Author
Kilian Wolfinger

Submission
Institute for System Software

Thesis Supervisor
Dipl.-Ing. Dr. Markus Weninger, BSc.

Oktober 2024

SMAVIZ: INTERACTIVE AND PLAYFUL VISUALIZATIONS OF STRING-MATCHING ALGORITHMS

Bachelor's Thesis
to confer the academic degree of

Bachelor of Science
in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITÄT LINZ

Bachelor's Thesis
SMAVIZ: Interactive and Playful Visualizations of String-Matching Algorithms

Dipl.-Ing. Dr. Markus Weninger, BSc
Institute for System Software
T +43-732-2468-4361
markus.weninger@jku.at

Student: Kilian Wolfinger
Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc
Start date: March 2024

In the course "Algorithmen und Datenstrukturen 1" (Algorithms and Data Structures 1) at the Johannes Kepler University, students are introduced to various string search algorithms. Currently, the teaching materials for these algorithms are limited to slides and blackboard explanations. To enhance the learning experience for students and provide lecturers with more effective teaching aids, interactive and playful visualizations of string-matching algorithms can be developed.

The goal of this bachelor's thesis is to create an interactive visualization tool that helps students learn and lecturers teach string search algorithms more effectively. The tool should include the following features:

1. Interactive visualizations of various string search algorithms:
   - Implement visualizations for multiple single-string search algorithms, such as the Brute-Force algorithm, the Knuth-Morris-Pratt algorithm, and the Rabin-Karp-Moore algorithm.
   - Include at least one multi-string search algorithm visualization, such as the Aho-Corasick algorithm or the Rabin-Karp algorithm.
   - Allow users to input their own strings and patterns to observe how the algorithms work in different scenarios.
2. Playing mode for student engagement:
   - Develop a playing mode that expects students to perform certain steps of the algorithms. For example, students might have to fill out the failure function table of the Knuth-Morris-Pratt algorithm.
   - Provide immediate feedback on the correctness of student answers and offer explanations for incorrect responses.
3. User-friendly interface and visualizations:
   - Design an intuitive and visually appealing user interface that allows easy navigation between different algorithms and modes.
   - Create clear and informative visualizations that effectively illustrate the workings of each algorithm, including step-by-step animations and highlighting of matched patterns.
   - Provide options to control the speed of the visualizations and to pause, resume, or step through the algorithm execution.

The interactive visualization tool should be developed using web technologies such as HTML, CSS, and JavaScript, making it accessible through a web browser. The implementation should be well-structured, maintainable, and extensible to allow for future additions or modifications.

Modalities:
The progress of the project should be discussed at least every four weeks with the advisor. A time schedule and a milestone plan must be set up within the first 3 weeks and discussed with the advisor. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 31.10.2024.

Abstract

String matching algorithms are fundamental in computer science, with applications ranging from text editors to bioinformatics. However, the complexity of these algorithms often poses challenges for students and practitioners attempting to understand and implement them. This thesis addresses this problem by introducing SMAVIZ, an interactive and playful web-based visualization tool for string matching algorithms.

Our approach focuses on four key algorithms: Naive Search, Knuth-Morris-Pratt (KMP), Rabin-Karp, and Aho-Corasick. SMAVIZ provides step-by-step visualizations, interactive elements, and a context-sensitive help system featuring "Frogbert Algorithmicus," a mascot designed to guide users through the learning process. By leveraging modern web technologies such as Vue.js and D3.js, we create an engaging and responsive user interface that allows learners to manipulate inputs, control visualization speed, and observe algorithm behavior in real-time.

The key impact of this work is twofold. First, it enhances the learning experience for computer science students by providing a hands-on, visual approach to understanding complex algorithms. Second, it serves as a valuable reference tool for practitioners, allowing them to quickly refresh their knowledge or explore the nuances of different string matching techniques. By combining theoretical foundations with interactive visualizations, SMAVIZ aims to bridge the gap between abstract algorithmic concepts and practical understanding, ultimately contributing to more effective education and application of string matching algorithms in computer science.

Kurzfassung

String-Matching-Algorithmen sind grundlegend in der Informatik, mit Anwendungen von Texteditoren bis zur Bioinformatik. Die Komplexität dieser Algorithmen stellt jedoch oft Herausforderungen für Studierende und Praktiker dar, die versuchen, sie zu verstehen und zu implementieren. Diese Arbeit adressiert dieses Problem durch die Einführung von SMAVIZ, ein interaktives und spielerisches webbasiertes Visualisierungstool für String-Matching-Algorithmen.

Der Ansatz konzentriert sich auf vier Schlüsselalgorithmen: Naive Suche, Knuth-Morris-Pratt (KMP), Rabin-Karp und Aho-Corasick. SMAVIZ bietet schrittweise Visualisierungen, interaktive Elemente und ein kontextsensitives Hilfesystem mit "Frogbert Algorithmicus", einem Maskottchen, das Benutzer durch den Lernprozess führt. Durch den Einsatz moderner Webtechnologien wie Vue.js und D3.js wird eine ansprechende und reaktionsschnelle Benutzeroberfläche geschaffen, die es Lernenden ermöglicht, Eingaben zu manipulieren, die Visualisierungsgeschwindigkeit zu steuern und das Verhalten der Algorithmen in Echtzeit zu beobachten.

Die Hauptauswirkung dieser Arbeit ist zweifach: Erstens verbessert sie die Lernerfahrung für Informatik-Studierende durch einen praktischen, visuellen Ansatz zum Verständnis komplexer Algorithmen. Zweitens dient sie als wertvolles Referenztool für Praktiker, das ihnen ermöglicht, ihr Wissen schnell aufzufrischen oder die Feinheiten verschiedener String-Matching-Techniken zu erforschen. Durch die Kombination theoretischer Grundlagen mit interaktiven Visualisierungen zielt SMAVIZ darauf ab, die Lücke zwischen abstrakten algorithmischen Konzepten und praktischem Verständnis zu schließen und letztendlich zu einer effektiveren Ausbildung und Anwendung von String-Matching-Algorithmen in der Informatik beizutragen.

Table of Content

Contents

1 Introduction

String matching algorithms (SMAs) are fundamental components in the field of computer science, serving as essential tools for processing and analyzing textual data. These algorithms, designed to locate occurrences of a specific pattern within a larger text, play a crucial role in a wide array of applications across various domains.

Overview of String Matching Algorithms At their core, string matching algorithms aim to efficiently identify whether and where a given string appears within a larger string. In the context of these algorithms, we often refer to the string being searched for as the "pattern" or "needle", while the larger string being searched through is called the "text" or "haystack". These terms are used interchangeably in the field, with "pattern" and "text" being more common in academic literature, while "needle" and "haystack" are often used in programming contexts. Throughout this thesis, we will use both sets of terms to familiarize readers with the variety of terminology they may encounter.

While this task may seem straightforward, the sheer volume of data processed in modern computing environments necessitates optimized approaches. From the simple Naive Search to more sophisticated methods like Knuth-Morris-Pratt (KMP), Rabin-Karp, and Aho-Corasick, each algorithm offers unique strategies to balance efficiency and effectiveness in different scenarios.

Importance in Computer Science and Applications The significance of string matching algorithms extends far beyond theoretical computer science. These algorithms form the backbone of numerous practical applications, including [1]:

- **Text Editors and Digital Libraries:** Enabling efficient search and retrieval functionalities.
- **Search Engines:** Facilitating rapid information retrieval from vast databases.
- **Bioinformatics and Computational Biology:** Analyzing DNA sequences, protein structures, and performing tasks like DNA sequencing.
- **Network Intrusion Detection Systems:** Identifying potential security threats in network traffic.
- **Multimedia and Music Information Retrieval:** Searching and comparing musical patterns in databases.
- **Cheminformatics:** Searching and comparing chemical structures in databases.
- **File Comparison:** Identifying differences between files.
- **Plagiarism Detection:** Finding similarities between documents.

These applications underscore the critical importance of SMAs in our data-driven world, from text processing to advanced research and cybersecurity.

Challenges in Teaching and Learning String Matching Algorithms String matching algorithms, while fundamental to computer science, present unique challenges in university-level education. The dynamic nature of these algorithms, involving complex, step-by-step processes, is difficult to capture in static textbook illustrations or whiteboard demonstrations. In my experience studying computer science, I've observed that students often find it challenging to discern the nuanced differences and efficiency improvements among algorithms such as Naive Search, KMP, and Rabin-Karp. Traditional teaching methods may not completely address the diverse learning styles present in university classrooms, and students often find it difficult to connect theoretical knowledge with practical applications. The lack of interactive tools for experimentation and self-assessment further compounds these challenges, making it hard for students to effectively prepare for exams and develop a deep, intuitive understanding of string matching concepts.

Thesis Contribution: Interactive Visualization Tool - SMAVIZ In response to these challenges, this thesis presents an interactive visualization tool for the browser of a computer designed to enhance the understanding and learning of string matching algorithms. SMAVIZ features dynamic visualizations that bring abstract concepts to life through step-by-step animations. It offers an interactive interface where users can input custom strings, allowing for exploration of algorithm behavior across various scenarios. SMAVIZ covers multiple string matching algorithms, including Naive Search, KMP, Rabin-Karp, and Aho-Corasick, enabling comparative analysis. A context-sensitive help system, featuring the mascot "Frogbert Algorithmicus," provides timely explanations, guidance and insights. SMAVIZ also includes performance comparison features to analyze efficiency across different input scenarios. By leveraging modern web technologies such as Vue.js and D3.js, our solution bridges the gap between theoretical understanding and practical application. This interactive approach not only complements traditional lectures with hands-on experience but also serves as a valuable resource for exam preparation and self-assessment. Students can use it to verify their understanding, practice with custom inputs, and prepare for exam-style questions. The visual feedback and experimentation platform foster a more comprehensive and practical understanding of string matching algorithms.

2 Background

This chapter provides essential background on string matching algorithms and the role of visualization in education, forming the foundation for the development of SMAVIZ.

2.1 String Matching Algorithms

SMAs are integral to numerous applications in computer science. This subsection delves into their fundamental concepts, classifications, and historical evolution.

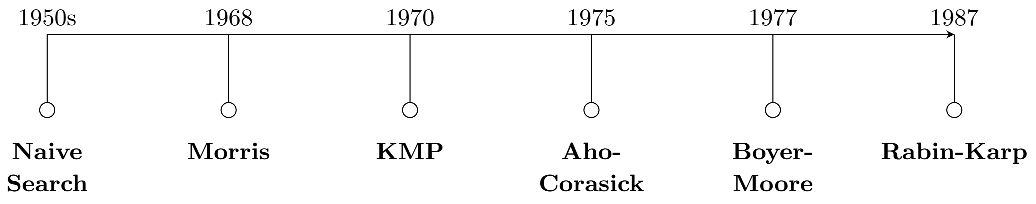
Fundamentals This section explores the core concepts and historical development of SMAs. These algorithms are fundamental tools in computer science, designed to efficiently locate occurrences of a specific pattern within a larger text. SMAs have widespread applications across various domains, playing a crucial role in modern computing. From powering search functions in text editors to enabling complex DNA sequence analysis in bioinformatics, these algorithms form the backbone of numerous technologies we interact with daily. As we delve into the fundamentals, we'll examine how SMAs have evolved to meet the growing demands of data processing in an increasingly digital world.

