Efficient Matching of Some Fundamental Regular Expressions with Backreferences

Taisei Nogami
Waseda University, Tokyo, Japan

Tachio Terauchi
Waseda University, Tokyo, Japan

Abstract

Regular expression matching is of practical importance due to its widespread use in real-world applications. In practical use, regular expressions are often used with real-world extensions. Accordingly, the matching problem of regular expressions with real-world extensions has been actively studied in recent years, yielding steady progress. However, *backreference*, a popular extension supported by most modern programming languages such as Java, Python, JavaScript and others in their standard libraries for string processing, is an exception to this positive trend. In fact, it is known that the matching problem of regular expressions with backreferences (rewbs) is theoretically hard and the existence of an asymptotically fast matching algorithm for arbitrary rewbs seems unlikely. Even among currently known partial solutions, the balance between efficiency and generality remains unsatisfactory. To bridge this gap, we present an efficient matching algorithm for rewbs of the form $e_0(e)_1e_1\backslash 1e_2$ where e_0, e, e_1, e_2 are pure regular expressions, which are fundamental and frequently used in practical applications. It runs in quadratic time with respect to the input string length, substantially improving the best-known cubic time complexity for these rewbs. Our algorithm combines ideas from both stringology and automata theory in a novel way. We leverage two techniques from automata theory, *injection* and *summarization*, to simultaneously examine matches whose backreferenced substrings are either a fixed *right-maximal repeat* or its *extendable prefixes*, which are concepts from stringology. By further utilizing a subtle property of extendable prefixes, our algorithm correctly decides the matching problem while achieving the quadratic-time complexity.

2012 ACM Subject Classification Theory of computation → Regular languages; Theory of computation → Pattern matching

Keywords and phrases Regular expressions, Backreferences, Regex matching, NFA simulation, Suffix arrays, Right-maximal repeats

Digital Object Identifier 10.4230/LIPIcs.MFCS.2025.81

Related Version *Full Version*: `https://arxiv.org/abs/2504.18247` [36]

Supplementary Material *Software (Analysis Script)*: `https://github.com/nogamita/MFCS2025` [34]

Funding This work was supported by JSPS KAKENHI Grant Numbers JP25KJ2137, JP23K24826, and JP20K20625.

1 Introduction

A regular expression is a convenient way to specify a language (i.e., a set of strings) using concatenation ($\cdot$), disjunction ($|$) and iteration ($*$). The *regular expression matching problem* (also known as *regular expression membership testing*) asks whether a given string belongs to the language of a given regular expression. This problem is of practical importance due to its widespread use in real-world applications, particularly in format validation and pattern searching. In 1968, Thompson presented a solution to this problem that runs in $O(nm)$ time where n denotes the length of the input string and m the length of the input regular expression [46]. We refer to his method, which constructs a nondeterministic finite automaton (NFA) and simulates it, as *NFA simulation*.

The matching problem of real-world regular expressions becomes increasingly complex due to its practical extensions such as *lookarounds* and *backreferences*. Unfortunately, reducing the matching of real-world regular expressions to that of pure ones is either inefficient or impossible. In fact, although adding positive lookaheads (a type of lookaround) does not increase the expressive power of regular expressions [31, 7], the corresponding NFAs can inevitably become enormous [30]. Worse still, adding backreferences makes regular expressions strictly more expressive, meaning that an equivalent NFA may not even exist.[1]

Nevertheless, most modern programming languages, such as Java, Python, JavaScript and more, support lookarounds and backreferences in their standard libraries for string processing. The most widely used implementation for the real-world regular expression matching is *backtracking* [44], an algorithm that is easy to implement and extend. On the other hand, the backtracking implementation suffers from a major drawback in that it takes exponential time in the worst case with respect to the input string length. This exponential-time behavior poses the risk of *ReDoS (regular expression denial of service)*, a type of DoS attack that exploits heavy regular expression matching to cause service downtime, making it a critical security issue (refer to Davis et al. [14] for details on its history and case studies). In response, RE2, a regular expression engine developed by Google, has deferred supporting lookarounds and backreferences, thereby ensuring $O(nm)$ time complexity using NFA simulation.[2]

Regarding lookarounds, several recent papers have proposed groundbreaking $O(nm)$-time solutions to the matching problem of regular expressions with lookarounds [29, 21, 5], yet regarding backreferences, the outlook is bleak. The matching problem of regular expressions with backreferences (rewbs for short) is well known for its theoretical difficulties. Aho showed that the rewb matching problem is NP-complete [2]. Moreover, rewbs can be considered a generalization of Angluin's *pattern languages* (also known as *patterns with variables*) [3]; even when restricted to this, its matching problem is NP-complete with respect to the lengths of both a given string and a given pattern [3, 16, 40], and its NP-hardness [18] as well as W[1]-hardness [19] are known for certain fixed parameter settings. The best-known matching algorithm for rewbs with at most k capturing groups runs in $O(n^{2k+2}m)$ time [41, 42] (With a slight modification, the time complexity can be reduced in $O(n^{2k+1}m)$ time; see Section 4).

Therefore, the existence of an efficient worst-case time complexity algorithm that works for any rewb seems unlikely, necessitating researchers to explore efficient algorithms that work for some subset of rewbs [39, 42, 20]. Nevertheless, all existing solutions either have high worst-case time complexity or impose non-trivial constraints on the input rewbs, and finding a good balance between efficiency and generality remains an open issue.

To bridge this gap, we present an efficient matching algorithm for rewbs of the form $e_0(e)_1e_1\backslash 1e_2$, where e_0, e, e_1, e_2 are pure regular expressions, which are fundamental and frequently encountered in practical applications. While the best-known algorithm for these rewbs is the one stated above and it takes $O(n^4m)$ (or $O(n^3m)$) time because $k = 1$ for these rewbs, our algorithm runs in $O(n^2m^2)$ time, improving the best-known time complexity for these rewbs with respect to the input string length n from cubic to quadratic. The key appeal of this improvement lies in the replacement of the input string length n with the expression length m. Because n is typically much larger than m, this improvement is considerable.

These rewbs are of both practical and theoretical interest. From a practical perspective, these rewbs account for a large proportion of the actual usage of backreferences. In fact, we have confirmed that, among the dataset collected in a large-scale empirical study conducted by Davis et al. [14], which consists of real-world regular expressions used in npm and PyPI projects, approximately 57% (1,659/2,909) of the non-pure rewbs[3] are of this form [34].

[1] The rewb $((a|b)^*)_1\backslash 1$ specifies $\{ww \mid w \in \{a, b\}^*\}$, which is non-context-free (and therefore non-regular).
[2] The development team declares, "Safety is RE2's raison d'être." [47]
[3] Non-pure rewbs are rewbs that use backreferences. Note that rewbs, in general, also include ones that

Additionally, the matching problem of rewbs of this form is a natural generalization of a well-studied foundational problem, making it theoretically interesting. A *square* (also known as *tandem repeat*) is a string $\alpha\alpha$ formed by juxtaposing the same string α. The problem of deciding whether a given string contains a square is of interest in stringology, and has been well studied [28, 12]. The rewb matching considered in this paper can be viewed as a generalization of the problem by regular expressions.[4]

We now provide an overview of our new algorithm. A novel aspect of the algorithm is that, *unlike previous algorithms for rewb matching or square finding, it combines ideas from both stringology and automata theory.* Our algorithm utilizes the *suffix array* of the input string to efficiently enumerate candidates for backreferenced substrings (i.e., the contents of $\backslash 1$). Once a candidate α is fixed, the matches whose backreferenced substring is α can be examined in $O(nm)$ time in almost the same way as NFA simulation. Because the number of candidate substrings is $\Theta(n^2)$, this gives an $O(n^3m)$-time algorithm (see Remark 11 for details). Next, we improve this baseline algorithm by extending the NFA simulation to simultaneously examine all matches whose backreferenced substrings are either a *right-maximal repeat* or its *extendable prefixes*, instead of examining each candidate individually. Because the new NFA simulation requires $O(m^2)$ time at each step and the number of right-maximal repeats is at most $n-1$, our algorithm runs in $O(n^2m^2)$ time.

A key challenge is how to do all the examinations within time linear in n for each fixed right-maximal repeat α. To address this, we incorporate two techniques from automata theory, *injection* and *summarization*. Each of these techniques is fairly standard on its own (see Section 4 for their applications in prior work), but the idea of combining them is, to our knowledge, novel. Additionally, we leverage a subtle property of α-extendable prefixes to do the examinations correctly within time linear in n even when occurrences of α may overlap (see Section 3.2).

The rest of the paper is organized as follows. Section 2 defines the key concepts in this paper, namely NFA simulation and right-maximal repeats. Section 3 presents our algorithm, which is the main contribution of this paper. Section 4 discusses related work and Section 5 presents the conclusion and future work. Omitted proofs are available in the full version [36].

