Longest Common Extensions with Wildcards: Trade-Off and Applications

Gabriel Bathie

DIENS, École normale supérieure de Paris, PSL Research University, France
LaBRI, Université de Bordeaux, Talence, France

Panagiotis Charalampopoulos

Birkbeck, University of London, UK

Tatiana Starikovskaya

DIENS, École normale supérieure de Paris, PSL Research University, France

Abstract

We study the Longest Common Extension (LCE) problem in a string containing wildcards. Wildcards (also called “don’t cares” or “holes”) are special characters that match any other character in the alphabet, similar to the character “`?`” in Unix commands or “`.`” in regular expression engines.

We consider the problem parametrized by G, the number of maximal contiguous groups of wildcards in the input string. Our main contribution is a simple data structure for this problem that can be built in $O(n(G/t)\log n)$ time, occupies $O(nG/t)$ space, and answers queries in $O(t)$ time, for any $t \in [1\mathinner{.\,.} G]$. Up to the $O(\log n)$ factor, this interpolates smoothly between the data structure of Crochemore et al. [JDA 2015], which has $O(nG)$ preprocessing time and space, and $O(1)$ query time, and a simple solution based on the “kangaroo jumping” technique [Landau and Vishkin, STOC 1986], which has $O(n)$ preprocessing time and space, and $O(G)$ query time.

By establishing a connection between this problem and Boolean matrix multiplication, we show that our solution is optimal up to subpolynomial factors when $G = \Omega(n)$ under a widely believed hypothesis. In addition, we develop a new simple, deterministic and combinatorial algorithm for sparse Boolean matrix multiplication.

Finally, we show that our data structure can be used to obtain efficient algorithms for approximate pattern matching and structural analysis of strings with wildcards. First, we consider the problem of pattern matching with k errors (i.e., edit operations) in the setting where both the pattern and the text may contain wildcards. The “kangaroo jumping” technique can be adapted to yield an algorithm for this problem with runtime $O(n(k+G))$, where G is the total number of maximal contiguous groups of wildcards in the text and the pattern and n is the length of the text. By combining “kangaroo jumping” with a tailor-made data structure for LCE queries, Akutsu [IPL 1995] devised an $O(n\sqrt{km}\operatorname{polylog} m)$-time algorithm. We improve on both algorithms when $k \ll G \ll m$ by giving an algorithm with runtime $O(n(k+\sqrt{Gk\log n}))$. Secondly, we give $O(n\sqrt{G}\log n)$-time and $O(n)$-space algorithms for computing the prefix array, as well as the quantum/deterministic border and period arrays of a string with wildcards. This is an improvement over the $O(n\sqrt{n\log n})$-time algorithms of Iliopoulos and Radoszewski [CPM 2016] when $G = O(n/\log n)$.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Longest common prefix, longest common extension, wildcards, Boolean matrix multiplication, approximate pattern matching, periodicity arrays

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.19

Related Version *Full Version*: `https://arxiv.org/abs/2408.03610`

Supplementary Material *Software (Source code)*: `https://github.com/GBathie/LCEW`
archived at `swh:1:dir:5114a79aef109b700d994cda9c036aa9bc47076e`

Funding Gabriel Bathie and Tatiana Starikovskaya were partially supported by the grant ANR-20-CE48-0001 from the French National Research Agency (ANR). A research visit during which part of the presented ideas were conceived was funded by a Royal Society International Exchanges Award.

1 Introduction

Given a string T, the *longest common extension* (LCE) at indices i and j is the length of the longest common prefix of the suffixes of T starting at indices i and j. In the LCE problem, given a string T, the goal is to build a data structure that can efficiently answer LCE queries.

Longest common extension queries are a powerful string operation that underlies a myriad of string algorithms, for problems such as approximate pattern matching [4, 5, 18, 31, 54, 55], finding maximal or gapped palindromes [11, 19, 36, 49], and computing the repetitive structure (e.g., runs) in strings [6, 48], to name just a few.

