Unraveling Exact String Matching Algorithms: Survey, Experiment and Comparison

SC203 - 22APCS2 - Group 03:
Nguyen Thanh Phuoc Loc - 22125050
Nguyen Hoang Phuc - 22125076
Ngo Hoang Tuan - 22125115
Le Thanh Lam - 22125046
Truong Chi Nhan - 22125068

Abstract

The task of exact string matching, as well as pattern matching in general, presents a challenge in various fields such as text, image, signal, and speech processing. However, choosing the best algorithm for a particular application proves to be complicated due to various factors, including pattern and text size and structure, alphabet size, memory usage, and overall performance. In this paper, we provide a comprehensive survey of five exact string-matching algorithms: Rabin-Karp, Knuth-Morris-Pratt, Boyer-Moore, Aho-Corasick, and Suffix Tree. Each algorithm is presented systematically, clarifying its fundamental intuition, origins, primary use cases, and common optimization strategies. Moreover, we conduct experiments to determine their performance across different problem categories, including single-pattern and multiple-pattern scenarios. Our findings offer conclusive insights into the strengths and weaknesses of each algorithm, providing valuable guidelines for algorithm selection based on specific problem requirements. This paper aims to help readers understand exact string-matching algorithms and their practical applications by delivering a cohesive and insightful overview.

Indexterm: String matching, Rabin-Karp, Knuth-Morris-Pratt, Boyer-Moore, Aho-Corasick, Suffix Tree, exact string matching, pattern matching

1 Introduction

String matching, a fundamental problem in various domains such as text, image, signal, and speech processing, involves finding occurrences of a particular sequence of characters within another. The naive approach to string matching, comparing each character sequentially, is remarkably slow and inefficient, especially with large datasets. Given its omnipresence in applications ranging from information retrieval[10][4], plagiarism detection[8] to bioinformatics[3][7], improving the efficiency of string matching algorithms is crucial for enhancing overall system performance.

Motivated by the need to address the inefficiencies of the naive approach, this paper embarks on a comprehensive survey of five distinct string matching algorithms: Rabin-Karp, Knuth-Morris-Pratt, Boyer-Moore, Aho-Corasick, and Suffix Tree. The primary goal is to provide readers with a deep understanding of each algorithm's implementation, strengths, and weaknesses, particularly concerning their performance in relation to varying input sizes. By comparing these algorithms comprehensively, we aim to offer valuable insights into the conditions under which each excels, aiding practitioners and researchers in selecting the most suitable algorithm for specific applications.

To execute this survey, the paper is organized into three main sections. In Section 2, we provide an overview and delve into the details of each of the five algorithms. This includes elucidating

their origins, main use cases, and common optimization techniques. In Section 3, we conduct a benchmarking exercise, rigorously comparing the algorithms across diverse problem scenarios, including single-pattern, multiple-pattern, and approximate matching. Finally, in Section 4, we draw together our findings to elaborate on the strengths and weaknesses of each algorithm, offering a conclusive perspective on their comparative performance and practical applicability. Through this structured approach, our survey aims to contribute valuable insights for both practitioners and researchers in the field of exact string matching algorithms.

2 Five String-Matching Algorithms

2.1 Rabin-Karp Algorithm

2.1.1 Overview

Rabin-Karp (or Karp-Rabin) algorithm was created by Richard M. Karp and Michael O. Rabin in an issue named "Efficient randomized pattern-matching algorithms"[5], which is published in the IBM Journal of Research and Development in 1987. This algorithm performs really well in the case of multidimensional pattern matching and it is also often used for documents plagiarism.

2.1.2 Algorithm Description

The algorithm works by associating each string with a "fingerprint" using a hash function. With each substring of the text, rather than directly comparing strings, it first compares the hashes of the substring and the pattern. Only if the hashes are equal does it perform the more expensive string comparison.

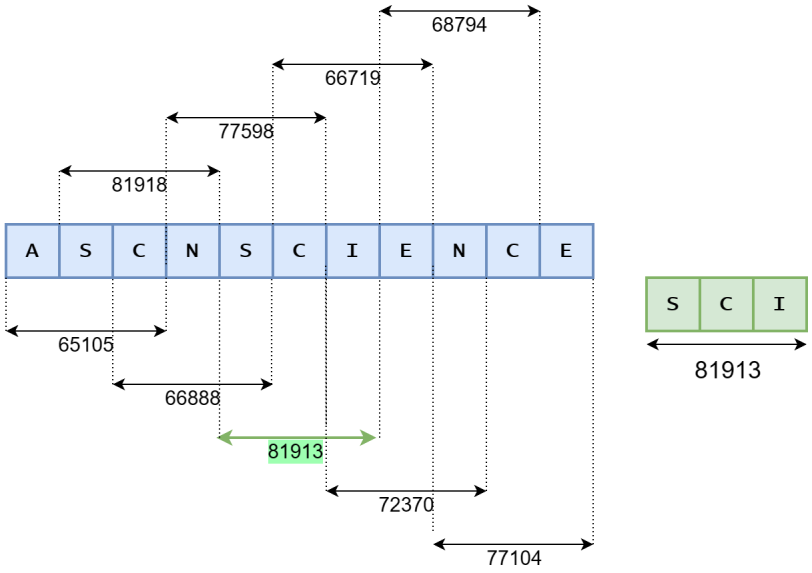
So now it comes to the choice of hash function, and Rabin-Karp algorithm uses the below:

$$H(S) = (\sum_{i=0}^{n-1} s[i] \times p^{n-1-i}) \mod m$$

where p, m are random prime numbers

The hash function provided is "good" enough to distinguish pattern and substrings, while allowing us to calculate the current hash in constant time if we already have the previous hash.

