Simon's Congruence Pattern Matching

Sungmin Kim
Department of Computer Science, Yonsei University, Seoul, Republic of Korea

Sang-Ki Ko
Department of Computer Science & Engineering, Kangwon National University, Chuncheon-si, Republic of Korea

Yo-Sub Han[1]
Department of Computer Science, Yonsei University, Seoul, Republic of Korea

Abstract

Testing Simon's congruence asks whether two strings have the same set of subsequences of length no greater than a given integer. In the light of the recent discovery of an optimal linear algorithm for testing Simon's congruence, we solve the Simon's congruence pattern matching problem. The problem requires finding all substrings of a text that are congruent to a pattern under the Simon's congruence. Our algorithm efficiently solves the problem in linear time in the length of the text by reusing results from previous computations with the help of new data structures called X-trees and Y-trees. Moreover, we define and solve variants of the Simon's congruence pattern matching problem. They require finding the longest and shortest substring of the text as well as the shortest subsequence of the text which is congruent to the pattern under the Simon's congruence. Two more variants which ask for the longest congruent subsequence of the text and optimizing the pattern matching problem are left as open problems.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases pattern matching, Simon's congruence, string algorithm, data structure

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2022.60

Funding This research was supported by the NRF grant (NRF-2020R1A4A3079947) and the AI Graduate School Program (No. 2020-0-01361) funded by the Korea government (MSIT).

1 Introduction

In the realm of string algorithms, subsequences are extensively studied with a lot of applications [3, 10, 14, 16]. Problems related to subsequences can be formulated using the string equivalence operation, which has many perks such as linear pattern matching time by the KMP algorithm [12] and linear construction of suffix trees [17]. For example, the longest common subsequence problem requires finding the longest subsequence of a string w_1 that is equal to some subsequence of a string w_2. Variants of the longest common subsequence such as the longest common increasing subsequence [1, 4] can also be interpreted as finding the longest common subsequence between three strings, where a string w_3 is obtained by sorting string $w_1 w_2$. Meanwhile, recent findings on a congruence relation made it feasible to construct new problems that capture stronger properties related to subsequences. This congruence relation is called Simon's congruence defined by Simon in his study of piecewise testable languages [15]. The relation is based on the equivalence of the subsequence set rather than the equivalence of individual subsequences.

Given a positive integer k and a string w, let $\mathbb{S}_k(w)$ denote the set of all subsequences of w with length no more than k. We say that two strings w_1 and w_2 are $\sim_k$-congruent if the subsequence sets $\mathbb{S}_k(w_1)$ and $\mathbb{S}_k(w_2)$ are equal. For example, for $k = 2$, strings $w_1 = ababb$

[1] Corresponding author

© Sungmin Kim, Sang-Ki Ko, and Yo-Sub Han;
licensed under Creative Commons License CC-BY 4.0

and $w_2 = baba$ satisfy $w_1 \sim_2 w_2$ because the subsequence sets $\mathbb{S}_k(w_1)$ and $\mathbb{S}_k(w_2)$ are both $\{aa, ab, ba, bb, a, b, \lambda\}$. On the other hand, if $k = 3$, then there exists a string $u = abb$ that is a subsequence of w_1 but not a subsequence of w_2. Thus, we have $w_1 \not\sim_3 w_2$. The Simon's congruence decision problem asks whether or not $w_1 \sim_k w_2$.

For binary strings, Hébrard [9] presented an $O(|w_1| + |w_2|)$ algorithm. For an arbitrary alphabet Σ, Garel [7] suggested an $O(|\Sigma||w_1|)$ algorithm when $w_1 = w_2\sigma$ for a character σ; namely, w_2 is obtained from w_1 by removing the last character σ. Fleischer and Kufleitner [5] designed a normalization algorithm that reduces all $\sim_k$-congruent strings into a single string called the ShortLex normal form string and tested Simon's congruence with the string equivalence operation in $O(|\Sigma|(|w_1| + |w_2|))$ time. Thus, until recently, testing Simon's congruence for general conditions was a superlinear process, which was a giant hurdle in formulating general problems.

Recently, Barker et al. [2] improved Fleischer and Kufleitner's normalization algorithm and obtained a linear time algorithm. Later, Gawrychowski et al. [8] proposed data structures that help find the maximum k value such that $w_1 \sim_k w_2$ in linear time. Since the decision and the optimization problems for Simon's congruence can be solved in linear time, we are now ready to apply Simon's congruence to practical contexts such as pattern matching.

While Gawrychowski and his co-authors [8] suggested an optimal algorithm for optimizing k for Simon's congruence, they also proposed three open problems, where two are based on Simon's congruence and the other is based on a variant of Simon's congruence. The first problem, named LangSimK, is a membership testing problem. Given a set S of strings, which is either regular or context-free, a string w and an integer k, the problem asks to decide if there is a string $x \in S$ such that $w \sim_k x$. Kim et al. [11] proved that the problem is NP-complete in general and presented an efficient algorithm when the alphabet size is fixed.

▶ **Problem 1** (Gawrychowski et al. [8]). *The remaining two problems are as follows:*

- *Simon's Congruence Pattern Matching (MatchSimK): Given a pattern* `P`*, a text* `T`*, and an integer k, find all substrings of* `T` *that are $\sim_k$-congruent to* `P`*.*
- *Subsequence Set Inclusion Problem (SubseqSetInclusion): Given two strings w_1 and w_2, find the maximum integer k such that $\mathbb{S}_k(w_1) \subseteq \mathbb{S}_k(w_2)$.*

We tackle MatchSimK from the string pattern matching perspective, and design efficient algorithms.

Our contributions

We solve MatchSimK in linear time in the size of a text `T`. Our linear-time algorithm relies on data structures called X-trees and Y-trees that reduce the number of testings and reuse the computation results of the previous computations. Then, we study possible variants of MatchSimK with different objectives: The first two variants are called the longest congruent substring problem (LCongStrK) and the shortest congruent substring problem (SCongStrK).

▶ **Problem 2.** *Since we aim to extend problems based on the string equivalence operation into problems that use Simon's congruence, we first modify the longest common substring problem to use Simon's congruence.*

- *Longest Congruent Substring (LCongStrK): Given a pattern* `P`*, a text* `T`*, and an integer k, find a longest substring of* `T` *that is $\sim_k$-congruent to* `P`*.*
- *Shortest Congruent Substring (SCongStrK): Given a pattern* `P`*, a text* `T`*, and an integer k, find a shortest substring of* `T` *that is $\sim_k$-congruent to* `P`*.*

LCongStrK immediately extends the longest common substring problem by specifying that the returned substring of T should be $\sim_k$-congruent to P, instead of being equal to some substring of P. On the other hand, SCongStrK is interesting because its string equivalence counterpart, the shortest common substring problem, is nonsensical. The next set of problems, the longest congruent subsequence problem (LCongSeqK) and the shortest congruent subsequence problem (SCongSeqK), modifies LCongStrK and SCongStrK to consider subsequences as follows:

▶ **Problem 3.** *The listed problems also extend the longest common subsequence problem.*

- *Longest Congruent Subsequence (LCongSeqK): Given a pattern* P*, a text* T*, and an integer k, find a longest subsequence of* T *that is $\sim_k$-congruent to* P*.*
- *Shortest Congruent Subsequence (SCongSeqK): Given a pattern* P*, a text* T*, and an integer k, find a shortest subsequence of* T *that is $\sim_k$-congruent to* P*.*

We solve the LCongStrK, SCongStrK SCongSeqK problems, and leave the LCongSeqK problem as an open problem.

2 Preliminaries

String Notations

For a string w over an alphabet Σ, $\texttt{alph}(w)$ denotes the set of characters that appears in w. The length $|w|$ of w is the number of characters in w. We denote the empty string as λ. Given a string $w = w[0]w[1]\cdots w[n-1]$ over Σ such that $w[i] \in \Sigma$ for $0 \le i < n$, we assign a *space position* of w from the space right before $w[0]$ as 0 toward the space right after $w[n-1]$ as n; namely, we map $0, 1, \ldots, n$ to the corresponding positions. Then, we call $w[i]$ the corresponding character of a space position i. Given two space positions i, j such that $i \le j$, we define a string $w[i]w[i+1]\ldots w[j-1]$ to be a *substring* of w, which is denoted by $w[i:j]$. For convenience, we express the last character of w as $w[-1]$ and the substring $w[0:|w|-1]$ as $w[:-1]$. A string $u = w[a_1]w[a_2]\cdots w[a_i]$ is a *subsequence* of w if $0 \le a_1 < a_2 < \cdots < a_i \le |w|-1$. The substring relation is written as $u \prec_s w$ and the subsequence relation is written as $u \prec w$. The set of subsequences of length at most k is denoted by $\mathbb{S}_k(w) = \{u \in \Sigma^* \mid u \prec w, |u| \le k\}$. Finally, the *reversal* $w^R = w[|w|-1]w[|w|-2]\cdots w[2]w[1]w[0]$ of string w is the string obtained by concatenating characters of w in the reverse order of w.