2 Preliminaries

Let Σ be a set called an *alphabet*, whose elements are called *characters*. A *string* w is a finite sequence $a_1 \cdots a_n$ of characters $a_1, \ldots, a_n$, and we write $|w|$ for the number n of characters in the sequence. The empty string is written as ε. For integers $i, j \geq 0$, we write $[i, j]$ for the set of integers between i and j. For $i, j \in [1, n]$, we write $w[i..j]$ for the substring $a_i \cdots a_j$. In particular, (1) $w[i] := w[i..i] = a_i$ is called the *character at position* i, (2) $w[..i] := w[1..i]$ is called the *prefix* up to position i and (3) $w[i..] := w[i..n]$ is called the *suffix* from position i. We regard $w[i+1..i]$ as ε. A *regular expression* e and its language denoted by $L(e)$ are defined in the standard way.

First, we define the matching problem for rewbs of the form mentioned earlier.

▶ **Definition 1.** Given regular expressions e_0, e, e_1, e_2, the language of rewb $r = e_0(e)_1 e_1 \backslash 1 e_2$, denoted by $L(r)$, is $\{w_0 \alpha w_1 \alpha w_2 \mid w_i \in L(e_i)(i = 0, 1, 2), \alpha \in L(e)\}$.

do not use backreferences (i.e., pure regular expressions).

[4] The problem is an instance of our rewb matching problem where $e_0, e, e_2 = \Sigma^*$ and $e_1 = \varepsilon$.

A regular expression e *matches* a string w if $w \in L(e)$. The *matching problem* for regular expressions is defined to be the problem of deciding whether a given regular expression matches a given string, and similarly for rewbs (of the form considered in this paper).

▶ Remark 2. Note that in full, rewbs may use a *capturing group* $(r)_i$ to assign a label i to a string that the captured subexpression r matches and a *reference* $\backslash i$ to denote the expression that matches only the string labeled i. Therefore, rewbs are capable of more versatile expressions, such as using reference more than once (e.g., $(a^*)_1\backslash 1\backslash 1$) or using multiple capturing groups (e.g., $(a^*)_1(b^*)_2\backslash 1\backslash 2$)). For the full syntax and semantics of rewbs, refer to [20]. We refer to [6, 35, 37] for studies on their expressive power.

Next, we review a classical solution of the regular expression matching.

▶ **Definition 3.** A *nondeterministic finite automaton* (NFA) N is a tuple (Q, δ, q_0, F) where Q is a finite set of *states*, $\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is a *transition relation*, $q_0 \in Q$ is an *initial state*, $F \subseteq Q$ is a set of *accept states*.

The transitive closure of a transition relation δ with the second argument fixed at ε is called *ε-closure operator* and written as cl_ε. Further, we lift $\mathrm{cl}_\varepsilon\colon Q \to \mathcal{P}(Q)$ to $\mathcal{P}(Q) \to \mathcal{P}(Q)$ by taking unions, i.e., $\mathrm{cl}_\varepsilon(S) := \bigcup_{q\in S} \mathrm{cl}_\varepsilon(q)$. For a state q and a character a, we define $\Delta(q, a)$ as $\mathrm{cl}_\varepsilon(\delta(q, a))$, which informally consists of states reachable by repeating ε-moves from states that are reachable from q by a. As before, we extend Δ by setting $\Delta(S, a) := \bigcup_{q\in S} \Delta(q, a)$. Also, for a string w, we define $\Delta(S, w)$ as $\Delta(S, \varepsilon) := S$ and $\Delta(S, wa) := \Delta(\Delta(S, w), a)$. Thus, the *language* $L(N)$ of an NFA N is the set of strings w such that $\Delta(\mathrm{cl}_\varepsilon(q_0), w) \cap F \neq \emptyset$.

The *NFA simulation* of N on a string w of length n is the following procedure for calculating $\Delta(\mathrm{cl}_\varepsilon(q_0), w)$ [46]. First, $S^{(0)} := \mathrm{cl}_\varepsilon(q_0)$ is calculated, and then $S^{(i)} := \Delta(S^{(i-1)}, w[i])$, called the *simulation set at position i*, is sequentially computed from $S^{(i-1)}$ for each $i \in [1, n]$. We have $w \in L(N) \iff S^{(n)} \cap F \neq \emptyset$.

This gives a solution to the regular expression matching in $O(nm)$ time and $O(m)$ space where n is the length of the input string w and m that of the input regular expression e. First, convert e to an equivalent NFA N_e whose number of states and transitions are both $O(m)$ using the standard construction, then run the NFA simulation of N_e on w. Finally, check whether the last simulation set $S^{(n)}$ contains any accept state of N_e. Each step of the NFA simulation can be done in $O(m)$ time using breadth-first search. Also, the procedure can be implemented in $O(m)$ space by reusing the same memory for each $S^{(i)}$.

▶ Remark 4. We call *acceptance testing* the disjointness testing of a simulation set S and a set F of accept states, as performed above. An acceptance test *succeeds* if $S \cap F \neq \emptyset$.

Acceptance testing of an intermediate simulation set is also meaningful. We can check whether e matches each prefix $w[..i]$ with the same asymptotic complexity by testing at each position i. The same can be done for the suffixes by reversing e and w.

Next, we review *right-maximal repeats* and *extendable prefixes*. Let w be a string. A nonempty string α *occurs* at position i in w if $w[i..i + |\alpha| - 1] = \alpha$. Two distinct occurrences of α at positions $i < j$ *overlap* if $j < i + |\alpha|$. A *repeat* of w is a nonempty string that occurs in w more than once. A repeat α is called *right-maximal* if α occurs at two distinct positions i, j such that the right-adjacent characters are different (i.e., $w[i + |\alpha|] \neq w[j + |\alpha|]$). In the special case when α occurs at the right end of w, we consider it to have a right-adjacent character distinct from all other characters when defining right-maximality. For example, the right-maximal repeats of `mississimiss` are `i`, `iss`, `issi`, `miss`, `s`, `si`, `ss` and `ssi` (`miss`, `iss`, `ss` are due to the special treatment).[5] In general, the number of repeats of a string of length n is $\Theta(n^2)$, while that of right-maximal repeats is at most $n - 1$ [23].

[5] This shows that, contrary to what the word suggests, a right-maximal repeat may contain another right-maximal repeat. In fact, the repeat `iss` and its proper substrings `i`, `s` and `ss` are all right-maximal.

For any non-right-maximal repeat α, all the right-adjacent positions of the occurrences of α have the same character. By repeatedly extending α with this, we will obtain the right-maximal repeat, denoted by $\overrightarrow{\alpha}$. This notation $\overrightarrow{\alpha}$ follows that of [25, 45, 33]. We define $\overrightarrow{\alpha} := \alpha$ for a right-maximal repeat α. A prefix β of α such that $\overrightarrow{\beta} = \alpha$ is called *α-extendable prefix* of α. Note that α itself is α-extendable. Two distinct occurrences of an α-extendable prefix β are *α-separable* in w if the two α's extended from the β's have no overlap.

Finally, we mention the enumeration subroutine used in our algorithm. Abouelhoda et al. presented an $O(n)$-time enumeration algorithm for right-maximal repeats [1]. By slightly modifying this, we obtain an $O(n^2)$-time algorithm EnumRM that enumerates each right-maximal repeat α with the sorted array Idx_α of all starting positions of the occurrences of α (see the full version [36] for details). We call Idx_α the *occurrence array* of α in w.

3 Efficient Matching

In this section, we show an efficient matching algorithm for rewbs of the form $e_0(e)_1e_1\backslash 1e_2$, which is the main contribution of this paper.

▶ **Theorem 5.** *The matching problem for rewbs of the form $e_0(e)_1e_1\backslash 1e_2$, where e_0, e, e_1, e_2 are regular expressions, can be solved in $O(n^2m^2)$ time and $O(n + m^2)$ space. Here, n denotes the length of the input string and m that of the input rewb. More precisely, let $m_{e_0}, m_e, m_{e_1}, m_{e_2}$ denote the length of e_0, e, e_1, e_2 respectively, and μ_{right} the number of right-maximal repeats of the input string, which is at most $n-1$. Then, the problem can be solved in $O(n^2 + n(m_{e_0} + m_{e_2}) + \mu_{\mathrm{right}} n(m_e + m_{e_1}^2))$ time and $O(n + \max\{m_{e_0}, m_e, m_{e_2}, m_{e_1}^2\})$ space.*

Let r be a rewb $e_0(e)_1e_1\backslash 1e_2$ and w be a string. The overview of our matching algorithm for r and w is as follows. We assume without loss of generality that e does not match ε because we can check if e matches ε and $e_0e_1e_2$ matches w in $O(nm)$ time by ordinary NFA simulation. We show an $O(n(m_e + m_{e_1}^2))$-time and $O(n + m_e + m_{e_1}^2)$-space subprocedure $\textsc{Match}(\alpha, \mathsf{Idx}_\alpha)$ ($\textsc{Match}(\alpha)$ for short) that takes a right-maximal repeat α and its occurrence array Idx_α, and simultaneously examines all matches whose backreferenced substrings (i.e., the contents of $\backslash 1$) are α-extendable prefixes of α. Before defining Match, we state the property that characterizes its correctness:

▶ **Lemma 6** (Correctness of Match)**.** *Let α be a right-maximal repeat of w. There exists a (not necessarily α-extendable) prefix β of α such that e matches β and $e_0\beta e_1\beta e_2$ matches w if* $\textsc{Match}(\alpha)$ *returns* **true**. *Conversely,* $\textsc{Match}(\alpha)$ *returns* **true** *if there exists an α-extendable prefix β of α such that e matches β and $e_0\beta e_1\beta e_2$ matches w.*

▶ Remark 7. Note that, interestingly, the correctness is *incomplete on its own.* That is, when there is a prefix β of α such that a match with β as the backreferenced substring exists, $\textsc{Match}(\alpha)$ is guaranteed to return **true** if β is α-extendable, but it can return **false** if β is not α-extendable. Still, the correctness of the overall algorithm Main described below holds because it runs Match on *every right-maximal repeat*, and a match with a non-α-extendable prefix β is guaranteed to be reported by another execution of Match (namely, by $\textsc{Match}(\overrightarrow{\beta})$). Formally, see the proof of Theorem 5 described below.

