# Aho-Corasick Pattern Matching Automata

Abrie Greeff
B.Sc Hons (Computer Science)
Department of Computer Science
University of Stellenbosch

March 9, 2006

# 1 Introduction

Pattern matching has many applications in every day computer use. Pattern matching is applied in areas such as searching text documents, text scanners for compilers and web search engines. This paper considers one method of searching for patterns called the Aho-Corasick method. This method was proposed by A. Aho and M. Corasick in 1975 [1]. This method consists of two phases, a pre-processing phase and a search phase.

In the pre-processing phase the search patterns are loaded from a text file and a trie is constructed. A trie is an ordered tree where each child of each node is labelled with a character of the alphabet [2]. In this trie the alphabet will be the alphabet of the deterministic finite automaton (DFA) associated with the trie. The failure functions [3] of the patterns will then be computed on this trie. Failure functions are transitions that occur in the trie when a certain pattern has failed but another pattern may have started occurring.

The second phase consists of searching through a text file for these patterns. The searching is done by traversing the trie and accepting if a pattern has been matched. When a pattern isn't found but there is a failure function the failure function will be followed to watch for other possible patterns. Failure functions will also be followed when a state has accepted because it is possible that two or more patterns may be found nested together.

# 2 Formal Description

## 2.1 Deterministic Finite Automaton

A deterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the states,
2. $\Sigma$ is a finite set called the alphabet,
3. $\delta$:Q $\times\Sigma \rightarrow$ Q is the transition function,
4. $q_0 \in$ Q is the start state, and
5. F $\subseteq$ Q is the set of accept states.

## 2.2 Failure Functions

A failure function is defined as the transition to the state with the longest matching suffix of the current prefix of a pattern being matched.

# 3 Implementation

The Aho-Corasick algorithm was implemented in Sun Java 2 SDK build 1.5.0. The algorithm was developed in three phases, the trie construction phase, the failure function construction phase and the search phase.

## 3.1 Trie Construction Phase

The trie construction phase was split into two phases, the parse phase and the construction phase.

### 3.1.1 Parse Phase

The parse phase of the algorithm opens the pattern text file and attaches to a stream tokenizer. The stream tokenizer splits the pattern file into a array of patterns that must be searched for in the search phase.

### 3.1.2 Construction Phase

The construction phase first creates the start state of the trie. Every node in the trie uses an ArrayList to store the transitions to their children. This implementation was chosen because the ArrayList can be indexed with a character to find the correct transition but no extra space is wasted because the ArrayList is a dynamic array and thus only contains the necessary characters of the alphabet needed to represent the transitions. In this phase every node that is created has its failure function set to null which means when it fails at this state the failure function returns to the start state of the trie. The trie is then constructed recursively from the start state by traversing with every pattern in the pattern file and adding transitions and accept states when it's necessary.

## 3.2 Failure Function Construction Phase

The construction of the failure function consists of traversing the whole trie and storing the patterns that must occur to reach each state. At each state this pattern is traversed through the trie again until the longest suffix that can reach another state is reached. This state is then stored as the state to where the failure function must jump to when it is called. States that doesn't find a state its failure function must jump to is left as null. The length of the longest suffix is also stored with each state to make the searching phase easier.

## 3.3 Search Phase

The dictionary file that must be searched for the patterns is also attached to a stream tokenizer. Every string token that is read from the file is then

read one character at a time. These characters are then used to traverse the trie. If an accept state is reached the count for that pattern is increased. If a state fails the failure function is called and the trie is traversed further until the end of the string token. The next token is then read and the process is repeated. After the whole file has been searched the results are displayed.

# 4 Conclusion

The performance that was observed on this algorithm was quite good. The algorithm was tested on the linux dictionary (linux.words) which consists of 483523 words. Different permutations of possible patterns was searched for in this file. The time it took to complete all the phases was 0.72 seconds on average. This can be improved if all the output to the screen is removed. The algorithm may be further improved by replacing the ArrayList with a different schema. Possible schemas that may offer better performance include a hash table, a triple array or a balanced tree.

# 5 References


1. A. Aho and M. Corasick. *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, 18(6):333-340, 1975.

2. M. Goodrich and R. Tamassia *Data structures and algorithms in JAVA*,2nd edition, 2001, p.512.

3. M. Crochemore and C. Hancart. *Handbook of Formal Languages*, p.407-422.

4. J. Nieminen. *Efficient implementation of Unicode string pattern matching automata in Java*, 2005.

5. M. Sipser *Introduction to the Theory of Computation*, 1st edition, 1997, p.35.