UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

EMERSON LEONARDO LUCENA

OPTIMIZING AHO-CORASICK FOR WORD COUNTING

CAMPINA GRANDE - PB

2020

EMERSON LEONARDO LUCENA

OPTIMIZING AHO-CORASICK FOR WORD COUNTING

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Rohit Gheyi.

CAMPINA GRANDE - PB

2020

L935o Lucena, Emerson Leonardo.
Optimizing Aho-Corasick for word counting. / Emerson Leonardo Lucena. – 2020.

12 f.

Orientador: Prof. Dr. Rohit Gheyi.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Aho-Corasick algoritm. 2. Pattern matching. 3. Word counting. 4. Algoritmo offline. 5. Processamento de textos. I. Gheyi, Rohit. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

EMERSON LEONARDO LUCENA

OPTIMIZING AHO-CORASICK FOR WORD COUNTING

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Trabalho aprovado em: 2020.

CAMPINA GRANDE - PB

Optimizing Aho-Corasick for Word Counting

Emerson Leonardo Lucena
emerson.lucena@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

Rohit Gheyi
rohit@dsc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

ABSTRACT

The Aho-Corasick algorithm is used to recognize all occurrences of a set of strings in a text. Its time complexity is $O(m + n + o)$, where m is the sum of the lengths of the keywords, n is the text length and o is the number of occurrences of all keywords in the text. However, when the input contains a large amount of matches with the dictionary patterns, the algorithm performance decreases. In many domains, such as information retrieval, natural language processing and DNA sequence analysis, Aho-Corasick is used for word counting, possibly with many repetitions. In this paper, we improve the Aho-Corasick algorithm to count the number of occurrences of a set of words in a text. The new algorithm works offline and does not depend on the frequencies of the dictionary words. Its time complexity is $O(m + n + u)$, where u is the number of distinct keywords found in the text, and its space complexity is the same as the Aho-Corasick. We compare the original and the new algorithm performances with texts varying up to 100MB and dictionaries with sizes 1KB, 1MB and 10MB. The new algorithm performed better in every experiment made, from 50% to 300% faster in comparison with the Aho-Corasick.

CCS CONCEPTS

• **Theory of computation** → **Pattern matching**; • **General and reference** → *Performance*.

1 INTRODUCTION

Multiple pattern matching is the task of finding occurrences of a set of words in a text. It is used in applications from many areas, such as information retrieval [7], natural language processing [8], intrusion detection [15] and DNA sequence matching [14]. The Aho-Corasick algorithm [1] is commonly used for this purpose. It extends the Knuth-Morris-Pratt [6] algorithm to a prefix tree, using concepts of finite state machines [11] to search for multiple patterns at once. It is used in many real-world applications. For example, the Unix command GNU Grep [4] uses Aho-Corasick to match multiple patterns in files.

However, this algorithm outputs every occurrence of every dictionary word in the text. Thus, its runtime depends on the total number of occurrences. It is possible that a large number of keyword occurrences slows down the execution of the algorithm. For example, consider the worst case where the keywords are {`a, aa, aaa, aaaa, ... , a`$^{\mathtt{k}}$} and the text is $\mathtt{a}^{\mathtt{n}}$ ($\mathtt{a}^{\mathtt{m}}$ means a concatenation of m characters `a`). Every character of the text will contain up to k new occurrences. Therefore, the Aho-Corasick algorithm (and every other pattern matching algorithm) would have to print each one of the $O(nk)$ occurrences of words.

Since the output itself depends on the number of occurrences of words, the time complexity cannot be better than the one achieved by Aho-Corasick. However, in some applications (for example, to count n-grams of a document) we will not need the positions of the occurrences in the text. In these cases, we are only interested in the frequencies of the keywords (i.e., the number of times the keywords appear in the text).

In this paper, we present a modification to the Aho-Corasick algorithm, called WORD COUNTING AHO-CORASICK, that outputs the frequencies of each keyword in a text instead of listing every occurrence and the positions where they happen. Both algorithms are divided into two steps: building a pattern matching machine with a set of keywords and searching a text with it. The method of building the structure is similar for both algorithms and the time complexity is $O(m)$, where m is the sum of the lengths of all keywords. For the search step, the WORD COUNTING AHO-CORASICK time complexity is $O(n + u)$, where n is the text length and u is the number of distinct keywords found in the text. It has a better time complexity compared to the Aho-Corasick algorithm, which is $O(n{+}o)$, where o is the number of occurrences of all keywords in the text. The WORD COUNTING AHO-CORASICK is an *offline* algorithm, which means the output is calculated only after the whole input has been fed. In some cases, the online nature of the Aho-Corasick is useful, for example in intrusion detection [15].

The WORD COUNTING AHO-CORASICK performed better than the original algorithm on every experiment made. We used keyword sets of sizes 1KB, 1MB and 10MB, two alphabets (DNA bases and alphanumeric characters) and texts varying up to 100MB (over 100 million characters). For most pattern sets, the proposed algorithm execution times were 100% faster than the original Aho-Corasick ones, and for one pattern set they were 300% faster.

This paper is organized as follows: Section 2 defines our notations and discusses approaches to the pattern matching problem. Section 3 presents our algorithm, the WORD COUNTING AHO-CORASICK. In Section 4 we present our experiments and discuss the achieved results. Finally, in Section 5 we make our conclusions and point out options for future work.[1]

2 BACKGROUND

In this section, we provide an example to the multiple pattern matching problem, give an overview of the previous approaches to the problem, briefly explains the Aho-Corasick algorithm and discusses the problem of having a large number of word occurrences in the text.