The importance of efficient string matching algorithms has grown significantly with the exponential increase in digital textual data. As the volume of information processed by modern computing systems continues to expand, the demand for fast and accurate string matching becomes ever more critical. These algorithms must be capable of handling vast amounts of data while maintaining high performance and precision. Consequently, the development and optimization of string matching algorithms remain active areas of research in computer science, driven by the need to process and analyze increasingly large datasets across diverse applications.

String matching algorithms can be broadly categorized into two types [2]:

1. **Single Pattern Matching**: Algorithms that search for a single pattern in a text. Examples include Naive Search, KMP, and Rabin-Karp algorithms.
2. **Multi-Pattern Matching**: Algorithms that can search for multiple patterns simultaneously in a given text. Examples include the Aho-Corasick and Commentz-Walter algorithms.

Brief History String matching algorithms have evolved significantly since the early days of computer science. Here's a brief timeline of key developments:



1. **Naive Search Algorithm**: This straightforward approach compares the pattern with every possible substring in the text. It's simple but inefficient for large texts. Works

well for short patterns or small texts. Unlike others, it doesn't use any preprocessing or advanced techniques.

2. **Knuth-Morris-Pratt (KMP) Algorithm**: Utilizes information from previous match attempts to skip unnecessary comparisons. Efficient for patterns with repeating substrings. Differs from Naive Search by avoiding backtracking in the text. [3]

3. **Aho-Corasick Algorithm**: Constructs a finite state machine from the patterns for efficient multiple pattern matching. Excels at searching for multiple patterns simultaneously in a text. Unlike single-pattern algorithms, it can find all occurrences of multiple patterns in one pass through the text. [4]

4. **Boyer-Moore Algorithm**: Scans the pattern from right to left and can skip large portions of the text. Performs well with large alphabets and long patterns. Distinguishes itself by potentially skipping more characters than other algorithms. **(not implemented in SMAVIZ)** [5]

5. **Rabin-Karp Algorithm**: Uses hashing to compare patterns, allowing for quick checks of potential matches. Effective for multiple pattern searching and plagiarism detection. Unique in its use of rolling hash functions to efficiently compute hash values. [6]

2.2 Importance of Visualization in Learning

The use of visualizations in learning has been recognized as a powerful tool to enhance understanding and retention of complex concepts. Research in multimedia learning has consistently demonstrated the benefits of combining visual elements with textual information to improve learning outcomes.

In general learning contexts, visualizations offer several key advantages [7]:

1. **Improved performance on pictorial recall and transfer tasks:** Visualizations enable learners to better understand and apply complex concepts, leading to superior performance in tasks that require visual understanding and knowledge transfer.

2. **Reduced cognitive load:** By offloading some of the mental processing onto visual representations, learners experience lower perceived difficulty, allowing them to focus more on understanding the content rather than struggling with its complexity.

3. **Increased generation of inferences:** The presence of visualizations encourages learners to make more connections and draw conclusions beyond what is explicitly stated, fostering deeper engagement with the material.

When it comes to understanding algorithms specifically, visualizations provide additional benefits:

1. **Concretization of Abstract Concepts**: Algorithms often involve abstract operations that can be challenging to grasp through text alone. Visualizations help to make these concepts more tangible and easier to understand.

2. **Step-by-Step Understanding**: Visual representations allow learners to see how an algorithm progresses step-by-step, making it easier to follow the logic and flow of the algorithm.

3. **Pattern Recognition**: Visualizations can help learners identify patterns in algorithm behavior, leading to deeper insights and understanding. This ability to recognize patterns is crucial in developing a more intuitive grasp of algorithmic principles.

The effectiveness of visualizations in learning is supported by research in cognitive science and educational psychology. However, it's important to note that the effectiveness of visualizations can depend on various factors, including the nature of the content, the design of the visualizations, and individual learner characteristics. Therefore, careful consideration should be given to the implementation of visualizations in educational contexts to maximize their benefits, particularly when dealing with complex subjects like algorithms. [7]

3 Overview

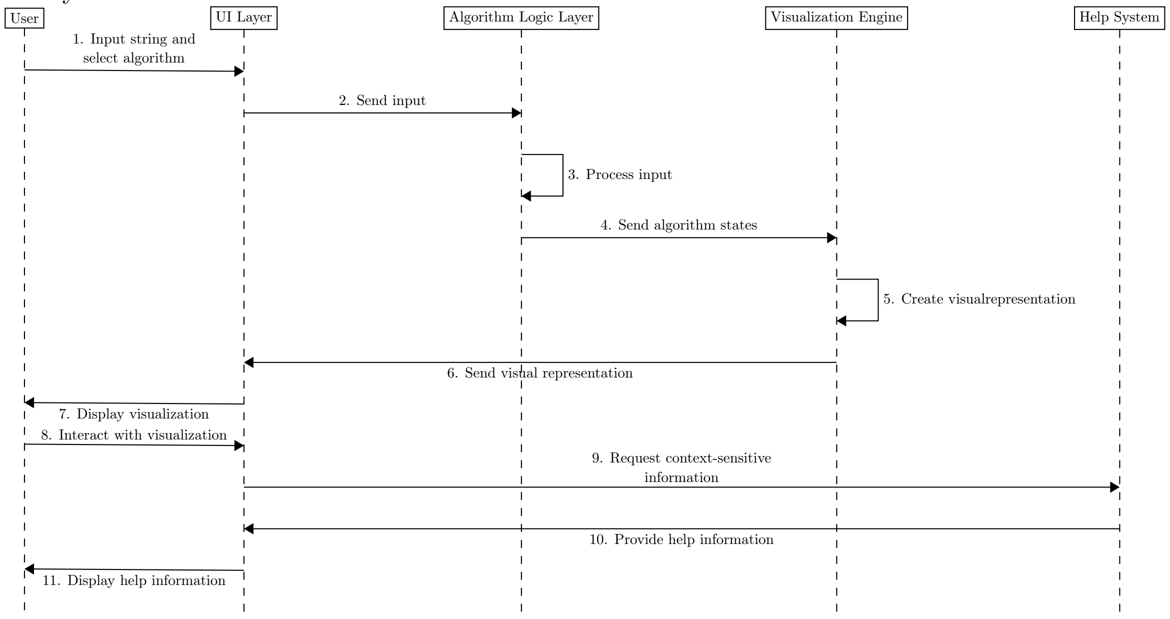
This chapter provides a high-level overview of SMAVIZ. We'll discuss the system architecture, main features, introduce our unique context-sensitive help system and key technologies used.

3.1 Architecture

SMAVIZ is designed as a web-based application, leveraging modern front-end technologies to provide an interactive and responsive user experience. The system is structured around a single-page application architecture, with the following key components:

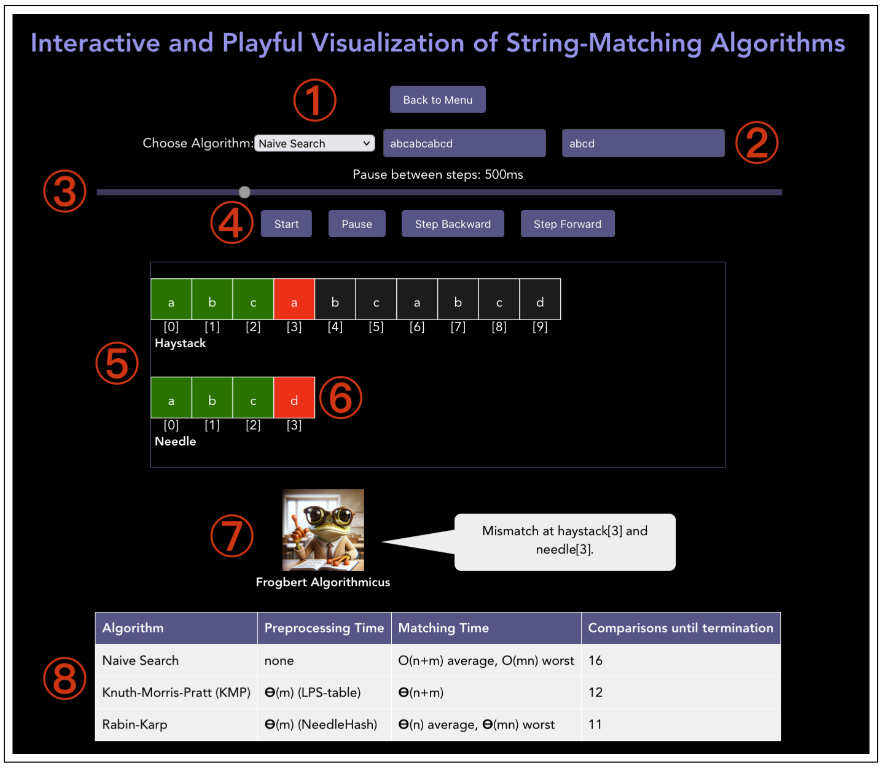
- **User Interface (UI) Layer:** Handles user interactions and displays visualizations.
- **Algorithm Logic Layer:** Implements the string matching algorithms.
- **Visualization Engine:** Translates algorithm states into visual representations.
- **Help System:** Provides context-sensitive guidance to users.

Workflow: The following sequence diagram illustrates the typical flow of interactions within our system:


User
UI Layer
Algorithm Logic Layer
Visualization Engine
Help System
1. Input string and select algorithm
2. Send input
3. Process input
4. Send algorithm states
5. Create visualrepresentation
6. Send visual representation
7. Display visualization
8. Interact with visualization
9. Request context-sensitive information
10. Provide help information
11. Display help information

3.2 Key Features

SMAVIZ incorporates a range of features specifically designed to facilitate the learning and comprehension of SMAs. These key components are illustrated in Figure 1, which provides a comprehensive overview of SMAVIZ's interface and functionality:



Interactive and Playful Visualization of String-Matching Algorithms

Choose Algorithm: Naive Search

abcabcabcd

abcd

Pause between steps: 500ms

Mismatch at haystack[3] and needle[3].

Algorithm	Preprocessing Time	Matching Time	Comparisons until termination
Naive Search	none	O(n+m) average, O(mn) worst	16
Knuth-Morris-Pratt (KMP)	Θ(m) (LPS-table)	Θ(n+m)	12
Rabin-Karp	Θ(m) (NeedleHash)	Θ(n) average, Θ(mn) worst	11

Figure 1: Screenshot of SMAVIZ's main page

1. **Interactive Algorithm Selection:** Users can choose from multiple string matching algorithms, including Naive Search, KMP and Rabin-Karp. Aho-Corasick can be found in a seperate section when returning to the Menu because it is not a Single-Pattern algorithm.

2. **Custom Input:** Users can input their own strings and patterns, allowing for experimentation with different scenarios.

3. **Speed Control:** The visualization speed can be adjusted, allowing users to slow down complex steps or speed through simpler ones.

4. **Step-by-Step Navigation:** Users can also manually move forward and backward through the algorithm's steps, facilitating deeper understanding by pausing and analyzing critical points.

5. **Real-time Visualization:** As the selected algorithm runs, users can see a step-by-step visual representation of its operation on the input string which will be demonstrated later.

6. **Color highlighting:** The user is provided with color-coded elements to highlight pattern matches, as well as comparisons and hash value mismatches.

7. **Context-Sensitive Help System:** Frogbert Algorithmicus provides useful information, guiding the user and explaining critical steps during algorithm execution.

8. **Algorithm Comparison:** SMAVIZ allows side-by-side comparison of different algorithms on the same input, highlighting their relative efficiencies by calculating how many comparisons each algorithm would need until termination.