Given this, the overall matching algorithm Main is constructed as follows. It first constructs an NFA N_{e_1} equivalent to the middle subexpression e_1 and two Boolean arrays Pre and Suf, which are necessary for Match. A Boolean array Pre (resp. Suf) is the array which stores whether each prefix (resp. suffix) of w is matched by e_0 (resp. e_2). More precisely, Pre and Suf are the arrays such that $\mathsf{Pre}[i] = \textbf{true} \iff w[..i] \in L(e_0)$ for $i \in [0, n]$ and

$\mathsf{Suf}[j] = \mathbf{true} \iff w[j..] \in L(e_2)$ for $j \in [1, n+1]$. Note that we can construct these arrays in $O(n(m_{e_0} + m_{e_2}))$ time and $O(n + \max\{m_{e_0}, m_{e_2}\})$ space as mentioned in Remark 4. Then, it runs the $O(n^2)$-time and $O(n)$-space enumeration algorithm EnumRM from the final paragraph of the previous section. Each time EnumRM outputs α and Idx_α, Match is executed with these as input and using N_{e_1}, Pre and Suf. Main returns **true** if Match(α) returns **true** for some α; otherwise, it returns **false**.

Proof of Theorem 5. It suffices to show the correctness of Main, namely Main returns **true** if and only if r matches w. If w has a match for r, the (nonempty) backreferenced substring is a repeat β of w that e matches. Then, Match($\overrightarrow{\beta}$) returns **true** by Lemma 6. The converse also follows from the lemma. ◀

In what follows, we present the detailed behavior of the subroutine Match, incrementally progressing from simple cases to more complex ones.

3.1 The Case of Nonoverlapping Right-Maximal Repeats

Let α be a fixed right-maximal repeat of the input string w. In this subsection, for simplicity, we consistently assume that *no occurrences of α overlap with each other*. We call such an α *nonoverlapping right-maximal repeat*. We first introduce in Section 3.1.1 a way to examine matches whose backreferenced substring is α itself, namely NFA simulation with auxiliary arrays and a technique called *injection*. Then, in Section 3.1.2, we extend it to simultaneously examine all matches whose backreferenced substrings are α-extendable prefixes of α, instead of examining α individually. There, in addition to injection, we use a technique called *summarization*.[6]

3.1.1 Only the right-maximal repeat itself

Let r_α denote the (pure) regular expression $e_0 \alpha e_1 \alpha e_2$. We give an $O(n(m_e + m_{e_1}))$-time algorithm Match1(α) that checks whether r_α matches w. It runs the NFA simulation of N_{e_1} using the arrays Pre, Suf and Idx_α as oracles. We begin by explaining the injection technique. It is grounded on the following property:

▶ **Lemma 8.** *For any sets of states S and T, and strings u and v, we have $\Delta(S, uv) = \Delta(\Delta(S, u), v)$ and $\Delta(S, u) \cup \Delta(T, u) = \Delta(S \cup T, u)$.*

This implies the following equation:

$$\begin{aligned}\Delta(\mathrm{cl}_\varepsilon(q_0), uv) \cup \Delta(\mathrm{cl}_\varepsilon(q_0), v) &= \Delta(\Delta(\mathrm{cl}_\varepsilon(q_0), u), v) \cup \Delta(\mathrm{cl}_\varepsilon(q_0), v) \\ &= \Delta(\Delta(\mathrm{cl}_\varepsilon(q_0), u) \cup \mathrm{cl}_\varepsilon(q_0), v).\end{aligned}$$

Therefore, we can simultaneously check whether e matches *either a string uv or its suffix v* as follows: in the NFA simulation on uv, when u has been processed, replace the current simulation set $\Delta(\mathrm{cl}_\varepsilon(q_0), u)$ with the union of it and $\mathrm{cl}_\varepsilon(q_0)$, and then continue with the remaining simulation on v. We call this replacement *injection*. More generally, for any positions $i_1 < i_2 < \cdots < i_l \le j$, we can check whether e matches any of $w[i_1..j], \ldots, w[i_l..j]$ by testing the injected simulation set immediately after the character $w[j]$, namely $\Delta(\cdots \Delta(\Delta(\mathrm{cl}_\varepsilon(q_0), w[i_1..i_2-1]) \cup \mathrm{cl}_\varepsilon(q_0), w[i_2..i_3-1]) \cup \mathrm{cl}_\varepsilon(q_0) \cdots, w[i_l..j])$.

[6] As noted in the introduction, these techniques are fairly standard on their own and often used without being given names (see Section 4 for details), but our uses of them are novel and we give them explicit names to clarify how and where they are used in our algorithm.

Algorithm 1 MATCH1(α).

Correctness: See Lemma 10.

```
1  i_prev ← ⊥;  S ← ∅;  i_que ← ⊥;  (Δ, cl_ε(q_0), F) ← N_{e_1}
2  if e does not match α then return false
3  for i_next ∈ Idx_α do
4      if i_que ≠ ⊥ then
5          for i ← i_prev to i_next − 1 do
6              if S ≠ ∅ then S ← Δ(S, w[i])
7              if i = i_que then S ← S ∪ cl_ε(q_0)                    /* Injection */
8          if S ∩ F ≠ ∅ and Suf[i_next + |α|] then return true
9      i_prev ← i_next
10     if Pre[i_prev − 1] then i_que ← i_prev + |α| − 1
11 return false
```

We now explain MATCH1(α) whose pseudocode is shown in Algorithm 1. First, it checks if e matches α and returns **false** if it is false; otherwise, the algorithm continues running. Let $i_1 < i_2 < \cdots$ be the positions in Idx_α. Then, it searches for the starting position i_{j_1} of the leftmost occurrence of α such that e_0 matches the prefix $w[..i_{j_1} - 1]$ to the left of the α by looking at $\mathsf{Pre}[i_j - 1]$ sequentially for each $i_j \in \mathsf{Idx}_\alpha$ (lines 3, 9 and 10). Remark that the α at position i_{j_1} is the leftmost candidate that the left α of r_α may correspond to in a match.

If i_{j_1} is found, it starts an NFA simulation of N_{e_1} from the position $i_{j_1} + |\alpha|$ immediately to the right of the α at position i_{j_1} by setting S to be $\mathrm{cl}_\varepsilon(q_0)$ immediately after the character $w[i_{j_1} + |\alpha| - 1]$ (line 7). Note that S is $\emptyset$ prior to the assignment. In what follows, let $i_{j_2}, i_{j_3}, \ldots$ denote the starting positions of the α's to the right of the α at position i_{j_1} in sequence. Note that, unlike in the case of i_{j_1}, we do not assume that e_0 matches the prefix $w[..i_{j_{k-1}}]$ for i_{j_k}, i.e., $j_k = j_1 + k - 1$ ($k \geq 2$).

Next, the algorithm resumes the simulation and proceeds until the character $w[i_{j_2} - 1]$ (lines 5 and 6). Then, it performs the acceptance testing $S \cap F \neq \emptyset$ (line 8). If it succeeds, $e_0 \alpha e_1 \alpha$ matches $w[..i_{j_2} + |\alpha| - 1]$ by matching the two α's to $w[i_{j_1}..i_{j_1} + |\alpha| - 1]$ and $w[i_{j_2}..i_{j_2} + |\alpha| - 1]$. Accordingly, the algorithm further checks whether e_2 matches the remaining suffix $w[i_{j_2} + |\alpha|..]$ by looking at $\mathsf{Suf}[i_{j_2} + |\alpha|]$. If it is **true**, r_α matches w and the algorithm returns **true**. Otherwise, there is no possibility that r_α matches w in a way that the right α of r_α matches the α at position i_{j_2}, and the algorithm continues running.