[1]The authors retain the rights, under a Creative Commons Attribution CC BY license, to all content in this article (including any elements they may contain, such as pictures, drawings, tables), as well as all materials produced by authors that are related to the reported work and are referenced in the article (such as source code and databases). This license allows others to distribute, adapt and evolve their work, even commercially, as long as the authors are credited for the original creation.

2.1 Example

Next we use the following notation: the data in which the words are searched is called *text* and its length is n; the searched words are called *patterns* or *keywords* and the set of patterns is called a *dictionary*; a dictionary contains k patterns and the sum of the lengths of all patterns is m; *alphabet* is the set of symbols used in the text and the patterns; the number of occurrences, or the frequency, of a pattern p in a text t is the number of indexes of t where p appears as a substring.

The exact multiple pattern matching problem receives as input a text and a dictionary and outputs all the occurrences of each pattern in the text. For example, if we are looking for the words cab, ab and aba in the text cababaab, then we have one occurrence of cab, three occurrences of ab and two occurrences of aba. Table 1 shows the locations of such occurrences in the text.

Table 1: String matching example for patterns cab, ab and aba, and text cababaab.

Position	Text Prefix	Occurrences
1	**c**ababaab	
2	**ca**babaab	
3	**cab**abaab	ab, cab
4	**caba**baab	aba
5	**cabab**aab	ab
6	**cababa**ab	aba
7	**cababaa**b	
8	**cababaab**	ab

2.2 Previous Approaches

The simplest algorithm for this task is the brute force one: for every starting position, scan the text and the pattern and compute whether there is an occurrence of the pattern starting there. The time complexity is $O(nm)$.

A more efficient solution is the Knuth-Morris-Pratt algorithm [6]. It builds an automaton that recognizes occurrences of one keyword. Figure 1 shows an example of an automaton for the keyword aaba, which will be explained later in this section. Unlike the brute force algorithm, we only need to scan the text once per pattern. Hence, the time complexity is $O(nk + m)$.

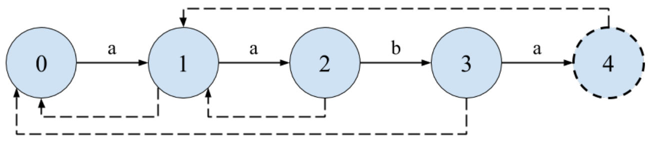


Figure 1: Automaton that finds occurrences of aaba. The state with a dashed circle represents an occurrence of aaba.

The brute force solution is inefficient as for each pattern the text needs to be scanned multiple times. For the Knuth-Morris-Pratt algorithm be able to search the text in only one pass, it needs to keep track of all the possible prefixes of the keyword that matches with the current text at each iteration of the scan.

For example, consider that we are searching for the pattern aaba in text baaaba. At each moment, we have knowledge of not only the longest prefix of the keyword but also every prefix of the keyword that matches the text up to the current index. Therefore, if the next symbol of the text does not match with the next symbol of the pattern, we do not need to backtrack the text (like the brute force would do). We only need to discard that keyword prefix and use the second largest one. See Table 2 for this example. In the first iteration, there is no matching prefix, since the first letter of the pattern is a and the first letter of the text is b. Note that, in baa, we have two possible pattern prefixes, a and aa. This is the difference between the Knuth-Morris-Pratt and the brute force, which only keeps the longest matching pattern prefix: when moving to the next iteration baaa, the pattern prefix aa cannot be expanded to aab and the matching fails. However, we do not need to move back in the text. Instead, we use the next possible prefix that can be matched with the next letter of the text, which is a.

Table 2: Prefixes of the pattern aaba for each iteration of the text baaaba.

Text prefix	Pattern prefixes
b	
ba	a
baa	a, aa
baaa	a, aa
baaab	aab
baaaba	a, **aaba**

We keep track of the next possible prefix thanks to the *failure function*. The failure function of a prefix points to the longest prefix which is also a proper suffix of it. In other words, a failure function points to the second longest prefix that can also match the current text. See Figure 1 for an example. The failure functions are drawn as dashed arrows. There, $failure(\text{aaba}) = \text{a}$, $failure(\text{aa}) = \text{a}$, and both $failure(\text{aab})$ and $failure(\text{a})$ equals to an empty string.

2.3 Aho-Corasick

The Aho-Corasick [1] algorithm uses the concept of Knuth-Morris-Pratt and extends to a prefix tree, or a *trie* [2], to find occurrences of multiple patterns simultaneously (see Figure 2 for an example, which will be explained later in this section). Thus, we only need to scan the text once. The algorithm can be divided into two parts: building the structure and searching the text. Building the structure only depends on the patterns and takes time proportional to the sum of their lengths (i.e., the time complexity for constructing the machine is $O(m)$). The search step does not depend on the number of patterns. Since it only needs to pass through the text once, and it outputs each occurrence, the time complexity for searching a text is $O(n + o)$, where o is the total number of occurrences of all patterns in the text. Hence, the total time complexity for searching a set of keywords in a text is $O(n + m + o)$.