3.3 Context-Sensitive Help System: Frogbert Algorithmicus

To enhance the learning experience, we've introduced a unique context-sensitive help system personified by our mascot, Frogbert Algorithmicus illustrated in Figure 2.

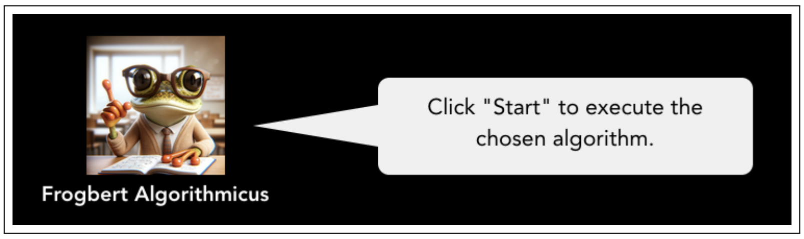


Figure 2: Mascot Frogbert Algorithmicus giving an instruction

This system provides:

- **Dynamic Commentary:** Frogbert offers context-specific comments tailored to the current step of the algorithm, helping users understand what's happening at each stage.
- **Proactive Hints:** Without user prompting, Frogbert provides hints and suggestions, guiding users on what to consider or try next in the visualization process.
- **Algorithmic Insights:** Frogbert shares brief, relevant insights into algorithm behavior and efficiency considerations at appropriate moments during the visualization.
- **Engaging Presentation:** The mascot's friendly, approachable persona makes complex algorithmic concepts more accessible and less intimidating to learners, presenting information in a lighthearted manner.

Frogbert Algorithmicus does not engage in direct question-answering or interactive dialogue. Instead, it provides pre-programmed, context-sensitive guidance that automatically appears based on the user's current position in the algorithm visualization. This approach ensures that learners receive relevant information and explanations at the right moments, supporting their understanding without interrupting the flow of the visualization.

Through this combination of interactive visualization, powerful technologies, and automated contextual guidance, SMAVIZ aims to provide a comprehensive and engaging platform for learning and understanding string matching algorithms.

3.4 Technologies Used

Our application leverages two primary technologies:

- **Vue.js:** A progressive JavaScript framework used for building the user interface and managing application state. Vue.js's reactive data model and component-based architecture allow for efficient updates and modular design.

- **Graphviz-D3:** A library that internally utilizes D3.js to create dynamic, interactive visualizations of the string matching algorithms. This library provides an abstraction layer over D3.js, allowing us to generate detailed and responsive visualizations without directly interacting with D3.js.

Vue.js Framework: A Brief Introduction Vue.js is a progressive JavaScript framework ideal for building interactive user interfaces, making it perfect for creating dynamic algorithm visualizations. The term "progressive" means that it can be incrementally adopted, allowing developers to integrate it into existing projects gradually or use it to build full-fledged Single-Page Applications. At the core of Vue.js is its reactivity system. This system automatically tracks JavaScript object property changes through a process called dependency tracking. When a property used in a component's template is modified, Vue efficiently updates only the parts of the DOM that depend on that property. This is achieved through a virtual DOM implementation, which minimizes direct manipulation of the browser's DOM for better performance. The framework's component-based architecture promotes the creation of reusable and maintainable code, which is particularly beneficial in complex applications. Vue's virtual Document Object Model (DOM) implementation ensures efficient rendering and updates, contributing to smooth and responsive user experiences. For our string matching algorithm visualization tool, Vue.js offers several advantages. Its declarative rendering system simplifies the process of binding the algorithm's state to the UI, while its reactive data model ensures that changes in the algorithm's execution are immediately reflected in the visualization. The component structure allows us to modularize different parts of the visualization, such as the input controls, algorithm display, and performance metrics, making the codebase more organized and easier to maintain. [8]

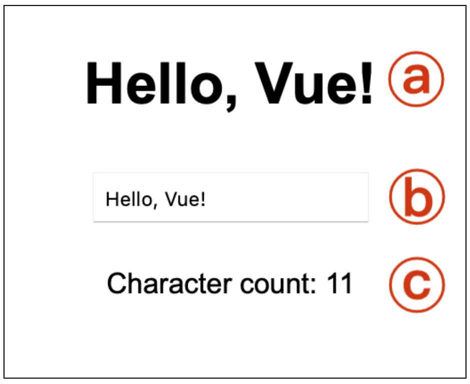
Let's examine a simple "Hello World" example demonstrating Vue's reactivity:

Listing 1: Vue.js component with character counter

```
1  <template>
2    <div>
3      <h1>{{ message }}</h1>
4      <input v-model="message">
5      <p>Character count: {{ characterCount }}</p>
6    </div>
7  </template>
8
9  <script>
10 export default {
11   data() {
12     return {
13       message: 'Hello, Vue!'
14     }
15   },
16   computed: {
17     characterCount() {
18       return this.message.length
19     }
20   }
21 }
22 </script>
23
24 <style scoped>
25 div {
26   font-family: Arial, sans-serif;
27   text-align: center;
28   margin-top: 20px;
29 }
30 input {
31   margin: 10px 0;
32   padding: 5px;
33 }
34 </style>
```

In this Vue.js component (Listing 1), we can observe the following key aspects of data binding and reactivity. For easier understanding we will take a look at Figure 3, which displays the rendered component where `(a)` shows the `<h1>` tag (line 3), `(b)` the input field (line 4) and `(c)` the computed value of `characterCount()` (line 17):

1. **Data Binding:**
   - One-way binding: The `{{ message }}` in the `<h1>` tag (a) shows the value of `message` from the JavaScript. If `message` changes in the JavaScript, the `<h1>` (a) text updates automatically, but changing the `<h1>` text doesn't affect the JavaScript `message`.
   - Two-way binding: The `v-model="message"` on the `<input>` (line 4) connects the input's text to the `message` in JavaScript. Typing in the input (b) updates `message`,


Hello, Vue!

Hello, Vue!

Character count: 11

Figure 3: Rendered Vue.js component with character counter

and changing `message` in JavaScript updates the input's text (b).

2. **Reactivity:**
   - The `data()` function (lines 11-15) creates the `message` variable that Vue watches for changes.
   - Vue keeps track of where `message` is used in the HTML and updates all occurrences automatically.
   - The `computed` section (lines 16-20) creates `characterCount`, which also automatically updates when `message` changes.
3. **Reactivity in Action:**
   - When typing in the input box, Vue updates the `message` variable.
   - Vue then automatically updates:
     (a) The `<h1>` text to show the new `message`.
     (b) The `characterCount` to reflect the new length of `message`.
     (c) The displayed character count in the HTML.

This example shows how Vue automatically updates the webpage when data changes, without needing to manually change the HTML. The two-way binding (`v-model`) keeps the input box and `message` in sync, while computed properties like `characterCount` automatically update based on other data changes.

We will now examine another example in Figure 4 where we modify the string in the input field `(b)`:

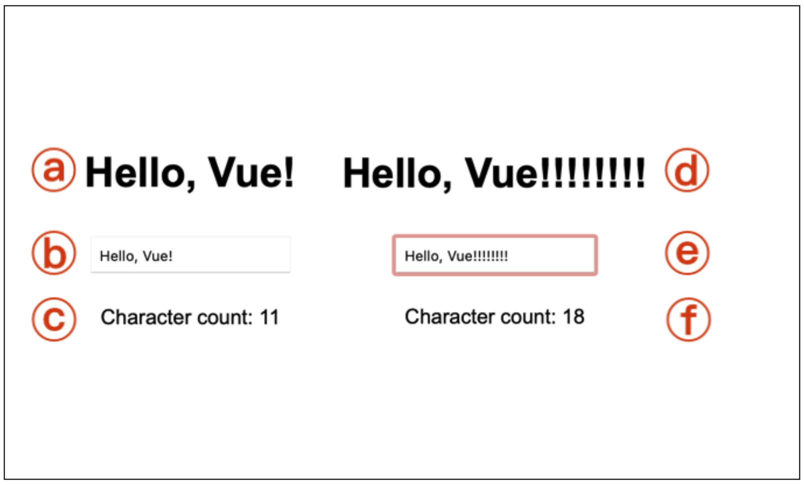


Figure 4: Binding of a computed property to a reactive property

On the left, we see the initial state with the message "Hello, Vue!" in both the `<h1>` heading `(a)` and input field `(b)`, and a character count of 11 `(c)`. The right side demonstrates the component's state after user interaction, where seven additional exclamation marks have been added to the input `(e)`. Vue automatically updates the displayed message in the `<h1>` heading `(d)` and increases the character count to 18 `(f)`, all in real time without manual DOM manipulation.

This reactivity is fundamental to SMAVIZ. When the algorithm's state changes or users interact with the visualization, Vue automatically updates the visual representation. Vue tracks these dependencies and re-renders affected DOM parts when changes occur. By leveraging Vue's reactivity, we create algorithm visualizations that are both informative and interactive. Users can manipulate inputs, control algorithm flow, and see immediate results, fostering a deeper understanding of how string matching algorithms work in practice. This immediate feedback loop is particularly valuable for visualizing how changes in input affect an algorithm's behavior and efficiency.

4 Visualizations

Building upon the foundational sections and technologies discussed in the previous chapters, we now turn our attention to the core of SMAVIZ: the individual SMA visualizations.

In the following sections, we will explore the four implemented SMAs: Naive Search, KMP, Rabin-Karp, and Aho-Corasick. For each algorithm, we will delve into its underlying principles, implementation details, and most importantly, how we've translated these concepts into visual, interactive elements within SMAVIZ by providing example executions of different algorithms on custom input strings.

We will now proceed to examine the visual representations of string matching algorithms, beginning with the fundamental Naive Search approach.

4.1 Naive Search Algorithm

To begin our exploration of string matching algorithms, we'll start with the most fundamental approach: the Naive Search algorithm. This method serves as a foundation for understanding more complex algorithms that follow.

4.1.1 Algorithm Overview

The Naive Search algorithm, also known as the brute force string matching algorithm, is the most straightforward approach to string matching. It operates by sliding a search window, equal in length to the pattern, across the text. At each position, the algorithm compares the characters in the window with those in the pattern. If a mismatch is found, the window shifts one position to the right, and the comparison begins anew. If all characters match, a successful pattern occurrence is reported. This process continues until the window has traversed the entire text. The algorithm's simplicity makes it an excellent starting point for understanding the fundamentals of string matching.

The algorithm works as follows:

1. It starts by aligning the pattern with the beginning of the text.
2. It compares the characters of the pattern with the corresponding characters in the text, one by one.
3. If all characters match, it reports a successful match.
4. If a mismatch is found or after a successful match, it shifts the pattern one position to the right.
5. Steps 2-4 are repeated until the end of the text is reached.

The algorithm can be represented in pseudocode as follows:

```
for i = 0 to n - m
    for j = 0 to m - 1
        if T[i + j] != P[j]
```

```
            break
    if j == m
        report match at position i
```

Despite the fact this algorithm is almost trivial and has a worst-case time complexity of O(nm), it often already performs efficiently in practice, especially with natural language texts. This time complexity means that in the worst case, the algorithm's running time grows linearly with both the length of the text (n) and the length of the pattern (m). In practical terms, this implies that for very long texts or patterns, the algorithm's performance may degrade significantly. For instance, searching for a 1000-character pattern in a 1,000,000-character text could require up to 1 billion character comparisons in the worst case. However, for shorter texts and patterns or in average cases, the algorithm can still be quite efficient. Its conceptual simplicity makes it an ideal candidate for visualization. In SMAVIZ, we leverage this to provide a clear, step-by-step representation of the matching process, allowing users to easily grasp the fundamental concepts of string matching before exploring more complex algorithms.