Rankers and Coordinates

A function *ranker* is defined as follows: given a string w, a space position i of w, and a character $\sigma \in \Sigma$, $R(w, i, \sigma)$ returns a space position of w whose corresponding character is the nearest occurrence of σ from i towards a specific direction [5, 13, 18]. There are two types of ranker directions; from left to right (X-type) and from right to left (Y-type). Thus, an *X-ranker* $R_X(w, i, \sigma)$ returns a space position $j > i$ if $w[j-1] = \sigma$ and the substring $w[i:j-1]$ does not contain σ. If such a position does not exist, $R_X(w, i, \sigma) = \infty$. Similarly, a *Y-ranker* $R_Y(w, i, \sigma)$ returns a space position $j < i$ if $w[j] = \sigma$ and the substring $w[j+1:i]$ does not contain σ. When the output is not defined, then $R_Y(w, i, \sigma) = -1$. For example, for $w = aabcbabc$, we have $R_X(w, 2, a) = 6$ since $w[2:5] = bcb$ does not contain a and $w[5] = a$.

A *ranker chain* $R(w, i, x)$ is a ranker that takes a string instead of a single character for its third argument. Ranker chains can be recursively defined as $R_X(w, R_X(w, i, x[0]), x[1:|x|])$ or $R_Y(w, R_Y(w, i, x[-1]), x[:-1])$. Semantically, $R_X(w, 0, x)$ refers to the smallest space

position j such that $x \prec w[0:j]$. Similarly, $R_Y(w, |w|, x)$ is the largest space position j such that $x \prec w[j:|w|]$. For example, the X-ranker value for $w = ababaab$, $i = 0$, and $x = abb$ is $R_X(w, 0, abb) = R_X(w, 1, bb) = R_X(w, 2, b) = 4$.

Fleischer and Kufleitner [5] defined the *X- and Y-coordinates* of space position i of string w to be the length of the shortest string x such that $R_X(w, 0, x) = i + 1$ and $R_Y(w, |w|, x) = i$, respectively. We denote each as $X(w, i)$ and $Y(w, i)$. For the same example $w = ababaab$, we have $X(w, 6) = 3$ because $x = bbb$ is the shortest string such that $R_X(w, 0, x) = 6$. Kim et al. [11] extended the notion of coordinates by defining *X- and Y-vectors*, which are Σ-indexed arrays of potential X- and Y-coordinates if such a character is inserted at space position i. Formally, $\overrightarrow{X}(w, i)[\sigma] = X(w[0:i]\sigma w[i:|w|], i)$ and $\overrightarrow{Y}(w, i)[\sigma] = Y(w[0:i]\sigma w[i:|w|], i)$. Computing $\overrightarrow{X}(w, i{+}1)$ from $\overrightarrow{X}(w, i)$ or computing $\overrightarrow{Y}(w, i)$ from $\overrightarrow{Y}(w, i{+}1)$ is completed in $|\Sigma|$ time by incrementing the cell corresponding to character $w[i]$ by one and then setting the value of all cells that exceed the value at cell $w[i]$ to that value. Recall our example $w = ababaab$. We have $\overrightarrow{X}(w, 6) = [5, 3]$ because $X(ababaaab, 6) = 5$ and $X(ababaabb, 6) = 3$ If we compute $\overrightarrow{X}(w, 7)$ from $\overrightarrow{X}(w, 6)$, we first set $\overrightarrow{X}(w, 7)[b] = \overrightarrow{X}(w, 6)[b] + 1$ because $w[6] = b$. Since $\overrightarrow{X}(w, 6)[a] = 5$ exceeds $\overrightarrow{X}(w, 7)[b] = 4$, we set $\overrightarrow{X}(w, 7)[a] = \overrightarrow{X}(w, 7)[b]$. We refer to this single-step vector computation as one *iteration* of the vector computation and denote the function as $\text{Iter}(\overrightarrow{v}, \sigma)$ for a vector $\overrightarrow{v}$ and a character σ. Also, for an integer i we define a *uniform vector* $\overrightarrow{U}(i)$ as the vector with all cell values equal to i. Vector addition and subtraction follow the general definition for vector operations. Barker et al. [2] defined the *k-universality* as follows: A string w is k-universal if $\mathbb{S}_k(w)$ is the set of all strings of length no greater than k. An example of a 3-universal string over $\Sigma = \{a, b, c\}$ is $abcbacccba$.

Arch Factorization

Given a string w, the jth *arch* of w, written as $\texttt{ar}_j(w)$, is the minimal substring of w that starts at the end of the $j-1$th arch and satisfies $\texttt{alph}(\texttt{ar}_i(w)) = \Sigma$. We denote the sum of the lengths of arches 1 to i by $\texttt{ArchSum}(i, w) = \sum_{j=1}^{i} |\texttt{ar}_j(w)|$ and define $\texttt{ArchSum}(0, w) = 0$. If no such space position $j > \texttt{ArchSum}(i, w)$ satisfies $\texttt{alph}(w[\texttt{ArchSum}(i, w) : j]) = \Sigma$, then the integer i, which is the number of arches of w, becomes the *universality index* $\iota(w)$ [2] of w. The universality index refers to the maximum k for which w is k-universal. We call the string $w[\texttt{ArchSum}(\iota(w), w) : |w|]$ – the suffix of w that starts at the space position $\texttt{ArchSum}(\iota(w), w)$ – the *rest*, and denote it as $\texttt{rest}(w)$. Finally, w can be decomposed as $\texttt{ar}_1(w)\texttt{ar}_2(w) \cdots \texttt{ar}_{\iota(w)}(w)\texttt{rest}(w)$. This factorization scheme is called the *arch factorization* of w [9]. The arch factorization defines the *modus* of a string such that $\texttt{modus}(w) = \texttt{ar}_1(w)[-1]\texttt{ar}_2(w)[-1] \cdots \texttt{ar}_{\iota(w)}(w)[-1]$. For example, if we have $w = aacabccbcbaacbcbc$ over $\Sigma = \{a, b, c\}$, we find $\texttt{ar}_1(w) = aacab$, $\texttt{ar}_2(w) = ccbcba$, and $\texttt{ar}_3(w) = acb$. Since there cannot be any more arches, we set $\iota(w) = 3$ and $\texttt{rest}(w) = cbc$. Thus, we have $w = \texttt{ar}_1(w)\texttt{ar}_2(w)\texttt{ar}_3(w)\texttt{rest}(w)$ and $\texttt{modus}(w) = bab$. Note that we can apply arch factorization to w^R as well as w. We call the arches of w and w^R the *X-arches* and *Y-arches* of w, respectively. From a pair of overlapping X-arch and Y-arch, we construct an *arch link*, which is the maximal substring of w where the two arches overlap. They are called arch links because X- and Y-arches are chained together by a series of arch links. There are two types of arch links. We define *YX-links* of w as the substrings $w[\texttt{ArchSum}(i, w) : |w| - \texttt{ArchSum}(\iota(w) - i, w^R)]$ for all nonnegative integers $i \le \iota(w)$. Likewise, we define *XY-links* of w as the substrings $w[|w| - \texttt{ArchSum}(\iota(w) - i, w^R) : \texttt{ArchSum}(i{+}1, w)]$ for all nonnegative integers $i < \iota(w)$. For YX-links, the Y-arch comes before the X-arch, while for XY-links, the X-arch comes before the Y-arch. Using our example $w = aacabccbcbaacbcbc$, the arch factorization of the

reverse of w results in $\texttt{ar}_1(w^R)^R = acbcbc$, $\texttt{ar}_2(w^R)^R = cba$, $\texttt{ar}_3(w^R)^R = abccb$, and finally $\texttt{rest}(w^R)^R = aac$. Thus, we have YX-links $w[0:3]$, $w[5:8]$, $w[11:11]$, and $w[14:17]$. On the other hand, XY-links of w include $w[3:5]$, $w[8:11]$, and $w[11:14]$. A similar factorization scheme was used by Fleischmann et al. [6] as well.