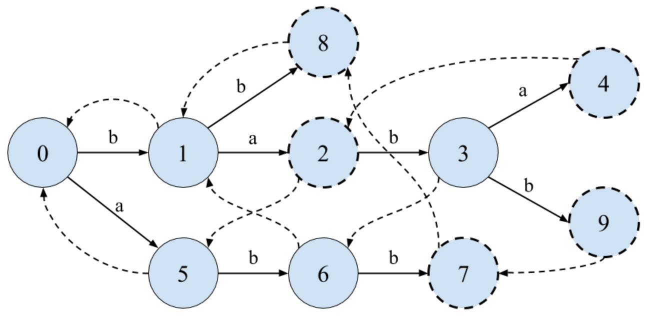


Figure 2: Automaton that finds occurrences of ba, baba, abb, bb and babb. Dashed circles represent keyword endings.

2.3.1 Combining more automata. The Aho-Corasick uses the automata introduced by Knuth-Morris-Pratt to search for multiple keywords simultaneously. Similarly, the failure function of a state in the prefix tree will point to the longest prefix of smaller depth which is also a suffix of the current state, among all the keywords of the dictionary. See Figure 2 for an example. The failure functions are drawn as dashed arrows.

Note that a failure function of a state can lead to a prefix of another word. For example, in Figure 2, the failure of state 3 (bab) is state 6 (ab), and the failure of state 6 (ab) is state 1 (b). Also note that we may have multiple occurrences simultaneously. Since the failure of state 7 (abb) is state 8 (bb), every occurrence of abb implies in another occurrence of bb.

2.3.2 Searching a text. Algorithm 1 was proposed in the original Aho-Corasick paper [1] and explains how to search the input text with an already built pattern matching machine. The *goto function* receives a state and a symbol and looks for a transition in the dictionary prefix tree using that symbol. If there is no valid transition, it returns *fail*. A failed *goto* transition from the root points to itself. For example, in Figure 2, $goto(1, \mathsf{b}) = 8$ and $goto(2, \mathsf{a}) = fail$. The *output function* returns the set of keywords that appears in one state. In Figure 2, $output(4) = \{\mathsf{baba}, \mathsf{ba}\}$, $output(9) = \{\mathsf{babb}, \mathsf{abb}, \mathsf{bb}\}$ and $output(3) = \emptyset$. Note that the output function traverses the prefix tree using the failure functions until it reaches the root (i.e., the empty prefix). Whenever it finds a state that represents the end of a pattern (shown in the Figure 2 as dashed circles), it adds that word to the set of found patterns.

Table 3 shows how the algorithm works for the text abbababba and the automaton shown in Figure 2.

2.3.3 Problem. Due to the nature of the output function, we can see that multiple keywords may appear in a single position of the text. There will be cases where an occurrence of a state will be counted many times, especially the states closer to the root, and might slow down the execution of the algorithm. If we change the output to count the frequencies of the words instead of their locations, we can modify the search algorithm in such a way that each state is visited at most once.

Algorithm 1 Aho-Corasick search algorithm. This algorithm uses an already built state machine, receives an input text and outputs the occurrences of the keywords and their locations.

Input: A string $x = a_1 a_2 \dots a_n$ where each a_i is an input symbol, a pattern matching machine M with a *goto function*, a *failure function* and an *output function*.
Output: Occurrences of keywords in x, indexed by their position.

```
function AHO-CORASICK($x, M$)
    $occurrences \leftarrow \emptyset$
    $state \leftarrow 0$
    for $i \leftarrow 1$ until $n$ do
        while $goto(state, a_i) = fail$ do
            $state \leftarrow failure(state)$
        $state \leftarrow goto(state, a_i)$
        if $output(state) \neq \emptyset$ then
            $occurrences(i) \leftarrow output(state)$
    return $occurrences$
```

Table 3: Execution of Algorithm 1 for the text abbababba and the automaton shown in Figure 2.

Processed text	Action	state	output(state)
	start search	0	
abbababba	$state = goto(state, \mathsf{a})$	5	
abbababba	$state = goto(state, \mathsf{b})$	6	
abbababba	$state = goto(state, \mathsf{b})$	7	abb, bb
	$state = failure(state)$	8	
	$state = failure(state)$	1	
abbababba	$state = goto(state, \mathsf{a})$	2	ba
abbababba	$state = goto(state, \mathsf{b})$	3	
abbababba	$state = goto(state, \mathsf{a})$	4	baba, ba
	$state = failure(state)$	2	
abbababba	$state = goto(state, \mathsf{b})$	3	
abbababba	$state = goto(state, \mathsf{b})$	9	babb, abb, bb
	$state = failure(state)$	7	
	$state = failure(state)$	8	
	$state = failure(state)$	1	
abbababba	$state = goto(state, \mathsf{a})$	2	ba

3 WORD COUNTING AHO-CORASICK

First of all, recall that the failure function points to a smaller prefix which also matches the current text. Therefore, the failure function always points to a state with a smaller depth in the prefix tree. Since every state with a nonempty prefix has a failure link, we can build a directed tree, where the edges are the failure transitions and the root is the root of the prefix tree (i.e., the empty prefix). We will call it the *failure tree*. Figure 3 shows the failure tree of the example given in Figure 2.

