

# String Matching Application in Google Ngram Viewer: Analyzing Linguistic Trends Over Digitized Historical Texts

Shanice Feodora Tjahjono - 13523097

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jalan Ganesha 10 Bandung

E-mail: [xianfeodora@gmail.com](mailto:xianfeodora@gmail.com) , [13523097@std.stei.itb.ac.id](mailto:13523097@std.stei.itb.ac.id)

**Abstract**—This paper studies the application of string-matching concepts in Google Ngram Viewer (GNV) for analyzing linguistic trends across digitized historical texts. The study includes studies on two fundamental pattern matching algorithms—Knuth-Morris-Pratt (KMP) and Boyer-Moore—and their role in enabling efficient text analysis at scale. Through the development of a simplified GNV implementation, this research demonstrates how classical string-matching techniques can be applied to identify word frequency patterns over time within historical corpora. The implementation utilizes Project Gutenberg's public domain texts, processed through both KMP and Boyer-Moore algorithms to generate decade-based frequency analysis. While GNV's actual architecture employs sophisticated preprocessing and indexing techniques for handling millions of books, this study illustrates the fundamental string-matching principles that underpin such large-scale text analysis systems. The experimental results show how both algorithms successfully identify patterns within the corpus, with Boyer-Moore demonstrating superior average-case performance for longer patterns and KMP providing consistent linear-time guarantees. This work contributes to understanding how foundational computer science algorithms enable modern digital humanities tools, bridging theoretical algorithm design with practical applications in computational linguistics and historical text analysis.

**Keywords**—string matching, pattern matching, Google Ngram Viewer, KMP algorithm, Boyer-Moore algorithm, digital humanities, computational linguistics

## I. INTRODUCTION

The study of language evolution and cultural shifts has long relied on the analysis of historical texts. With the advent of large-scale digitization, tools such as Google Ngram Viewer (GNV) have enabled researchers to explore linguistic trends across vast corpora of books spanning centuries. GNV functions by identifying and charting the frequency of specific sequences of words—known as n-grams—over time, offering valuable insights into how language and society have evolved.

At the core of this functionality lies the fundamental concept of *string matching*, a computational process that involves locating occurrences of a substring within a larger

body of text. Efficient string-matching algorithms are essential for processing large datasets like those used by GNV. In this paper, we explore the application of string-matching techniques in GNV, focusing particularly on how they contribute to identifying word trends in digitized historical texts.

To deepen our understanding, this study presents a simplified implementation of GNV that models its core functionality. This implementation utilizes two well-known string-matching algorithms—Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM)—to search for specific word patterns in a time-stamped corpus. By comparing the performance and results of these algorithms, we aim to demonstrate how string matching not only underpins the GNV's analytical capabilities but also offers a practical approach to understanding linguistic trends over time.

## II. STRING MATCHING

### A. Pattern Matching and String Matching

Pattern matching is the process of searching for the presence of a pattern within a given structure, especially in strings. In the context of text, this is known as string matching, where the goal is to find all occurrences of a pattern string (P) within a longer text string (T). String matching algorithms are essential in analyzing large collections of text, particularly in applications such as text search and natural language analysis. Exact string matching seeks positions where P matches exactly with a substring of T, while approximate matching allows for limited mismatches. Several algorithms exist for solving the exact matching problem, including the brute-force method, the Knuth-Morris-Pratt (KMP) algorithm, and the Boyer-Moore algorithm [1].

### B. The Brute Force Algorithm

The brute force algorithm, also known as the naive string-matching algorithm, is the most straightforward method for finding all occurrences of a pattern P of length m in a text T of length n. It works by checking for a match at every possible

alignment of the pattern in the text, starting from the leftmost character and shifting one character at a time. The following is a pseudocode that describes the algorithm,



Figure 1. Pseudocode of Brute Force Algorithm

In the brute force string matching algorithm, the pattern of length  $m$  is compared to every possible substring of the text of length  $n$  that could potentially contain it. This results in  $n - m + 1$  possible starting positions in the text where the pattern can be aligned. At each of these positions, the algorithm performs a character-by-character comparison between the pattern and the corresponding substring in the text.

In the worst-case scenario, most characters in the pattern may match with the text at each position, but a mismatch occurs at the very last character of the pattern. This forces the algorithm to perform up to  $m$  comparisons at each of the  $n - m + 1$  positions, without ever finding a complete match. A classic example of this occurs when both the text and pattern contain repeated characters, except for the final character in the pattern, which differs.

