Proceedings of CONF-SEML 2026 Symposium: Computational Analysis and Modeling in Complex Intelligent Systems
DOI: 10.54254/2755-2721/2026.GU33104

# *A Review of String Matching Algorithms with Emphasison Finite Automata*

**Minghao Lu**

*College of Science, The University of Arizona, Tucson, USA*
*aluminghao@arizona.edu*

***Abstract.*** String matching is a fundamental problem in computer science with applications in text processing, bioinformatics, and information retrieval. Over the past decades, a vari- ety of algorithms have been developed; however, selecting an appropriate algorithm for specific applications remains challenging. This paper provides a narrative review of major string matching algorithms, including classical approaches, automata-based methods, bit-parallel techniques, and hashing algorithms, to analyze their performance and applicability. Special attention is given to finite automata, which provide a theoretical framework for various string-matching approaches; however, their limitations, such as high preprocessing cost and memory consumption, are also critically examined. By examining existing literature, this paper identifies key characteristics, strengths, and limitations of different algorithms. The review also compares these approaches from a practical perspective and shows that hybrid algorithms, which combine multiple techniques, often achieve better performance in large-scale data environments. The findings suggest that no single algorithm is optimal for all scenarios, and future research should focus on combining different techniques to improve performance.

***Keywords:*** String Matching, Finite Automata, Pattern Matching, Algorithms, Text Processing

## 1. Introduction

String matching is a fundamental topic in computer science and has long been regarded as an important problem in both theory and practice. In general, string matching refers to the process of finding one or more occurrences of a pattern within a larger text. Although the problem appears simple at first glance, it becomes much more challenging as the data size grows or when matching must be per- formed under strict time constraints. Because of this, string matching has remained an active research area for many years.

The importance of string matching is evident across a wide range of applications. In text processing, it is used to search for words, phrases, or symbols in digital documents. In information retrieval, it supports search engines and indexing systems by enabling the efficient location of relevant content.

In bioinformatics, string matching is used to compare DNA or protein sequences. It is also widely ap- plied in areas such as compiler design, cybersecurity, data mining, and pattern recognition.

© 2026 The Authors. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

DOI: 10.54254/2755-2721/2026.GU33104

Previous studies have noted that string matching plays an important role in applications such as text min- ing, natural language processing, pattern recognition, and information security [1,2], underscoring its value that extends far beyond simple text search.

As digital data continues to expand, the efficiency of pattern matching becomes increasingly important [3]. A straightforward comparison method may work for short texts, but it often performs poorly on large datasets or in real-time tasks. For this reason, researchers have developed many algo- rithms to improve matching speed and reduce unnecessary comparisons. Over time, these algorithms have formed several major categories, such as classical character-comparison methods, automata- based methods, bit-parallel approaches, and hashing techniques. Survey studies have shown that string matching algorithms are commonly organized into these groups because each category reflects a dif- ferent design principle and computational strategy [1,4].

Among these approaches, finite automata deserve special attention. Finite automata provide a the- oretical framework for recognizing patterns in strings and build an important bridge between formal language theory and practical algorithm design. In many cases, a pattern can be represented as an au- tomaton, and the matching process can then be understood as a sequence of state transitions driven by the input text. This idea is especially valuable because it allows pattern matching to be described pre- cisely and systematically. Previous studies in automata theory describe deterministic finite automata as formal models for recognizing strings and regular languages, highlighting their close connection to pattern-matching problems [5].

Therefore, this paper reviews major string-matching algorithms, with an emphasis on the role of finite automata. It first introduces the theoretical background of automata and pattern matching, then examines representative studies on different algorithmic approaches, and finally discusses their strengths, limitations, and future directions. By doing so, this review aims to show not only how string- matching algorithms work but also why finite automata remain an essential concept for understanding this field.

## 2. Theoretical background

String matching is not only a practical problem but also closely related to theoretical concepts in computer science, especially formal language theory and computational models. Among these, finite automata provide one of the most important frameworks for understanding how pattern matching works at a fundamental level. By modeling the matching process as a sequence of state transitions, finite automata provide a formal, systematic way to describe how patterns are recognized in strings.

A finite automaton is a mathematical model used to recognize patterns in strings [6]. It can be formally defined as a 5-tuple

$$M = (Q, \Sigma, \delta, q0, F), \tag{1}$$

where Q is a finite set of states, Σ is the input alphabet, δ is the transition function, $q0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states.