The failure tree allows us to intuitively see how the Aho-Corasick output function works. For a state s, whenever there is an occurrence of s in the text, there are also occurrences of every state in the path from s to the root. So, by traversing the tree using the

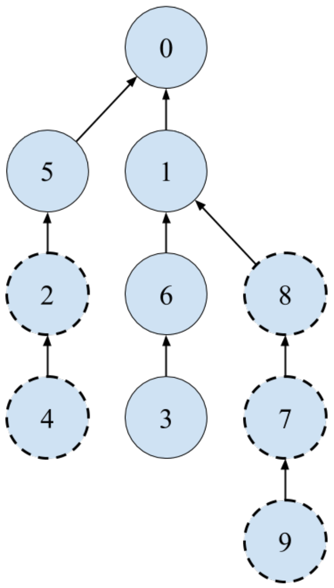


Figure 3: Failure tree of pattern matching machine given in Figure 2.

failure links, the algorithm prints every keyword that appears in one position of the text.

For the sake of efficiency, the original Aho-Corasick paper [1] suggests that $output(state)$ be implemented using linked lists. So, if $state$ marks the ending of a pattern (shown in this paper as a dashed circle), then $output(state)$ will be a linked list with that pattern and a pointer to $output(failure(state))$. Otherwise, $output(state)$ will be equal to $output(failure(state))$. This approach skips the internal states which are not an ending of any keyword. Thus, $output(state)$ will be linear on the number of occurrences of keywords in $state$. Table 4 shows the output function for each state in the example given in Figure 2.

Table 4: Output function for each state in example of Figure 2.

state	output(state)
0	$\emptyset$
1	$\emptyset$
2	`{ba}`
3	$\emptyset$
4	`{baba, ba}`
5	$\emptyset$
6	$\emptyset$
7	`{abb, bb}`
8	`{bb}`
9	`{babb, abb, bb}`

3.1 Algorithm

We will modify the search algorithm and the output function in such a way that each state is visited at most once. The modified search will make the same transitions as the original algorithm, but instead of printing at each iteration the occurrences in that position, the algorithm will increment a counter in the current state. That counter indicates how many times that state appeared in the text. Since an occurrence of $state$ implies an occurrence of $failure(state)$, after scanning the text we will propagate the counter of each state to its parent on the tree. In other words, $counter(failure(state))$ will be incremented by $counter(state)$.

For the counters to be properly propagated through the failure links, the only requirement is that the states are processed in a topological order [2, 12, 13]. In other words, for each pair of states u and v such that $failure(u) = v$, a topological ordering ensures that u is processed before v. Since our structure is a tree, there is always a correct topological ordering and we can achieve that by traversing the tree in a depth-first-search manner.

To maintain efficiency of the algorithm, we will introduce two functions: *patternId* and *outputLink*. If $state$ is marked as a pattern ending, then *patternId*$(state)$ will be equal to the index of that word in the set. *outputLink*$(state)$ points to the first state in the failure tree path to the root which is marked as a pattern ending, or the root itself. Algorithm 2 shows how to calculate *outputLink*. Both functions can be processed during the construction of the pattern matching machine and can be stored as a one-dimensional array and accessed in constant time.

Algorithm 2 Calculation of *outputLink*.

```
function CALCULATE_OUTPUTLINK($state$)
    if $state$ is marked as a keyword ending or $state = 0$ then
        $outputLink(state) \leftarrow state$
    else
        $outputLink(state) \leftarrow outputLink(failure(state))$
```

Thanks to the *outputLink* function, we can traverse the tree using only the relevant states (i.e., states marked as a pattern ending). Now, we can create another tree, called *outputTree*, including only the marked states which appear in the text. Thus, a traversal in $outputTree$ will only use states for keywords that have at least one occurrence in the text. Figure 4 shows the $outputTree$ for the example of Figure 2 with the text `abbababba`.

Algorithm 3 searches a text and outputs the number of occurrences of each keyword in the dictionary. Since we are traversing the $outputTree$, only the states that appear in the text at least one time are visited.

For example, for a set of words $K = \{\text{ba}, \text{baba}, \text{abb}, \text{bb}, \text{babb}\}$ and an input text `abbababba`, the Algorithm 3 will output a set of tuples $\{(0,3),(1,1),(2,2),(3,2),(4,1)\}$. The first element of the tuple is the index of the pattern in the dictionary and the second element is the number of occurrences of that pattern. In other words, if the pattern K_i appeared in the text j times ($j > 0$), then Algorithm 3 will include tuple (i, j) in its output.

3.2 Analysis

Let u be the number of distinct patterns found in the text. Since the text is scanned only once, the size of $outputTree$ is u and each state in $outputTree$ is visited only once, then the time complexity of Algorithm 3 is $O(n+u)$. Recall that the time complexity of the original

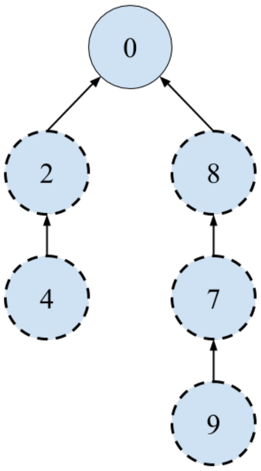


Figure 4: outputTree of pattern matching machine given in Figure 2 with input text abbababba.

Algorithm 3 Word Counting Aho-Corasick. This algorithm scans the input text, creates a tree with the appearing output nodes and traverses it, and returns the frequencies of each keyword. Keywords with no occurrences are not included in the output.

Input: A string $x = a_1 a_2 \dots a_n$ where each a_i is an input symbol, a pattern matching machine M with a *goto function* and a *failure function*.