Suppose that the text string is "ASCNSCIENCE" and the pattern is "SCI", where p is 31 and m is 1001. Then we have the illustration for the algorithm as below:



At first, we calculate the hash of the pattern "SCI" (the value of each character is its decimal representation in ASCII):

$$H(\text{"SCI"}) = \text{'S'} \times 31^2 + \text{'C'} \times 31^1 + \text{'I'} \times 31^0 = 81913$$

And the hash of the first substring "ASC":

$$H(\text{"ASC"}) = \text{'A'} \times 31^2 + \text{'S'} \times 31^1 + \text{'C'} \times 31^0 = 65105$$

That is a mismatch, we continue to calculate the hash of "SCN". But we don't need to do everything from scratch as we can utilize the hash of "ASC":

$$H(\text{"SCN"}) = ((H(\text{"ASC"}) - \text{'A'} \times 31^2) \times 31 \mod 1001) + \text{'N'} = 81918$$

That is, removing previous window's first character ('A') and adding the newly emerged character ('N'). However, mismatch still occurs and we need to continue until finding out the matched hash or reaching the end. When the hash matches, we need to compare the pattern and substring character by character.

Then we have the pseudocode for this algorithm:

Algorithm 1.1 Rabin-Karp Algorithm

```
1:  procedure RabinKarp(pattern, text)
2:      base ← 26                                          ▷ Assuming alphabet is a-z
3:      patternLength ← length(pattern)
4:      textLength ← length(text)
5:      prime ← random_prime()                             ▷ Choose a random prime number
6:      patternHash ← 0
7:      textHash ← 0
8:      h ← base^(patternLength−1) mod prime
9:      for i ← 0 to patternLength − 1 do
10:         patternHash ← (base ∗ patternHash + pattern[i]) mod prime
11:         textHash ← (base ∗ textHash + text[i]) mod prime
12:     end for
13:     for i ← 0 to textLength − patternLength do
14:         if patternHash == textHash then
15:             if text[i . . . i + patternLength − 1] == pattern then
16:                 print i                                ▷ Pattern found at position i
17:             end if
18:         end if
19:         if i < textLength − patternLength then
20:             textHash ← (base ∗ (textHash − text[i] ∗ h) + text[i + patternLength]) mod prime
21:             if textHash < 0 then
22:                 textHash ← textHash + prime
23:             end if
24:         end if
25:     end for
26: end procedure
```

2.1.3 Complexity Analysis

Time Complexity The Rabin-Karp algorithm has a time complexity of $O(n + m)$ in the average case, and a worst-case time complexity of $O(nm)$, where n is the length of the text and m is the length of the pattern. The average case assumes that the hash function will distribute values uniformly over the hash table, and that hash collisions (different inputs producing the same hash) are rare. In this case, the algorithm only needs to perform a constant amount of work for each character in the text, resulting in a linear time complexity.

Space Complexity We can see that Rabin-Karp algorithm only uses constant $O(1)$ space, which is a big advantage when compared to other string-matching algorithm.

2.2 Knutt-Morris-Pratt Algorithm

2.2.1 Overview

Introduced by Donald Knuth, Vaughan Pratt, and James H. Morris in 1977[6], the Knuth-Morris-Pratt (KMP) algorithm strategically boosts efficiency by leveraging inherent information within the pattern. This algorithm preprocesses the pattern, constructing a partial match table, or "pi" function, based on prefix-suffix matches. The resultant table provides valuable insights into potential fallback positions in the event of a mismatch, thereby minimizing unnecessary character comparisons during the search phase.

The heart of the KMP algorithm lies in constructing the partial match table. This table, computed in linear time complexity relative to the pattern length, defines the longest proper prefix that is also a suffix for each prefix substring of the pattern. By utilizing this information, the algorithm efficiently skips unnecessary comparisons when a mismatch occurs.

2.2.2 Algorithm Description

The algorithm operates in two phases: the pre-processing phase, where the partial match table is constructed, followed by the search phase utilizing this table for efficient pattern matching.

Pre-processing Phase **Partial Match Table Generation:** Traverse the pattern and populate the partial match table, identifying the longest proper prefix that is also a suffix for each prefix substring. This step determines potential fallback positions during the search phase.

Algorithm 2.1 outlines the procedure for constructing the partial match table, a crucial step within the Knuth-Morris-Pratt algorithm. This algorithm efficiently generates the table, known as the Pi function, facilitating optimized pattern-matching operations.

Algorithm 2.1 Construct Partial Match Table (Pi Function)

```
1:  procedure ConstructPiFunction(pattern)
2:      m ← length of pattern
3:      pi[1..m] ← array of zeros
4:      k ← 0
5:      for q ← 2 to m do
6:          while k > 0 and pattern[k + 1] ≠ pattern[q] do
7:              k ← pi[k]
8:          end while
9:          if pattern[k + 1] = pattern[q] then
10:             k ← k + 1
11:         end if
12:         pi[q] ← k
13:     end for
14:     return pi
15: end procedure
```

Below, Table 1 presents the partial match table (Pi function) for the pattern 'ABABCAB'. Note that the first element of the table is always zero, as a proper prefix cannot be identical to the entire pattern.

Index	0	1	2	3	4	5	6
Pattern	A	B	A	B	C	A	B
Pi[i]	0	0	1	2	0	1	2

Table 1: Partial Match Table (Pi Function) for Pattern "ABABCAB"

Search Phase **Pattern Matching:** Slide through the text, aligning the pattern with the text for comparison. Upon a mismatch, consult the partial match table to determine the next possible position for pattern alignment, minimizing unnecessary character comparisons.