Due to its importance, the LCE problem and its variants have received a lot of attention [8, 9, 10, 11, 13, 29, 32, 34, 38, 44, 47, 45, 51, 60, 61, 63, 64, 65, 46]. The suffix tree of a string of length n occupies $\Theta(n)$ space and can be preprocessed in $O(n)$ time to answer LCE queries in constant time [29, 38]. However, the $\Theta(n)$ space requirement can be prohibitive for applications such as computational biology that deal with extremely large strings. Consequently, much of the recent research has focused on designing data structures that use less space without being (much) slower in answering queries. Consider the setting when we are given a read-only length-n string T over an alphabet of size polynomial in n. Bille et al. [10] gave a data structure for the LCE problem that, for any given user-defined parameter $\tau \leq n$, occupies $\mathcal{O}(\tau)$ space on top of the input string and answers queries in $\mathcal{O}(n/\tau)$ time. Kosolobov [50] showed that this data structure is optimal when $\tau = \Omega(n/\log n)$. A drawback of the data structure of Bille et al. [10] is its rather slow $\mathcal{O}(n^{2+\varepsilon})$ construction time. This motivated studies towards an LCE data structure with optimal space and query time and a fast construction algorithm. Gawrychowski and Kociumaka [32] gave an optimal $\mathcal{O}(n)$-time and $\mathcal{O}(\tau)$-space Monte Carlo construction algorithm and Birenzwige et al. [13] gave a Las Vegas construction algorithm with the same complexity provided $\tau = \Omega(\log n)$. Finally, Kosolobov and Sivukhin [51] gave a deterministic construction algorithm that works in optimal $\mathcal{O}(n)$ time and $\mathcal{O}(\tau)$ space for $\tau = \Omega(n^{\varepsilon})$, where $\varepsilon > 0$ is an arbitrary constant. Another line of work [8, 9, 34, 60, 63, 64, 65, 45, 46] considers LCE data structures over compressed strings.

One important variant of the LCE problem is that of LCE with k-mismatches (k-LCE), where one wants to find the longest prefixes that differ in at most k positions, for a given integer parameter k. Landau and Vishkin [55] proposed a technique, dubbed "kangaroo jumping", that reduces k-LCE to $k+1$ standard LCE queries. This technique is a central component of many approximate pattern matching algorithms, under the Hamming [5, 18] and the edit [4, 18, 55] distances.

In this work, we focus on the variant of LCE in strings with *wildcards*, denoted LCEW. Wildcards (also known as *holes* or *don't cares*), denoted $\diamond$, are special characters that match every character of the alphabet. Wildcards are a versatile tool for modeling uncertain data, and algorithms on strings with wildcards have garnered considerable attention in the literature [20, 25, 57, 42, 43, 23, 35, 30, 4, 21, 22, 58, 5, 59, 7, 24, 62, 12, 3, 2, 41, 56, 14, 16, 17].

Given a string T, and indices i, j, LCEW(i, j) is the length of the longest matching prefixes of the suffixes of T starting at indices i and j. For all $\tau \in [1\,..\,n]$, Iliopoulos and Radoszewski [41] showed an LCEW data structure with $\mathcal{O}(n^2 \log n/\tau)$ preprocessing time, $\mathcal{O}(n^2/\tau)$ space, and $\mathcal{O}(\tau)$ query time. In the case where the number of wildcards in T is bounded, more efficient data structures exist. The LCEW problem is closely related to k-LCE: if we let $\widehat{T}$ be the string obtained by replacing each wildcard in T with a new character, the i-th wildcard replaced with a fresh letter $\#_i$, then an LCEW query in T can be reduced to a D-LCE query in $\widehat{T}$, where D is the number of wildcards in T. Consequently, an LCEW query can be answered using $\mathcal{O}(D)$ LCE queries. In particular, if we use the suffix tree to answer LCE queries, we obtain a data structure with $\mathcal{O}(n)$ space and construction time and $\mathcal{O}(D)$ query time. At the other end of the spectrum, Blanchet-Sadri and Lazarow [15] showed that one can achieve $\mathcal{O}(1)$ query time using $\mathcal{O}(nD)$ space after an $\mathcal{O}(nD)$-time preprocessing.

