Matching Regular-Typed Pattern Languages: Quadratic-Time Algorithms

Yuya Uezato
CyberAgent, Inc., Tokyo, Japan
National Institute of Informatics, Tokyo, Japan

Abstract

Pattern languages (PAT) are a class of languages generated by expressions called patterns that may contain variables. In a pattern, each variable can be instantiated with an arbitrary string. Typed pattern languages extend PAT by associating a type (constraint) with each variable that restricts the domain of allowed substitutions. In this paper, we study regular-typed PAT (PATwRT), where all types are represented either by a regular expression or by an ε-NFA. We consider the PATwRT matching problem for patterns with a single repeated variable of the form $P = \alpha_1\beta\alpha_2\beta\cdots\beta\alpha_K$. We present simple algorithms whose running time is linear in K and quadratic in the input length N, with polynomial dependence on the sizes of the type representations. Our results extend previous quadratic-time work in two directions: (1) the quadratic-time algorithm for untyped PAT of Fernau et al. (STACS 2015), and (2) the quadratic-time algorithm for the restricted PATwRT $K = 3$, i.e., $\alpha_1\beta\alpha_2\beta\alpha_3$ of Nogami and Terauchi (MFCS 2025).

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Pattern matching

Keywords and phrases Pattern languages, Regular expressions, String algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2026.11

Funding This work was supported by JST, CREST Grant Number JPMJCR21M3.

Acknowledgements The author is deeply grateful to Soh Kumabe for fruitful discussions during the early stages of this research and for providing valuable comments. The author also thanks the anonymous reviewers for their insightful feedback, which significantly improved the presentation of this paper. In particular, one of the reviewers kindly suggested using an $O(n^\omega)$-time algorithm for computing the reflexive transitive closure of a Boolean matrix, which improved the running time of our algorithm for dense ε-NFAs.

1 Introduction

In this paper, we study the following decision problem for the formalism of *regular-typed pattern languages* (**PATwRT**).

Matching Problem for Regular-Typed Pattern Languages

Input 1: An input string W.
Input 2: A pattern P and a type constraint $\mathcal{C}$.
Task: Deciding whether $W \in \mathcal{L}_{\mathcal{C}}(P)$ where $\mathcal{L}_{\mathcal{C}}(P)$ denotes the language generated by P under $\mathcal{C}$.

To explain **PATwRT**, we first recall *pattern languages* (**PAT**), since **PATwRT** is an extension of them. Pattern languages were introduced by Angluin in the context of automata learning theory [4, 5]. In **PAT**, variables can be used to describe repeated substrings. For example, consider the following pattern:

$P = \alpha\,\beta\,\#\,\beta\,\alpha$ where α, β are variables and $\#$ is a letter.

Over an alphabet Σ, the language $\mathcal{L}(P)$ generated by P consists of all strings obtained by substituting strings $w_\alpha, w_\beta \in \Sigma^*$ for α and β, respectively; that is,

$$\mathcal{L}(P) = \{w_\alpha\, w_\beta \,\#\, w_\beta\, w_\alpha : w_\alpha, w_\beta \in \Sigma^*\}.$$

For example, let $\Sigma = \{\#, 0, 1\}$ and consider an assignment φ such that $\varphi(\alpha) = 01$ and $\varphi(\beta) = 111$. Then, the instantiated string $\varphi(P) = 01\,111\,\#\,111\,01$ belongs to $\mathcal{L}(P)$. For $\Sigma = \{\#, 0, 1\}$, the language $\mathcal{L}(P)$ is neither regular nor context-free (cf. [31, Theorem 8]). In general, the language class **PAT** is contained in **NL**, the class of languages recognized by nondeterministic log-space Turing machines [6, 23]. On the other hand, **PAT** does not include all regular languages. For example, the regular language $\mathcal{L}((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\}$ cannot be generated by any pattern.

To restrict the domains used when assigning strings to variables, *typed pattern languages* (**PATwTy**) were introduced by Wright [36] and studied by Koshiba [25] in the context of learning theory. In **PATwTy**, in addition to pattern expressions, we consider a type constraint $\mathcal{C}$ that assigns a *type* $\mathcal{C}(\nu)$ to each variable ν. For example, in the pattern P above, if we set $\mathcal{C}(\beta) = \{w \in \Sigma^* : \text{the numbers of 0s and 1s in } w \text{ are equal}\}$, then $\mathcal{L}_\mathcal{C}(P)$ contains $11\,0101\,\#\,0101\,11$, but not $11\,010\,\#\,010\,11$.

If all types are *regular languages*, we obtain *regular-typed pattern languages* (**PATwRT**). Let us consider the following example pattern Q and type constraint $\mathcal{C}$:

$$Q = \alpha\,\#\,\beta\,\alpha\,\gamma, \qquad \mathcal{C}(\alpha) = (\Sigma \setminus \{\#\})^+, \quad \mathcal{C}(\beta) = \mathcal{C}(\gamma) = (\Sigma \setminus \{\#\})^*,$$

where $\#$ is a separator letter. The generated language $\mathcal{L}_\mathcal{C}(Q)$ under $\mathcal{C}$ is the following:

$$L = \{w\,\#\,w_1\,w\,w_2 : w \in (\Sigma \setminus \{\#\})^+, w_1, w_2 \in (\Sigma \setminus \{\#\})^*\}.$$

Using this language L, we can perform pattern matching as follows. To test whether a target string $w_{\text{target}} \in (\Sigma \setminus \{\#\})^+$ occurs in a text $T \in (\Sigma \setminus \{\#\})^+$, it suffices to check whether $w_{\text{target}}\,\#\,T \in L$. As this example illustrates, and since **PATwRT** is strictly more expressive than both **REG** and **PAT**, developing faster algorithms for the **PATwRT** matching problem is of both practical and theoretical interest. Moreover, **PAT** and **PATwRT** can be viewed as important fragments of regular expressions with backreferences, REWB [33, 24]. Efficient matching algorithms for **PATwRT** provide efficient procedures for a non-trivial fragment of REWB and play an important role in practical text processing.

Our Contribution

To state our main results clearly, let us formalize our matching problem and its parameters. For an integer $K \geq 3$, we consider the matching problem for the pattern

$$P_K = \alpha_1\,\beta\,\alpha_2\,\beta\,\alpha_3\,\beta\cdots\beta\,\alpha_K,$$

where $\alpha_1, \ldots, \alpha_K, \beta$ are distinct variables, and only β occurs more than once in P_K. Let W be an input string of length $N := |W|$. For the type constraints $\mathcal{C}$ associated with the variables, let $m := \max\{|\mathcal{C}(\alpha_1)|, \ldots, |\mathcal{C}(\alpha_K)|, |\mathcal{C}(\beta)|\}$ be the maximum size of the representations, where $|\mathcal{C}(\nu)|$ denotes the expression size $|E|$ if given by a regular expression E, or the number of states $|Q_\mathcal{A}|$ if given by an ε-NFA $\mathcal{A}$.

▶ **Theorem 1** (Complexity with Regular-Expression Representation). *The matching problem $W \in \mathcal{L}_\mathcal{C}(P_K)$ with regular-expression types can be solved in $O((K+1)\,N^2\,m^2)$ time.*

▶ **Theorem 2** (Complexity with ε-NFA Representation). *The matching problem $W \in \mathcal{L}_{\mathcal{C}}(P_K)$ with ε-NFA types can be solved in $O((K+1)\,N^2\,m^\omega)$ time.*[1]

In particular, if K is fixed, then Theorems 1 and 2 yield $O(N^2\,m^2)$-time and $O(N^2\,m^\omega)$-time algorithms, respectively.

As we will see in Section 2, our result extends previous work in two directions. We generalize the quadratic-time result of Fernau et al. [12, 13] for untyped pattern languages to the regular-typed setting. We also extend the result of Nogami and Terauchi [29] from the restricted form $\alpha_1\beta\alpha_2\beta\alpha_3$ to the general form considered here.

Organization. In Section 2, we compare our results with previous work by Fernau et al. [12, 13] and by Nogami and Terauchi [29]. We also review existing results on pattern languages. In Section 3, we introduce the necessary preliminaries. In Section 4, we define and compute our key building block $\mathbf{Subs}_{\mathrm{occ}}$. In Section 5, we introduce and compute our data structure $\mathcal{R}$. In Section 6, we combine $\mathbf{Subs}_{\mathrm{occ}}$ and $\mathcal{R}$ to obtain a quadratic-time algorithm for the basic case $\alpha_1\beta\alpha_2\beta\alpha_3$. In Section 7, we extend this algorithm to the general form $\alpha_1\beta\alpha_2\beta\alpha_3\beta\cdots\beta\alpha_K$.

2 Related Work

We first review the result of Fernau et al. [12, 13] for (untyped) pattern languages.

▶ **Theorem** ([13, Theorem 4.2]; [12, Lemma 2]). The matching problem for $\mathrm{PAT}_{\mathrm{rvar}} \leq 1$ is solvable in $O(N^2 + N|P|)$ time, where $|P|$ is the length of the input untyped pattern P.

In their notation, $\mathrm{PAT}_{\mathrm{rvar}} \leq 1$ denotes the class of patterns with at most one repeated variable. Their result addresses the *untyped* case of this restriction. The present paper studies the regular-typed matching problem for patterns of the form $\alpha_1\beta\alpha_2\beta\cdots\beta\alpha_K$ and shows that, for fixed K, it is solvable in quadratic time in N. In this sense, our result can be viewed as a regular-typed extension of the quadratic-time result of Fernau et al. [13, 12].

We next review the result of Nogami and Terauchi [29].

▶ **Theorem** ([29, Theorem 5 (Recap.)]). The matching problem for REWB of the form $e_0\,(e)_1\,e_1\,\backslash 1\,e_2$, where e_0, e, e_1, e_2 are regular expressions, can be solved in $O(N^2\,M^2)$ time, where $M := |e_0| + |e| + |e_1| + |e_2|$.

