CSC 428 (Spring 2010)
Final Project

Exact-Set Matching with the Aho-Corasick Automaton

Tom Spreen*
Andre Van Slyke†

April 12, 2010

*University of Victoria
†University of Victoria

1. Introduction

The problem of exact-set matching (that is, searching some arbitrary text T for one or more members of some set P of patterns $p_1, p_2, p_3, \ldots p_n$) is well known, and a variety of different solutions now exist. This paper will explore one such solution, the Aho-Corasick automaton[1], in detail. This automaton is significant for its linear-time performance and its simplicity relative to other exact-set matching methods.

We will first examine the history of this automaton, including motivation for its invention. We will then examine the underlying algorithm in detail, including pre-construction of the automaton and the path of the automaton through an arbitrary text. Next, we will look at its running time and verify the claim that the automaton runs in linear time. Finally, we will briefly compare the Aho-Corasick automaton with other methods of accomplishing similar tasks, and draw some conclusions about its efficacy as a tool for exact-set matching, especially as compared to other methods available.

Note: we will variously refer to the Aho-Corasick automaton as the *AC automaton* or simply the *ACA*.

[1] Aho, Alfred V. and Corasick, Margaret J. *Efficient String Matching: An Aid to Bibliographic Search.* Communications of the ACM, June 1975, Volume 18, Number 6, pp 333-340.

2. Definitions and Conventions

It is useful to predefine some commonly used terms, most of which should be familiar to the reader.

text: a set of characters, typically large, which is the target of the search (for example, a book or a sample of DNA)

pattern: a set of characters, typically much smaller than the associated text, that we are looking for in the text. For example, the word "bioinformatics".

dictionary: a set of patterns.

exact-string matching, or *exact-pattern matching:* the problem of finding an exact instance of a pattern within a text (for example, finding "dog" within "antidogmaticism".

exact-set matching, or *exact-dictionary matching:* the problem of finding an exact instance of one of many patterns in a dictionary, within a text (for example, finding "dog" and "tido" and "tici" within "antidogmaticism".

prefix tree, or *trie:* a tree defined in the mathematical sense, and constructed from a dictionary, with the fundamental characteristics that each edge is labeled with a single character from some pattern in the dictionary, and that all edge labels from any one node are distinct.

failure link: a link from the longest suffix of the current pattern that also exists as a prefix in the keyword tree, to that prefix in the tree.

In most examples we denote a text by T (for Text), a dictionary by P (for Patterns), and a keyword tree by K (for Keywords).

We will use set-theoretic indices; that is, numbering will start at 1, not 0. For example, the first pattern in some dictionary P will be referred to as p_1, and the first character in that pattern will be denoted by p_{1_1}.

3. History


Figure 1: Alfred Aho and Margaret Corasick.

AT&T Bell Laboratories (also known as Bell Labs), a research and development centre in Murray Hill, New Jersey, has been (and continues to be) the birthplace of many inventions in the fields of physics, engineering and, more interestingly, computer science For instance, the UNIX operating system and the C programming language were developed there.

It was at Bell Labs in 1975 that researchers Alfred V. Aho and Margaret J. Corasick were searching for a fast way to perform bibliographic searches. At the time, the widely used method of searching a given text T for a dictionary P was to repeatedly use an exact-string matching solution such as the Knuth-Morris-Pratt algorithm[2](or KMP Algorithm) on T for each p_i individually in the set P. Though a simple solution, this naive approach proved extremely slow when the cardinality of T and/or P was large.

Aho and Corasick's breakthrough was to combine the Knuth-Morris-Pratt algorithm, a clever and efficient linear-time method for exact-string matching, with a carefully designed finite automaton, pre-constructed from a known list of keywords. Running this automaton, with its KMP-inspired speed and

[2]The paper by Donald Knuth, James Morris, and Vaughan Pratt, "Fast Pattern Matching in Strings", was published by Stanford University in 1974, one year prior to Aho and Corasick's paper. The KMP paper gained worldwide notoriety when it was republished in the *SIAM Journal on Computing*, Volume 6, Issue 2, pp. 323-350, 1977.

tweaked for a particular dictionary, on an arbitrary body of text produced a very fast method for exact-set matching.