Output: Set of tuples (i, j) which indicates ith keyword appeared j times in x $(j > 0)$.

```
function WORD COUNTING AHO-CORASICK($x, M$)
    $counter \leftarrow \emptyset$
    $outputTree \leftarrow \emptyset$
    $state \leftarrow 0$
    for $i \leftarrow 1$ until $n$ do
        while $goto(state, a_i) = fail$ do
            $state \leftarrow failure(state)$
        $state \leftarrow goto(state, a_i)$
        $s \leftarrow outputLink(state)$          ▷ Jump to next occurrence
        $counter(s) \leftarrow counter(s) + 1$
        while $s \neq 0 \wedge s \notin outputTree$ do
            $outputTree \leftarrow outputTree \cup \{s\}$
            $s \leftarrow outputLink(failure(s))$
    TRAVERSE_TREE(0)
    $occurrences \leftarrow \emptyset$
    for all $s \in outputTree$ do
        $occurrences(patternId(s)) \leftarrow counter(s)$
    return $occurrences$

function TRAVERSE_TREE($s$)
    for all $c \in outputTree \mid outputLink(failure(c)) = s$ do
        TRAVERSE_TREE($c$)
        $counter(s) \leftarrow counter(s) + counter(c)$
```

Aho-Corasick search algorithm is $O(n + o)$, where o is the total number of keyword occurrences in the text. Also note that $u \leq o$ (i.e., the number of keyword occurrences is at least the number of distinct found keywords). In the worst case, each pattern appears only once in the text ($u = o$). In that case, Algorithm 3 and 1 will have similar runtimes. But the more a keyword is found repeated times, the better Algorithm 3 performs in comparison with the original Aho-Corasick algorithm.

The additional structures used for this modification can be stored as a one-dimensional array, allowing direct access in constant time. The pattern matching machine can be built in a similar way to the original algorithm. Hence, the time complexity to create the structure is $O(m)$, with the same space complexity as the Aho-Corasick. The *outputTree* edges can be stored using adjacency lists, so the tree traversal time is proportional to the number of nodes. Therefore, the total time complexity of the Word Counting Aho-Corasick, to create the machine and search for a text, is $O(m + n + u)$.

3.3 Properties

This section shows that Algorithm 3 produces valid outputs. We will first define properties of the failure tree, to then advance to the *outputLink* function, the *outputTree*, the *traverse_tree* function and finally the Algorithm 3 itself.

For the next lemmas, we say that $failure^k(s)$ is the result of applying the failure function k times, starting from the initial value s. For example, $failure^3(s) = failure(failure(failure(s)))$. In particular, $failure^0(s) = s$. We also say a state is a keyword ending if we follow the *goto* links of some keyword and end in that state. In this paper, we represent the keyword ending states in the images as dashed circles.

The first and second lemmas define properties of the failure tree.

Lemma 3.1. *Let s and t be two distinct states in the failure tree. Then, there is a path from s to t in the failure tree if and only if $t = failure^k(s)$, for some $k \geq 1$.*

Proof. We will prove this lemma by induction on the failure tree depth. By the failure tree definition, there is a directed edge from s to t if and only if $failure(s) = t$. Since the failure function is not defined for the root, the root has no outgoing edges, thus the lemma is true for depth 0. Now, assume the lemma is true for all states with depth $\leq i$. Then, for all states s where $depth(s) = i + 1$, let $f = failure(s)$. It is clear that $depth(f) = i$, so the lemma is true for f. Since there is an edge from s to f, then f is reachable by s and every state t reachable by f is also reachable by s, using one more application of the failure function (i.e., if $t = failure^k(f)$ and $f = failure(s)$, then $t = failure^{k+1}(s)$). Thus, the lemma is true for depth $i + 1$ and the proof is complete. □

Lemma 3.2. *Let t be a state where a keyword K_i ends. Then, $K_i \in output(s)$ if and only if t is reachable by s in the failure tree.*

Proof. By the output function definition, $output(s)$ contains every keyword that appears as a suffix of s. By definition, we also know that $failure(s)$ points to the longest proper suffix of s that is also a prefix of a keyword (both affirmations were proven in the original Aho-Corasick paper [1]). Then by Lemma 3.1, this statement is true. □

The following lemma defines the *outputLink* function.

Lemma 3.3. *$outputLink(s) = t$ if and only if, among all p where p is either a keyword ending state or the root and p is reachable by s, t has the maximum depth.*

Proof. We will prove this lemma by induction on the failure tree depth. By Algorithm 2, $outputLink(root) = root$ (state 0 is the root, or the empty prefix), so the lemma is true for depth 0. Now, assume the lemma is true for all states with depth $\leq i$. Then, for all states s where $depth(s) = i + 1$, if s is a pattern ending state, then $outputLink(s) = s$ and the statement holds true. If s is not a pattern ending state, then $outputLink(s) = outputLink(failure(s))$, and by Lemmas 3.1 and 3.2, $outputLink(s)$ is reachable by s. Since $depth(failure(s)) \leq i$, then the statement holds true, the lemma is true for depth $i + 1$ and the proof is complete. □

The remaining lemmas describes the *outputTree* and how the function *traverse_tree* uses it to correctly produce the output.