With a foundational understanding of the Naive Search algorithm's mechanics, we now turn to its implementation within SMAVIZ and how we visualize each step to enhance learner comprehension.

4.1.2 Implementation Details

Our implementation of the Naive string matching algorithm emphasizes its straightforward, single-phase nature. Unlike more complex algorithms, the Naive approach does not require any preprocessing or additional data structures.

Single-Phase Search The Naive algorithm operates in a single phase, directly comparing characters between the needle (pattern) and the haystack (text). Our main window in Figure 5 displays this process:

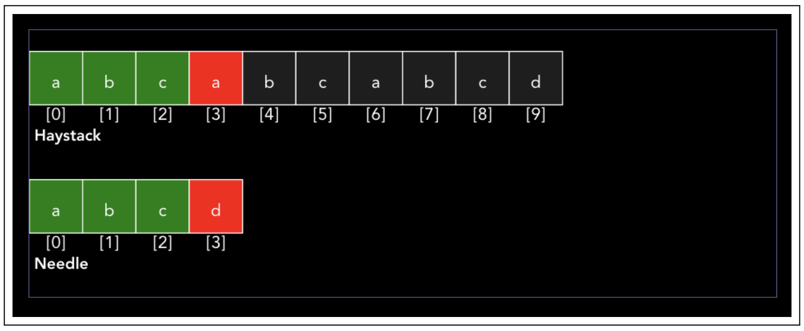


Figure 5: Main window showing character comparison until mismatch

The visualization presents the haystack and needle in separate, labeled character arrays. As the algorithm progresses, each comparison is visually highlighted, enabling users to follow the process character by character. Figure 5 demonstrates how matching characters are highlighted in green, while mismatches are shown in red. The current position of the needle within the haystack is clearly indicated, and Figure 6 illustrates how the algorithm shifts the pattern after

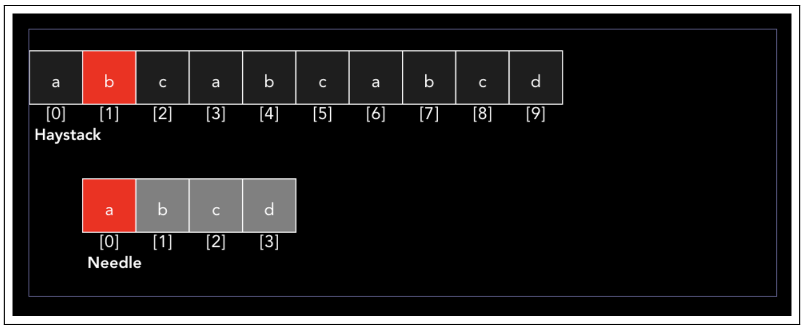


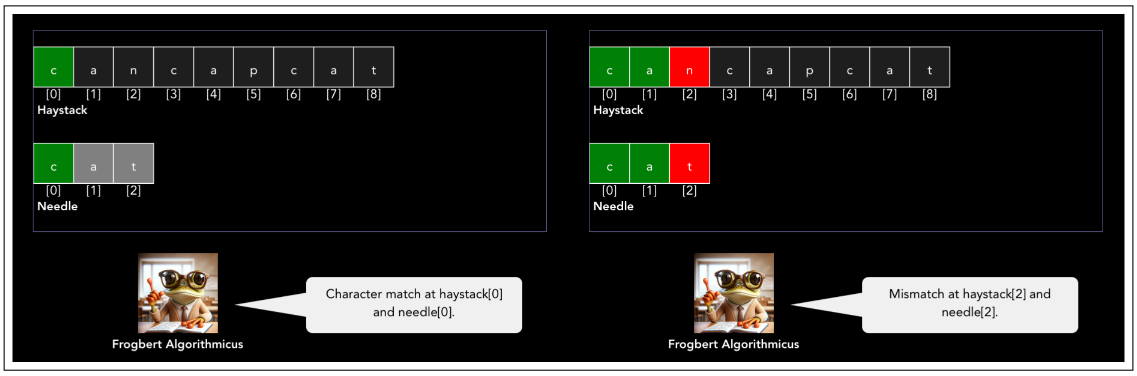
Figure 6: Main window showing needle shift after a mismatch

encountering a mismatch. This direct visualization approach allows users to observe the algorithm's simplicity and understand its step-by-step execution, including potential inefficiencies when dealing with longer texts or patterns with many partial matches. This hands-on, visual approach helps reinforce the conceptual understanding of the Naive string matching algorithm and sets the stage for understanding the improvements offered by more advanced algorithms.

4.1.3 Example Usage

To better illustrate how the Naive String Matching Algorithm operates in practice, let's walk through a step-by-step example using SMAVIZ. This demonstration will showcase the algorithm's behavior as it searches for a specific pattern within a given text.

Figure 7 illustrates the initial steps of the Naive String Matching Algorithm searching for the pattern "cat" (needle) in the text "cancapcat" (haystack). The image shows the very first attempt. The algorithm begins by comparing the haystack and needle starting at index 0 for both (left). Two steps later the first two characters "ca" match (shown in green), but a mismatch occurs at the third position where 't' in the needle doesn't match 'n' in the haystack (right). Frogbert Algorithmicus points out this mismatch at haystack[2] and needle[2].



Character match at haystack[0] and needle[0].

Mismatch at haystack[2] and needle[2].

Figure 7: Character comparison until first mismatch using Naive Search

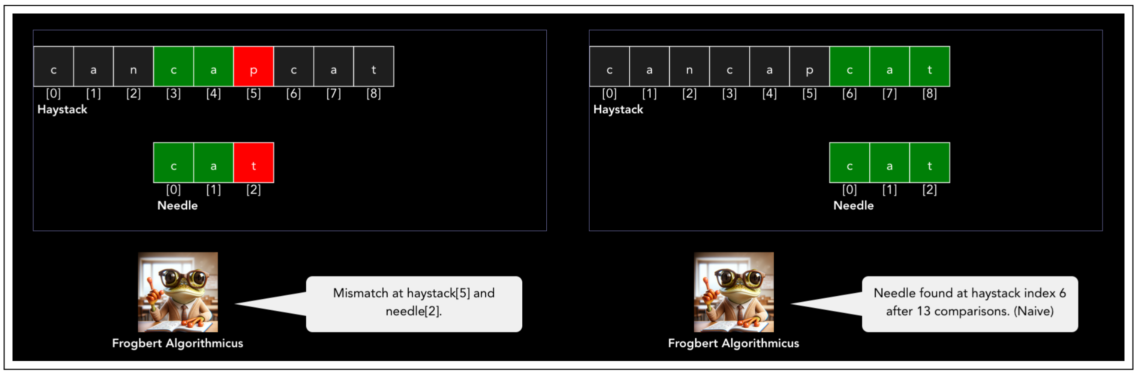


Figure 8: Further comparison until full match using Naive Search

Figure 8 shows the algorithm's progression and final step. The image (left) depicts an intermediate step after the needle has been shifted right multiple times due to mismatches. Here, only the first two characters 'c' and 'a' match with the corresponding indices in the haystack. The third character 't' of the needle does not match with the respective character above 'p', so Frogbert points out once again that a mismatch has occurred at haystack[5] and needle[2]. The right image displays the successful match, which is the final step. The needle "cat" is aligned with the haystack starting at index [6], with all characters highlighted in green to indicate a complete match. Frogbert informs us that the needle was found at haystack index 6 after 13 comparisons using the Naive approach. These visualizations effectively demonstrate the naive approach's systematic comparison and shifting process. The algorithm progresses through the haystack, comparing characters and shifting the needle one position to the right after each mismatch. This process continues until either the full pattern is found or the end of the text is reached, highlighting the simplicity and potential inefficiency of the Naive String Matching Algorithm for larger texts or patterns.

4.2 Knuth-Morris-Pratt (KMP) Algorithm

The KMP algorithm represents a significant advancement in string matching efficiency, offering improved performance over naive approaches. This section examines the algorithm's fundamental principles and operational methodology.

4.2.1 Algorithm Overview

The KMP algorithm improves upon the Naive Search by eliminating unnecessary comparisons, particularly when dealing with patterns containing repeating substrings. The key to KMP's efficiency is its preprocessing of the pattern to create a failure function. This failure function, also known as the partial match table or Longest Prefix Suffix table (LPS), is a crucial component of the KMP algorithm. Throughout this thesis, we will consistently refer to this concept as the failure function for clarity.
KMP operates in two main phases:

1. **Preprocessing Phase:** The algorithm analyzes the pattern to build the failure function,

which determines how far the pattern should be shifted when a mismatch occurs.

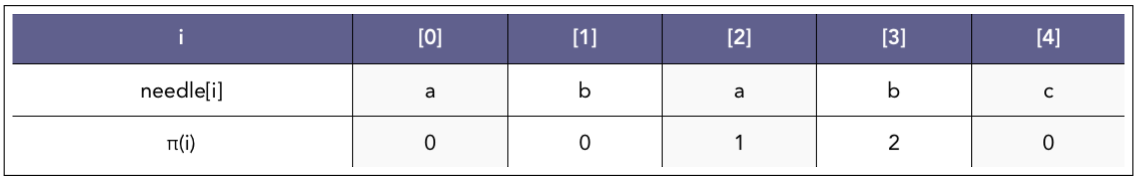
2. **Searching Phase:** Using the preprocessed information, the algorithm scans the text, comparing characters just like in the Naive Search and using the failure function to make intelligent jumps when mismatches occur.

KMP's efficiency stems from its use of preprocessed information, significantly improving upon the Naive Search algorithm. This improvement is particularly evident in two scenarios: when dealing with patterns containing repeating substrings and when searching through extensive texts.