The resulting paper by Aho and Corasick, "Efficient String Matching: An Aid to Bibliographic Search" was published in the June 1975 issue of *Communications of the ACM* (Association for Computing Machinery, Inc.)

4. How Aho-Corasick Works

There are three main steps to accomplishing a linear-time exact-set search with the Aho-Corasick Automaton. (We assume there exists some dictionary P which is of interest as the subject of a search.)

i) Build a keyword tree K from the elements $p_1, p_2, \ldots p_n$ of the dictionary P containing n patterns.

ii) Create failure links within the tree K.

iii) Run the pattern matching automaton with the tree K, and using some arbitrary text T as input. Importantly, this step may be repeated *ad infinitum* with as many target texts as are desired.

We will look at each of these steps in detail.

4.i) Build the Keyword Tree

Begin with a tree composed of only a root node. Add each element p_i in P, such that a single character p_{i_j} is added at a time, and only if some prefix of the partial pattern $p_{i_1} \ldots p_{i_j}$ does not already exist in the tree.

In this way, create a branch using each of the characters in p_i, possibly as a sub-branch of an existing branch. Each character in p_i is represented by

an edge, the "trailing" node from that edge is labeled by the contents of p_i to that point (i.e. the prefix pattern $p_{i_1} \dots p_{i_j}$, where j is the character in p_i that we are currently adding). In this way, common prefixes are shared.

When the last character of some pattern p_i in P has been added, add an additional label to the "trailing" node from that edge which contains a corresponding number, i. This will be a numbered node, which the automaton will use as an indicator that the pattern p_i has been found. (see step iii).

An example of a prefix tree is shown in Figure 3. It is a slightly enhanced version of a tree which appears in Aho and Corasick (1975).