As a result, the total number of character comparisons in the aforementioned scenario leads to a worst-case time complexity of  $O(mn)$ . Although it is sometimes simplified as  $O(n \cdot m)$ , the more precise bound reflects the actual behaviour of the algorithm in the most computationally expensive cases. On the other hand, in the best-case scenario, a match is found immediately at the first alignment, and all characters match without any need for further shifting or comparisons. In this case, only  $m$  comparisons are performed, followed by a return. Since this happens on the first iteration over the text, the best-case time complexity is  $O(n)$ , representing a single linear scan with minimal character comparisons [1].

```

NOBODY NOTICED HIM
1 NOT
2  NOT
3   NOT
4    NOT
5     NOT
6      NOT
7       NOT
8        NOT

```

Figure 2. Illustration of Brute Force Algorithm

Source: [https://www.researchgate.net/figure/Example-of-brute-force-algorithm-string-matching\\_fig1\\_369174858](https://www.researchgate.net/figure/Example-of-brute-force-algorithm-string-matching_fig1_369174858)

The image illustrates how the Brute Force String Matching algorithm works by comparing a pattern ("NOT") with all possible substrings of the same length in the text "NOBODY NOTICED HIM". The algorithm checks each position in the text sequentially, attempting to match the pattern one character at a time. As shown, it starts from position 1, then shifts one character to the right at each step (positions 2 to 8), until it finds a match at position 8 where "NOT" aligns exactly with the substring in the text.

The illustration further reinforces the working principle of the Brute Force algorithm, where the pattern is aligned at every possible position in the text and checked one character at a time until a match is found. This simple and systematic process reflects the algorithm's core nature: it examines all possible alignments without exception.

The Brute Force algorithm's primary advantage lies in its generality—it can be applied to any text and pattern without prior preprocessing. Additionally, it performs relatively well when the alphabet used is large, as mismatches tend to occur early in the comparisons. However, this algorithm also has its limitations; its performance is inefficient in the worst case, as it may require many unnecessary comparisons. Moreover, it does not exploit any information about the structure of the pattern or the text to optimize the matching process. As a result, despite its simplicity and broad applicability, the Brute Force algorithm is often outperformed by more advanced string-matching techniques in practical scenarios [1].

### C. The Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt (KMP) algorithm is an efficient string-matching algorithm that improves upon the brute force approach by avoiding unnecessary character comparisons. It was developed by Donald Knuth, Vaughan Pratt, and James H. Morris in 1977. The core idea behind KMP is to preprocess the pattern  $P$  to build a failure function (also called a border or prefix table), which is then used to determine how far the pattern can be shifted when a mismatch occurs, without rechecking characters that are known to match. In the brute force method, when a mismatch occurs, the pattern is shifted by just one position, and all previous comparisons are repeated. KMP avoids this by reusing the knowledge of matched characters from the failure function.

$j$	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a

$k$	0	1	2	3	4
$b(k)$	0	0	1	1	2

Figure 3. Border Function Example

Source:

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-\(2025\).pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-(2025).pdf)

The image presents an example of the border function table (or failure function) used in the Knuth-Morris-Pratt (KMP) string matching algorithm, specifically for the pattern  $P = \text{"abaaba"}$ . The first row ( $j$ ) represents the indices of the pattern from 0 to 5, and the second row ( $P[j]$ ) shows the corresponding characters of the pattern. The third row ( $k$ ) serves as a helper index used in the computation of the border values, while the fourth row ( $b(k)$ ) contains the computed border values for each prefix of the pattern.

This precomputed border function table plays a crucial role in the efficiency of the KMP algorithm. It allows the pattern to be shifted intelligently during mismatches, avoiding unnecessary re-evaluation of characters that have previously been matched. By doing so, KMP achieves linear time complexity in the worst case, making it significantly more efficient than the brute force approach in many practical scenarios.

The Knuth-Morris-Pratt (KMP) algorithm consists of two main phases. The first is the preprocessing phase, during which a border function table—is constructed based on the input pattern. This table records, for each position in the pattern, the length of the longest proper prefix that is also a suffix of the substring ending at that position. The second phase is the matching phase, where the pattern is compared to the text. When a mismatch occurs, the algorithm uses information from the border function table to determine how far the pattern can be shifted, thereby avoiding redundant comparisons and enabling a more efficient matching process. To better understand how the KMP algorithm operates, the following pseudocode outlines the steps involved in both the preprocessing and matching phases,

```

1 FUNCTION kmpMatch(text, pattern)
2   n ← length of text
3   m ← length of pattern
4   b[] ← computeBorder(pattern)
5
6   i ← 0    // index for text
7   j ← 0    // index for pattern
8
9   WHILE i < n DO
10    IF pattern[j] = text[i] THEN
11      IF j = m - 1 THEN
12        RETURN i - m + 1 // match found
13      ENDIF
14      i ← i + 1
15      j ← j + 1
16    ELSE IF j > 0 THEN
17      j ← b[j - 1]
18    ELSE
19      i ← i + 1
20    ENDIF
21  ENDWHILE
22
23  RETURN -1 // no match found
24 END FUNCTION