Given an input string $w = a1\ a2 \cdots an \in \Sigma^*$ , the automaton processes the string symbol by symbol according to the transition function δ . At each step, the automaton moves from one state to another based on the current input symbol. If the automaton ends in a state $q \in F$ after reading the entire input, the string is considered a match. This formal representation enables a precise, systematic description of pattern matching.

There are two main types of finite automata: deterministic finite automata (DFA) and nondeterministic finite automata (NFA).

DOI: 10.54254/2755-2721/2026.GU33104

In a DFA, the transition function is defined as

$$\delta : \mathbb{Q} \times \Sigma \to \mathbb{Q}, \tag{2}$$

which means that for each state and input symbol, there is exactly one next state. This makes the automaton's behavior fully predictable.

In contrast, an NFA allows multiple possible transitions for the same input. Its transition function is defined as

$$\delta : \mathbb{Q} \times \Sigma \to 2\mathbb{Q}, \tag{3}$$

where 2Q denotes the power set of Q. This formulation enables greater flexibility in representation, since the automaton can move to a set of possible states rather than a single state.

Although NFAs are more expressive in structure, they can always be converted into equivalent DFAs, which are typically used in practical implementations.

In the context of string matching, a pattern can be represented as a finite automaton. Each state corresponds to the amount of the pattern that has been matched so far.

Given a text T = t1t2 ⋯tn , the automaton processes the text sequentially, reading one character at a time and updating its state accordingly. The sequence of state transitions can be formally described as

$$\mathrm{qi} + 1 = \delta(\mathrm{qi}, \mathrm{ti}), \mathrm{i} = 1, 2, \ldots, \mathrm{n}, \tag{4}$$

where qi denotes the current state after processing the first i — 1 characters.

If at some position the automaton reaches an accepting state q ∈ F, a match is identified. In the case of a mismatch, the automaton transitions to an appropriate state without restarting the entire matching process. This property helps avoid redundant comparisons and improves overall efficiency.

Another key concept in this area is computational complexity. The performance of string-matching algorithms is usually evaluated in terms of time and space complexity.

For automata-based methods, the preprocessing phase, which constructs the automaton, typically requires O(m|Σ|) time, where m is the length of the pattern and |Σ| is the size of the input alphabet. Once the automaton is built, the matching phase processes a text of length n in O(n) time, since each character is examined exactly once.

Efficient algorithms aim to achieve linear-time matching while minimizing additional memory us- age [1]. Automata-based approaches are especially valuable in this regard because of their predictable performance and clear computational structure.

In addition to automata theory, string matching is also related to concepts such as prefix functions, suffix structures, and hashing mechanisms. These theoretical tools offer different approaches to optimizing the matching process and are often combined in modern algorithms.

For example, the prefix function used in the Knuth–Morris–Pratt (KMP) algorithm can be formally defined as

$$\pi(\mathrm{i}) = \max\{\mathrm{k} < \mathrm{i} \mid \mathrm{P}[1 : \mathrm{k}] = \mathrm{P}[\mathrm{i} \longrightarrow \mathrm{k} + 1 : \mathrm{i}]\}, \tag{5}$$

which represents the length of the longest proper prefix of the pattern that is also a suffix. Such formulations help reduce redundant comparisons and improve matching efficiency.

Overall, the theoretical background of string matching highlights the importance of formal models in algorithm design. Finite automata, in particular, provide a clear and powerful framework for

DOI: 10.54254/2755-2721/2026.GU33104

explaining how different string-matching algorithms achieve efficiency and correctness.

## 3. Literature review

Research on string matching algorithms has advanced significantly over the past several decades, moving from simple comparison-based methods to more sophisticated, application-oriented techniques. Early research mainly focused on improving efficiency by reducing unnecessary comparisons, while more recent studies have emphasized scalability, flexibility, and adaptability to different data environments.

### 3.1. Classical string matching algorithms

The earliest approaches to string matching are based on direct character comparison. The brute-force algorithm is a typical example in which the pattern is aligned with the text at every possible position and compared character by character. Although this method is straightforward to implement, its O(mn) time complexity makes it impractical for large-scale applications. As datasets grew larger, researchers recognized the need for more efficient solutions.

A major advancement in this field was the introduction of the Knuth–Morris–Pratt (KMP) algorithm. Instead of restarting the comparison from the beginning after a mismatch, KMP uses information about the pattern itself to determine how far the pattern can be shifted. This is achieved by constructing a prefix function that records the longest proper prefix that is also a suffix. By using this information, the algorithm avoids redundant comparisons and achieves linear time complexity [5]. This idea represents an important step toward more structured algorithm design, where preprocessing plays a key role in improving performance.