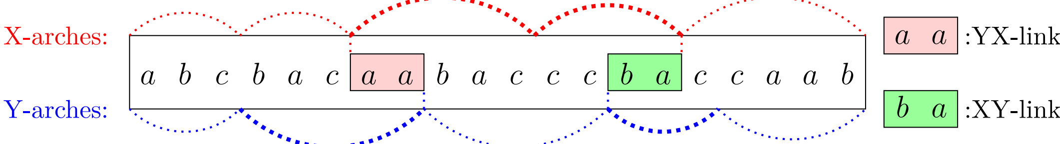

Figure 1 An illustration of X- and Y-arches of the string *abcbacaabacccbaccaab*. An XY-link, a YX-link, and the arches that produce them are highlighted to emphasize the difference between the two types of links.

Simon's Congruence

Given two strings w_1 and w_2, we say w_1 and w_2 are $\sim_k$-congruent if $\mathbb{S}_k(w_1) = \mathbb{S}_k(w_2)$. The congruence class defined by $\sim_k$ and a string w is written as $\text{Closure}_k(w) = \{x \in \Sigma^* \mid x \sim_k w\}$. Each congruence class $\text{Closure}_k(w)$ has a unique ShortLex normal form (SNF) string $\texttt{ShortLex}_k(w)$ which is the lexicographically least string among the shortest strings in $\text{Closure}_k(w)$. For example, for an integer $k = 2$ and a string $w = babaabacaabba$, the SNF string of w is $abcab$. Thus, $\text{Closure}_k(w)$ is a set of strings u_1cu_2 such that u_1 and u_2 are strings over $\{a, b\}$ and both strings contain at least one a and one b. Fleischer and Kufleitner [5] presented a normalization algorithm that takes $O(|\Sigma||w|)$ time, which performs $O(|\Sigma|)$-time computations for each position of the input string. The algorithm was improved by Barker et al. [2] to run in $O(|w|)$ time.

▶ **Proposition 4** (Fleischer and Kufleitner [5]). *For a string w, an integer k, and a space position i of w, if $X(w,i) + Y(w,i) > k+1$, then $w \sim_k w[0:i]w[i+1:|w|]$. If no such i satisfies the above, then the string w is length-minimal among the $\sim_k$-congruent strings with w. Moreover, for all length-minimal strings $z \in \text{Closure}_k(w)$ and any string x with $|x| = |z|$, $x \sim_k z$ if and only if x can be obtained by repetitively rearranging characters in contiguous positions i, j such that $X(w,i) + Y(w,i) = k+1$, $Y(w,i) = Y(w,j)$ and $X(w,i) = X(w,j)$.*

Based on Proposition 4, both algorithms [2, 5] repeatedly remove characters in w to derive a length-minimal string in $\text{Closure}_k(w)$. Then, the algorithms rearrange contiguous characters that have the same pair of X- and Y-coordinates whose sums are $k+1$, and obtain a lexicographically smallest string among the shortest strings in $\text{Closure}_k(w)$. This two-step procedure is called the *ShortLex normalization algorithm.*

▶ **Proposition 5** (Barker et al. [2]). *Given an integer k and two strings w_1 and w_2, we can check whether or not $w_1 \sim_k w_2$ in $O(|w_1| + |w_2|)$ time. Moreover, for a string w, we can obtain $\texttt{ShortLex}_k(w)$ in optimal time $O(|w|)$.*

Finally, in the context of MATCHSIMK, we say that a substring w of $\texttt{T}$ is a *match* of $\texttt{P}$ if w is $\sim_k$-congruent to $\texttt{P}$.

3 Main Contributions

3.1 Simon's Congruence Pattern Matching

3.1.1 A simple algorithm

Before we design an efficient algorithm, let us consider a simple algorithm based on the number of possible matches.

▶ **Lemma 6.** *There exist a pattern* $\mathtt{P}$*, a text* $\mathtt{T}$ *and a number* k *such that the total number of matches of* $\mathtt{P}$ *is quadratic in* $|\mathtt{T}|$*.*

Note that a naive algorithm checking the congruence of all substrings of $\mathtt{T}$ would have a cubic running time in $|\mathtt{T}|$, since it takes $O(|\mathtt{T}|)$ time to check whether or not each substring is $\sim_k$-congruent to $\mathtt{P}$. With the goal of performing fewer tests of Simon's congruence, we present the following lemma.

▶ **Lemma 7.** *For a string* u *over* Σ*, let* $A = \{a \in \Sigma \mid u \sim_k ua\}$*. Then, for any string* $w \in A^*$ *and any character* $b \in \Sigma \setminus A$*, we have* $u \sim_k uw$ *and* $u \not\sim_k uwb$*.*

Lemma 7 implies that it is sufficient to check Simon's congruence exactly once for each starting space position i of $\mathtt{T}$. Specifically, we can test for a shortest match candidate $\mathtt{T}[i : m]$ ending at space position m for which the match candidate has a $\sim_k$-congruent character rearrangement of $\mathtt{ShortLex}_k(\mathtt{P})$ as a subsequence. First, assume that there exists a match $\mathtt{T}[i : l]$ for some space position l. Then, by Proposition 4, there exists a subsequence z of $\mathtt{T}[i : l]$ such that $|z| = |\mathtt{ShortLex}_k(\mathtt{P})|$ and $z \sim_k \mathtt{P}$. Thus, $l \geq m$. Moreover, if a subsequence of $\mathtt{T}[i : m]$ is $\sim_k$-congruent to $\mathtt{P}$, then we have $\mathbb{S}_k(\mathtt{P}) \subseteq \mathbb{S}_k(\mathtt{T}[i : m])$. It follows that if $\mathtt{T}[i : m] \not\sim_k \mathtt{P}$, then there exists some $u \in \mathbb{S}_k(\mathtt{T}[i : m])$ that is not a member of $\mathbb{S}_k(\mathtt{P})$, and in turn, no such l exists. Finally, if $\mathtt{T}[i : m] \sim_k \mathtt{P}$, then, we can extend our match result based on Lemma 7 to find all matches of $\mathtt{P}$ that start at i. Otherwise, there cannot be any match that starts at i.

On the other hand, we assume that $\mathtt{ShortLex}_k(\mathtt{P})$ takes the form of a stack of sets. Specifically, by Proposition 4, we can obtain $\mathtt{ShortLex}_k(\mathtt{P})$ from the shortest subsequence of $\mathtt{P}$ that is $\sim_k$-congruent to $\mathtt{P}$ by rearranging characters in substrings with indices that have the same X- and Y-coordinates whose sum equals $k + 1$. Thus, we construct a stack by grouping the rearrangeable positions together into a set and repetitively pushing the sets into a stack starting with the set with the highest indices. Note that indices that are not rearrangeable with any other index each produce a singleton set. The peek operation returns the set at the top of the stack. However, the pop operation removes a given character from the set at the top of the stack and removes the set if the resulting set is empty. The stack representation of $\mathtt{ShortLex}_k(\mathtt{P})$ is convenient for finding a subsequence of $\mathtt{T}$ that is $\sim_k$-congruent to $\mathtt{P}$.

Based on these observations, we can solve MatchSimK in quadratic time in $|\mathtt{T}|$. For each space position i of $\mathtt{T}$, we find the minimal match candidate by finding the minimum space position m such that $\mathtt{T}[i : m]$ has a $\sim_k$-congruent character rearrangement of $\mathtt{ShortLex}_k(\mathtt{P})$ as a subsequence. We use the stack representation of $\mathtt{ShortLex}_k(\mathtt{P})$ to keep the testing time linear for each iteration. Thereon, we apply Lemma 7 to obtain all matches that start at space position i.

▶ **Theorem 8.** *Given a pattern* $\mathtt{P}$*, a text* $\mathtt{T}$*, and an integer* k*, we can find all space position pairs* (f, b) *of* $\mathtt{T}$ *for which* $\mathtt{T}[f : b] \sim_k \mathtt{P}$ *in* $O(|\mathtt{T}|(|\mathtt{T}| + |\Sigma|))$ *time.*

Note that $|\Sigma|$ is usually a constant. Even without the assumption, $|\mathtt{T}|$ dominates $|\Sigma|$ and thus the bound becomes $O(|\mathtt{T}|^2)$.