```

Figure 4. KMP Matching Pseudocode

```

1 FUNCTION computeBorder(pattern)
2   m ← length of pattern
3   b[0] ← 0
4
5   j ← 0
6   i ← 1
7
8   WHILE i < m DO
9     IF pattern[i] = pattern[j] THEN
10      j ← j + 1
11      b[i] ← j
12      i ← i + 1
13     ELSE IF j > 0 THEN
14      j ← b[j - 1]
15     ELSE
16      b[i] ← 0
17      i ← i + 1
18     ENDIF
19   ENDWHILE
20
21   RETURN b
22 END FUNCTION

```

Figure 5. KMP Border Function Table Pseudocode

The `kmpMatch` function attempts to find the starting index of a pattern inside a text. It begins by computing the border function (also called the failure function) using the `computeBorder` function. This function precomputes, for every position in the pattern, the length of the longest proper prefix which is also a suffix of the substring up to that point.

During the matching phase, the algorithm uses two pointers:  $i$  for the text and  $j$  for the pattern. If the characters match, both pointers advance. If a mismatch occurs:

- i) If  $j > 0$ , the pattern index  $j$  is updated using the border table to avoid rechecking known matched characters.
- ii) If  $j = 0$ , only  $i$  is incremented, as no useful prefix-suffix information exists.

If  $j$  reaches the end of the pattern ( $j = m$ ), a match is found, and the function returns the starting index of the match in the text. If the loop ends without a match,  $-1$  is returned.

The preprocessing step (`computeBorder`) runs in  $O(m)$  time, and the matching process (`kmpMatch`) runs in  $O(n)$  time, making the entire KMP algorithm run in  $O(n + m)$  — significantly faster than the brute force approach in the worst case [1].

#### D. The Boyer-Moore Algorithm

The Boyer-Moore algorithm is one of the most efficient string-matching algorithms, especially when the pattern is relatively short and the text is long. Developed by Robert S. Boyer and J Strother Moore in 1977, the key innovation of the Boyer-Moore algorithm is that it starts matching from the end of the pattern rather than the beginning. This allows the algorithm to skip sections of the text, potentially making fewer comparisons than Knuth-Morris-Pratt or brute force.

The Boyer-Moore algorithm utilizes two main techniques to improve string matching efficiency: the looking-glass technique and the character-jump technique. In the looking-glass technique, the pattern is compared against the text from right to left, starting at the last character of the pattern. This is different from most other string-matching algorithms, which executes comparisons from left to right.

When a mismatch is found, the character-jump technique is applied. This technique uses information about the last occurrence of the mismatched character in the pattern. Specifically, if a mismatch occurs at position  $T[i] \neq P[j]$ , the algorithm uses a precomputed last occurrence function, which records the rightmost position of each character in the pattern. The pattern is then shifted so that the mismatched character in the text aligns with its last occurrence in the pattern. If the character does not appear in the pattern at all, the pattern can be shifted beyond the mismatched character entirely. This approach allows the algorithm to potentially skip large sections of the text, making Boyer-Moore particularly efficient in practice.

Last occurrence function $f(x)$						
Pattern	a	b	a	c	a	b
	0	1	2	3	4	5
x	a	b	c	...	y	z
$f(x)$	4	5	3	...	-1	-1

Figure 6. Last Occurrence

Source:

<https://koding4fun.wordpress.com/2010/05/29/boyer-moore-algorithm/>

The image above illustrates the last occurrence function, denoted as  $f(x)$ , which is a key component of the Boyer-Moore string matching algorithm. This function maps each character  $x$  in the alphabet to the rightmost index at which it appears in the given pattern. If a character does not appear in the pattern at all, its value is set to -1.

In the example shown, the pattern consists of the characters "abacab" at indices 0 through 5. The corresponding values of the  $f(x)$  function indicate the last position at which each character occurs within the pattern. For instance, the character 'a' appears last at index 4, 'b' at index 5, and 'c' at

index 3. Characters absent in the pattern, such as 'y' and 'z', are assigned a value of -1.