Following the development of KMP, the Boyer–Moore (BM) algorithm introduced a different strategy that further improved practical efficiency. Unlike KMP, which scans from left to right, BM compares characters from right to left and uses heuristic rules to skip sections of the text. The two main heuristics, known as the bad-character rule and the good-suffix rule, allow the algorithm to shift the pattern by more than one position in many cases. As a result, BM often performs faster than KMP in real-world applications, particularly when the pattern is relatively long or the alphabet size is large [4].

These classical algorithms laid the foundation for subsequent research, but they also revealed certain limitations. Their performance can vary with input data, and they may not always achieve optimal efficiency under all conditions. This has led researchers to explore alternative approaches grounded in different theoretical principles.

### 3.2. Automata-based algorithms

One such direction is the development of automata-based algorithms. These methods are closely related to formal language theory and use finite automata to represent patterns. In this framework, the pattern is converted into a deterministic finite automaton (DFA), and matching is performed via state transitions. As each character of the text is processed, the automaton transitions between states according to predefined rules. When an accepting state is reached, a match is identified.

Automata-based approaches offer several advantages [1]. One key benefit is that, after preprocess- ing, the matching process runs in linear time, since each character of the text is examined only once. In addition, these methods provide a clear and systematic way to describe pattern matching, making them easier to analyze theoretically. As highlighted in previous studies,

DOI: 10.54254/2755-2721/2026.GU33104

automata-based algorithms constitute an important category of string-matching methods and are often used as a reference point for understanding other approaches.

A well-known example of an automata-based algorithm is the Aho–Corasick algorithm, which is designed for multiple-pattern matching. It constructs a trie structure from a set of patterns and augments it with failure links that allow the automaton to transition efficiently when mismatches occur. During the matching phase, the algorithm processes the text in a single pass while simultaneously checking for all patterns. The preprocessing phase requires O(m) time, where m is the total length of all patterns, and the matching phase runs in O(n + z) time, where n is the length of the text and z is the number of matches.

Another important structure is the Directed Acyclic Word Graph (DAWG), also known as a suffix automaton. Unlike standard finite automata constructed for a single pattern, DAWGs compactly represent all substrings of a given text. This structure enables efficient substring queries and is widely used in advanced string-processing tasks. The construction of a DAWG can be achieved in linear time O(n), and it provides efficient support for various matching operations.

However, automata-based methods also have limitations. The preprocessing step required to construct the automaton may be time-consuming and may require additional memory. This makes them less suitable for applications where patterns change frequently or where memory resources are limited. Despite these challenges, automata-based approaches remain an important part of the literature due to their strong theoretical foundation.

### 3.3. Bit-parallel algorithms

In addition to classical and automata-based approaches, another important direction in string matching research is the development of bit-parallel algorithms. These methods use bitwise operations to simulate multiple comparisons simultaneously, enabling several positions in the text to be processed at once. One representative example is the Shift-Or algorithm, which encodes the pattern into bit masks and updates matching states using bitwise operations. Similarly, the BNDM (Backward Nondetermin- istic DAWG Matching) algorithm improves performance by combining bit-parallelism with backward scanning techniques. These approaches are particularly effective when the pattern is relatively short, as they can fully exploit machine-word operations [2].

Despite their efficiency, bit-parallel algorithms also have inherent limitations. Their performance is closely tied to the machine's word size, which limits the maximum pattern length that can be processed efficiently. When the pattern exceeds this length, additional processing is required, which may reduce their advantage. Nevertheless, these algorithms remain an important contribution to the literature, especially in scenarios where fast matching of short patterns is required.

### 3.4. Hashing-based algorithms

Another widely studied category is hashing-based algorithms. The Rabin–Karp algorithm is a well-known example [7] that uses hash functions to compare substrings instead of performing direct character comparisons. By computing hash values for both the pattern and the text's substrings, the algorithm can quickly identify potential matches. Only when the hash values are equal does it perform a detailed comparison to confirm the match. This significantly reduces the number of comparisons in many cases, making the algorithm suitable for applications involving multiple pattern searches.

However, hashing-based methods introduce the possibility of hash collisions, where different sub- strings produce the same hash value. Although collision handling mechanisms can reduce errors, they may also increase computational overhead. As a result, the effectiveness of hashing-

DOI: 10.54254/2755-2721/2026.GU33104

based algorithms depends heavily on the choice of hash function and the characteristics of the input data.