They consider regular expressions *with backreferences* (REWB) $e_0\,(e)_1\,e_1\,\backslash 1\,e_2$, rather than (regular-typed) pattern languages. However, this expression is equivalent to the following simple instance of **PATwRT**:

$$\text{the pattern: } \alpha_1\,\beta\,\alpha_2\,\beta\,\alpha_3 \quad \text{where } \mathcal{C}(\alpha_1) = e_0,\ \mathcal{C}(\beta) = e,\ \mathcal{C}(\alpha_2) = e_1,\ \mathcal{C}(\alpha_3) = e_2.$$

It should be noted that they only consider the very restricted form $\alpha_1\beta\alpha_2\beta\alpha_3$ and do not consider the general case $\alpha_1\,\beta\,\alpha_2\,\beta\,\alpha_3\,\beta\,\cdots\,\beta\,\alpha_K$. Since $O(M^2) = O(m^2)$ holds for the parameter M in their statement, our result extends theirs to the general form. Furthermore, our approach is technically simpler and relies only on elementary tools such as suffix tries.

[1] We recall that the constant ω is the exponent of fast Boolean matrix multiplication: given two $n \times n$ Boolean matrices M_1 and M_2, we can compute their matrix product M_1M_2 in $O(n^\omega)$ time. The current value of ω is < 2.371339 [3].

For REWB, Schmid shows that every fixed REWB expression can be simulated by a two-way multihead finite automaton (2WMFA) [33, Lemma 18]. Since the language class of 2WMFA coincides with **NL** [19], the language class of REWB is contained in **NL**. Schmid also gives an automata-theoretic characterization of REWB by introducing *memory automata (MFA)* [34]. Freydenberger and Schmid consider a deterministic subclass of REWB (*DRX*) and give an efficient linear-time matching algorithm [17].

It is natural to employ regular languages as types. Similar systems have been studied from several directions [10, 30], [6, Definition 4.7]. It should be noted that Câmpeanu and Yu's formalization [10] permits nesting of pattern variables, and thus is different from standard typed pattern languages (cf. [10, Example 1]).

The general matching problem – deciding, for an input pattern P and a word w, if $w \in_? \mathcal{L}(P)$ – is **NP**-complete even without types [4, 11, 22, 27]. This **NP**-hardness does not contradict the above quadratic-time algorithm by Fernau et al. [13, 12], since it concerns the general matching problem, whereas their (and our) algorithm treats the number of repeating variables as a fixed parameter. Fernau and Schmid [14] refine the **NP**-completeness by analyzing bounded-parameter restrictions (such as the number of variables and the alphabet size $|\Sigma|$) and determining when they remain **NP**-complete or not. Fernau et al. [15] show that many natural parameterizations are $\mathbf{W}[\mathbf{1}]$-hard. Under ETH (Exponential Time Hypothesis, cf. [20]), they also rule out algorithms with subexponential dependence on the number of variables even when $|\Sigma|$ is bounded. For a comprehensive survey on matching patterns with variables, refer to [26].

As mentioned in Introduction, the language class of patterns **PAT** is incomparable with the language class **REG** (the class of regular languages) and **CFL** (the class of context-free languages) [4, Theorem 3.10]. Besides, the language class **PAT** is contained in **NL** [6, 23]. Jain et al. studied conditions under which a given pattern belongs to **REG** (and **CFL**) [21]. By their results, it can be shown that a pattern $P_1 = \alpha\beta\beta\alpha$ forms a *non*-context-free language [21, Lemma 11] (if $|\Sigma| \geq 2$). On the other hand, a similar pattern $P_2 = \alpha\beta\beta\gamma$ generates a regular language: indeed, $\mathcal{L}(P_2) = \Sigma^*$. Reidenbach and Schmid studied the setting $|\Sigma| = 2$ or $|\Sigma| = 3$ [31]. They showed that the language of a (fixed) pattern varies significantly by changing the alphabet size $|\Sigma|$. For a pattern $Q = \alpha\,0\,\beta\,\beta\,1\,\gamma$, (1) if $\Sigma = \{0, 1\}$, then $\mathcal{L}(Q)$ is regular; however, (2) if $\Sigma = \{0, 1, 2\}$, then $\mathcal{L}(Q)$ is not regular [31, Proposition 9].

3 Preliminaries

Strings and Languages

Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ be a finite alphabet. We write $|\Sigma|$ for its size. In this paper, we regard Σ as fixed. Hence, $|\Sigma|$ is treated as a constant in time complexity.

A word (string) over Σ is a finite sequence of symbols from Σ. The special symbol ε denotes the empty word. For a word w, we write $|w|$ for its length; in particular, $|\varepsilon| = 0$. We write $w_1 \cdot w_2$ (or simply $w_1 w_2$) for the concatenation of words w_1 and w_2. We extend concatenation to sets of words (languages) as follows: $A \cdot B := \{w_1 \cdot w_2 : w_1 \in A, w_2 \in B\}$ for two languages A and B. We write Σ^i for the set of all words of length i over Σ and write Σ^* for the set of all words over Σ: $\Sigma^* := \bigcup_{i \geq 0} \Sigma^i$ with $\Sigma^0 = \{\varepsilon\}$. We write $w[i]$ for the i-th letter of w. We use 1-based indexing; thus, $w[1]$ is the first letter. We write $w[i\,..\,j]$ to denote the substring $w[i]w[i+1]\cdots w[j]$ and $w[i\,..\,j)$ to denote $w[i]w[i+1]\cdots w[j-1]$. In particular, $w[i\,..\,j] = \varepsilon$ with $j < i$ and $w[i\,..\,j) = \varepsilon$ with $j \leq i$. To denote a prefix, we write $w[\,..\,i)$ instead of $w[1\,..\,i)$. To denote a suffix, we write $w[i\,..\,)$ instead of $w[i\,..\,|w|\,]$.

Boolean Matrices over the Boolean Semiring

Let $\mathbb{B} = (\{0,1\}, \vee, \wedge)$ be the Boolean semiring, where 0 and 1 represent false and true respectively, addition is the Boolean-OR $\vee$, and multiplication is the Boolean-AND $\wedge$.

Let M_1 and M_2 be $n \times n$ Boolean matrices. To denote the Boolean matrix multiplication, we write $M_1 \otimes M_2$. As mentioned in Introduction, we can compute $M_1 \otimes M_2$ in $O(n^\omega)$ time. In particular, it is known that $\omega < 2.371339$ [3].

We write $M_1 \oplus M_2$ for the elementwise Boolean OR of M_1 and M_2. It is clear that we can compute $M_1 \oplus M_2$ in $O(n^2)$ time.

Regular Expressions

Regular expressions over Σ are inductively defined by the following grammar:

$$E ::= \sigma \mid \varepsilon \mid \emptyset \mid E + E \mid E \cdot E \mid E^*$$

where $\sigma \in \Sigma$ is a symbol. We write $\textsc{Regex}(\Sigma)$ to denote the set of regular expressions over Σ.

The language $\mathcal{L}(\bullet) \subseteq \Sigma^*$ is defined inductively in the usual way:

$$\mathcal{L}(\sigma) = \{\sigma\},\ \mathcal{L}(\varepsilon) = \{\varepsilon\},\ \mathcal{L}(\emptyset) = \emptyset,$$
$$\mathcal{L}(E_1 + E_2) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2),\ \mathcal{L}(E_1 \cdot E_2) = \mathcal{L}(E_1) \cdot \mathcal{L}(E_2),\ \mathcal{L}(E^*) = (\mathcal{L}(E))^*.$$

For $E \in \textsc{Regex}(\Sigma)$, we write $|E|$ to denote the size (i.e., the number of symbols) of E.

ε-NFAs (Nondeterministic Finite Automata with ε-transitions)

An ε-NFA $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, \mathcal{E}, q_0, F)$ where Q is a finite set of states, Σ is an input alphabet, $\Delta \subseteq Q \times \Sigma \times Q$ is a set of transition rules, $\mathcal{E} \subseteq Q \times Q$ is the set of ε transitions, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of final (accepting) states. We write $|\mathcal{A}|$ to denote the number of its states $|Q_\mathcal{A}|$.

For each transition $(p, \sigma, q) \in \Delta$, we write $p \xrightarrow{\sigma} q$. For $(p, q) \in \mathcal{E}$, we write $p \xrightarrow{\varepsilon} q$. A rule $p \xrightarrow{\sigma} q$ means that we can move from p to q by consuming the letter σ. We write $p \overset{\varepsilon}{\Rightarrow} q$ if q is reachable from p by a (possibly empty) sequence of ε-transitions: $p \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} q$. We also write $p \overset{\sigma}{\Rightarrow} q$ if we can move from p to q while consuming exactly one letter σ: i.e., $\exists p', p'' \in Q.\ p \overset{\varepsilon}{\Rightarrow} p' \xrightarrow{\sigma} p'' \overset{\varepsilon}{\Rightarrow} q$. We naturally extend this notation to arbitrary words $w \in \Sigma^*$ as $p \overset{w}{\Rightarrow} q$. Then, the language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of words accepted by $\mathcal{A}$, formally defined as:

$$\mathcal{L}(\mathcal{A}) := \{w \in \Sigma^* : \exists q \in F.\ q_0 \overset{w}{\Rightarrow} q\}.$$

By well-known Thompson's construction, every regular expression E can be translated in linear time into a sparse ε-NFA $\mathcal{A}_E$.