3.1.2 Can we do better?

Pondering on the simple algorithm, we identify two main causes that make the algorithm quadratic in $|\mathtt{T}|$. First, we report all matches of $\mathtt{P}$ by putting every match in a set, one-by-one. Recall that Lemma 6 proves that the number of matches in the worst-case is quadratic in $|\mathtt{T}|$. However, for a given space position f, Lemma 7 proves that all space positions b such that $\mathtt{T}[f:b] \sim_k \mathtt{P}$ are contiguous. Thus, despite the quadratic worst-case lower bound on the number of matches obtained in Lemma 6, we can design a faster algorithm if, instead of listing all matching substrings, we report the intervals $[f_1, f_2]$ and $[b_1, b_2]$ where all starting and ending space positions $(f, b) \in [f_1, f_2] \times [b_1, b_2]$ satisfy $\mathtt{T}[f:b] \sim_k \mathtt{P}$.

The remaining cause of the quadratic running time of the simple algorithm is that we need to check whether the candidate string that starts at each space position of $\mathtt{T}$ is $\sim_k$-congruent to $\mathtt{P}$ every time, which leads to a quadratic runtime with respect to $|\mathtt{T}|$. Using the concept of arch factorization, we can improve the running time by reusing substrings of SNF strings of other match candidates. The following lemma illustrates how arches can be used to find reusable substrings in $\mathtt{T}$.

▶ **Lemma 9.** *For a string w, and all non-negative integers $i \le \iota(w)$, the X-vector at the right end of the ith X-arch $\mathtt{ar}_i(w)$ and the Y-vector at the left end of the ith Y-arch $\mathtt{ar}_i(w^R)^R$ are $[i+1, i+1, \ldots, i+1]$. In other words,*

$$\overrightarrow{X}(w, \mathtt{ArchSum}(i, w)) = \overrightarrow{Y}\left(w, |w| - \mathtt{ArchSum}(i, w^R)\right) = \overrightarrow{U}(i+1).$$

The key observation is that all X- or Y-vectors for space positions at the borders of every ith X- or Y-arch are static uniform vectors. Thus, for a space position i, the X- and Y-coordinates for i can be computed exclusively from the pair of X- and Y-arch which i is a member of. For example, let $w = aaaabaaaaaabaaaa$. Position $i = 8$ is covered by X-arch $\mathtt{ar}_2(w) = w[5:12]$ and Y-arch $\mathtt{ar}_2(w^R)^R = w[4:11]$. Using Lemma 9, we can directly let $\overrightarrow{X}(w, 5) = [2, 2]$ and $\overrightarrow{Y}(w, 11) = [2, 2]$. Finally, we can compute $X(w, 8) = 5$ and $Y(w, 8) = 4$ through only 6 calls of Iter(). Moreover, for a pattern $\mathtt{P}$ that is not k-universal, all matches x of $\mathtt{P}$ must satisfy $\iota(x) = \iota(\mathtt{P})$ and $\mathtt{alph}(\mathtt{rest}(x)) = \mathtt{alph}(\mathtt{rest}(\mathtt{P}))$ by the following lemma.

▶ **Lemma 10.** *For two strings w_1 and w_2, if $w_1 \sim_k w_2$ and w_1 is not k-universal, then $\iota(w_1) = \iota(w_2)$ and $\mathtt{alph}(\mathtt{rest}(w_1)) = \mathtt{alph}(\mathtt{rest}(w_2))$.*

Note that, for a k-universal pattern $\mathtt{P}$, we always have $\iota(\mathtt{P}) \ge \iota(\mathtt{ShortLex}_k(\mathtt{P})) = k$. Since we want to utilize Lemma 7 to extend borders of a minimal match, we first investigate the substrings of $\mathtt{T}$ that have the same universality index as $\mathtt{ShortLex}_k(\mathtt{P})$ following Lemma 10. Let w_1 and w_2 be substrings of $\mathtt{T}$ such that $\iota(w_1) = \iota(w_2)$. If a YX- or XY-link of w_1 is a YX- or XY-link of w_2, respectively, then we can reuse the substring of $\mathtt{ShortLex}_k(w_1)$ that corresponds to that arch link of w_1 in constructing $\mathtt{ShortLex}_k(w_2)$. Specifically, let the overlapping arch link be u. If u is a substring of the ith X-arch of w_1 as well as a substring of the jth X-arch of w_2, then the X-vectors for space positions of u in w_2 must differ by exactly $\overrightarrow{U}(j-i)$ from the X-vector values for space positions of u in w_1. The Y-arches' and Y-vectors' case is symmetric. Thus, if we let $w_1 = v_1 u x_1$ and $w_2 = v_2 u x_2$, then applying the repeated removal procedure of the ShortLex normalization algorithm on both strings will result in $z_1 = v_1' u' x_1'$ and $z_2 = v_2' u' x_2'$, where $v_1' \prec v_1$, $v_2' \prec v_2$, $x_1' \prec x_1$, $x_2' \prec x_2$, and finally $u' \prec u$.

It remains to find the borders of X- and Y-arches to determine which arch links can be reused. We solve this problem by building two trees, named the X-tree and Y-tree. An X-tree $T_X(\mathtt{T})$ or a Y-tree $T_Y(\mathtt{T})$ is a tree with at most $\iota(\mathtt{T})|\Sigma| + 1$ space positions of $\mathtt{T}$ as

nodes. The root is a virtual node that corresponds to space position ∞ for X-trees and -1 for Y-trees. For X-trees, each node is a right end point of an X-arch for some substring of $\mathtt{T}$. A node i's parent is the end point of the X-arch that starts at space position i. In other words, $\mathtt{prnt}(i) = \max\{R_X(\mathtt{T}, i, \sigma) \mid \sigma \in \Sigma\}$. Each node i maintains an interval of space positions $\mathtt{chld}(i)$ that would have i as the end point of an X-arch. Specifically, $\mathtt{chld}(i) = \{j \mid i = \max\{R_X(\mathtt{T}, j, \sigma) \mid \sigma \in \Sigma\}\}$. Finally, each node i holds a position $r(i)$ which is the minimum position for which $\mathtt{rest}(z) \prec \mathtt{T}[i : r(i)]$ for some $\sim_k$-congruent character rearrangement z of $\mathtt{ShortLex}_k(\mathtt{P})$. If there is no such position, then $r(i) = \infty$.

Conversely, a node in a Y-tree is a left end point of a Y-arch for some substring of $\mathtt{T}$. A node i's parent $\mathtt{prnt}(i)$ is the starting point of the Y-arch that ends at space position i. Formally, $\mathtt{prnt}(i) = \min\{R_Y(\mathtt{T}, i, \sigma) \mid \sigma \in \Sigma\}$. The child set $\mathtt{chld}(i)$ is the interval of space positions that would have i as a parent. Again, $\mathtt{chld}(i) = \{j \mid i = \min\{R_Y(\mathtt{T}, j, \sigma) \mid \sigma \in \Sigma\}\}$. Lastly, each node i holds a space position $r(i)$ which is the maximum position for which $\mathtt{rest}(z^R)^R \prec \mathtt{T}[r(i) : i]$ for some $\sim_k$-congruent character rearrangement z of $\mathtt{ShortLex}_k(\mathtt{P})$. If there is no such position, then $r(i) = -1$. The notation for an edge $\langle c, p\rangle$ of an X- or Y-tree follows the convention $\langle \text{child}, \text{parent}\rangle$, thus $c < p$ for X-trees and $c > p$ for Y-trees. Finally we abuse the notation $\mathtt{prnt}(i)$ for every space position i so that $\mathtt{prnt}(i) = j$ such that $i \in \mathtt{chld}(j)$.

Figure 2 is an example of an X-tree and a Y-tree constructed from $\mathtt{T}$ = *abcbacaabacccbaccaab*. Arrows represent the edges $\langle c, p\rangle$ of each tree. Since the SNF string *abcabcabc* of the pattern has $\mathtt{rest}(\mathtt{P}) = \mathtt{rest}(\mathtt{P}^R)^R = \lambda$, the value of $r(i)$ for each node i is itself.