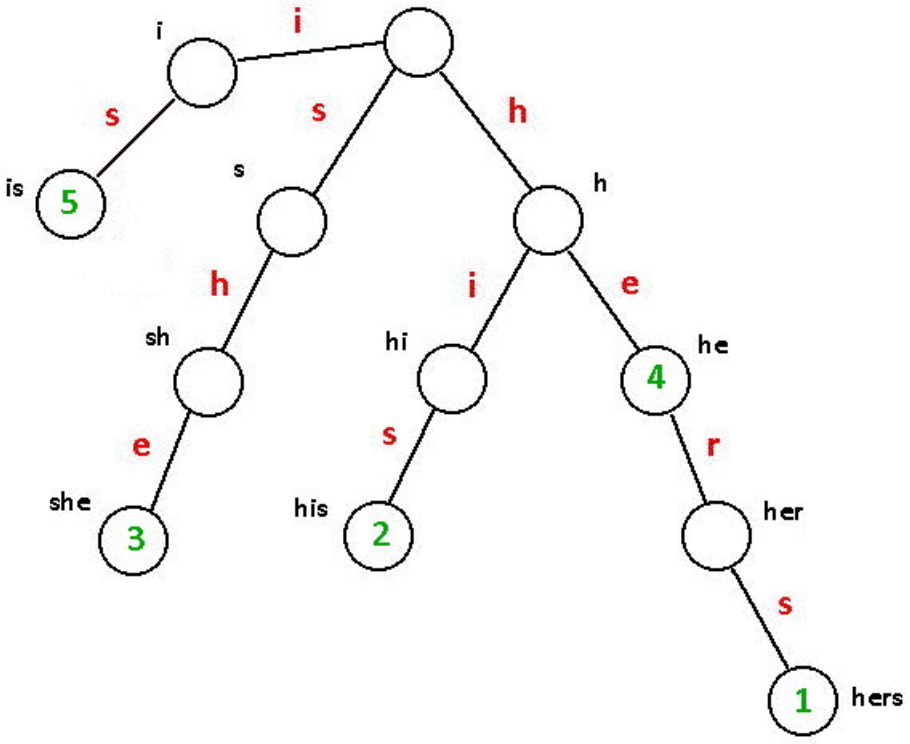

Figure 3: Sample prefix tree for $P = \{hers, his, she, he, is\}$

4.ii) Add the Failure Links

Next, take the newly created prefix tree K and add failure links to it[3]. To do this, perform an inorder traversal of the tree. At each node v, look for the longest suffix of v that exists elsewhere in the tree as a prefix. Create a directed path (v, n_v) where n_v is the node which has the same pattern label as v. If no such prefix pattern exists elsewhere in the tree, set $n_v = r$, where r is the root of K, and thus create a directed path (v, r).

Three facts are noteworthy here: first, since each edge emanating from the root is uniquely labeled, any vertex of depth ≤ 1 must have the root r as its failure link.

Secondly, Gusfield (1997) shows that failure links are unique; that is, for any node v in K, there exists only one corresponding failure link node n_v creating the unique ordered pair (v, n_v).

Finally, we note that consecutively linked failure links create a directed path. This feature will become important in part iii.

In figure 4 (next page) we present the same prefix tree as in Figure 3, but with failure links added. Failure links are shown as blue lines, except that those linking to the root have been omitted for clarity.

[3]Aho and Corasick (1975) present a linear-time algorithm for the construction of failure links for a given prefix tree; for brevity we have not included it here.

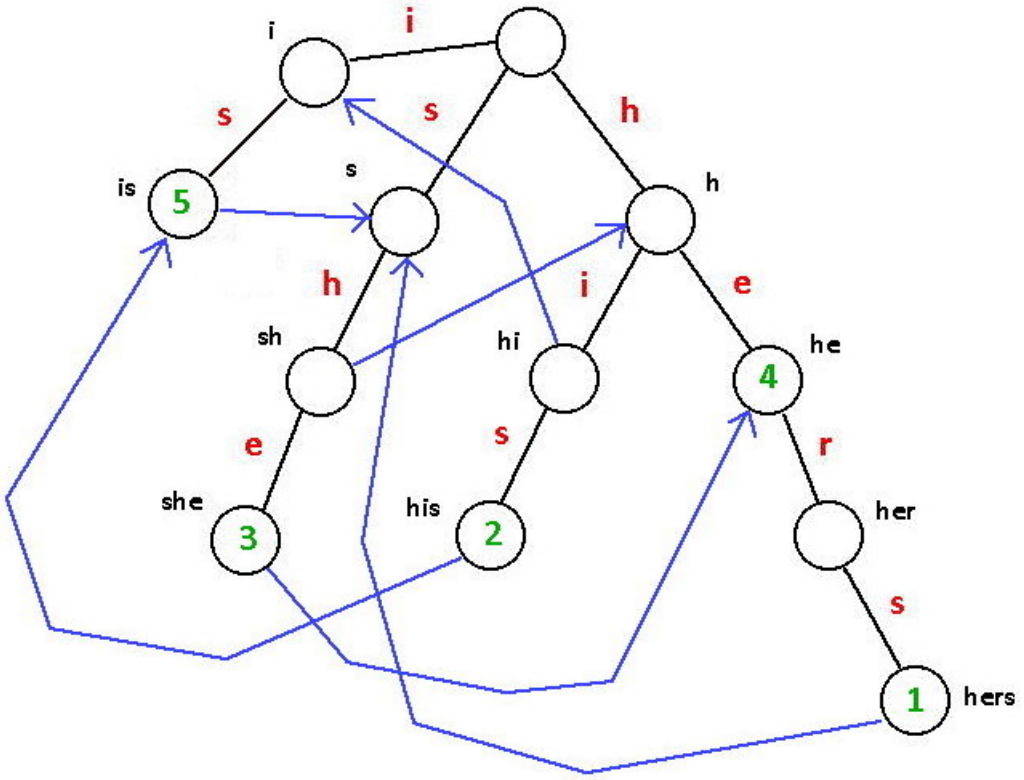

Figure 4: Sample prefix tree for $P = \{hers,\ his,\ she,\ he,\ is\}$, with failure links.