In this way, the algorithm proceeds from position $i_{j_{k-1}}$ to position i_{j_k} for $k = 2, 3, \ldots$ as follows, while assigning $i_{j_{k-1}}$ and i_{j_k} to variables i_{prev} and i_{next} respectively at each k: (i) it processes the substring $w[i_{prev}..i_{prev} + |\alpha| - 1]$ (lines 5 and 6), and then (ii) injects $\mathrm{cl}_\varepsilon(q_0)$ into the simulation set S if the α at position i_{prev} is a candidate that the left α of r_α may correspond to in a match (i.e., if e_0 matches $w[..i_{prev} - 1]$) (line 7). Next, (iii) it resumes the simulation and proceeds until the character $w[i_{next} - 1]$ (lines 5 and 6), and then (iv) performs the acceptance testing and checks whether e_2 matches the remaining suffix $w[i_{next} + |\alpha|..]$ (line 8). Finally, (v) it updates i_{prev} using i_{next} and if the α at i_{j_k} is a candidate that the left α of r_α may correspond to in a match, then keep its right-adjacent position $i_{j_k} + |\alpha| - 1$ in i_{que} for future injection (lines 9 and 10). If step (iv) succeeds for some j, it returns **true**; otherwise, it returns **false**.

MATCH1 runs in $O(n(m_e + m_{e_1}))$ time because it runs an NFA simulation of e and an NFA simulation of e_1 with injection. For correctness, the following is essential.

▶ **Proposition 9.** *Let $i_1 < i_2 < \cdots$ be the positions in Idx_α. Every time line 8 is reached at an iteration with $i_{next} = i_j$, we have $S = \bigcup_{j' \in J'} \Delta(\mathrm{cl}_\varepsilon(q_0), w[i_{j'} + |\alpha|, i_j - 1])$ where $J' = \{j' \in [1, |\mathsf{Idx}_\alpha|] \mid i_{j'} + |\alpha| \leq i_j \text{ and } w[..i_{j'} - 1] \in L(e_0)\}$.*

▶ **Lemma 10** (Correctness of MATCH1)**.** *Let α be a nonoverlapping right-maximal repeat of w. Then, MATCH1(α) returns* **true** *if and only if e matches α and r_α matches w.*

▶ Remark 11. In fact, MATCH1 works correctly for any repeat α and not only right-maximal ones. This gives an $O(n^3 m)$-time matching algorithm for rewbs of our form by modifying ENUMRM to output not only all right-maximal repeats but all repeats. As mentioned in the introduction, we note that this time complexity itself can also be achieved by existing algorithms. Further improvements in time complexity require additional ideas that we describe in the following sections as extensions of the baseline algorithm MATCH1.

▶ Remark 12. An essential and interesting property of the algorithm is that if it returns **true**, the existence of a match is guaranteed, *but we do not know where $(e)_1$ and $\backslash 1$ match.* This is because injecting $\mathrm{cl}_\varepsilon(q_0)$ into the simulation set in an NFA simulation means identifying the current position with the starting position of the NFA simulation.

3.1.2 The extendable prefixes of the right-maximal repeat

Recall that α is a fixed nonoverlapping right-maximal repeat of w. We extend MATCH1 to simultaneously examine all matches whose backreferenced substrings are α-extendable prefixes of α. To this end, we split the NFA simulation of MATCH1 in two phases.

We first introduce a technique called *summarization*. Let $q_1, \ldots, q_{m_1}$ be the states of N_{e_1}, the NFA equivalent to the middle subexpression e_1 of the input rewb that we fixed earlier.[7] While ordinary NFA simulation starts only from the initial state, *NFA simulation with summarization* (NFASS) starts its simulation from each state $q_1, \ldots, q_{m_1}$. Consequently, the "simulation set" of an NFASS is a vector of simulation sets $\mathcal{S} = \langle \mathcal{S}[1], \ldots, \mathcal{S}[m_1] \rangle$, where each $\mathcal{S}[l]$ is the simulation set of an ordinary NFA simulation but with q_l regarded as its initial state. We write $\Delta^{sum}(\mathcal{S}, u)$ for $\langle \Delta(\mathcal{S}[1], u), \ldots, \Delta(\mathcal{S}[m_1], u) \rangle$. Note that each step of an NFASS takes $O(m_1^2) = O(m_{e_1}^2)$ time.

Building on the above, we describe MATCH2(α) whose pseudocode is shown in Algorithm 2. Although the overall flow is similar to MATCH1 in Algorithm 1, it has two major differences.

One difference is that the NFA simulation using a simulation set S has been replaced by the NFASS using $\mathcal{S}$ (line 6). Similarly to step (ii) of MATCH1 (Section 3.1.1), when the NFASS reaches an occurrence of α at position i where e_0 matches the prefix $w[..i-1]$, it injects $\{q_l\}$ into $\mathcal{S}[l]$ for each $l \in [1, m_1]$ immediately after the character $w[i + |\alpha| - 1]$. Analogously to Proposition 9, the following proposition holds.

▶ **Proposition 13.** *Let $i_1 < i_2 < \cdots$ be the positions in Idx_α. Every time line 9 is reached at an iteration with $i_{next} = i_j$, we have $\mathcal{S}[l] = \bigcup_{j' \in J'} \Delta(\{q_l\}, w[i_{j'} + |\alpha|, i_j - 1])$ where $J' = \{j' \in [1, |\mathsf{Idx}_\alpha|] \mid i_{j'} + |\alpha| \leq i_j \text{ and } w[..i_{j'} - 1] \in L(e_0)\}$.*

The other major difference is the guard condition for the algorithm returning **true** (line 9). Each time the NFASS reaches the occurrence of α at position i_j, it runs another NFA simulation on the α with injection (line 8). The aim of this subsimulation is, roughly, to calculate a set of reachable states T from the initial state q_0 of N_{e_1} by any suffix $\alpha[k+1..]$

[7] Note that m_{e_1} denotes the length of e_1, whereas m_1 denotes the number of states in N_{e_1}. However, they may be regarded as the same because they differ only by a constant factor.

Algorithm 2 MATCH2(α).

Correctness: See Lemma 15.

```
1  i_prev ← ⊥;  S ← ⟨∅,...,∅⟩;  i_que ← ⊥;  (Q = {q_1,...,q_{m_1}}, Δ^sum, F) ← N_{e_1}
2  Construct an array Pre_α such that Pre_α[k] = true ⟺ α[..k] ∈ L(e) for k ∈ [1,|α|],
     which is used later in INTMED
3  for i_next ∈ Idx_α do
4      if i_que ≠ ⊥ then
5          for i ← i_prev to i_next − 1 do
6              if S ≠ ⟨∅,...,∅⟩ then S ← Δ^sum(S, w[i])          /* Summarization */
7              if i = i_que then S ← ⟨S[1] ∪ {q_1},...,S[m_1] ∪ {q_{m_1}}⟩ /* Injection */
8          T ← INTMED(i_next, i_next + |α| − 1)
9          if ∃q_l ∈ T.S[l] ∩ F ≠ ∅ then return true
10     i_prev ← i_next
11     if Pre[i_prev − 1] then i_que ← i_prev + |α| − 1
12 return false
```

Algorithm 3 INTMED(i_{beg}, i_{end}).

Correctness: See Lemma 14.

```
1  T ← ∅;  (Δ, cl_ε(q_0), F) ← N_{e_1}
2  for i ← i_beg to i_end do
3      if Pre_α[i − (i_end − |α|)] and Suf[i + 1] then T ← T ∪ cl_ε(q_0) /* Injection */
4      if T ≠ ∅ and i < i_end then T ← Δ(T, w[i + 1])
5  return T
```

of α whose corresponding prefix $\alpha[..k]$ is a candidate at position i_j for the content of $\backslash 1$ in a match, namely $\alpha[k+1..]$ where e matches $\alpha[..k]$ and e_2 matches $w[i_j+k..]$. Then, MATCH2(α) composes T and $\mathcal{S}$, that is, checks if there exists a state q_l of T such that the acceptance testing of $\mathcal{S}[l]$ succeeds (line 9). If such q_l exists, we can build a match by concatenating a suffix $\alpha[k+1..]$ that takes q_0 to q_l and a substring u of w that lies in two occurrences of α that takes q_l to an accept state in N_{e_1}. In this scenario, r matches w and the algorithm returns **true**; otherwise, it continues running.

Algorithm 3 shows INTMED(i_{beg}, i_{end}), the algorithm which calculates T at line 8 of Algorithm 2. It uses a Boolean array Pre_α such that $\mathsf{Pre}_\alpha[k] = \textbf{true} \iff \alpha[..k] \in L(e)$ for $k \in [1, |\alpha|]$ which is precomputed at the beginning of MATCH2(α) (line 2 of Algorithm 2).