The subsequent pseudocode illustrates the Knuth-Morris-Pratt algorithm applied to search for a specific pattern within a given text. Leveraging the previously constructed partial match table (Pi function), this algorithm efficiently identifies occurrences of the pattern within the text.

Algorithm 2.2 KMP Algorithm for Pattern Searching

```
1:  procedure KMPSEARCH(text, pattern)
2:      n ← length of text
3:      m ← length of pattern
4:      pi ← CONSTRUCTPIFUNCTION(pattern)
5:      q ← 0
6:      for i ← 1 to n do
7:          while q > 0 and pattern[q + 1] ≠ text[i] do
8:              q ← pi[q]
9:          end while
10:         if pattern[q + 1] = text[i] then
11:             q ← q + 1
12:         end if
13:         if q = m then
14:             print "Pattern occurs at index" i − m
15:             q ← pi[q]
16:         end if
17:     end for
18: end procedure
```

2.2.3 Complexity Analysis

Time Complexity In the pre-processing phase, we see that the q pointer increases by at most one for each iteration of the outer loop. The inner loop, however, may cause q to decrease. Since q is initialized to zero and is only increased within the outer loop, it can only be decreased at most m times. Therefore, the inner loop is executed at most $2m$ times in total. As the outer loop iterates m times, the total number of iterations of the inner loop is $O(m)$. Since each iteration of the inner loop takes $O(1)$ time, the total time complexity of the pre-processing phase is $O(m)$.

Similar to the pre-processing phase, the outer loop of the search phase iterates n times, and the inner loop iterates at most $2m$ times. Therefore, the total time complexity of the search phase is $O(n + m)$.

Space Complexity The space complexity of the Knuth-Morris-Pratt algorithm is $O(m)$, as the partial match table (Pi function) is of size m. Note that the space complexity of the algorithm is independent of the length of the text, as the text is processed character by character.

2.3 Aho-Corasick Algorithm

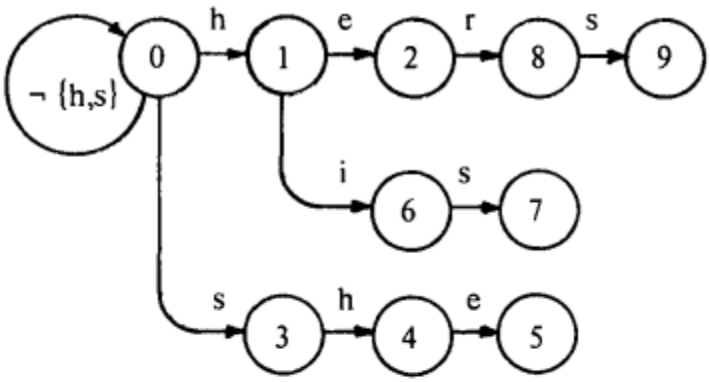
2.3.1 Overview

In 1975, Alfred Aho and Marget Corasick had proposed Aho-Corasick algorithm [1], which allows us to quickly search for a set of keywords in a text. We will denote the total length of its constituent keywords by m and the size of the alphabet by k. The algorithm constructs a finite state automaton based on a trie in $O(mk)$ time and then uses it to process the text. The time complexity when we process the text string is independent of the number of pattern strings.

2.3.2 Algorithm Description

The algorithm consists of two parts. First, it construct from the set of keywords a finite state pattern matching machine. After that, we apply the text string as input to the pattern matching machine. The machine signals whenever it has found a match for a keyword. The idea of using finite state machine in pattern matching applications is not new, but programmers are often avoid implementing them due to the complexity of programming the conventional algorithms for constructing finite automata from regular expressions. This paper shows that an efficient finite state pattern matching machine can be

constructed quickly and simply by the ideas in the Knuth-Morris-Pratt algorithm with those of finite state machines.



(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.

Figure 1: Fig 1. Pattern matching machine[1]

The construction of a pattern matching machine is is similar to trie. Formally, a trie is a rooted tree, where each edge of the tree is labeled with some letter and outgoing edges of a vertex have distinct labels. We will identify each vertex in the trie with the string formed by the labels on the path from the root to that vertex. Each vertex will also have a flag output which will be set if the vertex corresponds to a pattern in the dictionary. Accordingly, a trie for a set of strings is a trie such that each output vertex corresponds to one string from the set, and conversely, each string of the set corresponds to one output vertex.

The algorithms to construct the goto, failure and output functions from K are summarized below.

Algorithm 3.1 Construction of the goto function

Input. A set of keywords $K = y_1, y_2, \ldots, y_k$

Output. Goto function g and a partially computed output function $output$.

Note. We assume that $output(s)$ is empty when state s is first created, and $g(s, a) = fail$ if a is undefined or if $g(s, a)$ has not yet been defined. The procedure $enter(y)$ inserts into the $goto$ graph a path that spells out y.

```
1:  procedure ADDSTRING(K)
2:      newstate ← 0                                  ▷ 0 is designated as a start state
3:      for i ← 1 to k do
4:          enter(y_i)
5:      end for
6:      for all a such that g(0, a) = fail do
7:          g(0, a) ← 0
8:      end for
9:  end procedure
10: procedure ENTER(a_1 a_2 ... a_m)
11:     state ← 0; j ← 0
12:     while g(state, a_j) ≠ fail do
13:         state ← g(state, a_j)
14:         j ← j + 1
15:     end while
16:     for p ← j to m do
17:         newstate ← newstate + 1
18:         g(state, a_p) ← newstate
19:         state ← newstate
20:     end for
21:     output(state) ← a_1 a_2 ... a_m
22: end procedure
```

Algorithm 3.2 Construction of the failure function

Input. Goto function g and output function $output$ from Algorithm 1.