### 3.5. Hybrid approaches and recent trends

More recent research has shifted toward the classification and systematic analysis of string matching algorithms. Scholars have proposed various taxonomies that group algorithms by their underlying techniques, such as character comparison, automata-based methods, bit-parallel approaches, and hashing. These classifications provide a clearer understanding of how different algorithms relate to one another and help researchers select appropriate methods for specific applications [1].

In addition to classification, there has been increasing interest in hybrid algorithms that combine features from multiple approaches. Instead of relying on a single technique, hybrid methods aim to leverage the strengths of different algorithms while minimizing their weaknesses. For example, some approaches integrate automata-based models with bit-parallel operations to achieve both theoretical efficiency and practical speed. Others combine hashing techniques with filtering strategies to reduce unnecessary comparisons before performing exact matching.

Another important trend in the literature is the growing emphasis on application-driven design. String matching is no longer studied purely as a theoretical problem but as a practical tool that must adapt to different environments. In bioinformatics, for instance, algorithms are designed to handle large-scale DNA sequences and often require multiple-pattern matching capabilities. In cybersecurity, pattern matching is used to detect malicious signatures in network traffic, which requires both speed and accuracy. These applications highlight the need for algorithms that can perform efficiently under different constraints.

Scalability has also become a major concern in recent research. As data sizes continue to grow, traditional algorithms may struggle to remain efficient. This has led to the exploration of parallel computing and hardware-based implementations, such as using GPUs or specialized hardware.

For example, prior studies have proposed GPU-based implementations of string matching algorithms, demonstrating significant improvements in processing speed for large-scale data applications. These approaches exploit the massive parallelism of GPUs to process multiple portions of the text simultaneously, significantly improving throughput compared to CPU-based implementations. In par- ticular, parallel versions of automata-based and bit-parallel algorithms have been shown to achieve substantial speedups in applications such as bioinformatics sequence analysis and network intrusion detection [2].

In addition, hardware-accelerated solutions, including FPGA-based implementations, have been explored to further enhance performance in real-time systems. While these approaches can significantly improve efficiency, they also introduce additional complexity in implementation and resource management and may not always be practical in all settings.

Overall, the literature shows a clear evolution in string matching research. Early work focused on improving basic efficiency through better comparison strategies, while later studies introduced new computational models such as automata and bit-parallelism. More recent research emphasizes flexibility, scalability, and the integration of different techniques. Across all these developments, finite automata remain a central concept, providing both a theoretical framework and a foundation for many modern algorithms.

DOI: 10.54254/2755-2721/2026.GU33104

## 4. Discussion

The review of different string-matching algorithms reveals that each approach reflects a distinct trade- off among efficiency, complexity, and applicability. Rather than identifying a single "best" algorithm, it is more meaningful to compare these methods based on their underlying principles and performance across different scenarios.

Classical algorithms, such as KMP and Boyer–Moore, are widely used for their simplicity and strong theoretical guarantees. KMP ensures linear time complexity by avoiding redundant comparisons, making it reliable for a wide range of inputs. In contrast, the Boyer–Moore algorithm often achieves better practical performance because it can skip large portions of the text. However, the effectiveness of these algorithms can depend heavily on the characteristics of the input, such as pattern length and alphabet size. This variability highlights a key limitation of classical approaches: their performance is not always consistent across different datasets.

Automata-based algorithms provide a more structured and theoretically grounded solution. By representing patterns as finite automata, these methods ensure that each character of the text is processed exactly once after preprocessing. This leads to predictable performance and makes automata-based approaches particularly suitable for real-time applications. Moreover, they offer a clear connec- tion between theory and implementation. However, this advantage comes at the cost of preprocessing overhead and increased memory usage, which may limit their applicability in environments with con- strained resources or where patterns change frequently.

Bit-parallel algorithms offer a different perspective, focusing on hardware efficiency. These meth- ods leverage bitwise operations to perform multiple comparisons simultaneously, resulting in signif- icant speed improvements for short patterns. Compared to classical and automata-based approaches, bit-parallel algorithms are more dependent on the underlying hardware architecture. The machine's word size limits its performance, so it is not always suitable for long patterns. This highlights the trade-off between computational speed and flexibility.

Hashing-based algorithms, such as the Rabin–Karp algorithm, emphasize flexibility and are particularly useful for multiple-pattern matching. By comparing hash values instead of characters, they can reduce the number of direct comparisons. However, the possibility of hash collisions introduces uncertainty, requiring additional verification steps. As a result, these algorithms are often used in scenarios where approximate filtering is acceptable before exact matching.