This function is used during the matching phase of the Boyer-Moore algorithm to determine how far the pattern can be shifted when a mismatch occurs. By knowing the last occurrence of a mismatched character in the pattern, the algorithm can skip over sections of the text, thereby improving efficiency and reducing unnecessary comparisons. The following is a pseudocode to exemplify the Boyer-Moore algorithm,

```
1 FUNCTION bmMatch(text, pattern)
2   last[] ← buildLast(pattern)
3   n ← length of text
4   m ← length of pattern
5   i ← m - 1 // index for text (starting from end of pattern)
6
7   IF m > n THEN
8     RETURN -1 // pattern longer than text
9
10  j ← m - 1 // index for pattern (also from end)
11
12  DO
13    IF text[i] = pattern[j] THEN
14      IF j = 0 THEN
15        RETURN 1 // match found
16      ELSE
17        i ← i - 1
18        j ← j - 1
19    ELSE
20      lo ← last[text[i]] // last occurrence of text[i] in pattern
21      i ← i + MIN(j, 1 + j - lo) // character jump
22      j ← m - 1 // reset j
23  WHILE i ≤ n - 1
24
25  RETURN -1 // no match found
26 END FUNCTION
```

Figure 7. Boyer-Moore Matching Pseudocode

```
1 FUNCTION buildLast(pattern)
2   DECLARE last[128]
3   FOR i = 0 TO 127 DO
4     last[i] ← -1 // initialize to -1
5
6   FOR i = 0 TO length of pattern - 1 DO
7     last[pattern[i]] ← i // store last index of each char
8
9   RETURN last
10 END FUNCTION
```

Figure 8. Build Last Occurrence Table Pseudocode

The process starts with a preprocessing step, implemented in `buildLast(pattern)`. This function constructs a last table that maps each ASCII character to the last index at which it appears in the pattern. If a character does not occur in the pattern, it is mapped to -1. This table enables the character jump heuristic during the search.

The main matching function `bmMatch` begins comparison from the end of the pattern using a technique known as the looking-glass heuristic. If a mismatch occurs between `text[i]` and `pattern[j]`, the algorithm refers to the last table to determine how far the pattern can be shifted. Specifically, it computes a shift based on the distance between the mismatched character in the text and its last occurrence in the pattern. This reduces redundant comparisons and enables the algorithm to skip over sections of the text.

If a match is found (i.e., all characters in the pattern match in reverse order), the function returns the starting index `i` in the text. If no match is found after the loop ends, the function returns -1. Overall, Boyer-Moore is more efficient than brute force and even KMP in many cases, especially for long texts and large alphabets, due to its heuristic-based skipping mechanism.

### III. GOOGLE N-GRAM VIEWER

The Google N-gram Viewer (GNV) is a freely accessible online tool that allows users to visualize the frequency of n-grams—sequences of one to five words—across a vast corpus of digitized books over a historical timeline, ranging from the year 1500 to 2019. Introduced by Google in 2010, GNV provides a simple graphical interface where users can enter words or phrases, and the tool returns a time-series chart showing how often those terms appeared in published literature over the centuries [2], [3].

Unlike traditional search engines, GNV does not retrieve full documents or contexts. Instead, it analyses statistical trends based on the relative frequency of the queried n-gram, normalized by the total number of words published each year. This normalization helps ensure that frequency changes reflect linguistic or cultural shifts rather than fluctuations in publication volume [5]. The tool has been used in a variety of disciplines, including linguistics, history, sociology, and digital humanities, often to study the evolution of language, concepts, or cultural phenomena over time.

Although GNV was widely celebrated for democratizing access to historical language data, it also attracted criticism and doubt regarding the quality and representativeness of the underlying dataset. Researchers and scholars have pointed out issues such as OCR errors, inconsistent metadata, and the overrepresentation of certain genres or publication types [4]. Nonetheless, when interpreted cautiously and contextually, GNV remains a powerful and popular instrument for exploring large-scale linguistic patterns.

## IV. DISCUSSION

### A. Exact Lookup via Precomputed n-gram Index

When a user enters a word or phrase into the Google N-gram Viewer (GNV), the system performs an exact match against a precompiled database of n-grams extracted from digitized texts. These n-grams have been previously tokenized, time-stamped, and stored along with metadata such as their frequency and the number of books they appear in. Because the corpus has been fully processed beforehand, GNV does not need to scan entire texts during a query. Instead, it searches through an indexed data structure that allows for efficient string matching and rapid retrieval of matching records [6].

The nature of this exact-match retrieval closely corresponds to classic string-matching paradigms, where an input pattern is compared directly to an existing set of sequences for exact equivalence. Although GNV's backend is proprietary, it is reasonable to infer from available documentation that the system employs data structures such as hash maps, tries, or inverted indexes to support high-performance matching at scale [6]. Furthermore, the public documentation for GNV specifies that only exact matches are returned (no fuzzy or partial matching). Those matches are normalized by the total number of tokens published per year to ensure fair comparison across time periods [5].