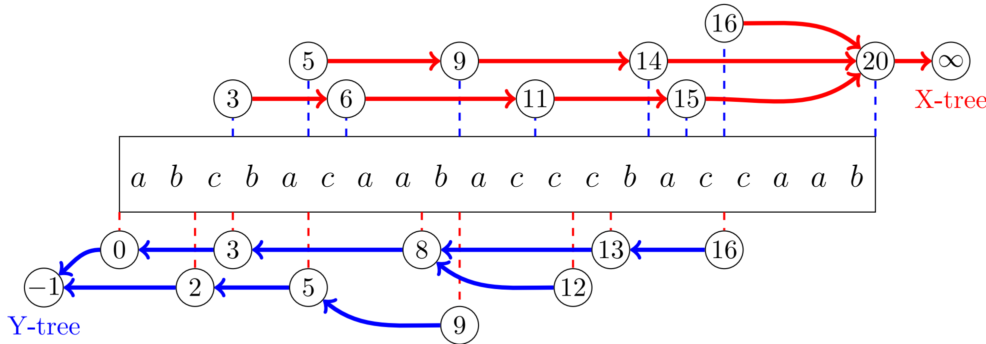

Figure 2 A pair of X-tree and Y-tree constructed from $\mathtt{T}$ = *abcbacaabacccbaccaab* and $\mathtt{ShortLex}_k(\mathtt{P})$ = *abcabcabc* when $k = 3$. The X-tree is drawn in red and the Y-tree is drawn in blue.

We establish the following result – the characterization of the number of nodes in each interval defined by an edge of an X- or Y-tree – for a running time bound on the construction of X-trees and Y-trees.

▶ **Proposition 11.** *For an X-tree or Y-tree constructed from a text* $\mathtt{T}$*, and a space position* j *of* $\mathtt{T}$*, there are at most* $|\Sigma|$ *nodes in every half-open interval* $[\mathtt{prnt}(j), \mathtt{prnt}(\mathtt{prnt}(j)))$.

Building on the characterization from Proposition 11, Lemma 12 bounds the maximum number of edges that includes a given space position of $\mathtt{T}$ along with the number of nodes of the tree.

▶ **Lemma 12.** *For an X-tree (or a Y-tree) constructed from a text* $\mathtt{T}$*, there are at most* $|\Sigma|$ *edges* $\langle c, p\rangle$ *that satisfy* $c < i \leq p$ *(or* $p \leq i < c$*, respectively) for every position* i *of* $\mathtt{T}$*. Moreover, an X- or Y-tree has at most* $\iota(\mathtt{T})|\Sigma| + 1$ *nodes.*

Using the bound obtained above, we compute the worst-case time complexity of the construction of an X-tree.

▶ **Lemma 13.** *Given a preprocessed X-ranker (or Y-ranker) array and the stack representation of* $\texttt{ShortLex}_k(\texttt{P})$*, an X-tree (or Y-tree) for a text* $\texttt{T}$ *can be constructed in* $O(|\texttt{T}||\Sigma|)$ *time.*

With our X- and Y-trees, we can obtain the X- and Y-arches for any minimal substring w of $\texttt{T}$ starting at space position i such that $\iota(w) = \iota(\texttt{P})$ and $\texttt{rest}(\texttt{P}) \prec \texttt{rest}(w)$.

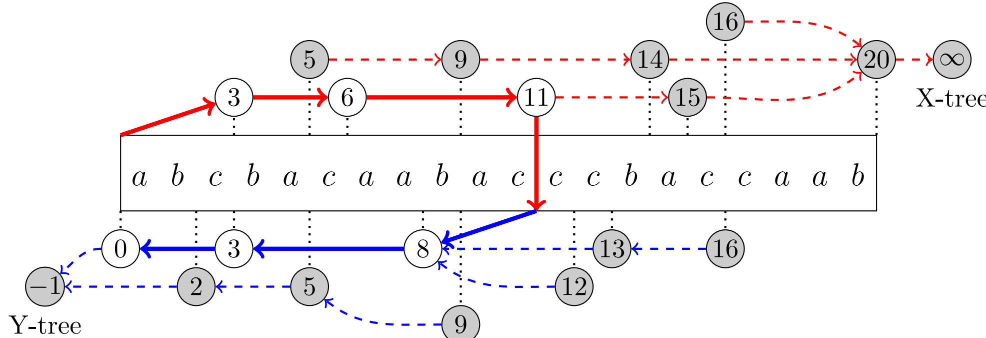

Figure 3 Using the X- and Y-trees $T_X(\texttt{T})$ and $T_Y(\texttt{T})$ to fetch the borders of arch links. The figure illustrates the traversal path for $\iota(\texttt{ShortLex}_k(\texttt{P})) = 3$ starting at space position 0. The thick arrows and the bright nodes indicate the traversal path of the X- and Y-tree.

Figure 3 illustrates the process of fetching the borders of X- and Y-arches from our running example $\texttt{T} = abcbacaabacccbaccaab$ and $\texttt{ShortLex}_k(\texttt{P}) = abcabcabc$. Starting with space position $i = 0$, we check the X-tree to obtain the node $T_X(\texttt{T}).\texttt{prnt}(i) = 3$. This step is indicated by the thick arrow starting at position 0 and ends at node 3. Afterwards, we repeat going up the X-tree for $\iota(\texttt{ShortLex}_k(\texttt{P})) - 1 = 2$ edges to visit node 6 and reach node 11. We read $T_X(\texttt{T}).r(11)$, but since $\texttt{rest}(\texttt{ShortLex}_k(\texttt{P})) = \lambda$, we stay at space position 11. By the definition of an X-tree, this is the smallest space position that satisfies $\texttt{rest}(\texttt{P}) \prec \texttt{rest}(w)$. From there on, we find the node of the Y-tree that has 11 in its child range. We find that node, which is 8, by following the thick arrow that starts at space position 11 and ends at node 8. Again, we traverse nodes 3 and 0 by repeatedly climbing the Y-tree for $\iota(\texttt{P}) - 1 = 2$ edges. The space positions that we have traversed are exactly the borders of X- and Y-arches for a minimal match candidate of $\texttt{P}$ that starts at space position 0.

However, we need $\Sigma = \texttt{alph}(\texttt{P})$ to ensure that arches from $\texttt{T}$ will line up with arches from $\texttt{P}$. Note that no substring w of $\texttt{T}$ such that $\texttt{alph}(w) \neq \texttt{alph}(\texttt{P})$ will be a match of $\texttt{P}$. Thus, we can let $\Sigma = \texttt{alph}(\texttt{P})$ and slice $\texttt{T}$ into maximal substrings $\texttt{T}'$ such that $\texttt{alph}(\texttt{T}') = \texttt{alph}(\texttt{P})$. Then, we can re-apply the same matching algorithm for each $\texttt{T}'$. Recall that arch links of w are determined by a pair of X-arch and Y-arch. Considering that X-vectors and Y-vectors can only be calculated in increasing or decreasing order of space positions, respectively, YX-links have two fixed vectors by Lemma 9. However, XY-links have no fixed vectors and thus we need a way to checkpoint and continue the application of the ShortLex normalization algorithm.

For a string w, let z be a length-minimal subsequence of w that is $\sim_k$-congruent to w. Also, let space positions i and j of w $(0 < i \leq j < |w|)$ be borders of some X- or Y-arch and map to space positions i' and j' of z such that

$$z \not\sim_k z[0 : i' - 1]z[i']z[i' - 1]z[i' + 1 : |z|],$$

$$z \not\sim_k z[0 : j' - 1]z[j']z[j' - 1]z[j' + 1 : |z|].$$

Then, we can decompose $z = v_1 u v_2$ where $v_1 \prec w[0 : i]$, $u \prec w[i : j]$, and $v_2 \prec w[j : |w|]$. Here, v_1 and v_2 do not share the same arches. Since Lemma 9 ensures that X-coordinate and Y-coordinate values of different arches are independent of each other, v_1 and v_2 can be independently obtained through the repeated removal procedure of the ShortLex normalization algorithm. Thus, we have $v_1 w[i : j]v_2 \sim_k w$. Moreover, we have $\overrightarrow{X}(v_1 w[i : j]v_2, |v_1|) = \overrightarrow{X}(z, |v_1|)$ as well as $\overrightarrow{Y}(v_1 w[i : j]v_2, |v_1 w[i : j]|) = \overrightarrow{Y}(z, |v_1 u|)$. It follows that we can compute u if we surely know $\overrightarrow{X}(z, |v_1|)$ and $\overrightarrow{Y}(z, |v_1 u|)$ without running the full ShortLex normalization algorithm on w. We use $\texttt{ShortLex}_k(w, i, j, \overrightarrow{X}, \overrightarrow{Y})$ to denote this checkpointed version of the ShortLex normalization algorithm. The second and third arguments are space positions i and j each at the border of some arch of w. They define the substring $w[i : j]$ that needs to be ShortLex normalized. The fourth and fifth arguments are X- and Y-vectors such that $\overrightarrow{X} = \overrightarrow{X}(z, |v_1|)$ and $\overrightarrow{Y} = \overrightarrow{Y}(z, |v_1 u|)$ following our decomposition scheme from earlier.