INTMED requires that i_{end} is the right end position of an occurrence of α and i_{beg} is a position that belongs to the α. Note that the algorithm is presented in a generalized form to be used later in Section 3.2, but MATCH2 always passes i_{next} to the argument i_{beg}. Each time it reaches a position $i \in [i_{beg}, i_{end}]$, it checks whether e matches the prefix of α which ends at i and e_2 matches the remaining suffix of w by looking at $\mathsf{Pre}_\alpha[i - (i_{end} - |\alpha|)]$ and $\mathsf{Suf}[i+1]$ (line 3). If both checks succeed, it starts an NFA simulation or injects $\mathrm{cl}_\varepsilon(q_0)$ into the ongoing simulation set T. The correctness of the algorithm is as follows:

▶ **Lemma 14** (Correctness of INTMED). *Let i_{beg} and i_{end} be positions of w where i_{end} is the right end of some occurrence of α and $i_{end} - |\alpha| < i_{beg} \le i_{end}$. Then,* INTMED($i_{beg}, i_{end}$) *returns $T = \bigcup_{i \in I} \Delta(\mathrm{cl}_\varepsilon(q_0), w[i+1..i_{end}])$ where $I = \{i \in [i_{beg}, i_{end}] \mid w[i_{end} - |\alpha| + 1..i] \in L(e)$ and $w[i+1..] \in L(e_2)\}$.*

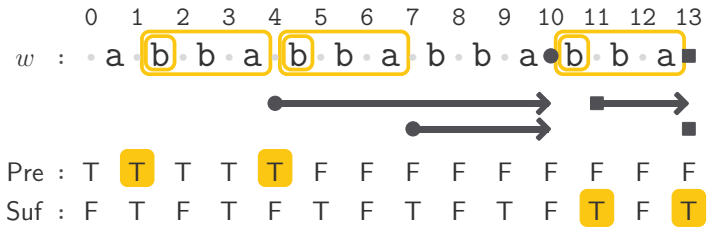


Figure 1 An example execution of MATCH2.

Given this, we prove the correctness of MATCH2 in the following lemma.

▶ **Lemma 15** (Correctness of MATCH2)**.** *Let α be a nonoverlapping right-maximal repeat of w. Then, there exists a prefix β of α such that e matches β and $e_0\beta e_1\beta e_2$ matches w if* MATCH2(α) *returns* **true**. *Conversely,* MATCH2(α) *returns* **true** *if there exists an α-extendable prefix β of α such that e matches β and $e_0\beta e_1\beta e_2$ matches w.*

Regarding the time complexity, MATCH2 runs in $O(n(m_e + m_{e_1}^2))$ time because its main loop processes each position of w at most twice (once by the NFASS and once by INTMED at line 8) and each step takes $O(m_{e_1}^2)$ time. Note that INTMED does not revisit any position because we assumed that α is a nonoverlapping right-maximal repeat.

▶ **Example 16.** We illustrate MATCH2 with its execution on the instance defined as follows. Let $w = \mathtt{abbabbabbabba}$ be the input string over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $e_0(e)_1 e_1 \backslash 1 e_2$ be the input rewb, where $e = \Sigma^*$, $e_2 = (\Sigma\Sigma)^*$, e_0 matches the strings that contain at most two $\mathtt{b}$'s, and e_1 matches those that contain $\mathtt{b}$ at least three and an odd number of times. We consider the case where $\alpha = \mathtt{bba}$, which is a nonoverlapping right-maximal repeat of w. Note that its occurrence array Idx_α is $[2, 5, 8, 11]$.

Figure 1 shows the point at which INTMED, which was called at line 8, has completed its execution during the loop for $i_{next} = 11$. The top row shows the position of w, and the two rows labeled Pre and Suf represent the Boolean values of the corresponding arrays at each position. Within the row of w, the bullet and the square mark the positions being currently processed by the NFASS and the execution of INTMED, respectively. Analogously, the arrows below w, starting from the bullets and the squares, indicate the past behaviors of those.

The bullet at position 4 marks where the NFASS started because the α at position 2 is the leftmost among the occurrences of α that e_0 matches the prefix of w to the left (i.e., $\mathsf{Pre}[1] =$ **true**). Similarly, the one at position 7 marks where injection was performed in the NFASS because the α at position 5 was such an α (i.e., $\mathsf{Pre}[4] =$ **true**). Also, the squares at positions 11 and 13 mark where the execution of INTMED started and injection was performed in it because $\mathtt{b}$ and $\mathtt{bba}$ are prefixes of α that e matches (which always holds in this instance) and e_2 matches whose remaining suffix of w (i.e., $\mathsf{Suf}[11] = \mathsf{Suf}[13] =$ **true**), respectively.

Then, the algorithm checks at line 9 if e_1 matches any of $w[3..10]$, $w[5, 10]$, $w[6..10]$ and $w[8, 10]$, and returns **true** because e_1 matches $w[3..10]$ and $w[6..10]$. Observe that, as stated in Remark 12, it cannot determine which of the four e_1 actually matches.

3.2 The General Case of Right-Maximal Repeats

Let α be a right-maximal repeat. In this subsection, we give the full version of our algorithm MATCH(α) that works for the general case where α is possibly overlapping.

We first explain why the aforementioned algorithm MATCH2(α) does not work correctly in this case. There are two main reasons. One is the time complexity. Note that in MATCH2(α), INTMED is called at every position right before where α occurs (line 8 of Algorithm 2). Under

Algorithm 4 MATCH3A(α).

Correctness: See Lemma 18.

```
1  i_prev ← ⊥;  S ← ⟨∅,...,∅⟩;  Que ← ⊥;  (Q = {q_1,...,q_{m_1}}, Δ^sum, F) ← N_{e_1}
2  Construct an array Pre_α such that Pre_α[k] = true ⟺ α[..k] ∈ L(e) for k ∈ [1,|α|],
     which is used later in INTMED
3  d ← max({0} ∪ {Idx_α[j−1] + |α| − Idx_α[j] | j ≥ 2})
4  for i_next ∈ Idx_α do
5      if Que ≠ ⊥ then
6          for i = i_prev to i_next − 1 do
7              if S ≠ ⟨∅,...,∅⟩ then S ← Δ^sum(S, w[i])        /* Summarization */
8              if Que ≠ [] and Que.top = i then
9                  S ← ⟨S[1] ∪ {q_1},...,S[m_1] ∪ {q_{m_1}}⟩      /* Injection */
10                 Que.dequeue()
11         T ← INTMED(i_next + d, i_next + |α| − 1)
12         if ∃q_l ∈ T.S[l] ∩ F ≠ ∅ then return true
13     i_prev ← i_next
14     if Pre[i_prev − 1] then
15         if Que = ⊥ then Que ← []
16         Que.enqueue(i_prev + |α| − 1)
17 return false
```

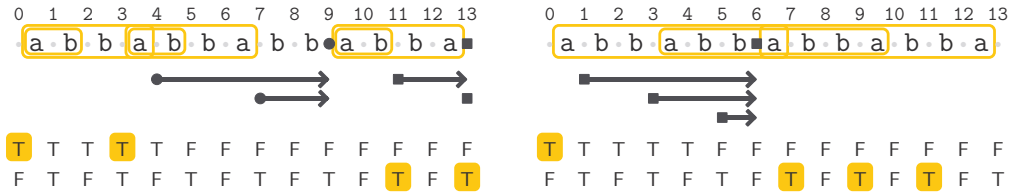
the nonoverlapping assumption, the total time INTMED takes is linear in n because the total length of all occurrences of α is also linear, but that does not necessarily hold when α may overlap. The other reason concerns the correctness of the algorithm. When overlaps are allowed, there may exist a match whose backreferenced substring occurs as nonoverlapping α-extendable prefixes of some overlapping occurrences of α. Because MATCH2 is not designed to find such matches, it may falsely report that no match exists.

The key observation to overcome these obstacles is, as stated in Remark 7, that it only needs to check if there are matches whose backreferenced substrings are α-extendable prefixes of α, rather than arbitrary prefixes of α. The following lemma gives a necessary condition for a prefix of α being α-extendable (a related statement appears as Lemma 5 in [45]):

▶ **Lemma 17.** *Let α be a right-maximal repeat. Suppose that α contains its prefix β at least twice: once as a prefix and once elsewhere. Then β is a non-α-extendable prefix of α. More generally, if two occurrences of α have an overlap of length d, the prefixes of α whose length is no more than d are non-α-extendable prefixes of α.*

In what follows, we divide the algorithm MATCH into two subalgorithms MATCH3A and MATCH3B, and describe how they address the obstacles noted above. MATCH3A(α) (resp. MATCH3B(α)) is an algorithm to detect a match whose backreferenced substring occurs as an α-extendable prefix of some nonoverlapping occurrences (resp. a nonoverlapping α-extendable prefix of some overlapping occurrences) of α. Consequently, MATCH(α) is an algorithm that returns **true** if and only if at least one of the subalgorithms returns **true**.

Algorithm 4 shows MATCH3A(α), the subalgorithm for detecting a match whose backreferenced substring occurs as an α-extendable prefix of some nonoverlapping occurrences of α. We explain the changes from MATCH2. Recall the first obstacle: INTMED takes too much time. Let d be the maximum length of the overlapping substrings of the occurrences of α,



(a) MATCH3A. **(b)** MATCH3B.