▶ **Proposition 3** (Thompson's construction [35];[32, Sec. 5]). *Every regular expression E can be translated into a* sparse *ε-NFA $\mathcal{A}_E = (Q, \Sigma, \Delta, \mathcal{E}, q_0, F)$ in $O(|E|)$ time such that $\mathcal{L}(E) = \mathcal{L}(\mathcal{A}_E)$. Moreover, $|Q|, |\Delta|, |\mathcal{E}| \in O(|E|)$.*

Pattern Languages

Let Σ be a finite alphabet, $\mathcal{V} = \{\alpha, \beta, \gamma, \cdots\}$ be a set of variables, disjoint from Σ. A *pattern* P is a string of $\Sigma \cup \mathcal{V}$, $P \in (\Sigma \cup \mathcal{V})^*$. For example, $P = 0\,1\,\alpha\,\beta\,\alpha\,d\,e\,f$ is a pattern where $\alpha, \beta \in \mathcal{V}$ are variables and $0, 1, d, e, f \in \Sigma$ are symbols. To instantiate the variables in a given pattern, we use *assignments* $\varphi : \mathcal{V} \to \Sigma^*$. We naturally extend φ to the monoid homomorphism $\hat{\varphi} : (\Sigma \cup \mathcal{V})^* \to \Sigma^*$ by letting $\hat{\varphi}(\sigma) = \sigma$ for every $\sigma \in \Sigma$. For example, when $\varphi(\alpha) = 1111$ and $\varphi(\beta) = 010$, then the above pattern P is instantiated as follows:

$$\hat{\varphi}(P) = 0 \cdot 1 \cdot 1111 \cdot 010 \cdot 1111 \cdot d \cdot e \cdot f.$$

The language generated by P is

$$\mathcal{L}(P) := \{\hat{\varphi}(P) : \varphi : \mathcal{V} \to \Sigma^* \text{ is an assignment } \}.$$

Note that we consider the erasing semantics of pattern languages, where variables can be replaced by the empty word ε.

Typed and Regular-Typed Pattern Languages

Typed pattern languages are an extension of pattern languages with constraints on variables. In plain pattern languages as defined above, we cannot restrict the domains of words assigned to variables. For example, let us consider generating the language $\{w\#w : w \in \Sigma^*, w \text{ does not contain } \#\}$ where $\# \in \Sigma$. However, the plain pattern $Q = \alpha\#\alpha$ generates $\mathcal{L}(Q) = \{w\#w : w \in \Sigma^*\}$, and in particular $\#\#\# \in \mathcal{L}(Q)$.

For practical applications, however, it is often useful to restrict the word domains for variables. To this end, in typed pattern languages, we introduce *constraints (types)* on the variables. Although the original papers introducing typed pattern languages allow a very general class of types [36, 25], in this paper, we use regular languages as types.

We write $\mathcal{C}(\nu)$ for the type assigned to ν, represented either by a regular expression over Σ or by an ε-NFA: i.e., $\mathcal{C}(\nu)$ is a regular expression E or an ε-NFA $\mathcal{A}$. Thus, $|\mathcal{C}(\nu)|$ means the size $|E|$ of the assigned expression E or the number of states $|\mathcal{A}|(= |Q_{\mathcal{A}}|)$ of $\mathcal{A}$.

An assignment $\varphi : \mathcal{V} \to \Sigma^*$ is *proper* with respect to $\mathcal{C}$ if $\varphi(\nu) \in \mathcal{L}(\mathcal{C}(\nu))$ for all variables ν occurring in P. The language generated by a regular-typed pattern P under $\mathcal{C}$ is

$$\mathcal{L}_{\mathcal{C}}(P) := \{\hat{\varphi}(P) : \varphi \text{ is proper with respect to } \mathcal{C}\}.$$

For example, let $\Sigma = \{\#, \ldots\}$ and $\mathcal{V} = \{\alpha\}$, and again consider $Q = \alpha\#\alpha$. Using the type $\mathcal{C}(\alpha) = (\Sigma \setminus \{\#\})^*$, we obtain $\mathcal{L}_{\mathcal{C}}(Q) = \{w\#w : w \in (\Sigma \setminus \{\#\})^*\}$, and $\#\#\# \notin \mathcal{L}_{\mathcal{C}}(Q)$.

4 Our Key Data Structure

Let W be the string used throughout this section, and let $N := |W|$ denote its length. We assume that the alphabet Σ is fixed in advance and is not part of the input.

In this section, we introduce our key data structure, $\mathbf{Subs}_{\mathrm{occ}}$, which stores, for every (distinct) non-empty substring of W, all of its occurrences in sorted order. Our construction is very simple and in fact uses only *suffix tries* (rather than suffix trees) to build $\mathbf{Subs}_{\mathrm{occ}}$ in $O(N^2)$ time.

4.1 Sorted Occurrences and Size Bound

Let $\mathbf{Subs}(W)$ (or simply, $\mathbf{Subs}$) be the set of *nonempty* substrings of W. In this paper, a *substring* is a contiguous factor of W. Namely, for $S = abcde$, bcd is a substring of S whereas ace is not.

Notation for Substrings

For any substring u of W (i.e., $u \in \mathbf{Subs}(W)$), we introduce the following notation:

- $\textsc{Occ}_W(u)$ denotes the set of all the occurrences of u in W. That is, the set of starting positions i with $1 \le i \le N - |u| + 1$ such that $W[i\,..\,i + |u|\,) = u$.
- $\textsc{Socc}_W(u) = [i_1, i_2, \ldots, i_n]$ denotes the sorted list of $\textsc{Occ}_W(u)$ in ascending order, so $i_1 < i_2 < \cdots < i_n$. In other words, $\textsc{Socc}_W(\bullet)$ is the sorted version of $\textsc{Occ}_W(\bullet)$.

If there is no ambiguity, we simply write $\textsc{Occ}(u)$ and $\textsc{Socc}(u)$ instead of $\textsc{Occ}_W(u)$ and $\textsc{Socc}_W(u)$, respectively.

▶ **Example.**

$$\text{If} \quad W = \begin{array}{c|cccccccccc} \text{index} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \hline \text{letter} & a & a & b & a & a & b & a & b & b & a \end{array}, \qquad \text{then } \textsc{Socc}(ab) = [2, 5, 7].$$

The following straightforward proposition bounds the total size of all occurrence lists, which corresponds to the total number of all substring occurrences in W.

▶ **Proposition 4.**

$$\sum_{u \in \mathbf{Subs}} |\textit{Occ}(u)| = \sum_{u \in \mathbf{Subs}} |\textit{Socc}(u)| = \Theta(N^2).$$

We can show this bound by a simple calculation. The proof can be found in Appendix A.

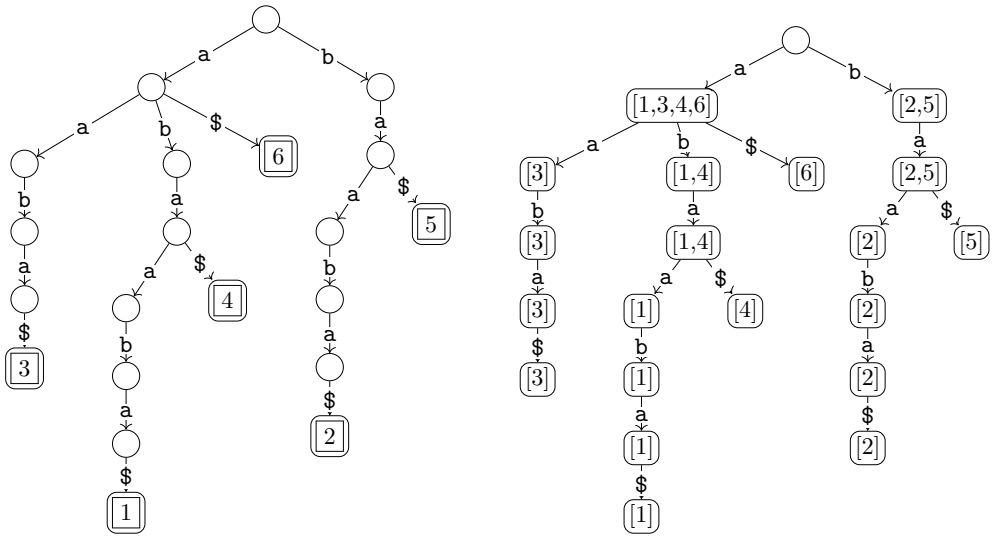
4.2 Building Our Data Structure in $O(N^2)$ Time

While $\mathbf{Subs}(W)$ is just the set of nonempty substrings of W, our actual data structure is

$$\mathbf{Subs}_{\text{occ}}(W) := \{\langle u, \textsc{Socc}(u)\rangle : u \in \mathbf{Subs}(W)\}.$$

We now show how to compute this data structure in $O(N^2)$ using the *suffix trie* of W. Before delving into the formal algorithm, we illustrate our approach.

Illustration of Our Approach. Our strategy is to augment the standard suffix trie of W into a decorated version, where each node is labeled with the sorted occurrence list of the substring it represents. The following example illustrates this target structure. The left tree is the (plain) suffix trie of $W\$ = \overset{1}{a}\overset{2}{b}\overset{3}{a}\overset{4}{a}\overset{5}{b}\overset{6}{a}\$$ constructed by our procedure, where $\$ \notin \Sigma$ is the standard extra terminal symbol used in suffix tries. The right tree is its decorated version, where each non-root node n is labeled with $\textsc{Socc}_{W\$}(\Gamma(n))$. Here $\Gamma(n)$ denotes the substring represented by node n.



Suffix Trie

We briefly recall the standard definition of suffix tries [9, 8]. A suffix trie of W is a rooted tree of maximum out-degree $|\Sigma| + 1$ representing all suffixes of the string $W_\$:= W\$$, where $\$ \notin \Sigma$ is a unique terminal symbol. The suffix trie can be constructed by the following procedure:

1. Initialize $\mathscr{T}$ to be the empty tree whose root is unlabeled.
2. For every suffix $s = a_1 a_2 \dots a_k\$$ of $W_\$$, traverse $\mathscr{T}$ from the root using the letters $a_1, a_2, \dots, a_k, \$$ in this order. When traversing, if there is no outgoing edge labeled by the current letter from the current node, add such an edge to a fresh child node.
3. Assign an integer $\mathsf{Idx}(n_\ell)$ to each leaf node n_ℓ, denoting the starting index of the suffix represented by n_ℓ in $W_\$$.