Using our checkpointed algorithm, let strings w_1 and w_2 be substrings of T that share some arch links. Let space positions i_1 and j_1 of w_1 and space positions i_2 and j_2 of w_2 mark the start and end of arch links that are shared between w_1 and w_2. For a length-minimal substring z of w_1 that is $\sim_k$-congruent to w_1, we let $z = v_1 u v_2$ where $v_1 \prec w_1[0 : i_1]$, $u \prec w_1[i_1 : j_1]$, and $v_2 \prec w_1[j_1 : |w_1|]$. Moreover, let arches $\texttt{ar}_{x_1}(w_1)$ and $\texttt{ar}_{y_1}(w_1^R)^R$ of w_1 and arches $\texttt{ar}_{x_2}(w_2)$ and $\texttt{ar}_{y_2}(w_2^R)^R$ be the X-arch and Y-arch that produces the arch link u.

First, while computing $\texttt{ShortLex}_k(w_1)$, we must checkpoint the progress of the ShortLex normalization algorithm for all borders of the arch links of w_1. If $w_1[i_1 : j_1]$ is a YX-link, we associate u, $\overrightarrow{X}(z, |v_1 u|) - \overrightarrow{U}(x_1)$, and $\overrightarrow{Y}(z, |v_1|) - \overrightarrow{U}(y_1)$ to the arch pair $(\texttt{ar}_x(w_1), \texttt{ar}_y(w_1^R)^R)$ using nodes in the X- and Y-tree. Later, when computing $\texttt{ShortLex}_k(w_2)$, we can skip the computation for $\texttt{ShortLex}_k(w_2, i_2, j_2, \overrightarrow{X}(w_2, i_2), \overrightarrow{Y}(w_2, j_2))$ and use u instead. Thus, we can decompose the length-minimal subsequence of w_2 that is $\sim_k$-congruent to w_2 as $v_1' u v_2'$, where

$$v_1' = \texttt{ShortLex}_k(w_2, 0, i_2, \overrightarrow{U}(1), \overrightarrow{Y}(z, |v_1|) - \overrightarrow{U}(y_1) + \overrightarrow{U}(y_2)),$$

$$v_2' = \texttt{ShortLex}_k(w_2, j_2, |w_2|, \overrightarrow{X}(z, |v_1 u|) - \overrightarrow{U}(x_1) + \overrightarrow{U}(x_2), \overrightarrow{U}(1)).$$

On the other hand, if $w_1[i_1 : j_1]$ is a XY-link, we only remember that the arch link defined by the arch pair $\texttt{ar}_x(w_1)$ and $\texttt{ar}_y(w_1^R)^R$ produces u, because $\overrightarrow{X}(w_2, j_2)$ and $\overrightarrow{Y}(z, i_2)$ are directly obtainable through Lemma 9. Thus, the length-minimal subsequence of w_2 that is $\sim_k$-congruent to w_2 is decomposed as

$$\texttt{ShortLex}_k(w_2, 0, i_2, \overrightarrow{U}(1), \overrightarrow{U}(y_2 + 1))u\texttt{ShortLex}_k(w_2, j_2, |w_2|, \overrightarrow{U}(x_2 + 1), \overrightarrow{U}(1)).$$

Returning to our running example on Figure 3, the minimal match candidates that start at space position 0 and space position 3 share the YX-link $w[6 : 8]$. This is because arches $T_X(\texttt{T}).\langle 6, 11\rangle$ and $T_Y(\texttt{T}).\langle 8, 3\rangle$ are traversed for both match candidates. This allows us to skip the computation for $w[6 : 8]$ if checkpoints for space positions 6 and 8 were saved beforehand. Note that further decomposition is possible if more checkpoints are available.

Now we are ready to efficiently tackle the Simon's congruence pattern matching problem. Let $\Sigma = \texttt{alph}(\texttt{P})$. If $\texttt{alph}(\texttt{T}) \not\subseteq \Sigma$, arches of T will not align with arches of P. Thus, we first split T into maximal substrings T′ of T that satisfy $\texttt{alph}(\texttt{T}') \subseteq \Sigma$. Then, we repeat the following for every split substring T′ of T. We construct the X- and Y-trees for T′ as well as compute the X- and Y-ranker values beforehand. For each node n of the X-tree, we find the minimal match candidate by traversing the nodes of the X-tree and Y-tree. Then, we check whether the minimal match candidate is $\sim_k$-congruent to P through the checkpointed

ShortLex normalization algorithm. Along the way, we store checkpoints for arch links that are not yet checkpointed to use in future computations. If the minimal match candidate is indeed congruent to $\mathtt{P}$, then we extend the starting and ending points of the minimal match to find all matches of $\mathtt{P}$ which have n as the end space position of the first X-arch.

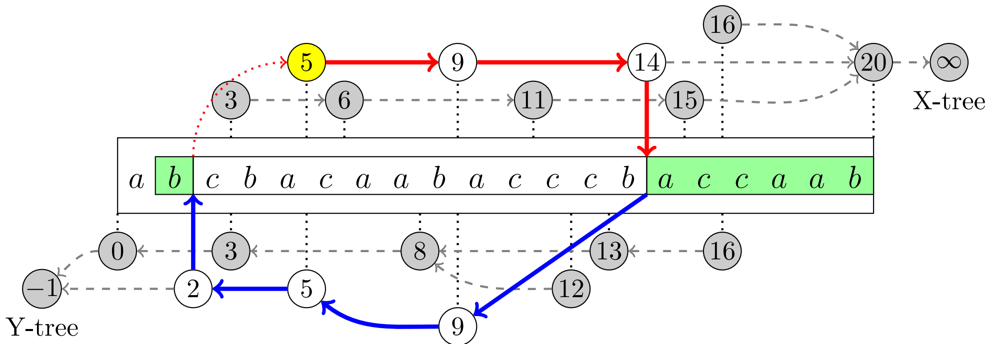

Figure 4 An illustration of the matching process where the current node is $i = 5$. If the minimal match candidate is a match of $\mathtt{P}$, the front and back of the minimal match are extended to the colored boxes at the left and right of the minimal match.

We explain the matching procedure using our running example illustrated in Figure 4, highlighting the matching process for candidate strings for which the first arch ends at node $i = 5$ using Figure 4. Let $k = 3$. We have already computed sets A and B where $A = \{\sigma \mid \mathtt{P}\sigma \not\sim_k \mathtt{P}\}$ as well as $B = \{\sigma \mid \sigma\mathtt{P} \not\sim_k \mathtt{P}\}$. Since $\mathtt{P}$ is 3-universal, we have $A = B = \emptyset$. Recall that we are searching for all pairs (f, b) of starting positions such that $\mathtt{T}[f : b] \sim_k \mathtt{P}$. All substrings of $\mathtt{T}$ that start at space positions in $T_X(\mathtt{T}).\mathtt{chld}(i)$ will have its first arch end at node 5, using the same traversal path in the X- and Y-tree. Thus, we have space positions in $[1, 2]$ as our candidates for f.

Next, we use our traversal method explained with Figure 3 to obtain the arch link borders. Note that $T_Y(\mathtt{T}).r(2) = 2$, so we set our minimal string to start at space position 2, which is the maximum position in $T_X(\mathtt{T}).\mathtt{chld}(i) \cap [-1, T_Y(\mathtt{T}).r(2)]$. The -1 comes from $B = \emptyset$. Moreover, our minimal string ends at space position 14, which is the largest value observed during the X- and Y-tree traversal. The minimal match candidate is indicated by the box in the middle of Figure 4.

Now, we check whether $\mathtt{T}[2 : 14] \sim_k \mathtt{P}$. Since no checkpoints are saved at the moment, we compute the shortest subsequence of $\mathtt{T}[2 : 14]$ that is $\sim_k$-congruent to $\mathtt{T}[2 : 14]$ without using any checkpoints. By applying the first step of the ShortLex normalization algorithm, we obtain $z = cbacabacb$. With the subsequence z, we compute checkpoints for later use for edge pairs $(\langle 5, 9\rangle, \langle 5, 2\rangle)$, $(\langle 5, 9\rangle, \langle 9, 5\rangle)$, and $(\langle 9, 14\rangle, \langle 9, 5\rangle)$ as well as the edge-rest pair $(\langle 9, 14\rangle, \langle 14, 9\rangle)$. Popping all characters in z from the stack representation of $\mathtt{ShortLex}_k(\mathtt{P})$ verifies that $z \sim_k \mathtt{P}$.