Figure 2 Example executions. The meaning of the rows is the same as in Figure 1.

i.e., $d = \max(\{0\} \cup \{\mathsf{Idx}_\alpha[j-1] + |\alpha| - \mathsf{Idx}_\alpha[j] \mid j \geq 2\})$.[8] MATCH3A precomputes d at line 3. Clearly, this can be done in $O(n)$ time by only considering each two adjacent occurrences of α. Then, by Lemma 17, the algorithm only needs to examine a match whose backreferenced substring is a prefix of α strictly longer than $\alpha[..d]$, namely any of $\alpha[..d+1], \ldots, \alpha[..|\alpha|] = \alpha$. Therefore, MATCH3A calls INTMED so that it starts from position $i_{next} + d$ rather than i_{next} (line 11), ensuring its $O(n(m_e + m_{e_1}^2))$ time complexity.

A subtle issue here is that in the NFASS of MATCH3A, there may be multiple timings of injection before reaching another occurrence of α. This is dealt with by managing them with an FIFO structure Que instead of a variable i_{que} as was done in MATCH2.

The following lemma states the correctness of MATCH3A. Recall the definition of α-separability from Section 2.

▶ **Lemma 18** (Correctness of MATCH3A)**.** *Let α be a right-maximal repeat of w. There exists a prefix β of α such that e matches β and $e_0\beta e_1\beta e_2$ matches w if* MATCH3A(α) *returns* **true**. *Conversely,* MATCH3A(α) *returns* **true** *if there exists an α-extendable prefix β of α such that (i) e matches β, (ii) $e_0\beta e_1\beta e_2$ matches w and (iii) the two occurrences of β are α-separable in w.*

▶ **Example 19.** Figure 2(a) shows the execution of the algorithm on the same instance as Example 16 where $\alpha = \mathtt{abba}$, which is an overlapping right-maximal repeat of w and whose occurrence array Idx_α is $[1, 4, 7, 10]$. It returns **true** because e_1 matches $w[3..9]$ and $w[6..9]$. Observe that it skips the check for a match whose backreferenced substring is $\mathtt{a}$ because $\mathtt{a}$ is non-α-extendable by $d = 1$ and Lemma 17. In fact, $\mathtt{a}$ is right-maximal and the check is instead performed by the execution of MATCH3A with $\alpha = \mathtt{a}$, as mentioned in Remark 7.

Next, we explain MATCH3B(α) shown in Algorithm 5, the subalgorithm for detecting a match whose backreferenced substring occurs as a nonoverlapping α-extendable prefix of some overlapping occurrences of α.

We first explain the challenges with detecting such matches in time linear in n. Fix an occurrence of α and suppose that no other α overlaps it from the left and it overlaps other α's to the right. We name them $\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_k$ from left to right with α_1 being the one we fixed earlier. It may seem that MATCH3B(α) has to check matches between every pair of these α_i's. Doing this naively takes $\Theta(k^3)$ time, and as $k = \Theta(n)$ in general, this becomes $\Theta(n^3)$ time (for example, the case $w = \mathtt{a}^{2n}$ and $\alpha_i = w[i..i+n-1]$ for $i \in [1, n]$).

[8] ENUMRM always returns Idx_α of size at least 2. Note that an α that occurs less than twice need not be considered.

Algorithm 5 MATCH3B(α).

Correctness: See Lemma 23.

```
   /* zip(A, B) = [⟨A[j], B[j]⟩ | 1 ≤ j ≤ |A|] provided that |A| = |B|          */
 1 Fwd ← []; j1 ← |Idx_α|; j2 ← |Idx_α|
 2 while j1 ≥ 1 do
 3     if Idx_α[j2] ≤ Idx_α[j1] + |α| − 1 then
 4         Fwd[j1] ← Idx_α[j2]
 5         j1 ← j1 − 1
 6     else j2 ← j2 − 1
 7 f_prev ← 0; S ← ∅; (Δ, cl_ε(q0), F) ← N_{e1}
 8 for ⟨i_next, f_next⟩ ∈ zip(Idx_α, Fwd) do
 9     if i_next < f_next, Pre[i_next − 1] and f_prev < f_next then
10         S ← ∅
11         for i ← max{i_next, f_prev} to f_next − 1 do
12             if Pre_α[i − i_next + 1] and Suf[f_next + i − i_next + 1] then
13                 S ← S ∪ cl_ε(q0)                                  /* Injection */
14             if S ≠ ∅ and i < f_next − 1 then S ← Δ(S, w[i + 1])
15         if S ∩ F ≠ ∅ then return true
16     f_prev ← f_next
17 return false
```

Our key observation is that MATCH3B(α) actually only needs to examine matches between each α and *at most one* α to its right. For example, in the above case of $\alpha_1, \ldots, \alpha_k$, when the algorithm checks if a match where e matches α_1 exists, it only examines the matches between α_1 and α_k. Moreover, the algorithm makes only one pass from α_1 to α_k to check all the necessary matches. The following definition and lemma explain why this is correct.

▶ **Definition 20.** For every $i \in \mathsf{Idx}_\alpha$, define $f(i) := \max\{j \in \mathsf{Idx}_\alpha \mid j \leq i + |\alpha| - 1\}$.

▶ **Lemma 21.** *For any positions $i, j \in \mathsf{Idx}_\alpha$ such that $i < j < f(i)$, neither $w[i..j-1]$ nor $w[j..f(i)-1]$, nor any of their prefixes, is α-extendable.*

Therefore, MATCH3B(α) only needs to examine matches between each α and the rightmost α it overlaps. Moreover, when checking each α at position j between i and $f(i)$, the algorithm can skip the steps from j to $f(i) - 1$ and start from $f(i)$. Thus, the overall checks can be done in $O(nm_{e_1})$ time using NFA simulation with oracles $\mathsf{Pre}, \mathsf{Suf}, \mathsf{Pre}_\alpha$ and injection. Recall that $\mathsf{Pre}[i] = \textbf{true} \iff w[..i] \in L(e_0)$ for $i \in [0, n]$, $\mathsf{Suf}[j] = \textbf{true} \iff w[j..] \in L(e_2)$ for $j \in [1, n+1]$ and $\mathsf{Pre}_\alpha[k] = \textbf{true} \iff \alpha[..k] \in L(e)$ for $k \in [1, |\alpha|]$.

We explain in detail how MATCH3B works. Algorithm 5 shows the pseudocode. First, MATCH3B computes the array Fwd which represents f in $O(n)$ time (lines 1 to 6). It uses two pointers j_1, j_2 and updates them so that the invariant $\mathsf{Fwd}[j_1] = f(\mathsf{Idx}_\alpha[j_1])$ holds every time line 4 is executed. Then, the algorithm scans each position i_{next} of Idx_α with the starting position f_{next} of the rightmost α which the α at i_{next} overlaps until the guard of the if statement at line 9 becomes true. The guard has the following purpose. In the if statement, the algorithm will check a match between a prefix of the α at i_{next} and a prefix of that at f_{next}. Prior to this, the guard excludes the cases (1) $i_{next} = f_{next}$ and (2) e_0 does not match the prefix to the left of the α at i_{next}. It also excludes the case (3) $f_{next} = f_{prev}$ to skip unnecessary checks.

If the guard holds, then the algorithm executes lines 10 to 15. The for loop in lines 11 to 14 is similar to that of IntMed (lines 2 to 4 of Algorithm 3). Each step performs injection and at line 15 the algorithm checks the existence of a match whose backreferenced substring is the prefix of α which starts at i_{next} and ends at i between the α at i_{next} and the one at f_{next}. Note that the length of the prefix is $i - i_{next} + 1$. The injection is performed only if e matches the prefix and e_2 matches the remaining suffix $w[f_{next} + i - i_{next}..]$. The for loop starts with $i = \max\{i_{next}, f_{prev}\}$ because the checks for the prefixes of $w[i_{next}..f_{prev} - 1]$ can be skipped when $i_{next} < f_{prev}$, as mentioned in the paragraph following Lemma 21. This and condition (3) above ensure the linear time complexity of the algorithm with respect to n.

We show the correctness of Match3B. The following is similar to Proposition 13.