A more quantitative comparison further highlights the differences among these approaches. Classical algorithms such as KMP achieve a worst-case time complexity of $O(n)$ with minimal additional space. In contrast, Boyer–Moore often achieves sublinear average-case performance due to its skipping heuristics. Automata-based algorithms require $O(m|\Sigma|)$ preprocessing time and space, where $m$ is the pattern length and $\Sigma$ is the alphabet, but guarantee $O(n)$ matching time. Bit-parallel algorithms can process multiple characters simultaneously and often achieve $O(n)$ time with very low constant factors when the pattern length fits within a machine word. In contrast, hashing-based algorithms such as Rabin–Karp have an average-case time complexity of $O(n)$ but may degrade to $O(nm)$ in the worst case due to hash collisions. These differences demonstrate that algorithm performance is shaped not only by asymptotic complexity but also by practical factors such as input characteristics and hardware constraints.

A key observation from this comparison is that no single approach performs optimally in all situations. Instead, an algorithm's effectiveness depends on the specific requirements of the application. For example, Boyer–Moore may be preferred for long patterns in large texts, while bit-parallel algorithms are more suitable for short patterns. Automata-based methods are advantageous when

DOI: 10.54254/2755-2721/2026.GU33104

pre- dictable performance is required, while hashing methods are useful for handling multiple patterns.

Recent research suggests that combining techniques can improve performance and adaptability [1,2]. Hybrid algorithms that integrate automata-based models with bit-parallel techniques or combine hashing with filtering strategies have shown promising results. These approaches aim to balance efficiency, accuracy, and flexibility, making them suitable for modern applications.

Another important trend is the increasing focus on scalability. With the growth of big data, string matching algorithms must handle large datasets efficiently. This has led to the exploration of parallel computing and distributed systems, though these approaches introduce new challenges in implementation complexity and resource management.

Overall, the discussion highlights that string matching is not a one-size-fits-all problem. Understanding the strengths and limitations of different algorithms is essential for selecting appropriate methods and designing more efficient solutions. Finite automata remain a unifying concept that helps explain and connect many of these approaches.

## 5. Conclusion

This paper reviewed the development of string-matching algorithms, including classical methods, automata-based approaches, bit-parallel techniques, and hashing algorithms. These categories reflect different design philosophies and offer distinct advantages in efficiency and applicability.

Among them, finite automata provide a clear theoretical framework for pattern matching, linking formal language theory with practical algorithm design. By modeling patterns as state transitions, automata-based methods enable systematic and efficient matching. Although they require additional preprocessing and memory, their predictable performance makes them valuable in many applications.

The comparison shows that no single algorithm is optimal for all scenarios; instead, performance depends on factors such as pattern length, text size, and application requirements. Classical methods remain widely used for their simplicity, while bit-parallel and hashing techniques perform well in specific contexts, and automata-based approaches balance theory and efficiency.

Future research may focus on hybrid algorithms and improving scalability for large-scale data processing, with parallel and distributed computing as potential directions. Overall, string matching remains an important research area, and understanding different approaches, particularly finite automata, is essential for developing more efficient solutions [3].

## References

[1] Hakak, S. I., Kamsin, A., Shivakumara, P., Gilkar, G. A., Khan, W. Z., & Imran, M. (2019). Exact string matching algorithms: Survey, issues, and future research directions. IEEE Access, 7, 69614–69638. https: //doi.org/10.1109/ACCESS.2019.2914071

[2] Faro, S., & Lecroq, T. (2013). The exact online string matching problem: A review of the most recent results. ACM Computing Surveys, 45(2), 1–42. https: //doi.org/10.1145/2431211.2431212

[3] Ren, J., Xia, F., Chen, X., Liu, J., Hou, M., Shehzad, A., Sultanova, N., & Kong, X. (2021). Match- ing algorithms: Fundamentals, applications and challenges. IEEE Transactions on Emerging Top- ics in Computational Intelligence, 5(3), 332–350. https: //doi.org/10.1109/TETCI.2021.3067655

[4] Navarro, G. (2001). A guided tour to approximate string matching. ACM Computing Surveys, 33(1), 31–88. https: //doi.org/10.1145/375360.375365

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.

[6] Crochemore, M., & Rytter, W. (2002). Jewels of stringology: Text algorithms. World Scientific Publishing. https: //doi.org/10.1142/4838

DOI: 10.54254/2755-2721/2026.GU33104

[7] Rabin, M. O., & Karp, R. M. (1987). Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, 31(2), 249–260.