Finally, using Lemma 7, we observe that all space positions no less than 14 are values for b that result in a match. Using Lemma 7 again on $\mathtt{T}^R$, we observe that space positions in $[-1, 2] \cap [1, 2]$ are values for f that result in a match. The extendable range for the starting and ending space positions are marked as the box on the left and right of Figure 4, respectively. Thus, we obtain the space position interval pair $([1, 2], [14, 20])$ for the iteration at node $i = 5$.

Now that we have an idea of how the algorithm works, we prove the algorithm solves MatchSimK in linear time in $|\mathtt{T}|$.

▶ **Theorem 14.** *Given a pattern* $\texttt{P}$*, a text* $\texttt{T}$*, and a number* k*, we can report all non-overlapping triples* $([f_1, f_2], [b_1, b_2], \text{offset})$ *such that for all space positions* $f \in [f_1, f_2]$ *and* $b \in [b_1, b_2]$*, we have* $\texttt{T}[f + \text{offset} : b + \text{offset}] \sim_k \texttt{P}$*. The computation takes* $O(|\texttt{T}||\Sigma|(|\Sigma|^2 + k))$ *time.*

Note that we need an additional offset value that denotes the starting point of each $\texttt{T}'$. This distinguishes matches of $\texttt{P}$ from different $\texttt{T}'$s and can be used to pinpoint the match from $\texttt{T}$. Moreover, in practice, the size of an alphabet is regarded as constant. Thus, given a fixed k, MatchSimK can be solved in $O(|\texttt{T}||\Sigma|^3) = O(|\texttt{T}|)$ time.

3.2 Algorithms for Pattern Matching Variants

By altering the algorithm for MatchSimK to remember the length of the shortest or longest congruent substring while iterating every node of the X-tree, we can solve LCongStrK and SCongStrK.

▶ **Theorem 15.** *Given a pattern* $\texttt{P}$*, a text* $\texttt{T}$*, and a number* k*, we can report the longest and shortest substring of* $\texttt{T}$ *that is* $\sim_k$*-congruent to* $\texttt{P}$ *in* $O(|\texttt{T}||\Sigma|(|\Sigma|^2 + k))$ *time.*

On the other hand, Proposition 4 shows that if a subsequence x of $\texttt{T}$ is $\sim_k$-congruent to $\texttt{P}$, then there must be a subsequence p' of x that is also $\sim_k$-congruent to $\texttt{P}$ and has length $|\texttt{ShortLex}_k(\texttt{P})|$. Since there does not exist a shorter string that is $\sim_k$-congruent to $\texttt{P}$, any algorithm that solves SCongSeqK must return a string of length $|\texttt{ShortLex}_k(\texttt{P})|$ if there exists a substring of $\texttt{T}$ that is $\sim_k$-congruent to $\texttt{P}$. In other words, the recognition of a shortest congruent subsequence of $\texttt{T}$ can be done by popping characters of $\texttt{T}$ from the stack representation of $\texttt{P}$. Scanning $\texttt{T}$ from left to right, we pop each character and mark the current space position if the corresponding character of the current space position is at the top of the stack. When the stack becomes empty, the marked positions yield a shortest subsequence of $\texttt{T}$ that is $\sim_k$-congruent to $\texttt{P}$.

▶ **Theorem 16.** *Given a pattern* $\texttt{P}$*, a text* $\texttt{T}$*, and a number* k*, we can report an instance of the shortest subsequence of* $\texttt{T}$ *that is* $\sim_k$*-congruent to* $\texttt{P}$ *in* $O(|\texttt{P}| + |\texttt{T}|)$ *time.*

Note that $\mathbb{S}_k(\texttt{P}) \subseteq \mathbb{S}_k(\texttt{T})$ if the search succeeds. Thus, one interesting idea is to use the algorithm for SCongSeqK in solving SubseqSetInclusion. However, the existence of an answer of SCongSeqK on strings $\texttt{P} = w_1$ and $\texttt{T} = w_2$ is only a sufficient condition for $\mathbb{S}_k(\texttt{P}) \subseteq \mathbb{S}_k(\texttt{T})$. Consider the example $w_1 = aabc$ and $w_2 = ccacbca$ and let $k = 2$. No character rearrangement of w_1 that is $\sim_k$-congruent to w_1 is a subsequence of w_2. Indeed, $\text{Closure}_k(w_1)$ is the singleton set $\{w_1\}$, while w_1 is not a subsequence of w_2. Thus, the algorithm for SCongSeqK will fail on this pair of strings. However, every element in the subsequence set $\mathbb{S}_2(w_1) = \{aa, ab, ac, bc, a, b, c, \lambda\}$ is a subsequence of w_2. This means that we need further characterization of the relation $\mathbb{S}_k(w_1) \subseteq \mathbb{S}_k(w_2)$ in order to solve SubseqSetInclusion. Although we conjecture that the solution for the Shortest Congruent Subsequence problem may be used in solving SubseqSetInclusion along with clever classification of w_1 and additional subprocedures, SubseqSetInclusion still remains open.

4 Conclusions

We have solved the open problem of finding all substrings of a text $\texttt{T}$ that are $\sim_k$-congruent to a pattern $\texttt{P}$ for an integer k proposed by Gawrychowski et al. [8]. We have devised tree data structures called X-trees and Y-trees to reuse results from previous computations and lower the asymptotic running time to be linear in the length of the text. Moreover, we have solved

two variants of the pattern matching problem using the efficient algorithm for MatchSimK as well as provided a linear algorithm that finds the shortest subsequence of the text that is $\sim_k$-congruent to the pattern. As future work, we plan to solve LCongSeqK, which is the remaining unsolved variant of MatchSimK. Moreover, we extend MatchSimK into an optimization problem, defined as the following:

▶ **Problem 17.** *Simon's Congruence Pattern Matching Optimization (ThreshMatchSimK): Given a pattern* P*, a text* T*, and a threshold t, find the maximum integer k for which there exist at least t congruent substrings of* T *that are $\sim_k$-congruent to* P*.*

Finally, we remark that the SubseqSetInclusion problem proposed by Gawrychowski et al. [8] is an interesting open problem to investigate.

References

1. Anadi Agrawal and Paweł Gawrychowski. A faster subquadratic algorithm for the longest common increasing subsequence problem. In *31st International Symposium on Algorithms and Computation*, volume 181 of *LIPIcs*, pages 4:1–4:12, 2020.
2. Laura Barker, Pamela Fleischmann, Katharina Harwardt, Florin Manea, and Dirk Nowotka. Scattered factor-universality of words. In *Developments in Language Theory – 24th International Conference*, volume 12086 of *Lecture Notes in Computer Science*, pages 14–28, 2020.
3. Richard Beal, Tazin Afrin, Aliya Farheen, and Don Adjeroh. A new algorithm for "the LCS problem" with application in compressing genome resequencing data. *BMC Genomics*, 17 (Supplement 4)(544):369–381, 2016.
4. Wun-Tat Chan, Yong Zhang, Stanley P. Y. Fung, Deshi Ye, and Hong Zhu. Efficient algorithms for finding a longest common increasing subsequence. *Journal of Combinatorial Optimization*, 13(3):277–288, 2007.
5. Lukas Fleischer and Manfred Kufleitner. Testing Simon's congruence. In *43rd International Symposium on Mathematical Foundations of Computer Science*, pages 62:1–62:13, 2018.
6. Pamela Fleischmann, Lukas Haschke, Annika Huch, Annika Mayrock, and Dirk Nowotka. Nearly k-universal words – investigating a part of Simon's congruence. In *Descriptional Complexity of Formal Systems – 24th International Conference, Proceedings*, volume 13439 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2022.
7. Emmanuelle Garel. Minimal separators of two words. In *4th Annual Symposium on Combinatorial Pattern Matching*, pages 35–53, 1993.
8. Paweł Gawrychowski, Maria Kosche, Tore Koß, Florin Manea, and Stefan Siemer. Efficiently testing Simon's congruence. In *38th International Symposium on Theoretical Aspects of Computer Science*, volume 187 of *LIPIcs*, pages 34:1–34:18, 2021.
9. Jean-Jacques Hébrard. An algorithm for distinguishing efficiently bit-strings by their subsequences. *Theoretical Computer Science*, 82(1):35–49, 1991.
10. James W. Hunt and M. Douglas McIlroy. An algorithm for differential file comparison. In *Computer Science Technical Reports 41*, 1975.
11. Sungmin Kim, Yo-Sub Han, Sang-Ki Ko, and Kai Salomaa. On Simon's congruence closure of a string. In *Descriptional Complexity of Formal Systems – 24th International Conference, Proceedings*, volume 13439 of *Lecture Notes in Computer Science*, pages 127–141. Springer, 2022.
12. Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
13. Thomas Schwentick, Denis Thérien, and Heribert Vollmer. Partially-ordered two-way automata: A new characterization of DA. In *Revised Papers from the 5th International Conference on Developments in Language Theory*, pages 239–250, 2001.