Output. Failure function f and $output$ function $output$.

```
1:  procedure CONSTRUCT
2:      queue ← empty
3:      for each a such that g(0, a) = s ≠ 0 do
4:          queue ← queue ∪ s
5:          f(s) ← 0
6:      end for
7:      while queue ≠ empty do
8:          let r be the next state in queue
9:          queue ← queue − r
10:         for each a such that g(r, a) = s ≠ fail do
11:             queue ← queue ∪ s
12:             queue ← f(r)
13:             while g(state, a) = fail do
14:                 state ← f(state)
15:             end while
16:             f(s) ← g(state, a)
17:             output(s) ← output(s) ∪ output(f(s))
18:         end for
19:     end while
20: end procedure
```

The trie vertices can be interpreted as states in a finite deterministic automaton, from any state we can transition - using the input letter - to other states, i.e., to another position in the set of strings. Thus we can understand the edges of the trie as transitions in an automaton according to the corresponding

letter. However, in an automaton we need to have transitions for each combination of a state and a letter. If we try to perform a transition using a letter, and there is no corresponding edge in the trie, then we nevertheless must go into some state. To solve this problem, the paper proposes the Failure function, which uses the ideas in the Knuth-Morris-Pratt algorithm. Suppose we are in a state corresponding to a string t, and we want to transition to a different state using the character c. If there is an edge labeled with this letter c, then we can simply go over this edge, and get the vertex corresponding to $t + c$. If there is no such edge, we find the longest string in the trie that's a proper suffix to the string t, and try to perform a transition (Goto) from there (and that longest string is output of the Failure function mentioned above).

Algorithm 3.3 Construction of a deterministic finite automaton

Input. Goto function g from Algorithm 1 and failure function f from Algorithm 2.
Output. Next move function δ.

```
1:  procedure NEXT(K)
2:      queue ← empty
3:      for each symbol a do
4:          δ(0, a) ← g(0, a)
5:          if g(0, a) ≠ 0 then
6:              queue ← queue ∪ g(0, a)
7:          end if
8:      end for
9:      while queue ≠ empty do
10:         let r be the next state in queue
11:         queue ← queue − r
12:         for each symbol a do
13:             if g(r, a) = s ≠ fail then
14:                 queue ← queue ∪ s
15:                 δ(r, a) ← s
16:             else
17:                 δ(r, a) ← δ(f(r), a)
18:             end if
19:         end for
20:     end while
21: end procedure
```

Algorithm 3.4 Pattern matching machine

Input. A text string $a = a_1a_2 \dots a_n$ where each a_i is an input symbol and a pattern matching machine M with goto function g, failure function f, and output function $output$, as described above
Output. Locations at which keywords occur in a.

```
1:  procedure AHOCORASICK(a, δ)
2:      state ← 0
3:      for i ← 1 to n do
4:          state ← δ(state, a_i)
5:          if output(state) ≠ empty then
6:              print i
7:              print output(state)
8:          end if
9:      end for
10: end procedure
```

2.3.3 Complexity Analysis

Time Complexity The overall running time for constructing the Goto, Output, Failure functions is $O(mk)$ with m denotes the total length of the set of keywords and k denotes the size of the alphabet by k. And running time for processing the text string is independent of the number of pattern strings, which is $O(n)$ with n denotes the length of the text string.

Space Complexity The space complexity of this algorithm is $O(mk)$ with m denotes the total length of the set of keywords and k denotes the size of the alphabet by k.

2.4 Boyer-Moore Algorithm

2.4.1 Overview

In 1977, Robert S. Boyer and J Strother Moore introduced the Boyer-Moore pattern matching algorithm[2], a transformative development in the realm of string searching. Confronting the limitations of sequential character examination, the authors implemented sophisticated preprocessing techniques, notably the "bad character" and "good suffix" rules, to strategically curtail the volume of necessary character comparisons. This deliberate optimization renders the Boyer-Moore algorithm an exemplar of computational efficiency.

The algorithm's impact extends beyond its conceptualization, permeating diverse domains such as text processing and bioinformatics. Its systematic approach to pattern matching, marked by a reduction in computational overhead, positions Boyer-Moore as an instrumental tool in research and application. The enduring relevance of this algorithm underscores the enduring contributions of Boyer and Moore to the field of computer science, underscoring its pivotal role in algorithmic design and practical implementation.

2.4.2 Algorithm Description

The Boyer-Moore algorithm searches for occurrences of P in T by performing character comparison. Instead of completely search for all alignments, Boyer-Moore preprocessed P to skip many alignments as possible.

The Brute-Force algorithm, a straightforward and intuitive approach, involves comparing the pattern with the text character by character, shifting one position at a time. While conceptually simple, this method can become computationally expensive, especially when dealing with large texts or patterns. Boyer-Moore, on the other hand, introduces a more strategic approach, instead of meticulously checking every character in the text against the entire pattern, it strategically compares the end of the pattern to the text. This approach allows the algorithm to make significant jumps along the text, avoiding unnecessary character-by-character checks.

The reason how this work lies in the fact that when aligning the pattern with the text, the last character of the pattern is initially compared to the corresponding character in the text. If there is a character does not match, there's no need to keep searching backward. Moreover, if the character in the text doesn't match any in the pattern, the algorithm efficiently jumps ahead by the length of the pattern. On the other hand, if there's a match, a partial shift of the pattern occurs to line up with the matching character, and the process repeats. This "jump and compare" strategy considerably reduces the overall number of comparisons, making the Boyer-Moore algorithm highly efficient in practice.