4.iii) Run the Aho-Corasick Automaton

We now present pseudocode for the ACA, which is due to Gusfield (1995). This algorithm is named "full AC Search" because it as extension of an earlier algorithm ("AC Search") which was simpler but placed an unwanted restriction on the dictionary elements: namely, that no pattern p_i in P can be a proper substring of any other pattern p_j in P. This restriction renders the AC algorithm much less useful in practice, so we will only examine the full algorithm. Note that we have edited the Gusfield algorithm slightly for clarity, and added some comments.

Algorithm full AC Search

```
01:   l := 1;          // l : starting pos of current search in the text
02:   c := 1;          // c : current character position in the text
03:   w := root;      // w : the node we are currently at in the tree
04:   repeat
05:       while there is an edge (w, w') labeled T(c)
06:       begin        // w' : some child of w that fits the description
07:           IF (w' is a numbered node), OR
08:               (there is a directed path of failure links from w'
09:               to a numbered node i)
10:           THEN
11:               report occurrence of P_i, ending at position c;
12:           w = w', and c = c + 1;
13:       end;
14:       w := n_w and l := c - lp(w);
15:   until c > n;
```

The notation $T(c)$ in line 5 indicates a single character of the text T, at position c. For example, if $T = abracadabra$, $T(3) = r$.

After initializing (lines 1 to 3), we begin a traversal of the tree. For each labeled edge (w, w') that we find, we examine the trailing node w' and determine if it is numbered. If it is, it indicates success - we have found a member of P in T - and we report it. If it is not numbered, but there is a failure link, we follow the trail of failure links and move our search to that portion of the tree. This allows us to avoid backtracking after a failed match - we can simply pick up the search again from the point we left off, at a different portion of the tree.

Continuing through the text T in this way, we create an occurrence report of every instance of a member of P in T.

The algorithm terminates when the character index, c, exceeds the length of T (line 15). The text T is traversed exactly once.

5. Running Time of the Aho-Corasick Automaton

The AC Automaton is composed of three stages: i) creation of the prefix tree from dictionary P, ii) adding the failure links, and iii) running the automaton with the text T as input.

For parts i) and ii), Aho and Corasick show in their paper (1975) that the prefix tree, with failure links can be created in linear time. We will not duplicate their proof here but instead provide a sketch. Consider a partially-completed tree in which we are to add a new pattern p_i. Since at each existing node, every edge is uniquely labeled, the time to add the new pattern will be at most $O(|p_i|)$. Since our alphabet is assumed to be of fixed length, the entire tree can be constructed on $O(n)$ time, where $n = \sum_{i=1}^{x_p} |p_i|$ and x_p is the number of patterns in P; and the failure function can be determined in constant time for each node.

For part iii), we need to consider the traversal of the tree with text T as input, and also the time taken by the output function (that is, the function which reports an occurrence of p_i in T). We will consider the time for this output to be k, where k is the number of matches; that is, a time unit of 1 for each match, to print the output. Then, by examining the AC algorithm from the previous section it is straightforward to see that we can run the automaton in $O(m + k)$ time, where $m = |T|$, and k is the number of matches.

Theorem (Gusfield, 1995): *If P is a set of patterns with total length n, and T is a text of total length m, then one can find all occurrences of T in patterns from P in $O(n)$ preprocessing time plus $O(m + k)$ search time, where k is the number of occurrences found.*

Thus the total running time for the automaton (including its construction) is $O(n + m + k)$, a linear bound.