4.2.2 Implementation Details

Our implementation of the KMP algorithm focuses on providing a clear visualization of both the preprocessing and searching phases, with an emphasis on interactive learning. Here's how we approach each phase:

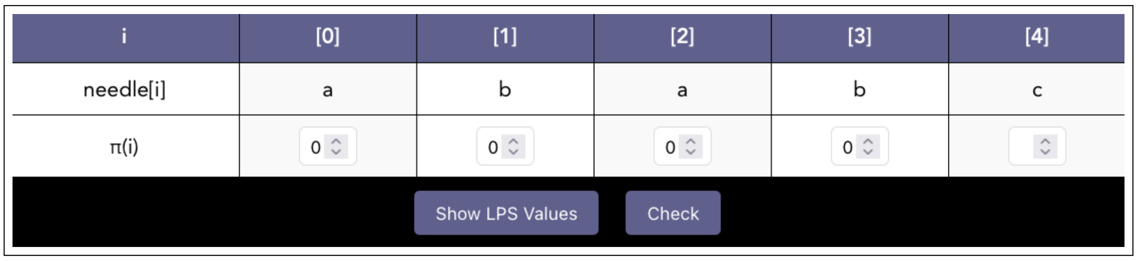
Preprocessing Phase For a given needle (pattern) string, SMAVIZ automatically generates the failure function which is illustrated in Figure 9. This table is a crucial component of the KMP algorithm and is visually presented to the user. The failure function determines how far the pattern can be shifted when a mismatch occurs, allowing for more efficient searching. To enhance understanding, we've made the failure function interactive:



i	[0]	[1]	[2]	[3]	[4]
needle[i]	a	b	a	b	c
π(i)	0	0	1	2	0

Figure 9: Failure function of KMP algorithm

- Users can hide the values of the table with a button click.
- They can then enter their own values into the table.
- Another button allows users to check if their entered values are correct.



i	[0]	[1]	[2]	[3]	[4]
needle[i]	a	b	a	b	c
π(i)	0	0	0	0	

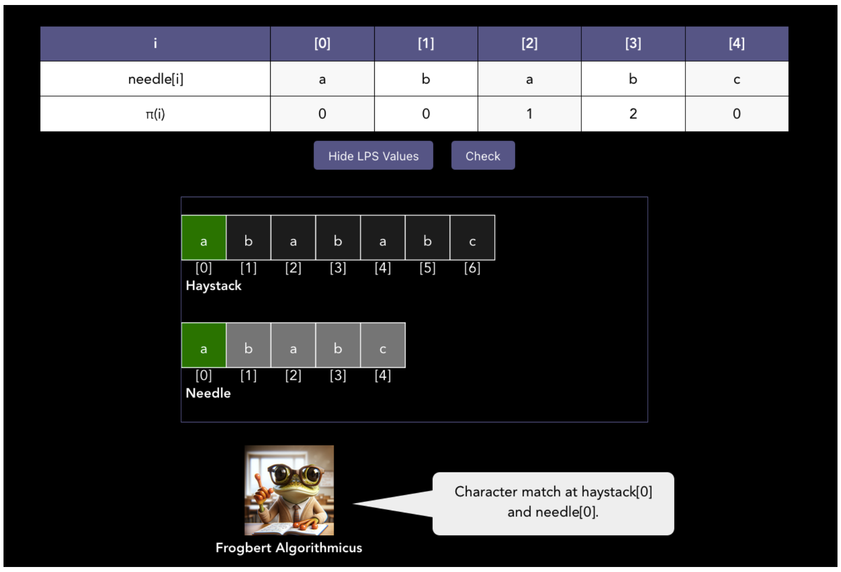
Figure 10: Failure function of KMP algorithm with hidden solution, ready for user input

This interactive feature serves as an excellent training tool, allowing users to test and reinforce their understanding of how the failure function is constructed for self chosen strings.

Searching Phase During the main search phase, the algorithm scans through the haystack string, comparing characters with the needle. When a mismatch occurs, instead of shifting the needle by just one position as in the Naive Search, the algorithm consults the precomputed failure function. This table guides the algorithm in making intelligent jumps, potentially skipping many unnecessary comparisons. By visualizing both the failure function table and the searching process, SMAVIZ allows users to observe how KMP leverages precomputed information to achieve its efficiency.

4.2.3 Example Usage

To better illustrate the efficiency and functionality of the KMP algorithm, let's walk through a step-by-step example using SMAVIZ. This demonstration will showcase how KMP utilizes the failure function to make intelligent shifts, highlighting its advantages over the Naive Search approach.



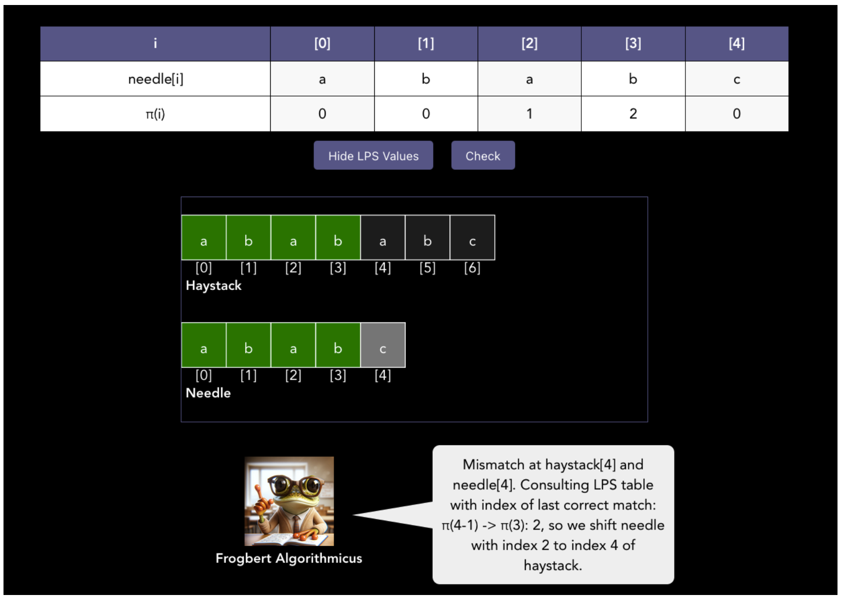
i	[0]	[1]	[2]	[3]	[4]
needle[i]	a	b	a	b	c
π(i)	0	0	1	2	0

Character match at haystack[0] and needle[0].

Figure 11: Starting character comparison using KMP

Figure 11 illustrates the KMP algorithm in action, starting by comparing the first character of the haystack 'abababc' with the first character of the needle 'ababc'. Initially, the algorithm proceeds similarly to the naive approach, successfully matching the first four characters. However, a mismatch occurs at haystack[4] and needle[4].

At this point, Frogbert Algorithmicus, our guide, highlights the mismatch, as seen in Figure 12, and explains the KMP algorithm's next step. Instead of shifting by just one position, KMP consults the failure function. The algorithm uses the index of the last correct match, which is the current index minus one (4 - 1 = 3), to look up the appropriate shift in the failure function.

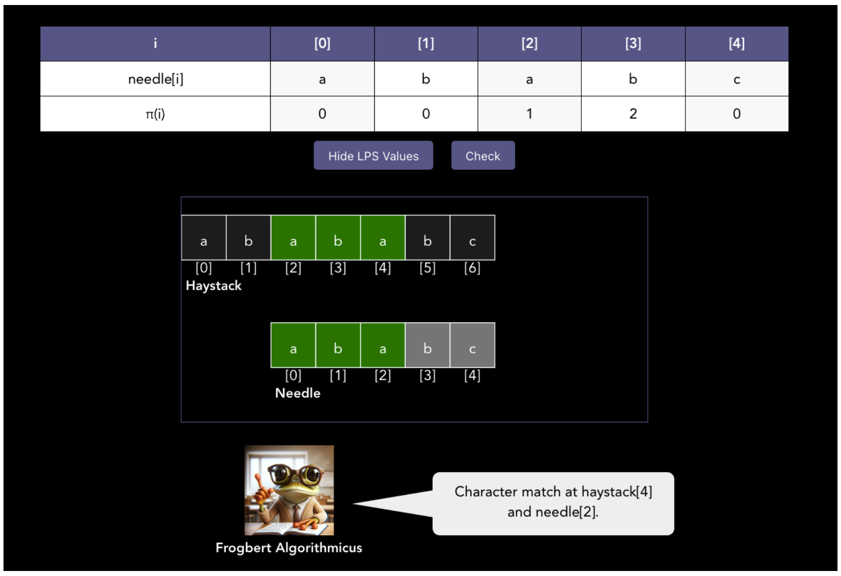


i	[0]	[1]	[2]	[3]	[4]
needle[i]	a	b	a	b	c
π(i)	0	0	1	2	0

Mismatch at haystack[4] and needle[4]. Consulting LPS table with index of last correct match: π(4-1) -> π(3): 2, so we shift needle with index 2 to index 4 of haystack.

Figure 12: Character comparison until first mismatch and failure function consultation

The failure function indicates that for index 3, pi(3) = 2. This means the algorithm can safely shift the needle so that its index 2 aligns with the current haystack index (4).

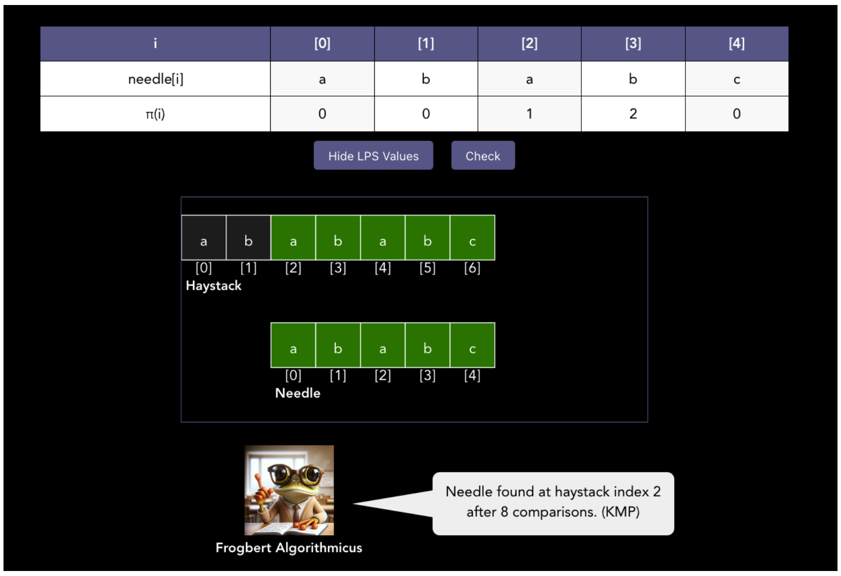


i	[0]	[1]	[2]	[3]	[4]
needle[i]	a	b	a	b	c
π(i)	0	0	1	2	0

Character match at haystack[4] and needle[2].

Figure 13: Intelligent shift skipping unnecessary comparison steps after consulting the failure function

This intelligent shift, shown in Figure 13, allows KMP to skip several comparisons that the naive approach would have made, demonstrating its improved efficiency. By leveraging the precomputed failure function information, KMP avoids unnecessary comparisons and backtracking, resulting in a more efficient string matching process.



i	[0]	[1]	[2]	[3]	[4]
needle[i]	a	b	a	b	c
π(i)	0	0	1	2	0

Needle found at haystack index 2 after 8 comparisons. (KMP)

Figure 14: Character comparison until full match using KMP

The algorithm continues the character matching until the needle is found at haystack index 2, as shown in Figure 14.

4.3 Rabin-Karp Algorithm

The Rabin-Karp algorithm also represents an advancement in string matching efficiency, offering a different approach that leverages hashing techniques. This section examines the algorithm's fundamental principles and operational methodology that distinguishes it from previously discussed algorithms.

4.3.1 Algorithm Overview

The Rabin-Karp algorithm enhances string matching efficiency by employing a hashing technique, significantly reducing the number of character comparisons required. The algorithm's key innovation lies in its use of a rolling hash function, enabling swift substring comparisons and efficient updates as it progresses through the text.

Rabin-Karp operates in two main phases:

1. **Preprocessing Phase:** The algorithm calculates the hash value of the pattern and the initial hash value of the first m characters in the text, where m is the pattern length.
2. **Searching Phase:** Employing a sliding window of length m, the algorithm traverses the text. It efficiently updates the window's hash value at each step. When a hash match occurs, it performs a character-by-character verification to confirm a true match.

The algorithm's efficiency stems from its ingenious use of rolling hash computation. As the window slides, instead of recalculating the entire hash, it merely subtracts the contribution of the exiting character and adds that of the entering character. This constant-time hash update mechanism significantly accelerates the search process compared to naive approaches.

4.3.2 Implementation Details

Our implementation of the Rabin-Karp algorithm focuses on providing a clear visualization of both the hash value comparison process and the character-by-character comparison. SMAVIZ is designed to highlight the algorithm's efficiency in reducing unnecessary comparisons.