### B. Efficient Matching Using Heuristics

Given the scale of the Google Books N-gram Corpus—spanning over 500 billion words across multiple languages—querying efficiency is essential. While exact-match searches in GNV are performed on pre-processed n-gram indexes, the responsiveness of the tool suggests the use of algorithmic optimizations typically found in classical string matching, such as heuristic-based shifting. One such heuristic is the last occurrence table from the Boyer-Moore algorithm, which allows the search process to skip sections of the pattern or text that cannot result in a match, based on the rightmost appearance of mismatched characters in the pattern. Though Google has not publicly disclosed that this specific heuristic is implemented, the sheer size of the dataset makes similar optimizations highly plausible [6].

In Google's own technical paper, the importance of memory-efficient and scalable storage for n-gram data is emphasized. The authors mention using compact binary encoding and indexing techniques to reduce memory footprint while allowing for quick look-up and access [6]. These strategies are conceptually aligned with the goals of classic pattern matching heuristics: minimizing unnecessary comparisons and maximizing throughput, especially when matching large patterns across vast precompiled corpora. Therefore, it is reasonable to conclude that the spirit of heuristics like the last occurrence rule influences the



architectural decisions underlying GNV's indexing and query engine.

### C. Handling Wildcards and POS Tags: Pattern Expansion

In addition to exact n-gram queries, Google N-gram Viewer (GNV) supports more flexible search functionalities, including wildcards (e.g., president of \*) and part-of-speech (POS) tags (e.g., run\_VERB). These features require the system to internally expand a query into multiple candidate n-grams, which are then matched against the index. For example, a wildcard query like the \* of prompts the system to return the most frequent completions of that pattern, such as "the end of," "the top of," or "the beginning of," each evaluated as a separate string match [5], [7]. Likewise, POS tagging enables users to restrict results based on grammatical category, thus introducing another layer of pattern specificity.

These types of flexible matching operations introduce computational challenges, as they require multi-pattern matching against potentially millions of n-gram candidates. Although Google has not disclosed the exact algorithms used, this functionality is conceptually comparable to multi-pattern matching techniques such as the Aho–Corasick algorithm, or iterative applications of single-pattern matching like KMP or Boyer–Moore. Such techniques are designed to handle many simultaneous string matches efficiently, a necessity in systems like GNV that support complex pattern expansion across massive, pre-processed datasets [6].

The system addresses this by limiting the scope of wildcard expansions to the top ten most frequent completions, thereby controlling both computational complexity and output interpretability [5]. This reflects a balance between expressive query design and performance feasibility, where core concepts of pattern matching algorithms inform how such features are implemented at scale.

## V. EXPERIMENT

### A. Program Design

To demonstrate how classical string-matching algorithms can be applied in large-scale linguistic trend analysis tools such as Google Ngram Viewer (GNV), a simplified version of GNV has been implemented. This program allows users to input a specific word (the "pattern") and analyse its frequency over time within a set of structured text data. The tool supports two pattern matching algorithms: Knuth–Morris–Pratt (KMP) and Boyer–Moore, both of which are efficient string-matching techniques with linear or near-linear performance.

The core functionality of the program mimics GNV's matching phase by reading text files containing year-stamped corpora, performing pattern matching on each file using either KMP or Boyer–Moore, and recording the number of occurrences per decade. This output is then used to generate a time-series view, illustrating how the usage of a word changes

over time. While this prototype lacks the full indexing and pre-processing power of GNV, it successfully demonstrates how exact string-matching algorithms can be applied to extract linguistic trends from historical texts. The implementation highlights how foundational string-matching concepts—such as prefix tables in KMP or last-occurrence heuristics in Boyer–Moore—can be leveraged to efficiently perform pattern analysis across time-stamped datasets, reinforcing their relevance in building scalable, text-driven analytical tools. The heart of the program lies in `pattern_matching.py`, where it contains the pattern-matching class with the necessary algorithms as shown in the image below,



```
1 from typing import List
2
3
4 class PatternMatcher:
5     """Implements KMP and Boyer-Moore algorithms for string matching"""
6
7     @staticmethod
8     def kmp_search(text: str, pattern: str) -> List[int]:
9         """
10         Knuth-Morris-Pratt algorithm for pattern matching.
11
12         Args:
13             text: The text to search in
14             pattern: The pattern to search for
15
16         Returns:
17             List of starting positions where pattern is found
18         """
19         if not pattern:
20             return []
21
22         # Build failure function (partial match table)
23         def build_failure_function(pattern):
24             failure = [0] * len(pattern)
25             j = 0
26             for i in range(1, len(pattern)):
27                 while j > 0 and pattern[i] != pattern[j]:
28                     j = failure[j - 1]
29                 if pattern[i] == pattern[j]:
30                     j += 1
31             failure[i] = j
32             return failure
33
34         failure = build_failure_function(pattern)
35         matches = []
36         j = 0
37
38         for i in range(len(text)):
39             while j > 0 and text[i] != pattern[j]:
40                 j = failure[j - 1]
41             if text[i] == pattern[j]:
42                 j += 1
43             if j == len(pattern):
44                 matches.append(i - j + 1)
45                 j = failure[j - 1]
46
47         return matches
```