Writing in formal way, the Boyer-Moore algorithm initiates its search with the alignment set at $k = m$, where meaning the start of pattern P aligns with the start of text T. Characters in P and T are then compared starting at index m of P and index k in T, and moving backward. The strings are matched from the end of P to the start of P. The comparison continue until either the starting point of P is reached, indicating a match, or a mismatch occurs, prompting a shift to the right in alignment. The shift is determined by predefined rules. Subsequent comparisons are performed at the new alignment, and this cycle repeats until the alignment past the end of T, signaling the end of matching. This shifting and comparison process defines the core mechanics of the Boyer-Moore algorithm.

In order to understand this algorithm deeply, let take an example from 1977 original paper written by Boyer-Moore[2]. We will use ↑ to indicate the current character.

```
P =       AT-THAT
T =       WHICH-FINALLY-HALTS.--AT-THAT-POINT
                 |
```

Algorithm 4.1 Boyer-Moore Algorithm

```
 1: procedure BOYER-MOORE(T, P)
 2:     n ← length(T)
 3:     m ← length(P)
 4:     i ← m
 5:     j ← m
 6:     k ← m
 7:     while j > 0 and i ≤ n do
 8:         if P[j] = T[i] then
 9:             j ← j − 1
10:             i ← i − 1
11:         else
12:             k ← k + 1
13:             i ← k
14:             j ← m
15:         end if
16:     end while
17:     if j = 0 then
18:         return k
19:     else
20:         return −1
21:     end if
22: end procedure
```

Since character 'F' is not in P, we can shift the pattern to the right by 7 characters. This is because the character in P does not appears anywhere in that substring, then it is not worth to check the substring.

```
P =              AT-THAT
T =    WHICH-FINALLY-HALTS.--AT-THAT-POINT
                        |
```

Since there exists an character appears in P, we can move the pointer to the right to align two hyphens.

```
P =                  AT-THAT
T =    WHICH-FINALLY-HALTS.--AT-THAT-POINT
                           |
```

Now the character matches on P, we consider compare to one-by-one.

```
P =                  AT-THAT
T =    WHICH-FINALLY-HALTS.--AT-THAT-POINT
                          |
```

Since 'L' is not appears in P, we will use the same reason on top to move pointer to the right by 6. This is because the character in P does not appears anywhere in that substring, and since we already compared one character, we can move the pointer to the right by 6.

```
P =                        AT-THAT
T =    WHICH-FINALLY-HALTS.--AT-THAT-POINT
                                |
```

Again char matches the last character of P. Stepping to the left we see that the previous character in string also matches its opposite in P. Stepping to the left a second time produces:

```
P =                        AT-THAT
T =    WHICH-FINALLY-HALTS.--AT-THAT-POINT
                               |
```

Since we got a mismatch, we will have move the pointer to the right by 7 so it aligns the discovered substring "AT" with the beginning of PP.

```
P =                                    AT-THAT
T =      WHICH-FINALLY-HALTS.--AT-THAT-POINT
                                       |
```

This time we discover that each character of P matches the corresponding character in T so we have found the pattern. Note that we made only 14 references to string. Seven of these were required to confirm the final match. The other seven allowed us to move past the first 22 characters of string.

2.4.3 Complexity Analysis

Time Complexity The Boyer-Moore algorithm is a standout in the realm of string searching algorithms due to its remarkable performance in a variety of situations. It shines in terms of time complexity, with its best-case scenario of $O(n/m)$ happening when the pattern is located at the start of the text. Its average-case time complexity is often close to linear time, demonstrating its efficiency in common scenarios. However, its worst-case time complexity of $O(nm)$ can occur when the pattern and text are carefully designed to reduce the effectiveness of the heuristic.

Space Complexity In terms of space complexity, Boyer-Moore uses $O(256 + m)$ for the bad character heuristic and $O(m)$ for the good suffix heuristic, with the former being proportional to the size of the alphabet. The algorithm is particularly effective in practice, especially when dealing with large texts and patterns of moderate to long length, due to its ability to skip parts of the text based on heuristic principles.

When pitted against other string-searching algorithms, Boyer-Moore often surpasses others like the naive algorithm, Knuth-Morris-Pratt, and Rabin-Karp. Its prowess is especially noticeable when searching for longer patterns, and its ability to adapt to different patterns and text characteristics cements its position as a flexible and efficient tool for string searching.

2.5 Suffix Tree

2.5.1 Overview

A suffix tree is a fundamental data structure in computer science and string processing that provides an efficient solution to various string-related problems. The concept of suffix trees was first introduced by Kurt Manfred Weiner in 1973, who formulated the idea and presented it in his paper titled "Linear pattern matching algorithms"[9]. In this seminal work, Weiner addressed the need for an effective and versatile data structure to solve the substring search problem, a common challenge in text processing and pattern matching.

Weiner's groundbreaking paper laid the groundwork for the development of suffix trees as a powerful tool for string manipulation. The suffix tree is designed to represent all the suffixes of a given string in a way that allows for fast and convenient substring searches. The construction of the tree captures the inherent relationships and repetitions within the string, enabling efficient queries and searches.

The field of suffix trees further evolved with Esko Ukkonen's contribution in 1995 when he introduced an on-line algorithm for the construction of suffix trees. Ukkonen's algorithm significantly improved the practicality and efficiency of constructing suffix trees, making them applicable in real-time scenarios where the input string is processed incrementally.

2.5.2 Algorithm Description

Let's consider a string $T = t_1 t_2 \cdots t_n$ derived from an alphabet Σ. Any string x that can be expressed as $T = uxv$ where u and v are any strings (even empty ones), is referred to as a *substring* of T. Furthermore, any string $T_i = t_i t_{i+1} \cdots t_n$ where $1 \leq i \leq n+1$ is known as a *suffix* of T. Specifically, T_n is the *empty suffix* of T. The collection of all suffixes of T is symbolized by $\sigma(T)$. The *suffix trie* of T is a trie that represents $\sigma(T)$.