We write $\mathcal{T}_W$ to denote this suffix trie built for $W_\$$. We can construct $\mathcal{T}_W$ in $O(N^2)$ time.

▶ **Proposition 5** (Known Fact). *The suffix trie $\mathcal{T}_W$ can be constructed in $O(N^2)$ time.*

For every node n of $\mathcal{T}_W$, we write $\Gamma(n)$ for the string spelled on the path from the root to n. Namely, if root $\xrightarrow{\sigma_1} n_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_k} n$, then $\Gamma(n) = \sigma_1\sigma_2\cdots\sigma_k$. In particular, for every *leaf* node n_ℓ labeled with $\mathsf{Idx}(n_\ell)$, the string $\Gamma(n_\ell)$ is the suffix of $W_\$$ starting at $\mathsf{Idx}(n_\ell)$.

Decorated Suffix Trie

We define the decorated suffix trie $\widetilde{\mathcal{T}_W}$ as the suffix trie $\mathcal{T}_W$ augmented with an additional label $\Upsilon_{socc}(n)$ for every non-root node n, satisfying $\Upsilon_{socc}(n) = \textsc{Socc}_{W_\$}(\Gamma(n))$.

▶ **Lemma 6.** *We can transform $\mathcal{T}_W$ into the corresponding $\widetilde{\mathcal{T}_W}$ in $O(N^2)$ time.*

Proof. We traverse $\mathcal{T}_W$ by a DFS in postorder.

When visiting a leaf node n_ℓ of $\mathcal{T}_W$, we simply set $\Upsilon_{socc}(n_\ell) := [\,\mathsf{Idx}(n_\ell)\,]$.

Next, consider an internal node n. Suppose that n has k children $\{(\sigma_1, c_1), (\sigma_2, c_2), \dots, (\sigma_k, c_k)\}$ where $\sigma_i \in \Sigma \cup \{\$\}$ and the pair (σ_i, c_i) means that there exists an edge $n \xrightarrow{\sigma_i} c_i$ in $\mathcal{T}_W$. Then, $\Gamma(c_i) = \Gamma(n)\sigma_i$ holds.

In this postorder DFS, all lists $\Upsilon_{socc}(c_i)$ for $c_i \in \mathit{children}(n) = \{c_1, \ldots, c_k\}$ have already been computed when n is visited. Hence, we build $\Upsilon_{socc}(n)$ as follows:

$$\Upsilon_{socc}(n) := \mathtt{merge}(\Upsilon_{socc}(c_1), \Upsilon_{socc}(c_2), \ldots, \Upsilon_{socc}(c_k)).$$

Here $\Upsilon_{socc}(n)$ indeed equals $\textsc{Socc}_{W_\$}(\Gamma(n))$, because every occurrence of $\Gamma(n)$ in $W_\$$ is followed by exactly one character, which determines a unique child $c_i \in \mathit{children}(n)$. In particular, the lists $\Upsilon_{socc}(c_1), \ldots, \Upsilon_{socc}(c_k)$ are pairwise disjoint. Since each $\Upsilon_{socc}(c_i)$ is a sorted list, we can efficiently merge them using the well-known k-way merge (or k-pointer) technique. Because $k \le |\Sigma| + 1 = O(1)$, the time needed for this merging is $O\big(\sum_{i=1}^{k} |\Upsilon_{socc}(c_i)|\big)$.

Summing over all internal nodes, the total merge cost (i.e., the running time) is $\Theta(N^2)$ as follows:

$$\sum_{n:\text{internal node}} \sum_{c \in \mathit{children}(n)} |\Upsilon_{socc}(c)| = \sum_{n:\text{nodes}, n \neq \text{root}} |\Upsilon_{socc}(n)| = \sum_{u \in \mathbf{Subs}(W_\$)} |\textsc{Socc}_{W_\$}(u)| = \Theta(N^2).$$

For the last equation, we apply Proposition 4 to $W_\$$. Since $|W_\$| = N + 1$, this yields $\Theta(N^2)$. ◀

To build our goal $\mathbf{Subs}_{\mathrm{occ}}(W)$, we only need to visit every internal node except the root node of $\widetilde{\mathscr{T}_W}$ because the sentinel \$ appears only at the leaves, meaning that the internal nodes (excluding the root) exactly correspond to all nonempty substrings of W. This leads to the main result of this section.

▶ **Lemma 7.** *We can build* $\mathbf{Subs}_{occ}(W)$ *in* $O(N^2)$ *time.*

5 Data Structure for Range Query

Let W be the string used throughout this section and let $N := |W|$ denote its length.

Let $\mathcal{A} = (Q, \Sigma, \Delta, \mathcal{E}, q_0, F)$ be an ε-NFA. Recall that Δ is the set of labeled transitions $p \xrightarrow{\sigma} q$ and $\mathcal{E}$ is the set of ε-transitions $p \xrightarrow{\varepsilon} q$.

In this section, we show an efficient way to compute a Boolean table $\mathcal{R}[i][j][p][q]$ that satisfies the following: for $1 \le i \le j \le N$ and $p, q \in Q$,

$$\mathcal{R}[i][j][p][q] = 1 \iff q \text{ is reachable from } p \text{ by consuming the substring } W[i..j].$$

Preprocessing.

- For $\mathcal{E}$, we build its adjacency list $\textsc{Adj}_{\mathcal{E}}$ in $O(|\mathcal{E}|)$ time and adjacency matrix $\mathcal{M}_{\mathcal{E}}$ in $O(|Q|^2)$ time.
- For Δ, we build its adjacency list $\textsc{Adj}_{\Delta}(\sigma)$ in $O(|\Delta|)$ time and adjacency matrix $\mathcal{M}_{\Delta}(\sigma)$ in $O(|Q|^2)$ time for each $\sigma \in \Sigma$.

5.1 Computing $\mathcal{R}$ for Sparse NFAs

Let $S \subseteq Q$ be a set of states. We represent S as a Boolean array indexed by states, so that membership queries $q \in S$ can be answered in $O(1)$ time.

We first consider the following two functions:

$$\varepsilon\text{-}\mathsf{closure}(S) := \{q \in Q : \exists p \in S.\ p \overset{\varepsilon}{\Rightarrow} q\}, \quad \mathsf{move}(S, \sigma) := \{q \in Q : \exists p \in S.\ p \xrightarrow{\sigma} q\}.$$

(1) We can compute ε-$\mathsf{closure}(S)$ in $O(|Q| + |\mathcal{E}|)$ time as follows. We consider the directed graph $G_{\mathcal{E}} := (Q, \mathcal{E})$ consisting only of ε-transitions. Then ε-$\mathsf{closure}(S)$ is exactly the set of vertices reachable from the source set S in $G_{\mathcal{E}}$. Hence, using the adjacency list $\textsc{Adj}_{\mathcal{E}}$

of $G_{\mathcal{E}}$, we can compute ε-$\mathsf{closure}(S)$ by a standard multi-source graph search: initialize the queue with all states in S, mark them as visited, and traverse only ε-edges. By the standard linear-time analysis of BFS on adjacency-list graphs [7, Chapter 20], this takes $O(|Q| + |\mathcal{E}|)$ time. (2) We can compute $\mathsf{move}(S, \sigma)$ in $O(|\Delta|)$ time by scanning the labeled transition set Δ once and collecting all targets q such that $(\sigma, q) \in \textsc{Adj}_{\Delta}(p)$ with $p \in S$.

We now consider the following function that computes the set $\{q \in Q : \exists p \in S.\ p \overset{\sigma}{\Longrightarrow} q\}$:

$\mathsf{consume}(S, \sigma) := \varepsilon\text{-}\mathsf{closure}(\mathsf{move}(\varepsilon\text{-}\mathsf{closure}(S), \sigma))$.

It can be computed in $O(|Q|+|\mathcal{E}|+|\Delta|)$ time. This construction and time bound are standard in the context of efficient ε-NFA simulation; see, e.g., [2, Section 3.7].

Now we can compute the table $\mathcal{R}$ in $O(N^2 \cdot |Q| \cdot (|Q| + |\mathcal{E}| + |\Delta|))$ time as follows. We assume all entries of $\mathcal{R}$ are initialized to 0.

```
for each i ∈ [1 .. N] do
    for each p ∈ Q do
        for each q ∈ consume({p}, W[i]) do
            R[i][i][p][q] := 1;
    for each j ∈ [(i + 1) .. N] do
        for each p ∈ Q do
            S := {q ∈ Q : R[i][j − 1][p][q] = 1};
            for each q ∈ consume(S, W[j]) do
                R[i][j][p][q] := 1;
```

▶ **Proposition 8.** *We can compute $\mathcal{R}$ for $\mathcal{A}$ in $O(N^2 \cdot |Q| \cdot (|Q| + |\mathcal{E}| + |\Delta|))$ time.*

5.2 Computing $\mathcal{R}$ for Dense NFAs

When the ε-NFA is dense (i.e., $|\Delta| + |\mathcal{E}| = \Theta(|Q|^2)$), the above procedure for $\mathcal{R}$ requires $O(N^2|Q|^3)$ time. Here, for dense ε-NFAs, we give an alternative $O(N^2\,|Q|^{\omega})$-time algorithm.

Computing $\widetilde{\mathcal{E}}$ in $O(|Q|^{\omega})$ time. We first compute the ε-reachability relation $\widetilde{\mathcal{E}} \subseteq Q \times Q$ as a $|Q| \times |Q|$ Boolean matrix in $O(|Q|^{\omega})$ time. Note that computing $\widetilde{\mathcal{E}}$ by running the above **ε-closure**$(\{q\})$ for every $q \in Q$ would take $O(|Q|^2 + |Q||\mathcal{E}|)$ time, which becomes cubic when $|\mathcal{E}| = \Theta(|Q|^2)$.