Hashing Visualization For both the needle (pattern) and the current window of the haystack (text), SMAVIZ automatically computes and displays the hash values using a simple hash function. The specific implementation of this function is abstracted away, allowing users to focus on the algorithm's core concepts. The hash value of the needle is computed once and displayed above it in a speech bubble. As the algorithm progresses through the haystack, the hash value of the current window is continuously updated and displayed in a speech bubble above the relevant portion of the text, which is highlighted in grey. This visual representation allows users to observe how hash values change as the algorithm moves through the text, providing insight into the core mechanism of the Rabin-Karp algorithm.

Searching Phase During the main search phase, the algorithm compares the hash values of the needle and the current window of the haystack. When hash values differ, SMAVIZ highlights the differing digits in red, visually indicating why a full character comparison is unnecessary at that position. This immediate visual feedback helps users understand how Rabin-Karp efficiently skips unnecessary character comparisons. In cases where hash values match, SMAVIZ highlights all matching digits in green and proceeds to compare characters directly, similar to the naive approach. This step demonstrates how Rabin-Karp combines the efficiency of hash comparisons with the thoroughness of character-by-character matching when needed. As the window slides through the haystack, users can observe how the hash value updates, illustrating the rolling hash concept central to the algorithm's efficiency.

By visualizing both the hash computations and the searching process, SMAVIZ allows users to observe how Rabin-Karp leverages hash values to achieve its efficiency. This comprehensive visual approach provides users with a deep understanding of the algorithm's mechanics, from initial hash computation to final pattern matching.

4.3.3 Example Usage

The following figures illustrate the Rabin-Karp algorithm in action, demonstrating its key features and efficiency.

Figure 15 showcases the algorithm's initial steps. On the left, we see the hash of the needle compared to the hash of the current haystack window. The red highlighting of a digit in the haystack hash indicates a mismatch. The right image shows the result after shifting the window one position to the right and recalculating the hash. Again, the hashes do not match, necessitating another shift.

In Figure 16, we observe a critical moment in the algorithm's execution. After another shift, the hashes of the needle and the current haystack window match perfectly, as indicated by the green highlighting. This match triggers the next phase of the algorithm: a character-by-character comparison to rule out the possibility of a hash collision.

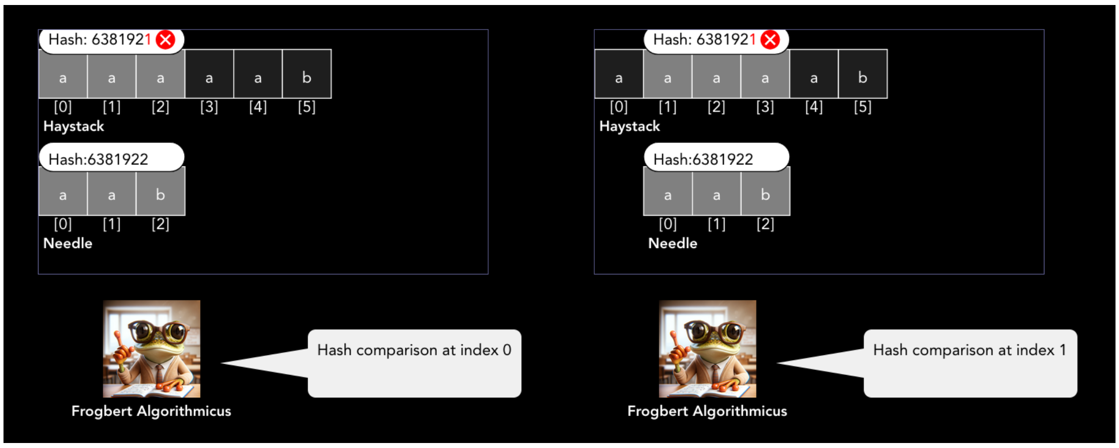


Figure 15: Comparing hash values using Rabin-Karp

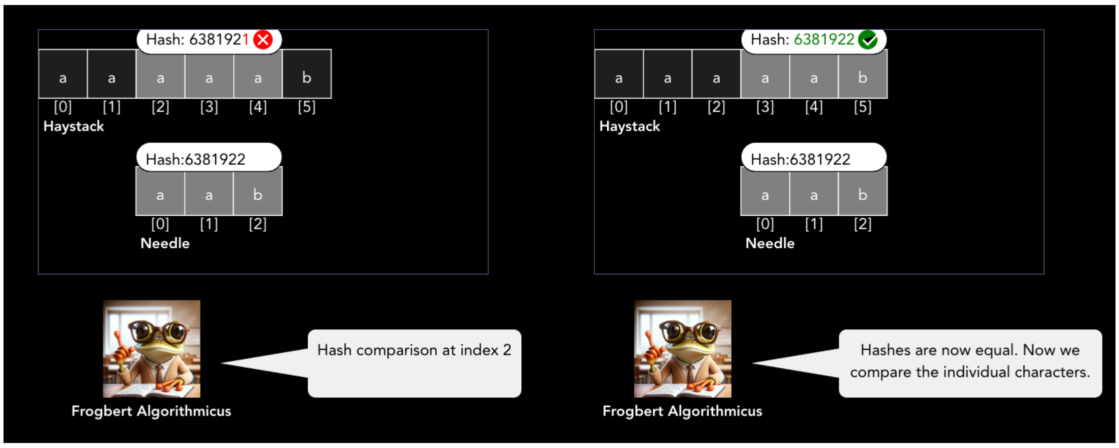


Figure 16: Discovery of a substring with matching hash value, initiating character-by-character comparison

The final stages of the algorithm are depicted in Figure 17. On the left, we see the individual character comparison in progress following the successful hash match. The right image shows the conclusion of this process, confirming a complete match between the pattern and the substring.

This sequence of images effectively demonstrates the Rabin-Karp algorithm's efficiency. By using hash comparisons to quickly identify potential matches and only performing detailed character comparisons when necessary, the algorithm significantly reduces the total number of comparisons required. This approach is particularly beneficial when searching for multiple patterns or in longer texts, showcasing why Rabin-Karp is often preferred over naive string-matching methods in such scenarios.

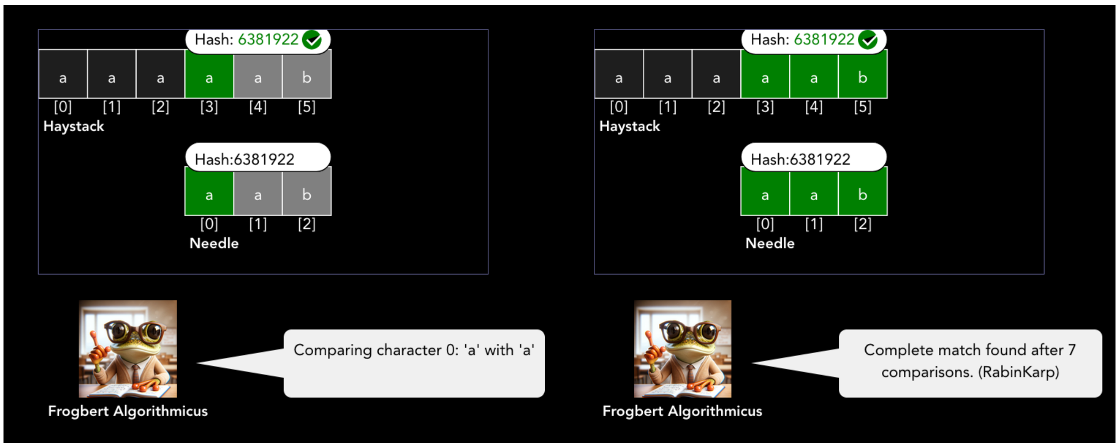


Figure 17: Character-by-character verification until full match

4.4 Single Pattern Algorithm Performance Visualization and Comparison

To provide a general overview of the efficiency gains offered by different string matching algorithms, SMAVIZ implements a performance comparison table. This table, as shown in Figure 18, offers a side-by-side comparison of the Naive Search, KMP, and Rabin-Karp algorithms by calculating the necessary number of comparisons needed until termination for the given input string and also information about the preprocessing time as well as matching time.

The table presents three crucial aspects of each algorithm:

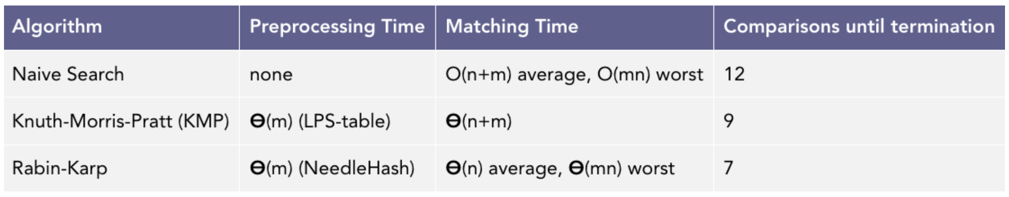
1. **Preprocessing Time:** This column highlights the initial setup required by each algorithm. While the Naive Search requires no preprocessing, both KMP and Rabin-Karp invest time in preprocessing steps that contribute to their improved efficiency.

2. **Matching Time:** This provides the time complexity for the main search phase of each algorithm, offering insights into their performance characteristics under average and worst-case scenarios.

3. **Comparisons until termination:** This practical metric showcases the number of character comparisons each algorithm performs before finding a match or concluding that no match exists.

For instance, in the example shown in the table in Figure 18, we can see that for the given input:

- The Naive Search, while simple, required 12 comparisons.
- KMP, leveraging its preprocessed LPS table, reduced this to 9 comparisons.
- Rabin-Karp, utilizing its hash-based approach, further reduced the comparisons to just 7.



Algorithm	Preprocessing Time	Matching Time	Comparisons until termination
Naive Search	none	O(n+m) average, O(mn) worst	12
Knuth-Morris-Pratt (KMP)	ϴ(m) (LPS-table)	ϴ(n+m)	9
Rabin-Karp	ϴ(m) (NeedleHash)	ϴ(n) average, ϴ(mn) worst	7

Figure 18: Performance comparison table showcasing preprocessing time, matching time, and comparisons until termination for Naive Search, KMP, and Rabin-Karp algorithms.

This practical demonstration of comparison counts allows users to directly observe how KMP's intelligent shifting and Rabin-Karp's hash-based filtering translate into reduced computational effort. Note that the Aho-Corasick algorithm is not included in this comparison table as it is designed for multi-pattern matching, and its performance metrics are not directly comparable to the single-pattern algorithms discussed here.

4.5 Aho-Corasick Algorithm

The Aho-Corasick algorithm represents a significant advancement in multi-pattern string matching, offering a sophisticated approach to simultaneously search for multiple patterns within a text. In this context, "patterns" refer to the set of strings we are searching for, also sometimes called dictionary in literature. This section covers the algorithm's core principles and operational methodology.

4.5.1 Algorithm Overview

While the previously discussed algorithms focussed on single-pattern matching, many real-world applications require searching for multiple patterns simultaneously. The Aho-Corasick algorithm addresses this need, providing an efficient solution for multi-pattern string matching. It builds upon the concepts we've explored in single-pattern algorithms, particularly the idea of preprocessing to enhance search efficiency.

The Aho-Corasick algorithm significantly improves upon repeated applications of single-pattern algorithms when searching for multiple patterns in a text. Its key innovation lies in constructing a finite state machine from all the search patterns, allowing for simultaneous matching of multiple patterns in a single pass through the text.