To be more precise, the suffix trie of T is represented as $STrie(T) = (Q \cup \perp, root, F, g, f)$ and is defined as an enhanced DFA (deterministic finite-state automaton) with a tree-like transition graph

that represents the trie for $\sigma(T)$. This is further enhanced with the so-called suffix function f and an auxiliary state $\perp$. The state set Q of $STrie(T)$ can be mapped in a one-to-one correspondence with the substrings of T. The state corresponding to a substring x is denoted by $\bar{x}$.

In the initial state, the root is associated with the empty string ϵ, and the set F of final states is associated with $\sigma(T)$. The transition function g is defined such that $g(\bar{x}, a) = \bar{y}$ for all $\bar{x}, \bar{y}$ in Q where $y = xa$ and a is an element of Σ.

The suffix function, denoted as f, is defined for each state x in Q. If x is not the root, then x can be expressed as ay for some a in Σ, and we set $f(x) = y$. Additionally, $f(root) = \perp$. The auxiliary state $\perp$ allows us to write algorithms without making explicit distinctions between the empty and nonempty suffixes, or between the root and other states. State $\perp$ is connected to the trie by $g(\perp, a) = root$ for every a in Σ. We leave $f(\perp)$ undefined. The suffix links, denoted as $f(r)$, are called the suffix link of state r and are used during the construction of a suffix tree. They also have many applications. The automaton $STrie(T)$ is identical to the Aho-Corasick string matching automaton for the keyword set $\{T_i | 1 \leq i \leq n + 1\}$.

The string w spelled out by the transition path in $STrie(T)$ between two explicit states s and r is represented in $STree(T)$ as generalized transition $g'(s, w) = r$. To save space the string w is actually represented as a pair (k, p) of pointers (the left pointer k and the right pointer p) to T such that $t_k \dots t_p = w$. In this way the generalized transition gets the form $g'(s, (k, p)) = r$.

Such pointers exist because there must be a suffix T_i such that the transition path for T_i in $STrie(T)$ goes through s and r. We could select the smallest such i, and let k and p point to the substring of this T_i that is spelled out by the transition path from s to r. A transition $g'(s, (k, p)) = r$ is called an a-transition if $t_k = a$. Each s can have at most one a-transition for each $a \in \Sigma$.

We again augment the structure with the suffix function f', now defined only for all branching states $\bar{x} \neq$ root as $f'(\bar{x}) = \bar{y}$ where y is a branching state such that $x = ay$ for some $a \in \Sigma$, and f'(root) $=\perp$. Such an f' is well defined: if x is a branching state, then $f'(x)$ is also a branching state. These suffix links are explicitly represented. It is sometimes helpful to speak about implicit suffix links, i.e., imaginary suffix links between the implicit states.

The suffix tree of T is denoted as $STree(T) = (Q' \cup \{\perp\}, root, g', f')$.

```
1:  procedure TEST-AND-SPLIT(s, (k, p), t)
2:      if k ≤ p then
3:          let g'(s, (k', p')) = s' be the t_k-transition from s;
4:          if t = t_{k'+p-k+1} then
5:              return (true, s);
6:          else
7:              replace the t_k-transition above by transitions
8:                  g'(s, (k', k' + p − k)) = r and g'(r, (k' + p − k + 1, p')) = s'
9:              where r is a new state;
10:             return (false, r);
11:         end if
12:     else
13:         if there is no t-transition from s then
14:             return (false, s);
15:         else
16:             return (true, s);
17:         end if
18:     end if
19: end procedure
```

```
1: procedure CANONIZE(s, (k, p))
2:     if p < k then return (s, k);
3:     else
4:         find the t_k-transition g'(s, (k', p')) = s' from s;
5:         while p' − k' ≤ p − k do
6:             k ← k + p' − k' + 1;
7:             s ← s';
8:             if k ≤ p then
9:                 find the t_k-transition g'(s, (k', p')) = s' from s;
10:            end if
11:        end while
12:    end if
13: end procedure
```

```
1: procedure UPDATE(s, (k, i))
2:     (s, (k, i − 1)) is the canonical reference pair for the active point;
3:     oldr ← root;
4:     (end-point, r) ← test-and-split(s, (k, i − 1), t_i);
5:     while not(end-point) do
6:         create new transition g'(r, (i, ∞)) = r' where r' is a new state;
7:         if oldr ≠ root then
8:             create new suffix link f'(oldr) = r;
9:         end if
10:        oldr ← r;
11:        (s, k) ← canonize(f'(s), (k, i − 1));
12:        (end-point, r) ← test-and-split(s, (k, i − 1), t_i);
13:    end while
14:    if oldr ≠ root then
15:        create new suffix link f'(oldr) = s;
16:    end if
17:    return (s, k);
18: end procedure
```

Algorithm 5.1 Construction of $STree(T)$ for string $T = t_1 t_2 \cdots \#$ in alphabet $\Sigma = \{t_{-1} t_{-2} \cdots t_{-m}\}$; # is the end marker not appearing elsewhere in T

```
1: create states root and ⊥;
2: for j ← 1, ..., m do
3:     create transition g(⊥, (−j, −j)) = root;
4: end for
5: create suffix link f'(root) = ⊥;
6: s ← root;
7: k ← 1;
8: i ← 0;
9: while t_{i+1} ≠ # do
10:    i ← i + 1;
11:    (s, k) ← update(s, (k, i));
12:    (s, k) ← canonize(s, (k, i));
13: end while
```

2.5.3 Complexity Analysis

You may take a look at the original paper[9] in order to deeply understand the time and space complexity of constructing suffix tree by using Ukkonen's algorithm.