Observe that $\widetilde{\mathcal{E}}$ is the reflexive transitive closure of $\mathcal{M}_{\mathcal{E}}$. Namely, $\widetilde{\mathcal{E}} = \mathcal{M}_{\mathcal{E}}^*$ holds. Now, we use the following classical result for efficiently computing $\mathcal{M}_{\mathcal{E}}^*$. On the Boolean semiring $\mathbb{B}$, for a given $n \times n$ Boolean matrix M, we can compute the reflexive transitive closure M^* of M in $O(n^{\omega})$ time [28, 18, 16, 1]. This leads to the following property.

▶ **Proposition 9** (Known Result [28, 18, 16];[1, § 5.9])**.** *We can compute $\widetilde{\mathcal{E}}$ in $O(|Q|^{\omega})$ time.*

Computing $\mathcal{R}$. Using $\widetilde{\mathcal{E}}$, we can compute $\mathcal{R}$ in $O(N^2|Q|^{\omega})$ time as follows.

For each $\sigma \in \Sigma$, we pre-compute $\widetilde{\Delta}_{\sigma} := \widetilde{\mathcal{E}} \otimes \mathcal{M}_{\Delta}(\sigma) \otimes \widetilde{\mathcal{E}}$ in $O(|Q|^{\omega})$ time. By definition, $\widetilde{\Delta}_{\sigma}[p][q] = 1$ iff $p \overset{\sigma}{\Longrightarrow} q$. We then run the following $O(N^2\,|Q|^{\omega})$-time DP algorithm. For notational convenience, we identify $\mathcal{R}[i][j]$ with the $|Q| \times |Q|$ Boolean matrix whose (p, q)-entry is $\mathcal{R}[i][j][p][q]$.

```
for i ∈ [1 .. N] do
    R[i][i] := Δ̃_{W[i]};
    for j ∈ [(i+1) .. N] do
        R[i][j] := R[i][j−1] ⊗ Δ̃_{W[j]};
```

This procedure leads to the following time complexity.

▶ **Proposition 10.** *We can compute $\mathcal{R}$ for $\mathcal{A}$ in $O(N^2\,|Q|^\omega)$ time.*

6 Quadratic Time Algorithm for $\alpha_1\,\beta\,\alpha_2\,\beta\,\alpha_3$

In this section, we consider the matching problem for the pattern $\alpha_1\,\beta\,\alpha_2\,\beta\,\alpha_3$. Namely, given an input string W, we decide whether there exists a decomposition

$$W = w_1\,w_\beta\,w_2\,w_\beta\,w_3 \quad \text{such that } w_i \in \mathcal{L}(\mathcal{C}(\alpha_i)) \text{ for } i \in \{1,2,3\} \text{ and } w_\beta \in \mathcal{L}(\mathcal{C}(\beta)).$$

In Section 7, we extend the algorithm presented here to the more general patterns $\alpha_1\,\beta\,\alpha_2\,\beta\,\alpha_3\,\beta\cdots\beta\,\alpha_K$.

Notation

We first set up the notation used in this section. Let W be an input string and $N := |W|$.

We work uniformly with ε-NFA representations: for $i \in \{1,2,3\}$, let $\mathcal{A}_i$ denote the ε-NFA for $\mathcal{C}(\alpha_i)$, and let $\mathcal{B}$ denote the ε-NFA for $\mathcal{C}(\beta)$. If a type is given by a regular expression, we first replace it by the equivalent ε-NFA obtained by Thompson's construction (Proposition 3).

Let $m = \max\{\,|\mathcal{C}(\alpha_1)|, |\mathcal{C}(\alpha_2)|, |\mathcal{C}(\alpha_3)|, |\mathcal{C}(\beta)|\}$ and $m_Q := \max(|Q_{A_1}|, |Q_{A_2}|, |Q_{A_3}|, |Q_B|)$. In the regular expression case, Thompson's construction yields $m_Q = O(m)$. In the ε-NFA case, by definition, $m_Q = m$.

For any ε-NFA $\mathcal{A} = (Q, \Sigma, \Delta, \mathcal{E}, q_0, F)$, once the table $\mathcal{R}_\mathcal{A}$ has been computed using Proposition 8 or 10, we define a Boolean table $\mathcal{ACC}_\mathcal{A}[i][j]$ for $1 \le i \le j \le N$ by:

$$\mathcal{ACC}_\mathcal{A}[i][j] = 1 \iff \mathcal{A} \text{ accepts the substring } W[i\,..\,j].$$

Given $\mathcal{R}_\mathcal{A}$, the table $\mathcal{ACC}_\mathcal{A}$ can be built in $O(N^2\,|Q|)$ time by checking, for each $1 \le i \le j \le N$, whether there exists $q \in F$ with $\mathcal{R}_\mathcal{A}[i][j][q_0][q] = 1$.

We also use half-open interval notation for these tables. For $i \le j$, we write $\mathcal{R}_\mathcal{A}[i\ldots j)$ for $\mathcal{R}_\mathcal{A}[i][j-1]$; in particular, $\mathcal{R}_\mathcal{A}[i\ldots i)$ is the ε-reachability matrix. Similarly, we write $\mathcal{ACC}_\mathcal{A}[i\ldots j)$ for $\mathcal{ACC}_\mathcal{A}[i][j-1]$ when $i < j$. We set $\mathcal{ACC}_\mathcal{A}[i\,..\,i) = 1$ iff $\varepsilon \in \mathcal{L}(\mathcal{A})$.

Finally, for substring u of W, we write $\textsc{Socc}_u$ for $\textsc{Socc}(u)$ and $\#u$ for $|\textsc{Socc}_u|$. For $1 \le x \le \#u$, we write x° for $\textsc{Socc}_u[x]$.

Preprocessing Cost

Building $\mathbf{Subs}_{\mathrm{occ}}(W)$ takes $O(N^2)$ time by Lemma 7.

Once the four tables $\mathcal{R}_{\mathcal{A}_1}, \mathcal{R}_{\mathcal{A}_2}, \mathcal{R}_{\mathcal{A}_3}, \mathcal{R}_\mathcal{B}$ have been computed, building the corresponding four $\mathcal{ACC}$ tables takes the following time in total:

$$O\big(N^2\big(|Q_{\mathcal{A}_1}| + |Q_{\mathcal{A}_2}| + |Q_{\mathcal{A}_3}| + |Q_\mathcal{B}|\big)\big) \le O(N^2 m_Q).$$

If the input types $\mathcal{C}(\nu)$ are given by regular expressions, then by Thompson's construction (Proposition 3), each type is translated into a sparse ε-NFA with $O(|\mathcal{C}(\nu)|)$ states and transitions. Hence, by Proposition 8, computing the four $\mathcal{R}$-tables takes the following time:

$$O\big(N^2(|Q_{\mathcal{A}_1}|^2 + |Q_{\mathcal{A}_2}|^2 + |Q_{\mathcal{A}_3}|^2 + |Q_\mathcal{B}|^2)\big) \le O(N^2 m_Q^2).$$

If the input types are given by ε-NFAs, then by Proposition 10, computing the four $\mathcal{R}$-tables takes the following time:

$$O\big(N^2(|Q_{\mathcal{A}_1}|^\omega + |Q_{\mathcal{A}_2}|^\omega + |Q_{\mathcal{A}_3}|^\omega + |Q_{\mathcal{B}}|^\omega)\big) \le O(N^2 m_Q^\omega).$$

Therefore, using $m_Q = O(m)$ in the regular-expression case and $m_Q = m$ in the ε-NFA case, we obtain the following time bound.

▶ **Proposition 11.** *The total preprocessing time is $O(N^2m^2)$ when the types are given by regular expressions, and $O(N^2m^\omega)$ when they are given by ε-NFAs.*

6.1 Our Procedure

Concentrating on $w_\beta \neq \varepsilon$. We first handle the easy case $w_\beta = \varepsilon$. If $\varepsilon \in \mathcal{L}(\mathcal{B})$, let $\mathcal{A}'$ be the ε-NFA obtained by concatenating $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$, where $|\Xi_{\mathcal{A}'}| = O(|\Xi_{\mathcal{A}_1}| + |\Xi_{\mathcal{A}_2}| + |\Xi_{\mathcal{A}_3}|)$ for $\Xi \in \{Q, \Delta, \mathcal{E}\}$. Then deciding whether $W \in \mathcal{L}(\mathcal{A}')$ is just the standard membership problem for ε-NFAs, which can be solved in $O(N \cdot (|Q_{\mathcal{A}'}| + |\Delta_{\mathcal{A}'}| + |\mathcal{E}_{\mathcal{A}'}|))$ time by standard ε-NFA simulation [2, Section 3.7]. If this test succeeds, we return **MATCH**.

Therefore, in the remainder of this section, we focus on the non-empty case.

Our Strategy. Our overall strategy is as follows:

1. For each pair $\langle u, \textsc{Occ}_u\rangle \in \mathbf{Subs}_{\mathrm{occ}}(W)$, we treat u as a candidate for w_β.
2. Check whether $u \in \mathcal{L}(\mathcal{B})$ in $O(1)$ time by picking arbitrary $i \in \textsc{Occ}_u$ and testing $\mathcal{ACC}_{\mathcal{B}}[i\,..\,i+|u|)$.
3. If $u \in \mathcal{L}(\mathcal{B})$, we decide whether there exists a valid decomposition $W = w_1\,u\,w_2\,u\,w_3$ in $O(\,\#u\,|Q_{\mathcal{A}_2}|^2\,)$ time. This deciding procedure will be given below (Section 6.2).

The total running time over all candidates u is

$$\sum_{\langle u,\textsc{Occ}_u\rangle \in \mathbf{Subs}_{\mathrm{occ}}(W)} O(\#u\,|Q_{\mathcal{A}_2}|^2) \;=\; O\Big(|Q_{\mathcal{A}_2}|^2 \cdot \sum_{\langle u,\textsc{Occ}_u\rangle \in \mathbf{Subs}_{\mathrm{occ}}(W)} \#u\Big) =_{\text{by Prop.4}} O(N^2\,|Q_{\mathcal{A}_2}|^2).\ (\bigstar)$$

6.2 Procedure for Deciding whether $W = w_1\,u\,w_2\,u\,w_3$

For a fixed substring u of W, we give a decision procedure for deciding whether there exists such a decomposition that runs in $O(\#u\,|Q_{\mathcal{A}_2}|^2)$ time.