▶ **Example 1.1.** For string $T = \texttt{abab}\diamond\diamond\diamond\texttt{aaaa}\diamond\diamond\diamond\diamond\texttt{ba}\diamond\diamond\diamond\texttt{bb}$, we have $D = 10$ and $G = 3$.

By using the structure of the wildcards inside the string, one can improve the aforementioned bounds even further. Namely, it is not hard to see that if the wildcards in T are arranged in G maximal contiguous groups (see Example 1.1), then we can reduce the number of LCE queries needed to answer an LCEW query to G by jumping over such groups, thus obtaining a data structure with $\mathcal{O}(n)$-time preprocessing, $\mathcal{O}(n)$ space, and $\mathcal{O}(G)$ query time. On the other hand, Crochemore et al. [27] devised an $\mathcal{O}(nG)$-space data structure that can be built in $\mathcal{O}(nG)$ time and can answer LCEW queries in constant time.

1.1 Our Results

In this work, we present an LCEW data structure that achieves a smooth space-time trade-off between the data structure based on "kangaroo jumps" and that of Crochemore et al. [27]. As our main contribution, we show that for any $t \le G$, there exists a set of $\mathcal{O}(G/t)$ positions, called *selected* positions, that intersects any chain of t kangaroo jumps from a fixed pair of positions. Given the LCEW information on selected positions, we can speed up LCEW queries on arbitrary positions by jumping from the first selected position in the common extension to the last selected position in the common extension. This gives us an $\mathcal{O}(t)$ bound on the number of kangaroo jumps we need to perform to answer a query. We leverage the fast FFT-based algorithm of Clifford and Clifford [20] for pattern matching with wildcards to efficiently build a dynamic programming table containing the result of LCEW queries on pairs of indices containing a selected position; this table allows us to jump from the first to the last selected position in the common extension in constant time. The size of the table is $\mathcal{O}(nG/t)$, while the query time is $\mathcal{O}(t)$. For comparison, the data structures of Crochemore et al. [27] and of Iliopoulos and Radoszewski [41] use a similar dynamic programming scheme that precomputes the result of LCEW queries for a subset of positions: Crochemore et al. use all transition positions (see Section 3 for a definition), while Iliopoulos and Radoszewski use one in every $\sqrt{n}$ positions. We use a more refined approach, that allows us to obtain both a dependency on G instead of n and a space-query-time trade-off. Our result can be stated formally as follows.

▶ **Theorem 1.2.** *Suppose that we are given a string T of length n that contains wildcards arranged into G maximal contiguous groups. For every $t \in [1\,..\,G]$, there exists a deterministic data structure that:*

- *uses space $\mathcal{O}(nG/t)$,*
- *can be built in time $\mathcal{O}(n(G/t)\log n)$ using $\mathcal{O}(nG/t)$ space,*
- *given two indices $i, j \in [1\,..\,n]$, returns LCEW(i, j) in time $\mathcal{O}(t)$.*

We further show that this trade-off can be extended to $t \ge G$ by implementing the kangaroo jumping method of Landau and Vishkin [55] with a data structure that provides a time-space trade-off for (classical) LCE queries. Using the main result of Kosolobov and Sivukhin [51], we obtain the following:

▶ **Corollary 1.3.** *Suppose that we are given a read-only string T of length n that contains wildcards arranged into G maximal contiguous groups. For every constant $\varepsilon > 0$ and $t \in [G\,..\,G \cdot n^{1-\varepsilon}]$, there exists a data structure that:*