Lemma 3.4. *State $s \in outputTree$ if and only if s is either the root or a keyword ending state that happens in the input text at least once.*

Proof. In Algorithm 3, after each *goto* transition, we traverse the failure tree from the current state s to the root, through the *outputLink* function. By Lemmas 3.2 and 3.3, every keyword ending state reachable by s is a suffix of s and will be added in *outputTree* during this loop. □

Lemma 3.5. *During Algorithm 3, for every state $s \in outputTree$, $traverse\_tree(s)$ will be called once.*

Proof. We will prove this lemma by induction on the depth of *outputTree*. Algorithm 3 explicitly calls $traverse\_tree(0)$, so the statement is true for depth 0. Now, assume the lemma is true for all states with depth $\leq i$. Then for a state s where $depth(s) = i + 1$, let $f = outputLink(failure(s))$. We know that $traverse\_tree(s)$ can only be called by $traverse\_tree(f)$. By Lemmas 3.3 and 3.4, we know that for every state $t \neq root$, if $t \in outputTree$, then $outputLink(failure(t)) \in outputTree$. Since $f \in outputTree$ and $depth(f) = i$, then the statement is true for f and $traverse\_tree(f)$ will be called once, therefore $traverse\_tree(s)$ will be called once as well. Then the lemma is true for depth $i + 1$ and the proof is complete. □

Lemma 3.6. *For all states $s \in outputTree$ different from the root, after $traverse\_tree(s)$ finishes, $counter(s)$ will be equal to the number of occurrences of the keyword ending in s.*

Proof. During the Algorithm 3 main loop, after each *goto* transition, the counter of the current state is incremented. Afterwards, during *traverse_tree*, the topological order of the traversal guarantees that if $depth(s) < depth(t)$, then $counter(t)$ is calculated before $counter(s)$. By Lemma 3.2, if s is reachable by t in the failure tree, then every occurrence of t implies in an occurrence of the keyword ending in s. Thus, $counter(s)$ is the sum of $counter(t)$, for all t such that $depth(s) < depth(t)$ and there is an edge between s and t in the *outputTree*. □

Theorem 3.7 addresses the validity of the Word Counting Aho-Corasick.

Theorem 3.7. *Algorithm 3 produces valid outputs.*

Proof. By Lemma 3.4, *outputTree* contains every keyword ending state that happens in a text at least once. Furthermore, by Lemma 3.5, for every s in *outputTree*, function $traverse\_tree(s)$ will be called once. Finally, by Lemma 3.6, after traversing the tree, every keyword ending state s will have the correct value of $counter(s)$ and will be mapped to the keyword index $patternId(s)$. Therefore, the output produced by Algorithm 3 is valid. □

4 EVALUATION

In this section, we will evaluate the new algorithm, comparing with the original Aho-Corasick. We will establish research questions, plan the experiments, discuss the results and point out threats to validity.

4.1 Goal and Research Questions

Our main goal for the evaluation is to compare the time performance of both the original Aho-Corasick and the Word Counting Aho-Corasick. Specifically, we have three questions:

RQ$_1$ To what extent our algorithm is better than the original one with respect to different text lengths?
RQ2 To what extent our algorithm is better than the original one with respect to different dictionary sizes?
RQ3 To what extent our algorithm is better than the original one with respect to different alphabet sizes?

4.2 Planning

Both Algorithms were implemented by the authors in C++ [5], and compiled using the `-O3` optimization flag. The original Aho-Corasick algorithm was accordingly modified to produce the same output as the optimization (i.e., the number of occurrences of each keyword), with no impact on its runtime.

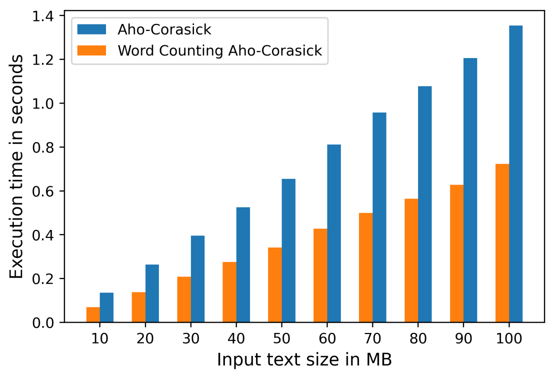
The experiments were run on a personal computer, with an Intel Core i7-8750H CPU, 2.2GHz clock speed, 384KB L1 cache, 1.5MB L2 cache and 9MB L3 cache, and a 16GB DDR4 memory at 2667MHz, running a 64-bit Windows 10 operating system.

To evaluate the proposed modification, we conducted experiments using two alphabets: English text (lowercase and uppercase letters, and digits) and DNA bases (*A*, *C*, *G* and *T*). The purpose is to test how well it performs on alphabets of different sizes (62 and 4, respectively).

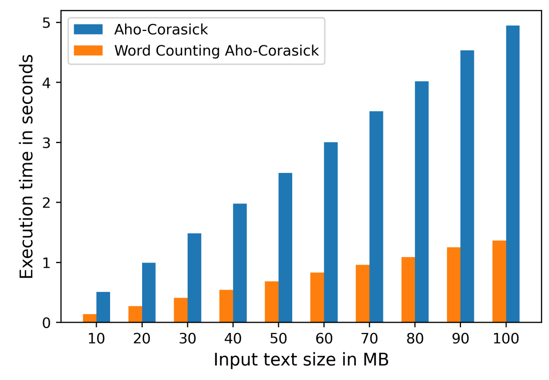
The English texts were generated randomly, and the DNA sequences were downloaded from the National Center for Biotechnology Information (NCBI) [10]. All patterns were generated randomly with varying lengths from 1 to 20 symbols. For each alphabet, we conducted three tests, with a dictionary size of 1KB, 1MB and 10MB. Table 5 shows the time taken to construct the structures for each alphabet and dictionary size. After constructing the pattern matching machines, we executed the search algorithms, increasing the text size from 10MB to 100MB.