Time Complexity The time complexity of Ukkonen's algorithm for constructing a suffix tree is $O(n)$. The key insight behind this linear time complexity lies in the amortized analysis of the algorithm. It can be explained briefly as follows:

- Each phase of Ukkonen's algorithm takes $O(1)$ time per character in the input string.
- During each phase, the algorithm processes one character of the input string for each active point.
- The total number of active points across all phases is $O(n)$.
- Therefore, the overall time complexity is $O(n)$.

Space Complexity The space complexity of the algorithm is $O(n)$ due to the storage requirements for the suffix tree nodes. The reasons behind this are:

- Each character in the input string results in the creation of a new leaf or an internal node in the suffix tree.
- The total number of nodes created during the construction of the suffix tree is $O(n)$.
- Each node requires a constant amount of space for storing its information.
- Thus, the space complexity is $O(n)$.

3 Benchmarking and Comparison

Experiment Setup

The input text was generated using the `testlib` library in C++, which allowed for the creation of a string of a specified length. The text was generated using a uniform distribution of characters, with the characters being randomly selected from the set of alphabets. The patterns were also generated using the same method, with the length of the patterns being varied to assess the impact of pattern length on the run time of the algorithms. The patterns were then searched for in the text using each of the algorithms, and the run times were recorded.

The experiments were conducted using a Dell G15 5511 with a 2.3 GHz Intel Core i7 processor and 16 GB of RAM. The programs were compiled using the GNU C++ compiler with the `-O3` flag to enable compiler optimization. The average run times were calculated using the `chrono` library in C++, which allowed for the measurement of time in microseconds.

3.1 Single Pattern Searching in a Text

For this experiment, the aim was to assess the efficiency of various string-matching algorithms - Rabin-Karp, Knuth-Morris-Pratt (KMP), Aho-Corasick, Boyer-Moore, and Suffix Trees - in the context of single pattern searching within a given text. The experiments were conducted using diverse input sizes, and the average run times were recorded for each algorithm.

3.1.1 Experiment Results

Table 2 outlines the average run times (in microseconds) for different input sizes.

Input Size	Rabin-Karp	KMP	Aho-Corasick	Boyer-Moore	Suffix Trees
100	2.19608	0.0196078	4.13725	0.352941	12.8235
10 000	12.5882	2	57.3529	2.11765	121.686
50 000	53.8824	11.2157	512.824	9.03922	542.725
100 000	98.3137	16.6275	1 058.29	16.1569	1 049.02
1 000 000	2 474.16	300.078	15 269.9	220.392	39 574.5
5 000 000	10 729.9	1 673.45	75 365.5	1 295.73	231 308

Table 2: Average run times (in microseconds) for different input sizes

3.1.2 Comparative Analysis

Performance with Increasing Input Sizes

As the input size increased, the run times for all algorithms showed an expected upward trend. Here are some observations:

- KMP: Maintains relatively low run times across all input sizes, showcasing a consistent performance and being particularly efficient for smaller inputs.
- Boyer-Moore: Demonstrates consistently efficient performance across different input sizes, remaining competitive and effective as the input scales up.
- Rabin-Karp: Shows moderate performance across smaller input sizes but starts to lag behind significantly as the input size grows, becoming less efficient compared to other algorithms for larger inputs.
- Aho-Corasick: While initially performing adequately, Aho-Corasick's run times escalate notably as the input size increases, becoming significantly less efficient compared to other algorithms.
- Suffix-Tree: Presents the highest run times across all input sizes, indicating scalability issues and reduced efficiency, particularly as the input size becomes larger.

Observations on Algorithm Efficiency

- Scalability: KMP and Boyer-Moore demonstrate better scalability with increasing input sizes compared to Rabin-Karp, Aho-Corasick, and Suffix-Tree.
- Consistency: Boyer-Moore maintains consistent and competitive performance, especially noticeable across various input sizes.
- Aho-Corasick and Suffix-Tree: Both algorithms exhibit significant scalability challenges, with run times increasing drastically with larger inputs, making them less suitable for scaling.

3.2 Multiple Pattern Searching in a Text

The objective was to evaluate the efficiency of various string-matching algorithms - Rabin-Karp, Knuth-Morris-Pratt (KMP), Aho-Corasick, Boyer-Moore, and Suffix Trees - specifically in the context of multiple pattern searching within a given text. Notes that the input string is given **before** the patterns. The experiments were conducted across different input sizes, and the average run times for each algorithm were recorded.

3.2.1 Experiment Results

Table 3 outlines the average run times (in microseconds) for different input sizes.

Text Size	Rabin-Karp	KMP	Aho-Corasick	Boyer-Moore	Suffix Trees
100	1.09091	2.90909	7	3.45455	0.363636
1 000	27.0909	17.1818	389.364	29.8182	2.90909
10 000	745.545	211.909	6 205.09	331.727	18
100 000	29 789.6	5 825.18	202 932	7 310.09	567.636
1 000 000	337 990	62 680.8	2 405 820	71 056.4	4 392.09

Table 3: Average run times (in microseconds) for different input sizes

3.2.2 Comparative Analysis

Performance with Increasing Input Sizes

As the input size increased, the run times for all algorithms showed an expected upward trend. Here are some observations:

- KMP: Exhibits increasing run times with growing input sizes, showcasing moderate efficiency for smaller inputs but experiencing considerable escalation in run times as the input size scales up.
- Boyer-Moore: Demonstrates relatively consistent and competitive performance across different input sizes, showcasing efficiency particularly for moderate input sizes but experiencing notable increases in run times for larger inputs.
- Rabin-Karp: Displays moderate efficiency for smaller inputs but shows a significant increase in run times as the input size grows, becoming less efficient compared to other algorithms for larger-scale problems.
- Aho-Corasick: Shows high run times across all input sizes, with a notable increase in inefficiency as the input size increases. It displays considerable challenges in handling larger inputs efficiently.
- Suffix-Tree: Initially presents exceptionally low run times for smaller inputs, demonstrating efficient performance. However, its run times increase notably as the input size scales up, albeit not as drastically as some other algorithms.