14 Jan Sedmidubský and Pavel Zezula. A web application for subsequence matching in 3d human motion data. In *19th IEEE International Symposium on Multimedia*, pages 372–373, 2017.
15 Imre Simon. Piecewise testable events. In *Proceedings of the 2nd GI Conference on Automata Theory and Formal Languages*, pages 214–222, 1975.
16 Petra Surynková and Pavel Surynek. Application of longest common subsequence algorithms to meshing of planar domains with quadrilaterals. In *Mathematical Methods for Curves and Surfaces – 9th International Conference*, volume 10521 of *Lecture Notes in Computer Science*, pages 296–311, 2016.
17 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
18 Philipp Weis and Neil Immerman. Structure theorem and strict alternation hierarchy for FO^2 on words. *Logical Methods in Computer Science*, 5(3), 2009.

A Appendix

Algorithm 1 $O(|\mathtt{T}|^2)$–MatchSimK$(\mathtt{P}, \mathtt{T}, k)$ for Theorem 8.

1: Given: a pattern $\mathtt{P}$, a text $\mathtt{T}$, and a number k
2: Returns: all space position pairs (f, b) that satisfy $\mathtt{T}[f : b] \sim_k \mathtt{P}$
3: preprocess all possible X-ranker values
4: $s_p \leftarrow$ stack representation of $\mathtt{ShortLex}_k(\mathtt{P})$
5: $B \leftarrow \{b \in \Sigma \mid \mathtt{P} \not\sim_k \mathtt{P}b\}$
6: $M \leftarrow \emptyset$
7: **for** $i = 0, 1, \ldots, |\mathtt{T}|$ **do**
8: $\quad s'_p \leftarrow copy(s_p)$
9: $\quad m \leftarrow i$
10: $\quad$**while** s'_p is not empty **do**
11: $\quad\quad \sigma \leftarrow \underset{a \in peek(s'_p)}{\arg\min}\ R_X(\mathtt{T}, m, a)$
12: $\quad\quad m \leftarrow R_X(\mathtt{T}, m, \sigma)$
13: $\quad\quad$ pop σ from s'_p
14: $\quad$**end while**
15: $\quad$**if** $\mathtt{ShortLex}_k(\mathtt{P}) \sim_k \mathtt{T}[i : m]$ **then**
16: $\quad\quad$ find $\sigma \in B$ that minimizes $R_X(\mathtt{T}, m, \sigma)$
17: $\quad\quad$**for** all space positions $j \in [m : R_X(\mathtt{T}, m, \sigma) - 1]$ **do**
18: $\quad\quad\quad$ add (i, j) to M
19: $\quad\quad$**end for**
20: $\quad$**end if**
21: **end for**
22: **return** M

Algorithm 2 X-tree construction for Lemma 13.

```
1: Given: preprocessed X-ranker values, ShortLex_k(P) s_p in stack form, and text T
2: Returns: an X-tree T_X(T) constructed from T
3: T_X(T).Nodes ← {∞}
4: for i = 1, 2, ..., ι(ShortLex_k(P)) do
5:     for j = 0, 1, ..., |ar_i(ShortLex_k(P))| − 1 do
6:         pop ar_i(ShortLex_k(P))[j] from s_p
7:     end for
8: end for
9: for i = 0, 1, 2, ..., |T| − 1 do
10:    parent ← max_{σ:σ∈Σ} {R_X(T, i, σ)}
11:    if parent ∉ T_X(T).Nodes then
12:        add parent to T_X(T).Nodes
13:        s'_p ← copy(s_p)
14:        T_X(T).r(i) ← i
15:        while s'_p is not empty do
16:            S ← peek(s'_p)
17:            σ ← arg min_{c:c∈S} {R_X(T, r(i), c)}
18:            T_X(T).r(i) ← R_X(T, T_X(T).r(i), σ)
19:            pop σ from s'_p
20:        end while
21:        T_X(T).chld(parent) ← [i, i)
22:    end if
23:    extend end point of T_X(T).chld(parent) by one
24:    if i ∈ Nodes then
25:        T_X(T).prnt(i) ← parent
26:    end if
27: end for
28: return T_X(T)
```

Algorithm 3 $O(|\mathtt{T}|)$–MatchSimK($\mathtt{P}, \mathtt{T}, k$) for Theorem 14.

1: Given: a pattern $\mathtt{P}$, a text $\mathtt{T}$, an integer k
2: Returns: a set S of triples where, for space positions f and b of $\mathtt{T}$, $\mathtt{T}[f : b] \sim_k \mathtt{P}$ if and only if there exists some element $e = ([f_1, f_2], [b_1, b_2], \text{offset})$ in S such that space positions $f - \text{offset} \in [f_1, f_2]$ and $b - \text{offset} \in [b_1, b_2]$
3: positions $\leftarrow \emptyset$
4: $s_p \leftarrow \mathtt{ShortLex}_k(\mathtt{P})$ in stack form
5: Slice $\mathtt{T}$ whenever $\mathtt{T}[i] \notin \mathtt{alph}(\mathtt{P})$
6: $A \leftarrow \{\sigma \mid \mathtt{P}\sigma \not\sim_k \mathtt{P}\}$
7: $B \leftarrow \{\sigma \mid \sigma\mathtt{P} \not\sim_k \mathtt{P}\}$
8: **for** all sliced substrings $\mathtt{T}'$ of $\mathtt{T}$ **do**
9: offset $\leftarrow$ the start space position of $\mathtt{T}'$ in $\mathtt{T}$
10: Map $\leftarrow$ empty map for saving vectors and substrings
11: Preprocess X- and Y-ranker array
12: Construct X-tree $T_X(\mathtt{T}')$ and Y-tree $T_Y(\mathtt{T}')$
13: **for** all nodes $i \in T_X(\mathtt{T}')$.nodes **do**
14: From i, go up the X-tree for $\iota(\mathtt{P}) - 1$ edges
15: $j_1 \leftarrow T_X(\mathtt{T}').r(\text{current node})$
16: if $j_1 = \infty$, break.
17: From j_1, go up the Y-tree using $\iota(\mathtt{P})$ calls of $T_Y(\mathtt{T}').\mathtt{prnt}()$
18: $n \leftarrow$ current node
19: $j_2 \leftarrow \max(T_Y(\mathtt{T}').\mathtt{chld}(i) \cap [\max_{\sigma \in B} R_Y(\mathtt{T}', n, \sigma) + 1, n])$
20: if no such value exists, continue.
21: $z \leftarrow \mathtt{ShortLex}_k(\mathtt{T}'[j_2 : j_1])$ using the checkpoint mechanism and Map
22: Save checkpoints for each arch link of $\mathtt{T}'[j_2 : j_1]$
23: **if** $z \sim_k \mathtt{ShortLex}_k(\mathtt{P})$ **then**
24: interval1 $\leftarrow T_X(\mathtt{T}').\mathtt{chld}(i) \cap [\max_{\sigma \in B} R_Y(\mathtt{T}', j_2, \sigma) + 1, j_2]$
25: interval2 $\leftarrow [j_1, \min_{\sigma \in A} R_X(\mathtt{T}', j_1, \sigma) - 1]$
26: add (interval1, interval2, offset) to positions
27: **end if**
28: **end for**
29: **end for**
30: **return** positions

Algorithm 4 Shortest Congruent Subsequence Problem for Theorem 16.

```
1: Given: a pattern P, a text T, an integer k
2: Returns: an instance of the shortest subsequence of T that is ~_k-congruent to P
3: s_p ← ShortLex_k(P) in stack form
4: sseq ← λ
5: for all indices i = 0, 1, . . . , |T| − 1 of T do
6:     if T[i] ∈ peek(s_p) then
7:         pop T[i] from s_p
8:         append T[i] to sseq
9:         if s_p is empty then
10:            return sseq
11:        end if
12:    end if
13: end for
14: return None
```