Figure 9. PatternMatcher Class (KMP Search)

```

1  @staticmethod
2  def boyer_moore_search(text: str, pattern: str) -> List[int]:
3      """
4      Boyer-Moore algorithm for pattern matching.
5
6      Args:
7          text: The text to search in
8          pattern: The pattern to search for
9
10     Returns:
11         List of starting positions where pattern is found
12     """
13     if not pattern:
14         return []
15
16     # Build bad character table
17     def build_bad_char_table(pattern):
18         table = {}
19         for i in range(len(pattern)):
20             table[pattern[i]] = i
21         return table
22
23     bad_char = build_bad_char_table(pattern)
24     matches = []
25     shift = 0
26
27     while shift <= len(text) - len(pattern):
28         j = len(pattern) - 1
29
30         while j >= 0 and pattern[j] == text[shift + j]:
31             j -= 1
32
33         if j < 0:
34             matches.append(shift)
35             shift += len(pattern) if shift + len(pattern) < len(text) else 1
36         else:
37             char = text[shift + j]
38             shift += max(1, j - bad_char.get(char, -1))
39
40     return matches

```

Figure 10. PatternMatcher Class (Boyer-Moore Search)

The PatternMatcher class is a utility class that implements two fundamental string-matching algorithms used in computer science for efficiently locating occurrences of a pattern string within a larger text string. Both algorithms are designed to overcome the inefficiencies of naive string-matching approaches by employing sophisticated preprocessing techniques and intelligent skip mechanisms to avoid redundant character comparisons.

The Knuth-Morris-Pratt (KMP) algorithm represents a significant advancement in pattern matching by utilizing a preprocessing phase that constructs a failure function, also known as a partial match table. This failure function analyses the pattern itself to identify the longest proper prefix that is simultaneously a suffix at each position within the pattern. During the actual search phase, when a mismatch occurs between the text and pattern, the algorithm leverages this precomputed information to determine exactly how many characters can be safely skipped without missing any potential matches. The implementation traverses the text from left to right while maintaining a position pointer in the pattern, and upon detecting a mismatch, it consults the failure function to reposition the pattern pointer optimally rather than restarting the comparison from the beginning.

The Boyer-Moore algorithm takes a fundamentally different approach by implementing a right-to-left matching strategy combined with character-based heuristics for pattern shifting. The algorithm begins by constructing a bad character table (otherwise known as the aforementioned last occurrence table) during preprocessing, which records the rightmost occurrence of each character within the pattern. During the

search phase, the algorithm aligns the pattern with the text and begins comparing characters from the rightmost position of the pattern moving leftward. When a mismatch is encountered, the bad character rule is applied to calculate an appropriate shift distance based on the mismatched character's position in the bad character table. This approach allows the algorithm to make substantially larger jumps through the text, particularly when the alphabet is large relative to the pattern length.

Both algorithms return identical results, but they achieve this outcome through distinctly different computational strategies. The choice between these algorithms often depends on specific use case requirements, with KMP providing predictable linear performance suitable for real-time applications, while Boyer-Moore offers superior average-case performance that makes it ideal for text processing applications where speed is paramount, and input characteristics are favourable to its heuristic approach.

Other complementary classes and modules that work in conjunction with the Pattern Matcher class, including components for dataset, file downloading, data visualization, n-gram analysis, and additional text processing capabilities, can be found in the repository linked below. These supporting modules provide a comprehensive framework for text analysis and pattern recognition tasks for the implementation of this simplified version of GNV.

## B. Testing

Upon execution, the program automatically downloads and processes books from Project Gutenberg, based on the metadata provided in the accompanying JSON file. Once the dataset setup is complete, the user is prompted to enter a search pattern (word), followed by a choice between the Knuth-Morris-Pratt (KMP) or Boyer-Moore string matching algorithm. The program then performs a search over the corpus using the selected algorithm and displays the n-gram results in the form of a decade-based bar graph. Additionally, it reports the execution time required to complete the search, offering both a visualization of linguistic trends and insight into the algorithm's performance.

```

C:\Users\xtani\Downloads\Simple-Text-Pattern-Analyzer>python main.py
Simple Ngram Viewer
=====
Inspired by Google Ngram Viewer
Using KMP and Boyer-Moore algorithms for pattern matching

Setting up data...
Loaded 24 books from data/books_database.json
Found 24 books in database
Downloading book 1342...
Book 1342 downloaded successfully!
✓ Processed: Pride and Prejudice by Jane Austen (1813)
Downloading book 1342...
Book 1342 downloaded successfully!
Downloading book 1342...
Book 1342 downloaded successfully!
✓ Processed: Pride and Prejudice by Jane Austen (1813)
Downloading book 11...
Book 11 downloaded successfully!
✓ Processed: Alice's Adventures in Wonderland by Lewis Carroll (1865)
Downloading book 74...
Book 74 downloaded successfully!
✓ Processed: The Adventures of Tom Sawyer by Mark Twain (1876)
Downloading book 1661...
Book 1661 downloaded successfully!
✓ Processed: The Adventures of Sherlock Holmes by Arthur Conan Doyle (1892)
Downloading book 84...
Book 84 downloaded successfully!