Before giving our procedure, we introduce an auxiliary procedure NextPhase as follows.

NextPhase(Cur, $\mathcal{A}$)

Input: Cur: $\#u$-length Boolean vector and $\mathcal{A}$: ε-NFA.
Output: Next: $\#u$-length Boolean vector that satisfies the following condition called (♦): for every $x \in [1\,..\,\#u]$,

$$\textsc{Next}[x] = 1 \iff \exists z \in [1\,..\,\#u].\ z^\circ + |u| \le x^\circ \wedge \textsc{Cur}[z] = 1 \wedge W[z^\circ + |u|\,..\,x^\circ) \in \mathcal{L}(\mathcal{A}).$$

We give a complete implementation of this procedure in Section 6.3. In particular, this procedure runs in $O(\#u\,|Q_{\mathcal{A}}|^2)$ time.

Using NextPhase, we build our procedure FindDecomposition(u) as follows:

```
1:  procedure FindDecomposition(u)
2:      check_{A_1} ← [0, 0, ..., 0];                ▷ Initialize #u-length Boolean array
3:      for all x ∈ [1 .. #u] do
4:          if ACC_{A_1}[.. x°) then
5:              check_{A_1}[x] ← 1;
6:      check_{A_2} ← NextPhase(check_{A_1}, A_2);
7:      for all x ∈ [1 .. #u] do
8:          if check_{A_2}[x] and ACC_{A_3}[x° + |u| .. ) then
9:              return 1 (as MATCH);
10:     return 0 (as UNMATCH);
```

▶ Remark. For any ε-NFA $\mathcal{A}$, if $i = N + 1$, $\mathcal{ACC}_{\mathcal{A}}[i\,..\,) = 1$ iff $\varepsilon \in \mathcal{L}(\mathcal{A})$.

By Lines 3–5, we have $\mathbf{check}_{\mathcal{A}_1}[x] = 1 \iff W[\,..\,x^\circ) \in \mathcal{L}(\mathcal{A}_1)$. We then have the following from the property (♦) of NextPhase:

$$\mathbf{check}_{\mathcal{A}_2}[x] = 1 \iff \exists z \in [1\,..\,\#u].\ z^\circ + |u| \le x^\circ \wedge W[\,..\,z^\circ) \in \mathcal{L}(\mathcal{A}_1) \wedge W[z^\circ + |u|\,..\,x^\circ) \in \mathcal{L}(\mathcal{A}_2).$$

This equivalence and Lines 7–10 immediately lead to the correctness of our procedure.

▶ **Lemma 12.** *FindDecomposition(u) returns 1 iff there exists a valid decomposition $W = w_1\,u\,w_2\,u\,w_3$ such that $w_1 \in \mathcal{L}(\mathcal{A}_1)$, $w_2 \in \mathcal{L}(\mathcal{A}_2)$, and $w_3 \in \mathcal{L}(\mathcal{A}_3)$.*

Time Complexity. The above procedure runs in $O(\#u\,|Q_{\mathcal{A}_2}|^2)$ for each fixed substring u. Hence, the total time is $O(N^2\,|Q_{\mathcal{A}_2}|^2)$ as (☆). Proposition 11 leads to the following results.

▶ **Lemma 13.** *The matching problem $W \in \mathcal{L}_{\mathcal{C}}(\alpha_1\beta\alpha_2\beta\alpha_3)$ can be solved in $O(N^2\,m^2)$ time when the types are given by regular expressions, and in $O(N^2 m^\omega)$ time when they are given by ε-NFAs.*

6.3 Implementation and Properties of NextPhase

A naive implementation of NextPhase checks condition (♦) for every pair (z, x) with $1 \le z < x \le \#u$. This takes $O(\#u^2)$ time, which is slow for our target bound.

To avoid this quadratic dependence on $\#u$, our approach is based on a monotone two-pointer scan. Let $\mathbf{Init}_{\mathcal{A}}$ be the $|Q_{\mathcal{A}}|$-length Boolean vector representing the ε-closure of the initial state of $\mathcal{A}$.

```
1:  procedure NextPhase(Cur, A)
2:      y ← 1;
3:      states ← [0, 0, ..., 0];                    ▷ |Q_A|-length array
4:      Next ← [0, 0, ..., 0];                      ▷ #u-length array
5:      for all x ∈ [2 .. #u] do
6:          states ← states ⊗ R_A[(x − 1)° .. x°);
7:          ▷ The invariant Inv holds here. ◁
8:          while y ≤ #u and y° + |u| ≤ x° do
9:              if Cur[y] then
10:                 states ← states ⊕ (Init_A ⊗ R_A[y° + |u| .. x°));
11:             y ← y + 1;
12:             ▷ The invariant Inv is restored here. ◁
13:         if states ∩ F_A ≠ ∅ then
14:             Next[x] := 1;
15:     return Next;
```

For a fixed target occurrence x°, after Line 6 the vector **states** already contains the contribution of all already-scanned indices $z < y$ with $\textsc{Cur}[z] = 1$, advanced from the previous endpoint $(x-1)^\circ$ to the current endpoint x°. The while-loop then scans newly eligible indices y satisfying $y^\circ + |u| \leq x^\circ$. Whenever $\textsc{Cur}[y] = 1$, we add the runs of $\mathcal{A}$ starting at $y^\circ + |u|$ and ending at x°. Hence, after the while-loop terminates, **states** represents exactly the states reachable from the initial state of $\mathcal{A}$ after reading $W[z^\circ + |u| \,..\, x^\circ)$ for all already-processed indices $z < y$ with $\textsc{Cur}[z] = 1$. We formalize this argument by the following invariant.

Invariant $\mathbf{Inv}(\mathbf{states}, x, y)$

For all states $q \in Q_{\mathcal{A}}$, the following conditions are equivalent:

- $\mathbf{states}[q] = 1$
- there is an index z such that
  - $z < y$ and $\textsc{Cur}[z] = 1$ and $z^\circ + |u| \leq x^\circ$ and
  - q is reachable from the initial state of $\mathcal{A}$ after reading $W[z^\circ + |u| \,..\, x^\circ)$.

With the invariant $\mathbf{Inv}(\mathbf{states}, x, y)$ formally defined, we now state the key preservation property of this invariant. This property serves as the cornerstone for proving the overall correctness of our algorithm.

▶ **Lemma 14.** *During the execution of the procedure* $\textsc{NextPhase}(\textsc{Cur}, \mathcal{A})$*, the invariant* $\mathbf{Inv}(\mathbf{states}, x, y)$ *holds at Line 7 and Line 12.*

The key point is that Line 6 advances all already represented runs from $(x-1)^\circ$ to x°, while the while-loop incorporates each newly eligible witness exactly once. The full proof is deferred to Appendix B.

We now use this preservation property to derive the correctness of NextPhase.

▶ **Lemma 15.** $\textsc{NextPhase}(\textsc{Cur}, \mathcal{A})$ *returns a vector* $\textsc{Next}$ *that satisfies the following condition: for every* $x \in [1 \,..\, \#u]$*,*

$$\textsc{Next}[x] = 1 \iff \exists z \in [1 \,..\, \#u].\ z^\circ + |u| \leq x^\circ \wedge \textsc{Cur}[z] = 1 \wedge W[z^\circ + |u| \,..\, x^\circ) \in \mathcal{L}(\mathcal{A}).$$

Using Lemma 14, we observe that, when the while-loop terminates, no unprocessed index can still satisfy $z^\circ + |u| \leq x^\circ$. Hence the invariant captures exactly all valid witnesses. The full proof is deferred to Appendix C.

We next analyze the running time of NextPhase.

▶ **Lemma 16.** *The time complexity of the procedure* $\textsc{NextPhase}(\textsc{Cur}, \mathcal{A})$ *is bounded by* $O(\#u \cdot |Q_{\mathcal{A}}|^2)$*, where* $|Q_{\mathcal{A}}|$ *is the number of states of* $\mathcal{A}$*.*

The key observation is that the pointer y moves only forward, so the body of the while-loop is executed at most $\#u$ times in total. The full amortized analysis is deferred to Appendix D.

7 Quadratic Time Algorithm for General Case

We now extend the construction of Section 6 to the general form $\alpha_1\beta\alpha_2\beta\cdots\beta\alpha_K$. As in Section 6, we first handle the easy case $w_\beta = \varepsilon$ separately: if $\varepsilon \in L(\mathcal{B})$, we test whether W is accepted by the ε-NFA obtained by concatenating $\mathcal{A}_1\mathcal{A}_2\cdots\mathcal{A}_K$. In the remainder of this section, we therefore focus on the case $w_\beta \neq \varepsilon$.

Procedure for a Fixed Candidate u

As a representative example, consider the pattern $\alpha_1\beta\alpha_2\beta\alpha_3\beta\alpha_4$. For a fixed nonempty candidate substring u, we extend our FindDecomposition as follows.

```
1: procedure FindDecomposition(u)
2:     check_{A_1} ← [0, 0, ..., 0];
3:     for all x ∈ [1 .. #u] do
4:         if ACC_{A_1}[ .. x°) then
5:             check_{A_1}[x] ← 1;
6:     check_{A_2} ← NextPhase(check_{A_1}, A_2);
7:     check_{A_3} ← NextPhase(check_{A_2}, A_3);
8:     for all x ∈ [1 .. #u] do
9:         if check_{A_3}[x] and ACC_{A_4}[x° + |u| .. ) then
10:            return 1;
11:    return 0;
```

By the same argument as in Lemma 12, FindDecomposition(u) returns 1 iff there exists a valid decomposition $W = w_1\,u\,w_2\,u\,w_3\,u\,w_4$ such that $w_i \in \mathcal{L}(\mathcal{A}_i)$ for $i \in \{1,2,3,4\}$.