▶ **Proposition 22.** *Let $i_1 < i_2 < i_3 < \cdots$ be the positions in Idx_α. Every time line 15 is reached at an iteration with $i_{next} = i_j$, we have $S = \bigcup_{i \in I} \Delta(\mathrm{cl}_\varepsilon(q_0), w[i+1..f(i_j)-1])$ where $I = \{i \in [\max\{i_j, f(i_{j-1})\}, f(i_j) - 1] \mid w[i_j..i] \in L(e)$ and $w[f(i_j) + i - i_j + 1] \in L(e_2)\}$. Here, we regard $f(i_{j-1}) = 0$ when $j = 1$.*

▶ **Lemma 23** (Correctness of Match3B)**.** *Let α be a right-maximal repeat of w. There exists a prefix β of α such that e matches β and $e_0\beta e_1\beta e_2$ matches w if* Match3B(α) *returns* **true**. *Conversely,* Match3B(α) *returns* **true** *if there exists an α-extendable prefix β of α such that (i) e matches β, (ii) $e_0\beta e_1\beta e_2$ matches w and (iii) the two occurrences of β are not α-separable in w.*

▶ **Example 24.** We illustrate Match3B using the same instance as in Examples 16 and 19. We consider the case where $\alpha = \mathtt{abbabba}$, which is an overlapping right-maximal repeat of w and whose occurrence array Idx_α is $[1, 4, 7]$. Figure 2(b) shows part of the execution. In this case, as mentioned in the paragraph immediately after Lemma 21, the algorithm only needs to examine matches between the occurrences of α at positions 1 and $f(1) = 7$. The squares at positions 1, 3 and 5 mark where injection was performed because `a`, `abb` and `abbab` are prefixes of α that e matches and e_2 matches whose remaining suffix of w (i.e., $\mathsf{Suf}[7] = \mathsf{Suf}[9] = \mathsf{Suf}[11] =$ **true**), respectively. It returns **false** because e_1 matches none of $w[2..6]$, $w[4..6]$ and $w[6]$. Note that `a` and `abb` are non-α-extendable prefix of α, and the checks for these in this execution are actually redundant.

4 Related Work

We first mention efficient solutions of the pure regular expression matching problem. The improvement of the $O(nm)$-time solution using NFA simulation was raised as an unsolved problem by Galil [22], but in 1992, Myers successfully resolved it in a positive manner [32]. Since then, further improvements have been made by researchers, including Bille [8, 9, 11]. On the other hand, Backurs and Indyk have shown that under the assumption of the strong exponential time hypothesis, no solution exists within $O((nm)^{1-\epsilon})$ time for any $\epsilon > 0$ [4]. Recently, Bille and Gørtz have shown the complexity with respect to a new parameter, the total size of the simulation sets in an NFA simulation $\sum_{i=0}^{n} |S^{(i)}|$, in addition to n and m [10].

We next discuss prior work on the matching problem of rewbs. The problem can be solved by simulating *memory automata* (MFA), which are a model proposed by Schmid [41] with the same expressive power as rewbs. An MFA has additional space called *memory* to keep track of matched substrings. A *configuration* of an MFA M with k memories is a tuple $(q, u, (x_1, s_1), \ldots, (x_k, s_k))$ where q is a current state, u is the remaining input string and (x_j, s_j) ($j \in [1, k]$) is the pair of the content substring x_j and the state s_j of memory j. Therefore, the number of configurations of M equivalent to a given rewb on a given

string is $O(n^{2k+1}m)$. Because each step of an MFA simulation may involve $O(n)$ character comparisons, this gives a solution to the rewb matching problem that runs in $O(n^{2k+2}m)$ time. Davis et al. gave an algorithm with the same time complexity as this [15]. Furthermore, by precomputing some string indices as in this paper, a substring comparison can be done in constant time, making it possible to run in $O(n^{2k+1}m)$ time. Therefore, for the rewbs considered in this paper, these algorithms take time cubic in n because $k = 1$ for these rewbs, and our new algorithm substantially improves the complexity, namely, to quadratic in n.

Regarding research on efficient matching of rewbs, Schmid proposed the *active variable degree* (avd) of MFA and discussed the complexity with respect to avd [42]. Roughly, avd is the minimum number of substrings that needs to be remembered at least once per step in an MFA simulation. For example, in a simulation of an MFA equivalent to the rewb $(a^*)_1\backslash 1(b^*)_2\backslash 2$, after consuming the substring captured by $(a^*)_1$ in the transition which corresponds to $\backslash 1$, configurations no longer need to keep the substring. In other words, it only needs to remember only one substring at each step of the simulation, and hence its avd is 1. On the other hand, $\mathrm{avd}((a^*)_1(b^*)_2\backslash 1\backslash 2)$ is 2. The avd of the rewbs considered in this paper is always 1, but their method takes quartic (or cubic with the simple modification on MFA simulation outlined above) time for them. Freydenberger and Schmid proposed *deterministic regular expression with backreferences* and showed that the matching problem of deterministic rewbs can be solved in linear time [20]. The rewbs considered in this paper are not deterministic in general (for example, $(a^*)_1\backslash 1$ is not).

Next, we mention research on efficient matching of pattern languages with bounded number of repeated variables. A *pattern with variables* is a string over constant symbols and variables. The matching problem for patterns is the problem of deciding whether a given string w can be obtained from a given pattern p by uniformly substituting nonempty strings of constant symbols for the variables of p. Note that, as remarked in the introduction, rewbs can be viewed as a generalization of patterns by regular expressions.

Fernau et al. discussed the matching problem for patterns with at most k repeated variables [17]. A *repeated variable* of a pattern is a variable that occurs in the pattern more than once. In particular, for the case $k = 1$, they showed the problem can be solved in quadratic time with respect to the input string length n. The patterns with one repeated variable and the rewbs considered in this paper are independent. While these patterns can use the variable more than twice, these rewbs can use regular expressions. Therefore, our contribution has expanded the variety of languages that can be expressed within the same time complexity with respect to n.

The algorithm by Fernau et al. [17] leverages *clusters* defined over the suffix array of an input string, which are related to right-maximal repeats. Moreover, it is similar to our approach in that it does the examination while enumerating candidate assignments to the repeated variable. The key technical difference is that, as demonstrated in their work, matching these patterns can be reduced to finding a canonical match by dividing the pattern based on wildcard variables. In contrast, matching the rewbs considered in this paper requires handling the general regular expression matching, particularly the substring matching of the middle expression e_1, which makes such a reduction not applicable even when the number of variable occurrences is restricted to 2.

Regarding the squareness checking problem mentioned in the introduction, Main and Lorentz [28] and Crochemore [12] showed a linear-time solution on a given alphabet. However, both solutions rely on properties specific to square-free strings, and extending them to the matching problem considered in this paper seems difficult.

Finally, we mention related work on the techniques and concepts from automata theory and stringology used in this paper. A similar approach to using oracles such as Pre and Suf in NFA simulation is used in research on efficient matching of regular expressions with lookarounds [29, 5]. Summarization has been applied to parallel computing for pure regular expression matching [27, 24, 43]. Regarding injection, NFA simulation itself uses injection internally to handle concatenation of regular expressions. That is, an NFA simulation of e_1e_2 can be seen as that of e_2 that injects the ε-closure of the initial state of the NFA N_{e_2} whenever e_1 matches the input string read so far. Nonetheless, our use of these automata-theoretic techniques for efficient matching of rewbs is novel. In fact, to our knowledge, this paper is the first to propose an algorithm that combines these techniques.

The right-maximal repeats of a string are known to correspond to the internal nodes of the *suffix tree* of the string [23]. Kasai et al. first introduced a linear-time algorithm for traversing the internal nodes of a suffix tree using a suffix array [26]. Subsequently, Abouelhoda et al. introduced the concept of the *LCP-interval tree* to make their traversal more complete [1]. As stated in Section 2, our EnumRM that enumerates the right-maximal repeats with the sorted starting positions of their occurrences is based on their algorithm. To our knowledge, our work is the first to apply these stringology concepts and techniques to efficient matching of rewbs.

5 Conclusion

In this paper, we proposed an efficient matching algorithm for rewbs of the form $e_0(e)_1e_1\backslash 1e_2$ where e_0, e, e_1, e_2 are pure regular expressions, which are fundamental and frequently used in practical applications. As stated in the introduction and Section 4, it runs in $O(n^2m^2)$ time, improving the best-known time complexity for these rewbs when $n > m$. Because n is typically much larger than m, this is a substantial improvement.

Our algorithm combines ideas from both stringology and automata theory in a novel way. The core of our algorithm consists of two techniques from automata theory, injection and summarization. Together, they enable the algorithm to do all the examination for a fixed right-maximal repeat and its extendable prefixes, which are concepts from stringology, instead of examining each individually. By further leveraging a subtle property of extendable prefixes, our algorithm correctly solves the matching problem in time quadratic in n.

A possible direction for future work is to further reduce the time complexity of the algorithm. A natural next step would be to use *maximal repeats* instead of right-maximal repeats. While this would not change the worst-case complexity with respect to n [13, 38], it could lead to faster performance for many input strings. Another possible direction is to extend the algorithm to support more general rewbs as mentioned in Remark 2. The extension of our algorithm with support for other practical extensions such as lookarounds is also challenging.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.

2 Alfred V. Aho. Algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 255–300. Elsevier and MIT Press, Cambridge, MA, USA, 1990.

3 Dana Angluin. Finding patterns common to a set of strings. *J. Comput. Syst. Sci.*, 21(1):46–62, 1980. doi:10.1016/0022-0000(80)90041-0.

4 Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In Irit Dinur, editor, *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 457–466. IEEE, IEEE Computer Society, 2016. doi:10.1109/FOCS.2016.56.