4.3 Results and Discussion

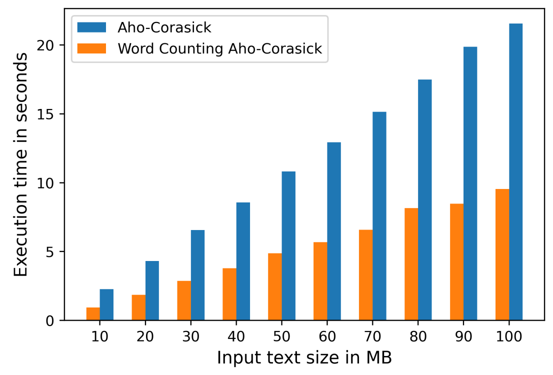
Figure 5 shows the results of the experiments.



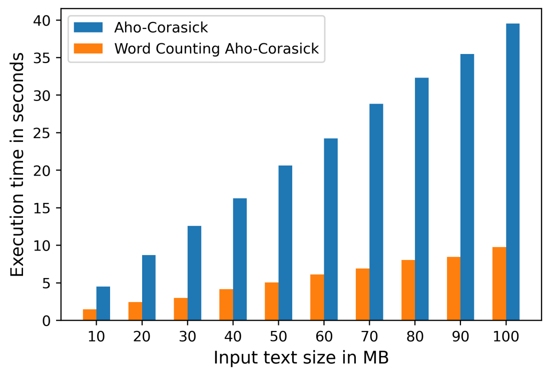
(a) English, 1KB pattern set



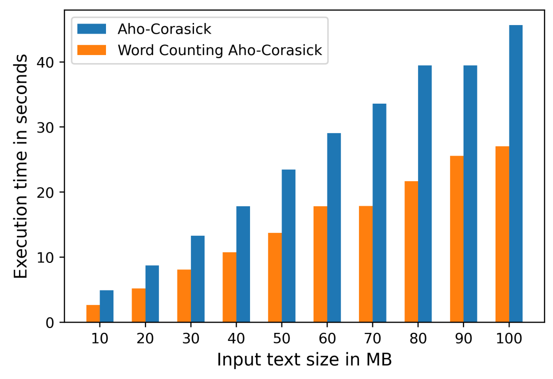
(b) DNA Sequences, 1KB pattern set



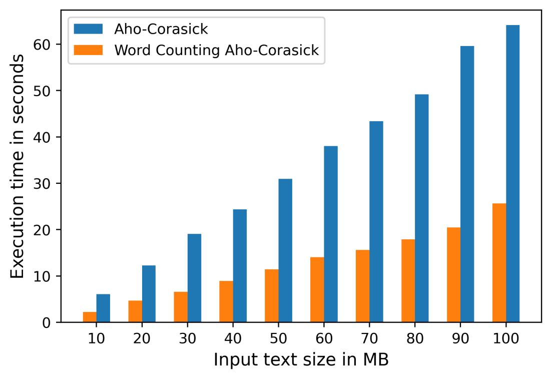
(c) English, 1MB pattern set



(d) DNA Sequences, 1MB pattern set



(e) English, 10MB pattern set



(f) DNA Sequences, 10MB pattern set

Figure 5: Conducted experiments for different pattern sets.

We conducted experiments with different text lengths, dictionary sizes and alphabets. With the results in mind, we can answer the questions made in Section 4.1:

RQ$_1$ As the text length increases, the disparity between the two algorithms increase, due to their time complexities: while the Aho-Corasick computes every keyword occurrence, the Word Counting Aho-Corasick only processes the first occurrence of each pattern.

RQ$_2$ The larger the dictionary gets, the more keyword occurrences a text can have. Our experiments have shown that the Word Counting Aho-Corasick had a better performance with different dictionary sizes.

Table 5: Time taken to build a pattern matching machine for each pattern set, for both implementations.

Alphabet	Dictionary		Build time (milliseconds)	
	Size	Words	Aho-Corasick	Optimization
English	1KB	100	< 1	< 1
English	1MB	93,974	346	359
English	10MB	910,937	4586	4676
DNA	1KB	87	< 1	< 1
DNA	1MB	79,731	96	112
DNA	10MB	751,033	525	590

RQ$_3$ A smaller alphabet means the keywords are closer to each other, causing potentially more occurrences per text character. The WORD COUNTING AHO-CORASICK performed better on a smaller alphabet size (the DNA bases). Nonetheless, the optimization still achieved better runtimes on a large alphabet (62 symbols).

The proposed optimization achieved better performance in every conducted experiment. In the majority of the experiments, WORD COUNTING AHO-CORASICK completed the search more than 100% faster than the original algorithm. Note that, in Figure 5d, the execution times were 300% faster in the optimization.

As for the build times, we can see in Table 5 that the construction of the WORD COUNTING AHO-CORASICK is slightly slower than the original algorithm since it creates additional arrays in memory. However, the arrays lengths are proportional to the total size of the keywords, so the space complexity of both algorithms is the same.