The same construction extends immediately to the general pattern $P_K = \alpha_1\beta\alpha_2\beta\cdots\beta\alpha_K$: for a fixed candidate u,

1. First, define $\mathbf{check}_{\mathcal{A}_1}[x] = 1$ iff $W[\ ..\ x^\circ) \in \mathcal{L}(\mathcal{A}_1)$; and then
2. Next, iteratively set $\mathbf{check}_{\mathcal{A}_i} := \textsc{NextPhase}(\mathbf{check}_{\mathcal{A}_{i-1}}, \mathcal{A}_i)$ for $i = 2, \ldots, K-1$.
3. Finally, accept iff there exists x such that $\mathbf{check}_{\mathcal{A}_{K-1}}[x] = 1$ and $\mathcal{ACC}_{\mathcal{A}_K}[x^\circ + |u|\ ..\ ) = 1$.

Time Complexity

For a fixed candidate u, the procedure performs $K-2$ calls to NextPhase. Hence, by Lemma 16, its running time is $O((K-2)\,\#u\,m^2)$ where $m = \max\{|\mathcal{C}(\alpha_1)|, \ldots, |\mathcal{C}(\alpha_K)|, |\mathcal{C}(\beta)|\}$. Summing over all candidates u and using $\sum_{u\in\mathbf{Subs}(W)} \#u = O(N^2)$, we obtain $O((K-2)\,N^2\,m^2)$. Recall that, for a type $\mathcal{C}(\nu)$, if $\mathcal{C}(\nu)$ is a regular expression E, then $|\mathcal{C}(\nu)|$ is its expression size $|E|$; if $\mathcal{C}(\nu)$ is an ε-NFA $\mathcal{A}$, then $|\mathcal{C}(\nu)|$ is its number of states $|Q_{\mathcal{A}}|$.

In the regular-expression case, together with the preprocessing cost for the $K+1$ ε-NFAs $\mathcal{A}_1, \ldots, \mathcal{A}_K$ and $\mathcal{B}$, this yields $O((K+1)\,N^2\,m^2)$. In the ε-NFA case, together with the preprocessing cost, this yields $O((K+1)\,N^2\,m^\omega)$.

Now we have the following main theorems, which are restated from Introduction.

▶ **Theorem 1** (Complexity with Regular-Expression Representation). *The matching problem $W \in \mathcal{L}_{\mathcal{C}}(P_K)$ with regular-expression types can be solved in $O((K+1)\,N^2\,m^2)$ time.*

▶ **Theorem 2** (Complexity with ε-NFA Representation). *The matching problem $W \in \mathcal{L}_{\mathcal{C}}(P_K)$ with ε-NFA types can be solved in $O((K+1)\,N^2\,m^\omega)$ time.*

References

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, USA, 1974. URL: https://dl.acm.org/doi/10.5555/578775.

2 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

3 Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. More asymmetry yields faster matrix multiplication. In *SODA 2025*, pages 2005–2039. SIAM, 2025. doi:10.1137/1.9781611978322.63.

4 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.

5 Dana Angluin. Inductive inference of formal languages from positive data. *Inf. Cont.*, 45(2):117–135, 1980. doi:10.1016/S0019-9958(80)90285-5.

6 Henning Bordihn, Jürgen Dassow, and Markus Holzer. Extending regular expressions with homomorphic replacement. *RAIRO Theor. Inf. Appl.*, 44(2):229–255, 2010. doi:10.1051/ita/2010013.

7 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 4th edition*. MIT Press, 2022. URL: https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/.

8 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. doi:10.1017/CBO9780511546853.

9 Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology*. World Scientific, 2002. doi:10.1142/4838.

10 Cezar Câmpeanu and Sheng Yu. Pattern expressions and pattern automata. *Inf. Process. Lett.*, 92(6):267–274, 2004. doi:10.1016/j.ipl.2004.09.007.

11 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.

12 Henning Fernau, Florin Manea, Robert Mercaş, and Markus L. Schmid. Pattern matching with variables: Fast algorithms and new hardness results. In *STACS 2015*, volume 30 of *LIPIcs*, pages 302–315. Schloss Dagstuhl, 2015. doi:10.4230/LIPIcs.STACS.2015.302.

13 Henning Fernau, Florin Manea, Robert Mercaş, 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.

14 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.

15 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.

16 Michael J. Fischer and Albert R. Meyer. Boolean matrix multiplication and transitive closure. In *SWAT 1971*, pages 129–131. IEEE Computer Society, 1971. doi:10.1109/SWAT.1971.4.

17 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.

18 M. E. Furman. Application of a method of rapid multiplication of matrices to the problem of finding the transitive closure of a graph. *Doklady Akademii Nauk SSSR*, 194(3):524, 1970. URL: https://www.mathnet.ru/eng/dan35686.

19 Juris Hartmanis. On non-determinancy in simple computing devices. *Acta Informatica*, 1(4):336–344, 1972. doi:10.1007/BF00289513.

20 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001. doi:10.1006/jcss.2001.1774.

21 Sanjay Jain, Yuh Shin Ong, and Frank Stephan. Regular patterns, regular languages and context-free languages. *Inf. Process. Lett.*, 110(24):1114–1119, 2010. doi:10.1016/j.ipl.2010.09.010.

22 Tao Jiang, Efim Kinber, Arto Salomaa, Kai Salomaa, and Sheng Yu. Pattern languages with and without erasing. *Int. J. Comp. Math.*, 50(3–4):147–163, 1994. doi:10.1080/00207169408804252.

23 Neil D. Jones and Sven Skyum. Recognition of deterministic ETOL languages in logarithmic space. *Inf. Control.*, 35(3):177–181, 1977. doi:10.1016/S0019-9958(77)90058-4.

24 Vladimir Komendantsky. Matching problem for regular expressions with variables. In *TFP 2012*, pages 149–166. Springer, 2013. doi:10.1007/978-3-642-40447-4_10.

25 Takeshi Koshiba. Typed pattern languages and their learnability. In *EuroCOLT 1995*, volume 904 of *LNCS*, pages 367–379. Springer, 1995. doi:10.1007/3-540-59119-2_192.

26 Florin Manea and Markus L. Schmid. Matching patterns with variables. In Robert Mercaş and Daniel Reidenbach, editors, *Combinatorics on Words*, pages 1–27. Springer, 2019. doi:10.1007/978-3-030-28796-2_1.

27 Alexandru Mateescu and Arto Salomaa. Aspects of classical language theory. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages, Volume 1: Word, Language, Grammar*, pages 175–251. Springer, 1997. doi:10.1007/978-3-642-59136-5_4.

28 Ian Munro. Efficient determination of the transitive closure of a directed graph. *Inf. Process. Lett.*, 1(2):56–58, 1971. doi:10.1016/0020-0190(71)90006-8.

29 Taisei Nogami and Tachio Terauchi. Efficient matching of some fundamental regular expressions with backreferences. In *MFCS 2025*, volume 345 of *LIPIcs*, pages 81:1–81:19. Schloss Dagstuhl, 2025. doi:10.4230/LIPIcs.MFCS.2025.81.

30 Dirk Nowotka and Max Wiedenhöft. The equivalence problem of E-pattern languages with length constraints is undecidable. In *CPM 2025*, volume 331 of *LIPIcs*, pages 4:1–4:23. Schloss Dagstuhl, 2025. doi:10.4230/LIPIcs.CPM.2025.4.

31 Daniel Reidenbach and Markus L. Schmid. Regular and context-free pattern languages over small alphabets. *Theor. Comput. Sci.*, 518:80–95, 2014. doi:10.1016/j.tcs.2013.07.035.

32 Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009. doi:10.1017/CBO9781139195218.

33 Markus L. Schmid. Inside the class of regex languages. *Int. J. Found. Comput. Sci.*, 24(7):1117–1134, 2013. doi:10.1142/S0129054113400340.

34 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.

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

36 Keith Wright. Inductive identification of pattern languages with restricted substitutions. In *COLT 1990*, pages 111–121. Morgan Kaufmann, 1990. URL: http://dl.acm.org/citation.cfm?id=92595.

A Proof of Proposition 4

▶ **Proposition 4.**

$$\sum_{u \in \mathbf{Subs}} |\mathrm{Occ}(u)| = \sum_{u \in \mathbf{Subs}} |\mathrm{Socc}(u)| = \Theta(N^2).$$

Proof. Let h be an integer with $1 \leq h \leq N$, and write $\mathbf{Subs}_h$ for the set of h-length substrings: $\mathbf{Subs}_h := \{u \in \mathbf{Subs} : |u| = h\}$.

For all positions i with $1 \leq i \leq N - h + 1$, there is a unique h-length substring $u = W[i\,..\,i+h)$. Thus, the multiset of all occurrences of all h-length substrings is in one-to-one correspondence with the set of all possible starting positions $\{1, \ldots, N-h+1\}$. Therefore, $\sum_{u \in \mathbf{Subs}_h} |\mathrm{Occ}(u)| = N - h + 1$ holds.

Summing over all lengths $h = 1, 2, \ldots, N$, we obtain

$$\sum_{u \in \mathbf{Subs}} |\mathrm{Occ}(u)| = \sum_{h=1}^{N} (N - h + 1) = \frac{N(N+1)}{2} = \Theta(N^2).$$

Since $|\mathrm{Socc}(u)| = |\mathrm{Occ}(u)|$ by our definition, the above equation completes the proof. ◀

B Proof of Lemma 14

▶ **Lemma 14.** *During the execution of the procedure* NextPhase($\textsc{Cur}, \mathcal{A}$)*, the invariant* **Inv**(**states**$, x, y$) *holds at Line 7 and Line 12.*

(Line 7 and Line 12 refer to the numbered comment lines in the pseudocode.)