```

Figure 11. Downloading the Dataset (not shown in full extent)

```

Data setup complete!
Total books loaded: 24
Decades covered: [1530, 1720, 1810, 1840, 1850, 1860, 1870, 1880, 1890, 1910, 1920]

Books by decade:
1530s: 1 books
1720s: 1 books
1810s: 2 books
1840s: 2 books
1850s: 4 books
1860s: 3 books
1870s: 1 books
1880s: 3 books
1890s: 5 books
1910s: 1 books
1920s: 1 books

```

Figure 12. Completed Data Setup

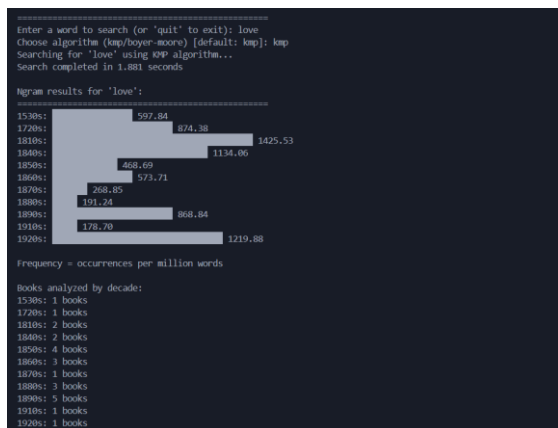


Figure 13. Search by KMP Algorithm

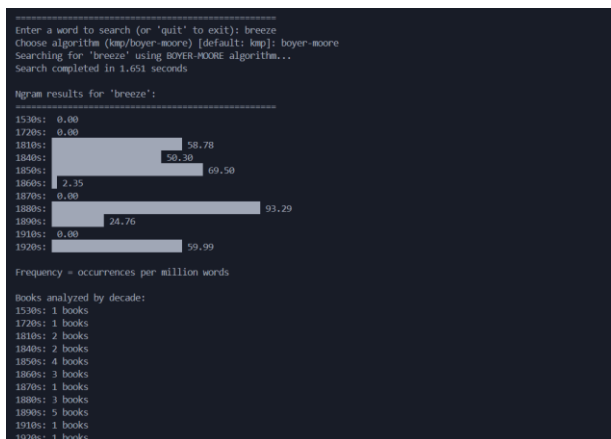


Figure 14. Search by Boyer-Moore Algorithm

The testing phase successfully demonstrates how classical string-matching algorithms can be effectively integrated into a simplified Google N-gram Viewer-like application. By supporting both KMP and Boyer-Moore, the program not only enables users to analyze word trends across historical texts—it also provides an opportunity to observe the behavior and efficiency of different pattern matching strategies in practice.

### C. Additional Notes

While this Simple N-gram Viewer provides a functional demonstration of pattern matching algorithms applied to textual analysis, it operates with significant limitations compared to the original Google N-gram Viewer. The program's scope is constrained by its reliance on Project

Gutenberg's public domain literary corpus, which represents only a small fraction of published works and is heavily skewed toward older texts that have entered the public domain, potentially creating bias in linguistic trend analysis.

Unlike Google's N-gram Viewer, which processes millions of books spanning multiple centuries with comprehensive metadata and multiple language support, this implementation operates on a much smaller dataset that may not accurately reflect broader linguistic patterns or cultural shifts. The program's frequency calculations are simplified and do not account for the complex statistical normalization techniques employed by professional corpus linguistics tools, such as smoothing algorithms, confidence intervals, or sophisticated weighting schemes that adjust for varying publication volumes across different time periods. Additionally, the terminal-based visualization lacks interactive features, comparative analysis capabilities, and advanced filtering options that make professional n-gram viewers powerful research tools.

The text preprocessing, while functional, employs basic cleaning techniques that may not adequately handle the diverse formatting inconsistencies, OCR errors, and encoding issues present in digitized historical texts, potentially affecting the accuracy of pattern matching results. Furthermore, the program's architecture does not support advanced linguistic features such as part-of-speech tagging, lemmatization, case-insensitive matching with proper statistical weighting, or multi-word phrase analysis that are essential for serious computational linguistics research.