4.4 Threats to Validity

Both algorithms were implemented by the authors. Although this may pose a threat to validity, we followed the algorithms presented in the original Aho-Corasick paper [1], making the implementations straightforward. The algorithms are not restricted to a specific language. They are designed to function with all kinds of alphabets and symbols (even integer numbers instead of characters).

The experiments were made in a personal computer, subject to time variations caused by loss of CPU to other running processes. Also, the pattern sets and the English texts were generated randomly, which may not reflect real-world scenarios. However, since pattern matching is a problem that finds applications in many domains (human language, DNA bases, bytes and integer numbers), we wanted to evaluate the proposed algorithm in a general way, so it would not be associated to a language, such as English. Although the validity of the performed evaluation might be threatened by these factors, the theoretical discussion of Sections 2 and 3 is not affected by it.

5 CONCLUSION

In this paper, we have proposed an optimization for the Aho-Corasick search algorithm, called WORD COUNTING AHO-CORASICK, for cases where we are only interested in the frequencies of the patterns. We have shown that the presented algorithm has a time complexity of $O(m + n + u)$, while the original Aho-Corasick time complexity is $O(m+n+o)$, where o is the total count of keyword occurrences and u is the number of different keywords that appears at least once in the text. We have seen that $u \leq o$, so the optimization is expected to have better runtimes in most cases, using slightly more memory. We performed tests and confirmed that the proposed modification achieved better execution times in all inputs of our dataset.

The proposed modification can be helpful in many specific applications, where the Aho-Corasick is being used only to count the number of appearances, and the locations of the occurrences are not relevant to the problem. Such applications might be in the area of natural language processing (to count n-grams in a document), or DNA sequence analysis (to count *k-mers*, equivalent to the n-grams, in a genome sequence), and so on.

5.1 Related Work

The Aho-Corasick algorithm is widely studied as researchers try to optimize it even further. Dori and Landau [3] described a method to build the Aho-Corasick finite state machine in time proportional to sum of pattern lengths, regardless of the alphabet size, using suffix arrays. Nishimura et al. [9] described an optimization using Huffman Codes and rearranging states. Both studies can be combined with the WORD COUNTING AHO-CORASICK to improve even more its performance.

Even though there are many contributions involving the Aho-Corasick algorithm, to the best of our knowledge, there is no other work related to the optimization proposed in this paper.

5.2 Future Work

For future work, a better testing methodology can be used, with more realistic texts and patterns. Also, a more controlled environment for the experiments, so the execution time can be calculated more accurately, alongside with the memory consumption for the algorithm. The proposed modification can be combined with other known Aho-Corasick optimizations [3, 9] to reduce even more the algorithm execution time. Furthermore, we plan to develop and demonstrate a proof of time complexity for the proposed optimization.

ACKNOWLEDGMENTS

We thank Dr. Rohit Gheyi for the invaluable guidance. We thank our institution, the Federal University of Campina Grande, and its Computer Science undergraduate course, for the support. We thank our colleagues Arthur Vinícius and Gustavo Ribeiro for helping in this study.

REFERENCES

[1] Alfred Aho and Margaret Corasick. 1975. Efficient string matching: An aid to bibliographic search. *Commun. ACM* 18 (06 1975), 333–340.
[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (third ed.). MIT Press.
[3] Shiri Dori and Gad M Landau. 2006. Construction of Aho Corasick automaton in linear time for integer alphabets. *Inform. Process. Lett.* 98, 2 (2006), 66–72.
[4] Free Software Foundation. 2020. GNU Grep: Print lines matching a pattern. Retrieved November 14, 2020 from https://www.gnu.org/software/grep/manual/grep.html#Performance
[5] ISO. 2017. *ISO/IEC 14882:2017 Information technology — Programming languages — C++* (fifth ed.). International Organization for Standardization, Geneva, Switzerland. 1605 pages. https://www.iso.org/standard/68564.html

[6] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. 1977. Fast pattern matching in strings. *SIAM journal on computing* 6, 2 (1977), 323–350.

[7] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, USA.

[8] Christopher D. Manning and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA.

[9] Toshino Nishimura, Shuichi Fukamachi, and Takeshi Shinohara. 2001. Speed-up of Aho-Corasick pattern matching machines by rearranging states. In *Proceedings Eighth Symposium on String Processing and Information Retrieval*. IEEE, 175–185.

[10] U.S. National Library of Medicine. 1988. National Center for Biotechnology Information (NCBI). Retrieved November 14, 2020 from https://www.ncbi.nlm.nih.gov/

[11] Michael Sipser. 1996. *Introduction to the Theory of Computation* (first ed.). International Thomson Publishing.

[12] Steven S. Skiena. 2008. *The Algorithm Design Manual* (second ed.). Springer-Verlag, USA.

[13] Steven S. Skiena and Miguel Revilla. 2003. *Programming Challenges: The Programming Contest Training Manual*. Springer-Verlag, Berlin, Heidelberg.

[14] Benfano Soewito and Ning Weng. 2007. Methodology for Evaluating DNA Pattern Searching Algorithms on Multiprocessor. In *IEEE Seventh International Symposium on BioInformatics and BioEngineering*. 570–577.

[15] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. 2004. Deterministic memory-efficient string matching algorithms for intrusion detection. In *IEEE INFOCOM 2004*, Vol. 4. 2628–2639.