Observations on Algorithm Efficiency

- KMP and Boyer-Moore: Both algorithms showcase moderate scalability but struggle noticeably with efficiency as the input size increases exponentially. Boyer-Moore displays better efficiency for moderate input sizes compared to KMP.
- Rabin-Karp: Demonstrates moderate efficiency for smaller inputs but faces significant challenges in scalability, leading to high run times for larger inputs.
- Aho-Corasick and Suffix-Tree: Both algorithms exhibit substantial inefficiencies in handling larger inputs, with Aho-Corasick displaying significantly high run times across all input sizes, while Suffix-Tree manages relatively better for smaller inputs but struggles with scalability.

3.3 Multiple Pattern Searching in a Text with Multiple Patterns

The objective was to evaluate the efficiency of various string-matching algorithms - Rabin-Karp, Knuth-Morris-Pratt (KMP), Aho-Corasick, Boyer-Moore, and Suffix Trees - in the context of multiple pattern searching across multiple texts. Notes that the input string is given **after** the patterns. The experiments were conducted with varying input sizes and multiple patterns, recording the average run times for each algorithm.

3.3.1 Experiment Results

Table 4 outlines the average run times (in microseconds) for different input sizes.

Text Size	Rabin-Karp	KMP	Aho-Corasick	Boyer-Moore	Suffix Trees
One, 1 000	14 235.8	1 182	14 785	772.5	321.5
One, 100 000	Timeout	85 077.8	110 282	15 819.2	32 910
Multiple, 1 000	Timeout	Timeout	177 150	5 777.75	312 226
Multiple, 100 000	Timeout	Timeout	237 458	Timeout	Timeout

Table 4: Average run times (in microseconds) for different input sizes

3.3.2 Comparative Analysis

Performance with Increasing Input Sizes and Multiple Patterns

- KMP: Shows varied performance, managing reasonable run times for smaller inputs with a single pattern but encountering timeouts for larger inputs or multiple patterns. It struggles significantly with scalability in these scenarios.

- Boyer-Moore: Demonstrates competitive performance for smaller inputs and single patterns, showcasing relatively lower run times compared to other algorithms. However, it also encounters timeouts for larger inputs or multiple patterns, indicating limitations in scalability.
- Rabin-Karp: Initially performs reasonably well for smaller inputs and single patterns but faces timeouts as the input size increases or multiple patterns are introduced. This algorithm shows challenges in handling larger-scale scenarios efficiently.
- Aho-Corasick: Presents high run times for larger inputs or multiple patterns but does not encounter timeouts. While it showcases scalability challenges, it manages to process these scenarios without timeouts, indicating better handling of larger inputs compared to some other algorithms.
- Suffix-Tree: Demonstrates competitive run times for smaller inputs and single patterns, but experiences timeouts for larger inputs or multiple patterns, suggesting scalability limitations despite initial efficiency.

Observations on Algorithm Efficiency

- KMP, Boyer-Moore, Rabin-Karp, and Suffix-Tree: All these algorithms face scalability challenges when dealing with larger inputs or multiple patterns, encountering timeouts or significantly high run times.
- Aho-Corasick: While it showcases high run times for larger inputs or multiple patterns, it manages to process these scenarios without encountering timeouts, indicating comparatively better scalability in handling larger inputs compared to some other algorithms.

4 Conclusion

In summary, this survey presents a thorough examination of the major exact string-matching algorithms. Through a systematic analysis of Rabin-Karp, Knuth-Morris-Pratt, Boyer-Moore, Aho-Corasick, and suffix trees, it provides a detailed and unified understanding of how each algorithm works, its origins, and applications. To offer practical guidance, experiments evaluate the performance and scalability of the algorithms under different conditions. The results illuminate the strengths and weaknesses of each approach in relation to varying input sizes and problem types. This informs recommendations about selecting the most suitable algorithm according to the specific demands of a given use case. Looking ahead, further research offers the potential to develop hybrid solutions by combining optimized aspects of different algorithms. There is also the opportunity to tailor the techniques reviewed here to emerging domains involving large-scale data processing across distributed systems.

References

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, jun 1975.

[2] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, oct 1977.

[3] Lok-Lam Cheng, David W Cheung, and Siu-Ming Yiu. Approximate string matching in dna sequences. In *Eighth International Conference on Database Systems for Advanced Applications, 2003.(DASFAA 2003). Proceedings.*, pages 303–310. IEEE, 2003.

[4] James C French, Allison L Powell, and Eric Schulman. Applications of approximate word matching in information retrieval. In *Proceedings of the sixth international conference on Information and knowledge management*, pages 9–15, 1997.

[5] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

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

[7] Nadia Ben Nsira, Mourad Elloumi, and Thierry Lecroq. On-line string matching in highly similar dna sequences. *Mathematics in Computer Science*, 11:113–126, 2017.

[8] Kusum Lata Pandey, Suneeta Agarwal, Sanjay Misra, and Rajesh Prasad. Plagiarism detection in software using efficient string matching. In *Computational Science and Its Applications–ICCSA 2012: 12th International Conference, Salvador de Bahia, Brazil, June 18-21, 2012, Proceedings, Part IV 12*, pages 147–156. Springer, 2012.

[9] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, sep 1995.

[10] Justin Zobel and Philip Dart. Phonetic string matching: Lessons from information retrieval. In *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 166–172, 1996.