## VI. CONCLUSION

This paper has explored the fundamental role of string-matching algorithms in enabling large-scale linguistic analysis tools such as Google N-gram Viewer. Through the examination of the Knuth-Morris-Pratt and Boyer-Moore algorithms, this study has demonstrated how classical pattern matching techniques serve as the computational foundation for modern digital humanities applications. The development and testing of a simplified GNV implementation successfully illustrated how these algorithms can be effectively applied to analyse word frequency trends across historical text corpora, providing both practical insights into algorithm performance and a deeper understanding of the underlying computational processes that power sophisticated text analysis tools.

The experimental results confirm that both KMP and Boyer-Moore algorithms are capable of efficiently processing textual data for pattern analysis, with each offering distinct advantages depending on the specific characteristics of the search task. KMP provides reliable linear-time performance with consistent behaviour across different input patterns, making it suitable for applications requiring predictable computational bounds. Boyer-Moore, with its heuristic-based approach, demonstrates superior performance in average-case scenarios, particularly when dealing with longer patterns and



diverse character distributions, making it ideal for large-scale text processing applications.

However, this study also reveals the significant gap between simplified academic implementations and production-scale systems like Google's N-gram Viewer. The limitations of the prototype—including its restricted corpus size, basic preprocessing techniques, and simplified frequency calculations—highlight the complexity and sophistication required to build truly comprehensive linguistic analysis tools. Professional systems must address challenges such as OCR error handling, sophisticated statistical normalization, multi-language support, and advanced linguistic features like part-of-speech tagging and lemmatization.

Despite these limitations, this research successfully demonstrates that fundamental string-matching algorithms remain central to modern text analysis applications. The principles underlying KMP's failure function and Boyer-Moore's character-jump heuristics continue to influence the design of contemporary information retrieval and text processing systems. As digital humanities and computational linguistics continue to evolve, understanding these foundational algorithms becomes increasingly important for developing efficient, scalable solutions for analysing the ever-growing volume of digitized textual data.

Future work could explore the integration of more advanced pattern matching techniques, such as multi-pattern algorithms like Aho-Corasick, or investigate how modern string-matching optimizations could be applied to improve the performance of text analysis tools in specialized domains. Additionally, research into hybrid approaches that combine classical string matching with modern machine learning techniques could yield new insights into how computational linguistics tools might evolve to meet the growing demands of digital scholarship and cultural analysis.

#### REPOSITORY LINK AT GITHUB

<https://github.com/feodorashanice/Simple-Text-Pattern-Analyzer.git>

#### ACKNOWLEDGMENT

The author expresses their deepest gratitude to all lecturers of IF2211 Algorithm Strategies, especially Dr. Ir. Rinaldi, M.T. as the lecturer of class K-02, for his constant guidance

and expertise throughout the semester. The author also extends appreciation to the Bandung Institute of Technology for its resources and facilities, and to friends and family for their unwavering support during the writing of this paper.


#### REFERENCES

- [1] R. Munir, *Pencocokan String*. [Online]. Available: [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-\(2025\).pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-(2025).pdf).
- [2] T. Kincaid, "Google's Ngram Viewer Charts the Cultural Rise of the Beatles, the Fall of the Word 'Golly'," *HuffPost*, Dec. 17, 2010. [Online]. Available: [https://www.huffpost.com/entry/google-ngram-database-tra\\_n\\_798150](https://www.huffpost.com/entry/google-ngram-database-tra_n_798150)
- [3] S. Shankland, "Google's new tool: the phrase hound of history," *CNET News*, Dec. 16, 2010. [Online]. Available: [https://web.archive.org/web/20140123012004/http://news.cnet.com/8301-1023\\_3-20025979-93.html](https://web.archive.org/web/20140123012004/http://news.cnet.com/8301-1023_3-20025979-93.html)
- [4] Meta Stack Exchange, "How reliable is Google Ngram?" [Online]. Available: <https://english.meta.stackexchange.com/questions/8015/how-reliable-is-google-ngram>
- [5] Google Books Ngram Viewer, "About Ngram Viewer." [Online]. Available: <https://books.google.com/ngrams/info>
- [6] P. Norvig, J. Orwant, W. Brockman, and S. P. Ghemawat, "Building a better n-gram viewer," Google Research Blog, 2012. [Online]. Available: <https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42490.pdf>
- [7] George Mason University Libraries, "Text Analysis Tools: Ngram Viewers," [Online]. Available: <https://infoguides.gmu.edu/textanalysis/tools/ngram>

#### PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 1 Juni 2025

  
Shanice Feodora Tjahjono - 13523097