6. Comparison With Other Methods, and Conclusions

It has been demonstrated already that the running time of the Aho-Corasick Automaton (ACA), with m the length of the text T, n the total (cumulative) length of all patterns in P, and k the total number of matches of P in T, is $O(n+m+k)$. Yet how significant is this, when compared to other algorithms?

First, compare the ACA with the naive exact-matching search, wherein T is searched once for each of the z patterns in P. Allowing that a single search can be run in $O(m)$ time (using the KMP Algorithm or others), there are z iterations of T, and $O(n)$ amount of work spent looking at the patterns. This results in a total running time of $O(n + mz)$, which is significant amount of time compared to the linear search time of the ACA.

Clearly, the ACA is more efficient than naive exact set matching algorithms. However, there is another highly efficient algorithm that can be considered. Rather than producing a keyword tree K of the patterns in P, what if a suffix tree S of the text T was created, and each pattern was searched for in the suffix tree? In this case, the suffix tree can be built in $O(m)$ time (this is due to Ukkonen[4]), and the search itself can be conducted in $O(n)$ time. Then for k matches, the running time is also $O(n + m + k)$, the same as the ACA algorithm. However, there is distinct difference between the two algorithms.

When working with two algorithms with similar running time and data of sufficiently large size, it is useful to consider constant work being done such as that of the size of the trees being built, and the work done in using each tree. The Aho-Corasick Automaton, which uses a keyword tree and performs its search in $O(m)$ time, builds a tree of size n in $O(n)$ time. The suffix tree will perform its search in $O(n)$ time, and build its tree of size m in $O(m)$ time.

Consider a dictionary P with size significantly larger than its corresponding search text T, and conversely a text T which is significantly larger than the dictionary P. In the first case, the suffix tree will be produced with a much smaller size, and in the second case the keyword tree will be constructed with a much smaller size. However, while the sizes of the trees are

[4]E. Ukkonen. (1995) On-line construction of suffix trees. Algorithmica 14(3):249-260.

relative to the problem, so is the constant work performed the algorithms it is a trade off between the tree and the search. Therefore, the work for the search will be greater with the suffix tree in the former problem, and will be greater for the keyword tree in the second problem.

Since this balance exists, neither algorithm can be universally determined to be superior, but one may be more efficient than the other based on the system (dictionary and text) on which the searches are being run. Clearly, for massive searches which will take significant amounts of real-world time, the problem system should be examined and the appropriate algorithm chosen based on whether the text is relatively large, or the dictionary is relatively large. Otherwise (that is, for reasonably balanced systems, or for smaller data sets) the Aho-Corasick algorithm is likely the better choice based on the easy construction of prefix trees.

Finally, it should be noted there exist certain specific cases for the suffix tree algorithm that will allow it to be modified to fit both the conditions mentioned above, trading off the size of the tree for search time. (However, the running time will still not be increased beyond that of $O(n + m + k)$.) This advantage is not shared in the ACA, and therefore the suffix tree may be considered more versatile for these specific cases.

The Aho-Corasick's efficiency, coupled with its relative simplicity make it an ideal algorithm for many types of exact-set searches. The lack of a requirement to preprocess the text T makes the linear-time performance of the AC algorithm especially impressive. Once a dictionary of interest has been processed, the resulting automaton can be put to work on arbitrary texts, without further preprocessing.

7. Bibliography

1. Aho, Alfred V. and Corasick, Margaret J. *Efficient String Matching: An Aid to Bibliographic Search.* Communications of the ACM, June 1975, Volume 18, Number 6, pp 333-340.

2. Aoe, Jun-ichi. Computer Algorithms: String Pattern Matching Strategies. Los Alamitos, CA: IEEE Computer Society Press, 1994.

3. Gusfield, Dan. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge, England: Cambridge University Press, 1997.

4. Jones, Neil C. and Pevzner, Pavel A. An Introduction to Bioinformatics Algorithms. Cambridge, MA: The MIT Press, 2004.