5 Aurèle Barrière and Clément Pit-Claudel. Linear matching of javascript regular expressions. *Proc. ACM Program. Lang.*, 8(PLDI):1336–1360, 2024. doi:10.1145/3656431.

6 Martin Berglund and Brink van der Merwe. Re-examining regular expressions with backreferences. *Theor. Comput. Sci.*, 940(Part):66–80, 2023. doi:10.1016/j.tcs.2022.10.041.

7 Martin Berglund, Brink van der Merwe, and Steyn van Litsenborgh. Regular expressions with lookahead. *J. Univers. Comput. Sci.*, 27(4):324–340, 2021. doi:10.3897/jucs.66330.

8 Philip Bille. New algorithms for regular expression matching. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part I*, volume 4051 of *Lecture Notes in Computer Science*, pages 643–654. Springer, Springer, 2006. doi:10.1007/11786986_56.

9 Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theor. Comput. Sci.*, 409(3):486–496, 2008. doi:10.1016/j.tcs.2008.08.042.

10 Philip Bille and Inge Li Gørtz. Sparse regular expression matching. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 3354–3375. SIAM, SIAM, 2024. doi:10.1137/1.9781611977912.120.

11 Philip Bille and Mikkel Thorup. Faster regular expression matching. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas, and Wolfgang Thomas, editors, *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I*, volume 5555 of *Lecture Notes in Computer Science*, pages 171–182. Springer, Springer, 2009. doi:10.1007/978-3-642-02927-1_16.

12 Maxime Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986. doi:10.1016/0304-3975(86)90041-1.

13 Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In Alberto Apostolico and Jotun Hein, editors, *Combinatorial Pattern Matching, 8th Annual Symposium, CPM 97, Aarhus, Denmark, June 30 - July 2, 1997, Proceedings*, volume 1264 of *Lecture Notes in Computer Science*, pages 116–129. Springer, Springer, 1997. doi:10.1007/3-540-63220-4_55.

14 James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 246–256. ACM, 2018. doi:10.1145/3236024.3236027.

15 James C. Davis, Francisco Servant, and Dongyoon Lee. Using selective memoization to defeat regular expression denial of service (redos). In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1–17. IEEE, IEEE, 2021. doi:10.1109/SP40001.2021.00032.

16 Andrzej Ehrenfeucht and Grzegorz Rozenberg. Finding a homomorphism between two words is np-complete. *Inf. Process. Lett.*, 9(2):86–88, 1979. doi:10.1016/0020-0190(79)90135-2.

17 Henning Fernau, Florin Manea, Robert Mercas, and Markus L. Schmid. Pattern matching with variables: Efficient algorithms and complexity results. *ACM Trans. Comput. Theory*, 12(1):6:1–6:37, 2020. doi:10.1145/3369935.

18 Henning Fernau and Markus L. Schmid. Pattern matching with variables: A multivariate complexity analysis. *Inf. Comput.*, 242:287–305, 2015. doi:10.1016/j.ic.2015.03.006.

19 Henning Fernau, Markus L. Schmid, and Yngve Villanger. On the parameterised complexity of string morphism problems. *Theory Comput. Syst.*, 59(1):24–51, 2016. doi:10.1007/s00224-015-9635-3.

20 Dominik D. Freydenberger and Markus L. Schmid. Deterministic regular expressions with back-references. *J. Comput. Syst. Sci.*, 105:1–39, 2019. doi:10.1016/j.jcss.2019.04.001.

21 Hiroya Fujinami and Ichiro Hasuo. Efficient matching with memoization for regexes with look-around and atomic grouping. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*, volume 14577 of *Lecture Notes in Computer Science*, pages 90–118. Springer, Springer, 2024. doi:10.1007/978-3-031-57267-8_4.

22 Zvi Galil. Open problems in stringology. In Alberto Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words*, pages 1–8. Springer, 1985.

23 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.

24 W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986. doi:10.1145/7902.7903.

25 Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. On-line construction of symmetric compact directed acyclic word graphs. In Gonzalo Navarro, editor, *Eighth International Symposium on String Processing and Information Retrieval, SPIRE 2001, Laguna de San Rafael, Chile, November 13-15, 2001*, pages 96–110. IEEE, IEEE Computer Society, 2001. doi:10.1109/SPIRE.2001.989743.

26 Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Amihood Amir and Gad M. Landau, editors, *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001. doi:10.1007/3-540-48194-X_17.

27 Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980. doi:10.1145/322217.322232.

28 Michael G. Main and Richard J. Lorentz. Linear time recognition of squarefree strings. In *Combinatorial Algorithms on Words*, pages 271–278. Springer, Springer Berlin Heidelberg, 1985. doi:10.1007/978-3-642-82456-2_18.

29 Konstantinos Mamouras and Agnishom Chattopadhyay. Efficient matching of regular expressions with lookaround assertions. *Proc. ACM Program. Lang.*, 8(POPL):2761–2791, 2024. doi:10.1145/3632934.

30 Takayuki Miyazaki and Yasuhiko Minamide. Derivatives of regular expressions with lookahead. *J. Inf. Process.*, 27:422–430, 2019. doi:10.2197/ipsjjip.27.422.

31 Akimasa Morihata. Translation of regular expression with lookahead into finite state automaton. *Computer Software*, 29(1):147–158, 2012. doi:10.11309/jssst.29.1_147.

32 Eugene W. Myers. A four russians algorithm for regular expression pattern matching. *J. ACM*, 39(2):430–448, 1992. doi:10.1145/128749.128755.

33 Kazuyuki Narisawa, Hideharu Hiratsuka, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Efficient computation of substring equivalence classes with suffix arrays. *Algorithmica*, 79(2):291–318, 2017. doi:10.1007/s00453-016-0178-z.

34 Taisei Nogami and Tachio Terauchi. nogamita/MFCS2025. Software, JSPS KAKENHI Grant Numbers JP25KJ2137, JP23K24826, and JP20K20625 (visited on 2025-08-04). URL: https://github.com/nogamita/MFCS2025, doi:10.4230/artifacts.24325.

35 Taisei Nogami and Tachio Terauchi. On the expressive power of regular expressions with backreferences. In Jérôme Leroux, Sylvain Lombardy, and David Peleg, editors, *48th International Symposium on Mathematical Foundations of Computer Science, MFCS 2023, August*

28 to September 1, 2023, Bordeaux, France, volume 272 of *LIPIcs*, pages 71:1–71:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.MFCS.2023.71.

36 Taisei Nogami and Tachio Terauchi. Efficient matching of some fundamental regular expressions with backreferences. *CoRR*, abs/2504.18247, 2025. doi:10.48550/arXiv.2504.18247.

37 Taisei Nogami and Tachio Terauchi. Measuring the expressive power of practical regular expressions by classical stacking automata models. *Inf. Comput.*, 305:105303, 2025. doi:10.1016/j.ic.2025.105303.

38 Mathieu Raffinot. On maximal repeats in strings. *Inf. Process. Lett.*, 80(3):165–169, 2001. doi:10.1016/S0020-0190(01)00152-1.

39 Daniel Reidenbach and Markus L. Schmid. A polynomial time match test for large classes of extended regular expressions. In Michael Domaratzki and Kai Salomaa, editors, *Implementation and Application of Automata - 15th International Conference, CIAA 2010, Winnipeg, MB, Canada, August 12-15, 2010. Revised Selected Papers*, volume 6482 of *Lecture Notes in Computer Science*, pages 241–250. Springer, Springer, 2010. doi:10.1007/978-3-642-18098-9_26.

40 Markus L. Schmid. A note on the complexity of matching patterns with variables. *Inf. Process. Lett.*, 113(19-21):729–733, 2013. doi:10.1016/j.ipl.2013.06.011.

41 Markus L. Schmid. Characterising REGEX languages by regular languages equipped with factor-referencing. *Inf. Comput.*, 249:1–17, 2016. doi:10.1016/j.ic.2016.02.003.

42 Markus L. Schmid. Regular expressions with backreferences: Polynomial-time matching techniques. *CoRR*, abs/1903.05896, 2019. doi:10.48550/arXiv.1903.05896.

43 Ryoma Sin'ya, Kiminori Matsuzaki, and Masataka Sassa. Simultaneous finite automata: An efficient data-parallel model for regular expression matching. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 220–229. IEEE Computer Society, 2013. doi:10.1109/ICPP.2013.31.

44 Henry Spencer. *A regular-expression matcher*, pages 35–71. Academic Press Professional, Inc., USA, 1994.

45 Masayuki Takeda, Tetsuya Matsumoto, Tomoko Fukuda, and Ichiro Nanri. Discovering characteristic expressions in literary works. *Theor. Comput. Sci.*, 292(2):525–546, 2003. doi:10.1016/S0304-3975(02)00185-8.

46 Ken Thompson. Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968. doi:10.1145/363347.363387.

47 WhyRE2. (Accessed: 2024-12-14). URL: https://github.com/google/re2/wiki/WhyRE2.