.



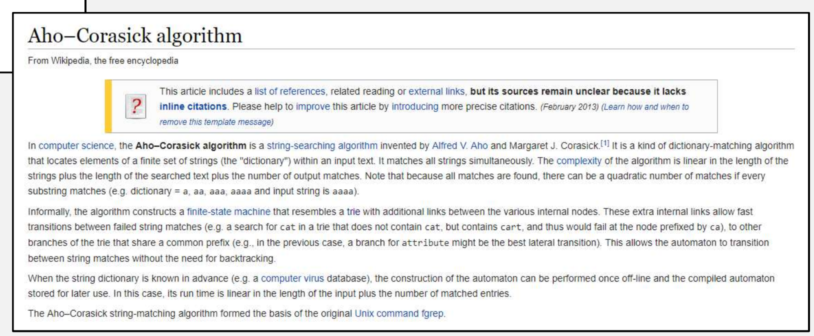
(a) Goto function.

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

(b) Failure function.

i	*output* (i)
2	{he}
5	{she, he}
7	{his}
9	{hers}

(c) Output function.



Aho–Corasick algorithm

From Wikipedia, the free encyclopedia

This article includes a list of references, related reading or external links, **but its sources remain unclear because it lacks inline citations**. Please help to improve this article by introducing more precise citations. *(February 2013) (Learn how and when to remove this template message)*

In computer science, the **Aho–Corasick algorithm** is a string-searching algorithm invented by Alfred V. Aho and Margaret J. Corasick.[1] It is a kind of dictionary-matching algorithm that locates elements of a finite set of strings (the "dictionary") within an input text. It matches all strings simultaneously. The complexity of the algorithm is linear in the length of the strings plus the length of the searched text plus the number of output matches. Note that because all matches are found, there can be a quadratic number of matches if every substring matches (e.g. dictionary = `a`, `aa`, `aaa`, `aaaa` and input string is `aaaa`).

Informally, the algorithm constructs a finite-state machine that resembles a trie with additional links between the various internal nodes. These extra internal links allow fast transitions between failed string matches (e.g. a search for `cat` in a trie that does not contain `cat`, but contains `cart`, and thus would fail at the node prefixed by `ca`), to other branches of the trie that share a common prefix (e.g., in the previous case, a branch for `attribute` might be the best lateral transition). This allows the automaton to transition between string matches without the need for backtracking.

When the string dictionary is known in advance (e.g. a computer virus database), the construction of the automaton can be performed once off-line and the compiled automaton stored for later use. In this case, its run time is linear in the length of the input plus the number of matched entries.

The Aho–Corasick string-matching algorithm formed the basis of the original Unix command fgrep.

AHO-CORASICK, CONT.

- A "trie walk", sort of
- The application dictates where matches can begin and end
- Combine with basic NLP to find linguistic variants

THE END