Proof. Fix an outer-loop index $x \in [2 \,..\, \#u]$. We show that **Inv**(**states**$, x, y$) holds at the following two evaluation points:

(1) Line 7: immediately before each test of the while-condition in the x-th iteration; and

(2) Line 12: immediately after the increment of y inside the while-loop.

For the base case, consider the first outer-loop iteration $x = 2$. At the first evaluation point (Line 7), Line 6 has already been executed. Since **states** was initialized to the all-zero vector, multiplying it by $\mathcal{R}_{\mathcal{A}}[(x-1)^\circ \,..\, x^\circ)$ still yields the zero vector. Besides, $y = 1$. As there is no index $z < 1$, both sides of the equivalence are false for every state q.

Next, assume the invariant holds after termination of the while-loop in the $(x-1)$-st outer iteration. Then Line 6 multiplies **states** by $\mathcal{R}_{\mathcal{A}}[(x-1)^\circ \,..\, x^\circ)$, which advances every currently represented run from endpoint $(x-1)^\circ$ to endpoint x°. Moreover, since the previous while-loop terminated only when no further eligible index remained, every index $z < y$ already satisfies $z^\circ + |u| \leq (x-1)^\circ$. Therefore, at Line 7, all runs already represented in **states** are correctly advanced to the new endpoint x°, and the corresponding witnesses $z < y$ remain valid. Hence the invariant also holds at the first evaluation point (Line 7) in the x-th iteration.

Now assume that the invariant holds before some test of the while-condition, and suppose the condition is true. If $\textsc{Cur}[y] = 0$, then Line 10 is skipped, **states** is unchanged, and incrementing y does not add any new witness; hence the invariant continues to hold. If $\textsc{Cur}[y] = 1$, then Line 10 adds exactly the states reachable from the initial state of $\mathcal{A}$ after reading $W[y^\circ + |u| \,..\, x^\circ)$, i.e., it adds precisely the missing witness $z = y$. After the increment of y, the invariant is restored in the form with $z < y$.

Thus the invariant holds at all claimed program points. ◀

C Proof of Lemma 15

▶ **Lemma 15.** NextPhase($\textsc{Cur}, \mathcal{A}$) *returns a vector* $\textsc{Next}$ *that satisfies the following condition: for every* $x \in [1 \,..\, \#u]$*,*

$$\textsc{Next}[x] = 1 \iff \exists z \in [1 \,..\, \#u].\ z^\circ + |u| \leq x^\circ \wedge \textsc{Cur}[z] = 1 \wedge W[z^\circ + |u| \,..\, x^\circ) \in \mathcal{L}(\mathcal{A}).$$

Proof. First, consider the base case $x = 1$. For $x = 1$, in the right-hand side of (♦), any candidate witness z satisfies $z \geq 1$, hence $z^\circ \geq x^\circ$. Since $|u| \geq 1$, we have $z^\circ + |u| > x^\circ$. Therefore the right-hand side of (♦) is false. Since $\textsc{Next}[1] = 0$ holds, (♦) also holds.

Next, fix an outer-loop index $x \in [2 \,..\, \#u]$. By Lemma 14, the invariant **Inv**(**states**$, x, y$) holds at every evaluation point of the while-loop: namely, immediately before each test of the while-condition (Line 7), and, whenever the loop body is executed, immediately after the increment of y (Line 12). Therefore, when the while-loop terminates, the invariant still holds: if the loop body has been executed at least once, this is given by the last occurrence of Line 12; otherwise, it is given by the first occurrence of Line 7.

Observe the termination condition of the **while**-loop. It terminates when either $y > \#u$ or $y^\circ + |u| > x^\circ$. Because the sequence of occurrence indices $z \mapsto z^\circ$ is strictly increasing, this termination condition guarantees that no index $z \geq y$ can satisfy $z^\circ + |u| \leq x^\circ$. In other words, any index $z \in [1 \,..\, \#u]$ satisfying $z^\circ + |u| \leq x^\circ$ must strictly satisfy $z < y$.

Therefore, we can safely drop the restriction $z < y$ from the invariant. It follows that for every state $q \in Q_{\mathcal{A}}$, we have $\mathbf{states}[q] = 1$ if and only if there exists an index $z \in [1\,..\,\#u]$ such that $\textsc{Cur}[z] = 1$, $z^\circ + |u| \leq x^\circ$, and q is reachable from the initial state of $\mathcal{A}$ after reading $W[z^\circ + |u|\,..\,x^\circ)$.

At Line 14, the procedure sets $\textsc{Next}[x] = 1$ if and only if $\mathbf{states} \cap F_{\mathcal{A}} \neq \emptyset$. This intersection is non-empty exactly when there exists an accepting state $q \in F_{\mathcal{A}}$ such that $\mathbf{states}[q] = 1$. Combining this with the above equivalence, we obtain:

$$
\begin{aligned}
& \textsc{Next}[x] = 1 \\
\iff\ & \exists q \in F_{\mathcal{A}}.\ \mathbf{states}[q] = 1 \\
\iff\ & \exists z \in [1\,..\,\#u].\ z^\circ + |u| \leq x^\circ \wedge \textsc{Cur}[z] = 1 \\
\wedge\ & \exists q \in F_{\mathcal{A}}.\ q \text{ is reachable from the initial state of } \mathcal{A} \text{ after reading } W[z^\circ + |u|\,..\,x^\circ) \\
\iff\ & \exists z \in [1\,..\,\#u].\ z^\circ + |u| \leq x^\circ \wedge \textsc{Cur}[z] = 1 \wedge W[z^\circ + |u|\,..\,x^\circ) \in \mathcal{L}(\mathcal{A}).
\end{aligned}
$$

Together with the base case $x = 1$, this establishes (♦) for all $x \in [1\,..\,\#u]$. ◀

D $O(\#u\,|Q_{\mathcal{A}}|^2)$-time Complexity Analysis of NextPhase

We show the following about $\textsc{NextPhase}(\textsc{Cur}, \mathcal{A})$.

▶ **Lemma 16.** *The time complexity of the procedure $\textsc{NextPhase}(\textsc{Cur}, \mathcal{A})$ is bounded by $O(\#u \cdot |Q_{\mathcal{A}}|^2)$, where $|Q_{\mathcal{A}}|$ is the number of states of $\mathcal{A}$.*

Proof. Recall that every precomputed $\mathcal{R}_{\mathcal{A}}[a\,..\,b)$ can be accessed in $O(1)$ time.

We break down the cost of each operation as follows.

1. Initialization Phase. Lines 2–4 perform the necessary initializations. Initializing the index y takes $O(1)$ time. The Boolean array states, which acts as a state configuration vector, is initialized to all zeros in $O(|Q_{\mathcal{A}}|)$ time. Similarly, the Boolean array $\textsc{Next}$ of length $\#u$ is initialized to all zeros in $O(\#u)$ time. Thus, the total initialization cost is $O(\#u + |Q_{\mathcal{A}}|)$.

2. Main Loop and Vector-Matrix Multiplications. The (outer) for-loop iterates $\#u - 1$ times. In each iteration, we perform the following key operations:

- *State Extension (Line 6):* We compute the multiplication of a Boolean row vector states of size $|Q_{\mathcal{A}}|$ and a Boolean matrix $\mathcal{R}_{\mathcal{A}}$ of size $|Q_{\mathcal{A}}| \times |Q_{\mathcal{A}}|$. Using standard matrix multiplication over the Boolean semiring, this operation takes $O(|Q_{\mathcal{A}}|^2)$ time per iteration. Over the entire for-loop, this contributes $O(\#u \cdot |Q_{\mathcal{A}}|^2)$ to the total running time.
- *Acceptance Check (Lines 13–14):* Checking the intersection $\mathbf{states} \cap F_{\mathcal{A}} \neq \emptyset$ corresponds to a bitwise AND operation between two Boolean arrays of size $|Q_{\mathcal{A}}|$, followed by a check if any bit is set. This takes $O(|Q_{\mathcal{A}}|)$ time per iteration, contributing $O(\#u \cdot |Q_{\mathcal{A}}|)$ overall. Assigning 1 to $\textsc{Next}[x]$ takes $O(1)$ time.

3. Amortized Analysis of the While Loop. The while-loop incrementally adds the contributions of newly eligible y with $\textsc{Cur}[y] = 1$ to $\mathbf{states}$. Although it is nested within the for loop, the index variable y is initialized to 1 and is only incremented at Line 11. Because y is never decreased and can reach at most $\#u$, the body of the while loop is executed at most $\#u$ times in total across all iterations of the outer for loop. The total number of evaluations of the while-condition is also $O(\#u)$, which does not affect the final bound.

Inside the while loop, the array lookup $\textsc{Cur}[y]$ takes $O(1)$ time. The dominant operation is Line 10, where we again multiply the Boolean vector $\mathbf{Init}_{\mathcal{A}}$ by the precomputed matrix $\mathcal{R}_{\mathcal{A}}[y^\circ + |u|\,..\,x^\circ)$ and compute the element-wise Boolean OR ($\oplus$) with the current states

array. The vector-matrix multiplication takes $O(|Q_{\mathcal{A}}|^2)$ time, and the Boolean OR takes $O(|Q_{\mathcal{A}}|)$ time. Since this inner body runs at most $\#u$ times globally, the aggregated time spent inside the while loop over the entire procedure is strictly bounded by $O(\#u \cdot |Q_{\mathcal{A}}|^2)$.

Summing the costs of initialization, the outer loop operations, and the globally bounded inner loop operations, the overall time complexity is:

$$O\big(\overbrace{\#u + |Q_{\mathcal{A}}|}^{\text{Lines 2–4}} + \overbrace{\#u \cdot |Q_{\mathcal{A}}|^2}^{\text{total Line 6}} + \overbrace{\#u \cdot |Q_{\mathcal{A}}|^2}^{\text{total Line 10}} + \overbrace{\#u \cdot |Q_{\mathcal{A}}|}^{\text{total Line 13}} \big) = O(\#u \cdot |Q_{\mathcal{A}}|^2). \qquad \blacktriangleleft$$