Aho-Corasick operates in two main phases:

1. **Preprocessing Phase:** The algorithm constructs a trie (a tree-like data structure) from all input patterns. This trie is then augmented with two types of special links:

   - Failure links, which connect states to the longest proper suffix that is also a prefix of some pattern
   - Output links, which connect states to other patterns that end at that state

   These links form the backbone of the algorithm's efficiency. Failure links allow the algorithm to transition to alternative states when a mismatch occurs, effectively bypassing unnecessary comparisons. Output links connect nodes to other possible patterns, enabling the detection of overlapping patterns without additional passes through the text.

2. **Searching Phase:** Using the preprocessed trie, the algorithm scans the text once. For each character:
   - If a match is found, it moves to the next state in the trie
   - If no match is found, it follows failure links until a match is found or it reaches the root
   - At each state, it reports any patterns that end at that state (following output links if necessary)

The power of Aho-Corasick stems from its ability to match multiple patterns simultaneously and to avoid backtracking in the text. When no match is found for the current character, the algorithm uses failure links to efficiently transition to the appropriate state, ensuring that no potential matches are missed. This approach allows it to find all occurrences of all patterns in a single pass through the text. [4]

4.5.2 Implementation Details

Our implementation of the Aho-Corasick algorithm allows users to engage directly with the algorithm's core mechanics. SMAVIZ is designed to enhance understanding through hands-on interaction with the trie structure. To facilitate a comprehensive learning experience, our implementation focuses on two key aspects of the Aho-Corasick algorithm: the initial setup and the interactive exploration of the algorithm's behavior.

Input and Trie Generation Users begin by entering the patterns and an input string. SMAVIZ then automatically generates and displays the trie structure, as illustrated in Figure 19, based on the provided patterns, complete with failure and output links. This visual representation serves as the foundation for the interactive learning experience.

Failure and Output Links In our visualization, the trie structure is enhanced with failure and output links, which are crucial for understanding the efficiency of the Aho-Corasick algorithm in multi-pattern matching. As shown in Figure 19, black arrows represent child connections, forming the basic structure of the trie. Blue dashed arrows indicate failure links, which are essential for fast mismatch recovery. These links allow the algorithm to quickly transition to the next potential match without backtracking. Green dashed arrows represent output links, connecting nodes to the next node that forms a complete pattern via suffix links. This feature is particularly important for identifying overlapping patterns. The nodes themselves are color-coded: grey nodes represent intermediate points on the way to a pattern, while purple nodes indicate endpoints of the patterns. The legend on the right in Figure 20 provides a detailed explanation of the different elements used in our trie visualization."

Interactive Trie Traversal Once the trie is generated, users can test their understanding of the Aho-Corasick algorithm by simulating its traversal. They are prompted to click on the trie nodes in the order they believe the algorithm would traverse them for the given input string.

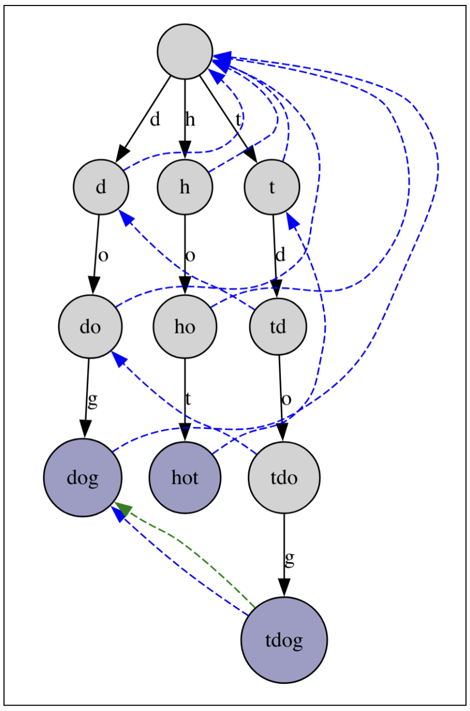


Figure 19: Automatically constructed trie with custom chosen patterns "dog,hot,tdog"

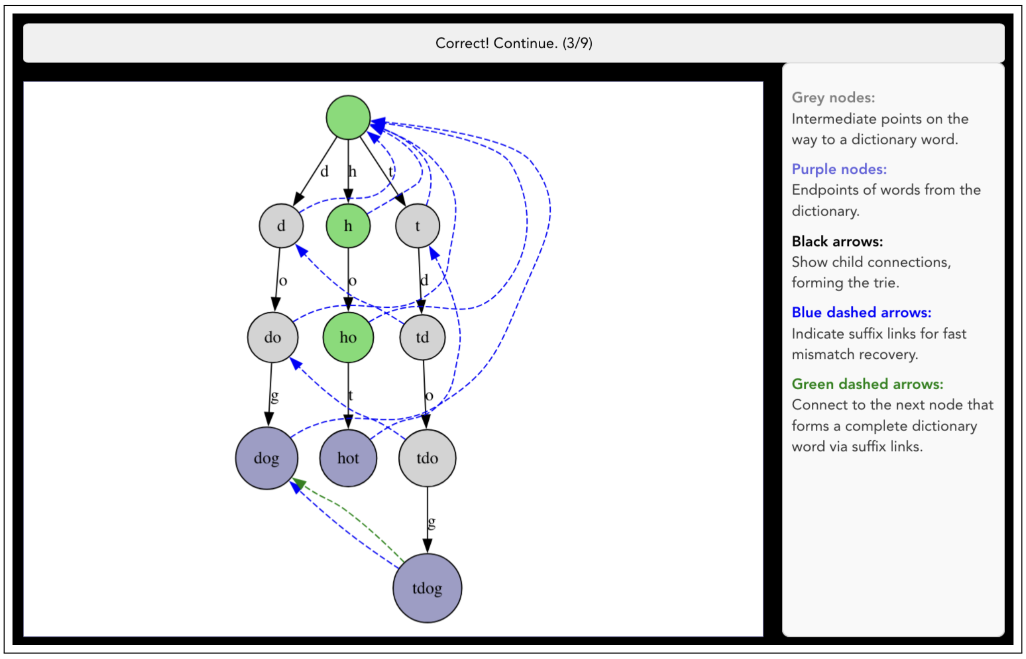
Correct node selections are visually confirmed by turning the node green, providing immediate positive feedback which is illustrated in Figure 20. If an incorrect node is selected, a message "Wrong node. Try again!" is displayed above the trie, encouraging users to reconsider their choice and reinforcing the correct traversal path. If the user successfully traversed the trie a message "Congratulations! You have completed the Aho-Corasick path!" is displayed as shown in Figure 21.

Learning Aids and Options To enhance the learning experience and accommodate different learning styles, SMAVIZ offers several options. Users can continue attempting to find the correct path even after an incorrect selection, promoting a trial-and-error learning approach. A "Reset" option allows users to start the traversal over from the beginning, useful for repeated practice or when exploring different strategies. For those who need additional guidance, a "Show Correct Path" button reveals the solution, displaying the correct traversal order. This feature helps users understand the correct algorithm behavior when they're stuck or want to verify their understanding.

4.5.3 Example Usage

We will now examine a practical example where we will use the very same patterns ("dog,hot,tdog") and input string ("hotdog") as before.

Our mascot Frogbert hints that if we want to generate an automaton we simply need to enter patterns where each word is separated by a comma. The trie is dynamically generated


Correct! Continue. (3/9)

Grey nodes:
Intermediate points on the way to a dictionary word.

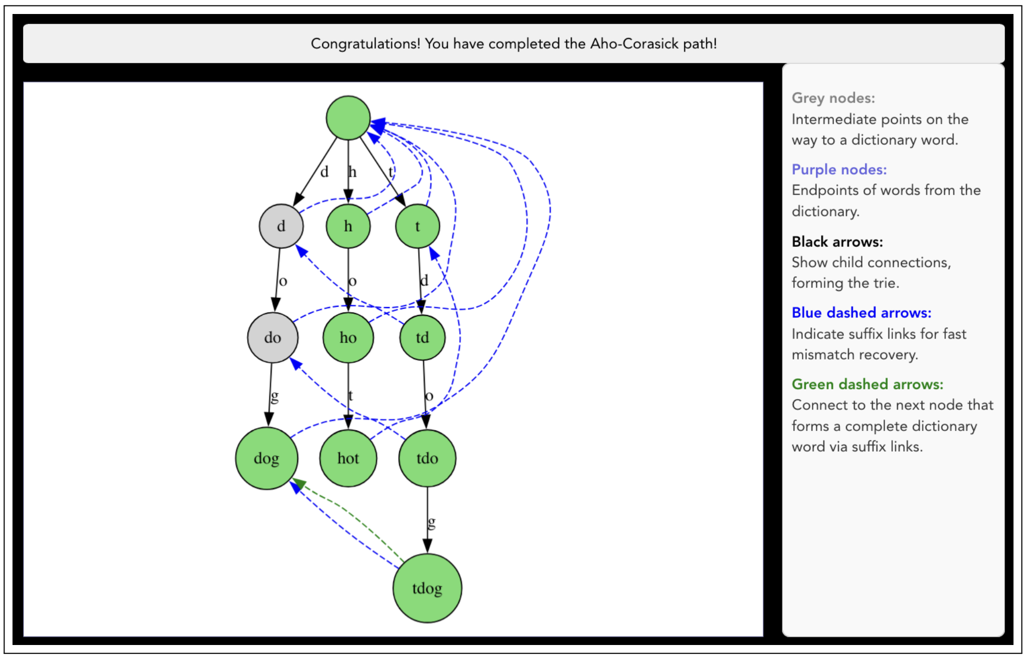
Purple nodes:
Endpoints of words from the dictionary.

Black arrows:
Show child connections, forming the trie.

Blue dashed arrows:
Indicate suffix links for fast mismatch recovery.

Green dashed arrows:
Connect to the next node that forms a complete dictionary word via suffix links.

Figure 20: Manual trie traversal in progress


Congratulations! You have completed the Aho-Corasick path!

Grey nodes:
Intermediate points on the way to a dictionary word.

Purple nodes:
Endpoints of words from the dictionary.

Black arrows:
Show child connections, forming the trie.

Blue dashed arrows:
Indicate suffix links for fast mismatch recovery.

Green dashed arrows:
Connect to the next node that forms a complete dictionary word via suffix links.

Figure 21: Manual trie traversal completed successfully

and updated as the patterns are changed, providing an immediate visual representation of the pattern relationships. Then the user is asked by a comment above "Start clicking the nodes in the correct order!"

The traversal of the trie proceeds as follows:

1. **Initial Match:** The user begins at the root node and clicks through the sequence "h", "ho", and finally "hot". This step identifies the first pattern "hot" in the input string. The nodes turn green as they are correctly selected as seen in Figure 22.

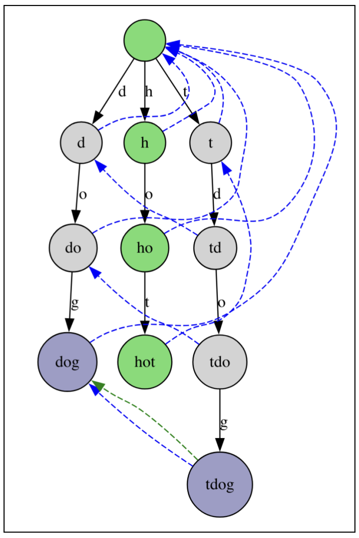


Figure 22: Initial match: Trie traversal highlighting the word 'hot' in green

2. **Failure Transition:** After matching "hot", the next character in the input string is "d". At this point, the algorithm demonstrates one of its key efficiency features. Instead of backtracking or starting over from the root, it utilizes a failure link (represented by a blue dashed arrow) to jump directly to the node containing "t" as we can see in Figure 23. This is important, because if we would start over from the root beginning with the node "d", we would only discover the pattern "dog" and we would miss the word "tdog". This is an example of overlapping pattern detection.

3. **Continuation of Matching:** From the "t" node, the user continues the traversal by clicking through "td", "tdo" and then "tdog". This step identifies the second pattern "tdog" within the input string. Again, the nodes turn green to visually confirm the correct path.

4. **Overlapping Pattern Detection:** In a particularly interesting step, after reaching the "tdog" node, the algorithm doesn't simply terminate. Instead, it leverages an output link (visualized as a green dashed arrow) to jump directly to the "dog" node. This crucial step, illustrated in Figure 25, highlights one of the most powerful features of the Aho-Corasick algorithm: its ability to identify overlapping patterns without any additional passes through the input string.

5. **Completion:** The traversal concludes at the "dog" node, having successfully identified all

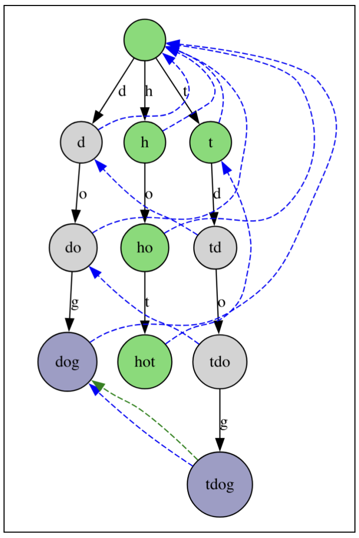


Figure 23: Failure transition: Blue dashed arrow showing the jump from 'hot' to 't' node

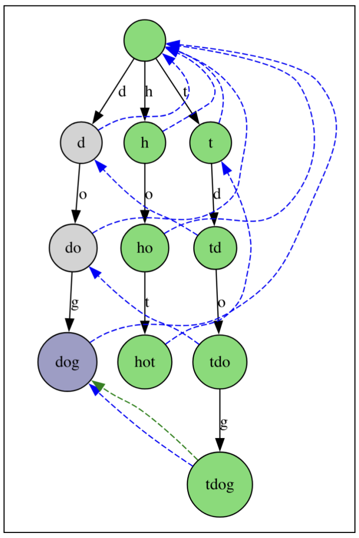


Figure 24: Continuation of matching: Trie traversal highlighting the word 'tdog' in green

occurrences of patterns within the input string "hotdog". SMAVIZ displays a congratulatory message, confirming the correct completion of the Aho-Corasick path.

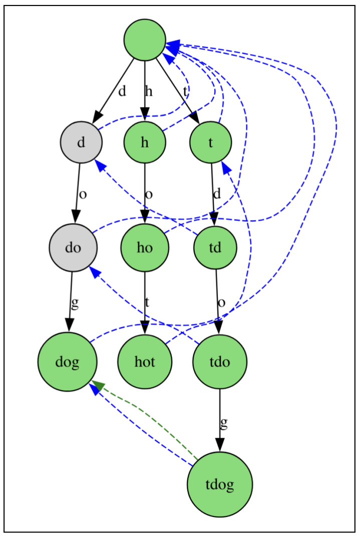


Figure 25: Overlapping pattern detection: Green dashed arrow showing the output link from 'tdog' to 'dog' node

5 Related Work

In the field of algorithm visualization, several tools have been developed to aid in the understanding of string matching algorithms. This section examines a selection of these tools, comparing their features with our implementation. We will highlight unique aspects of each tool, noting functionalities they offer that SMAVIZ does not, as well as features present in SMAVIZ that are absent in others.

5.1 Tool 1: String Matching Visualizer

`http://whocouldthat.be/visualizing-string-matching/`

This section examines *String Matching Visualizer*, a web-based tool that shares some similarities with SMAVIZ, highlighting key differences in design, functionality, and educational approach.

- **Design and User Interface:** *String Matching Visualizer* has a less intuitive design compared to our implementation. For example:
  1. The characters in the string are not numbered with their index
  2. Clicking "Step" performs multiple steps until a mismatch occurs, which can be irritating at first
  3. *Needle* and *Haystack* are not labeled
- **Visualization Control:** Unlike SMAVIZ, the referenced visualizer only offers playback and does not offer options to pause, step forward, or step backward single comparison steps through the algorithm execution. This lack of control can make it challenging for users to study specific steps of the algorithm at their own pace.

- **Speed Adjustment:** SMAVIZ implements a speed control feature, allowing users to adjust the pace of the visualization. *String Matching Visualizer* lacks this functionality, which can make it difficult for users to read and comprehend the step-by-step comments before the visualization moves to the next step.
- **Algorithm Coverage:** While *String Matching Visualizer* covers the Naive, KMP, and Boyer-Moore algorithms, our implementation includes Rabin-Karp and Aho-Corasick in addition to Naive and KMP.
- **Interactive Learning:** While both tools allow for custom string input, our implementation goes further in promoting interactive learning. It features a context-sensitive help system personified by the mascot Frogbert Algorithmicus, providing timely guidance and explanations. Additionally, SMAVIZ incorporates game-like elements that enhance engagement and reinforce understanding:
  - An interactive trie traversal exercise for the Aho-Corasick algorithm, where users can test their understanding by clicking nodes in the correct order.
  - A hands-on learning feature for the KMP algorithm, allowing users to fill in the values of the partial match table and verify their answers.
- **Performance Comparison:** Unlike *String Matching Visualizer*, our implementation includes a performance comparison feature, allowing users to directly compare the efficiency of different algorithms.

5.2 Tool 2: KMP Algorithm Visualizer

`https://cmps-people.ok.ubc.ca/ylucet/DS/KnuthMorrisPratt.html`

Another notable tool in the field is the KMP algorithm visualizer available at `https://cmps-people.ok.ubc.ca/ylucet/DS/KnuthMorrisPratt.html`. When comparing this tool to SMAVIZ, we find several differences:

- **Failure Function Visualization:** *KMP Algorithm Visualizer* offers a detailed visualization of the failure function (partial match table) creation process, which our current implementation does not feature.
- **Highlighting Mechanism:** It employs a direct highlighting approach, providing clear visual cues within the algorithm's steps. For instance, it highlights relevant parts of the partial match table during consultations. This method may be more immediately intuitive for some users compared to our mascot-delivered explanations in speech bubbles.
- **Canvas Adjustment:** Users can adjust the canvas size and reposition controls, allowing for the visualization of longer strings. SMAVIZ currently lacks this flexibility in display configuration.
- **Input Modification:** SMAVIZ offers greater flexibility, allowing users to edit the needle and haystack at any point during the visualization which terminates the simulation and jumps to the initial state using the new user input data. *KMP Algorithm Visualizer* blocks

user input until the user manually or automatically stepped through the algorithm until termination.

- **Interactive Learning:** SMAVIZ incorporates game-like elements such as the Aho-Corasick trie traversal exercise (Page 27) and the KMP partial match table fill-in activity (Page 17), which are not present in *KMP Algorithm Visualizer*.

6 Conclusions and Future Work

This thesis presents an interactive and playful visualization tool for SMAs, addressing the challenges in teaching and learning these fundamental computer science concepts. By focusing on four algorithms - Naive Search, KMP, Rabin-Karp, and Aho-Corasick - SMAVIZ provides a platform for understanding both single-pattern techniques as well as one multi-pattern matching technique.

The implementation of dynamic, step-by-step visualizations, coupled with interactive elements and a context-sensitive help system featuring "Frogbert Algorithmicus," has created an engaging learning environment. This approach bridges the gap between theoretical understanding and practical application, making complex algorithmic concepts more accessible to students.

Key achievements of this work include:

- **Enhanced learning experience:** The ability for users to input custom strings and patterns enables exploration of various scenarios, enhancing the learning process.
- **Improved understanding of algorithmic efficiency:** The implementation of the performance comparison table allows users to directly observe and compare the efficiency of different algorithms. By showcasing preprocessing time, matching time, and the number of comparisons until termination, users can concretely see how advanced algorithms like KMP and Rabin-Karp outperform the naive approach in various scenarios.
- **Integration into computer science curricula:** By aligning SMAVIZ with educational standards and incorporating it into coursework, educators can provide students with a hands-on tool that complements traditional teaching methods. This integration could offer advantages:
  - **For students:** It offers a hands-on, visual approach to understanding complex algorithmic concepts, bridging the gap between theory and practice.
  - **For educators:** It serves as a dynamic teaching aid, allowing for real-time demonstrations of algorithm behavior and performance comparisons, potentially enriching lectures and improving student engagement.

While the current implementation has successfully met its primary objectives as a learning tool, there are several avenues for future work:

- **Expansion of algorithm coverage:** Including additional SMAs, such as Boyer-Moore, could provide an even more comprehensive learning experience.
- **Integration with curriculum:** Developing structured learning modules or exercises that align with computer science curricula could enhance the tool's educational value.
- **Mobile optimization:** Adapting the tool for mobile devices could increase accessibility and allow for learning on-the-go.
- **User study and feedback integration:** Conducting a thorough user study to gather feedback from students and educators could inform future improvements and refinements.

- **Advanced visualization features:** Implementing more advanced visualizations, such as animated state transitions in the Aho-Corasick trie or a visualization of how exactly the partial match table of the KMP algorithm is constructed, could further enhance understanding of complex algorithmic processes.

In conclusion, this visualization tool represents a step forward in making SMAs more approachable and understandable for students. By combining modern web technologies with thoughtful design and interactive features, we have created a resource that aids in learning and serves as an educational tool for algorithm exploration and comparison. We hope it will contribute to more effective computer science education and foster a deeper appreciation for the elegance and efficiency of string matching algorithms.

List of Tables

List of Figures

Listings

References

[1] Nimisha Singla and Deepak Garg. String matching algorithms and their applicability in various applications. *International journal of soft computing and engineering*, 1(6):218–222, 2012.

[2] Chinta Someswararao and K. Raju. Single and multiple pattern string matching algorithm. *Indian Journal of Science and Technology*, 10, 01 2017. doi: 10.17485/ijst/2017/v10i3/84939.

[3] 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. doi: 10.1137/0206024. URL `https://doi.org/10.1137/0206024`.

[4] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, jun 1975. ISSN 0001-0782. doi: 10.1145/360825.360855. URL `https://doi.org/10.1145/360825.360855`.

[5] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, oct 1977. ISSN 0001-0782. doi: 10.1145/359842.359859. URL `https://doi.org/10.1145/359842.359859`.

[6] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, mar 1987. ISSN 0018-8646. doi: 10.1147/rd.312.0249. URL `https://doi.org/10.1147/rd.312.0249`.

[7] Ladan Shams and Aaron R. Seitz. Benefits of Stimulus Congruency for Multisensory Facilitation of Visual Learning. *Current Biology*, 18(15):1091–1095, 2008. doi: 10.1016/j.cub.2008.06.064. URL `https://doi.org/10.1016/j.cub.2008.06.064`.

[8] Vue.js. Introduction - Vue.js, 2023. URL `https://vuejs.org/guide/introduction.html`. Accessed: 14.08.2024.

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

This printed thesis is identical with the electronic version submitted.

Place, Date

Signature