- *uses space* $\mathcal{O}(nG/t)$*,*
- *can be built in time* $\mathcal{O}(n)$ *using* $\mathcal{O}(nG/t)$ *space,*
- *given two indices* $i, j \in [1\,..\,n]$*, returns* $\mathsf{LCEW}(i, j)$ *in time* $\mathcal{O}(t)$*.*

Proof. We build the LCE data structure of Kosolobov and Sivukhin [51] for parameter $\tau = nG/t = \Omega(n^{\varepsilon})$ in $\mathcal{O}(n)$ time and $\mathcal{O}(\tau)$ space. As an LCEW query reduces to G LCE queries, the constructed data structure supports LCEW queries in $\mathcal{O}(G \cdot n/\tau) = \mathcal{O}(t)$ time. ◀

By a reduction from Boolean matrix multiplication, we derive a conditional $\Omega(n^{2-o(1)})$ lower bound on the product of the preprocessing and query times of any combinatorial data structure for the LCEW problem (Theorem 4.2).[1] This is the first lower bound for this problem and matches the trade-off of our data structure up to subpolynomial factors when $G = \Theta(n)$.

Table 1 Overview of combinatorial deterministic sparse Boolean matrix multiplication algorithms. The values m_{in} (resp. m_{out}) refer to the total number of non-zero entries in the input matrices (resp., in the output matrix).

Source	Running Time
Gustavson [37]	$\mathcal{O}(n \cdot m_{in})$
Kutzkov [53]	$\mathcal{O}(n \cdot (n + m_{out}^2))$
Künnemann [52]	$\mathcal{O}(\sqrt{m_{out}} \cdot n^2 + m_{out}^2)$
Abboud, Bringmann, Fischer, Künemann [1]	$\tilde{\mathcal{O}}(m_{in}\sqrt{m_{out}})$
Our algorithm	$\tilde{\mathcal{O}}(n\sqrt{m_{in} \cdot (n + m_{out})})$

Surprisingly, one can also use the connection between the two problems to derive an algorithm for sparse Boolean matrix multiplication (BMM). Existing algorithms for BMM can be largely categorised into two types: combinatorial and those relying on (dense) fast matrix multiplication. However, the latter are notorious for their significant hidden constants, making them unlikely candidates for practical applicability. By using the connection to the LCEW problem, we show a deterministic combinatorial algorithm with runtime $\tilde{\mathcal{O}}(n\sqrt{m_{in} \cdot (n + m_{out})})$. Our algorithm ties or outperforms all other known deterministic combinatorial algorithms [37, 53, 66, 52] for some range of parameters m_{in} and m_{out}, e.g., when $m_{in} = \Theta(n^{3/2})$ and $m_{out} = \Theta(n^{4/3})$, except for the one implicitly implied by the result of Abboud et al. [1]. Namely, by replacing fast matrix multiplication (used in a black-box way) in [1, Theorem 4.1] with the naive matrix multiplication algorithm, one obtains a deterministic combinatorial algorithm with runtime $\tilde{\mathcal{O}}(m_{in}\sqrt{m_{out}})$, which is always better than our time bound. See Table 1 for a summary. However, our algorithm is much simpler than that of Abboud et al. [1]: while our algorithm relies solely on standard tools typically covered in undergraduate computer science courses, theirs requires an intricate construction of a family of hash functions with subsequent derandomisation. We provide a (non-optimized) proof-of-concept implementation at `https://github.com/GBathie/LCEW`.

[1] In line with previous work, we say that an algorithm or a data structure is *combinatorial* if it does not use fast matrix multiplication as a subroutine during preprocessing or while answering queries.

Applications

We further showcase the significance of our data structure by using it to improve over the state-of-the-art algorithms for approximate pattern matching and the construction of periodicity-related arrays for strings containing wildcards.

As previously mentioned, LCE queries play a crucial role in string algorithms, especially in approximate pattern matching algorithms, such as for the problem of *pattern matching with k errors* (k-PME, also known as pattern matching with k *edits* or *differences*). This problem involves identifying all positions in a given text where a fragment starting at that position is within an edit distance of k from a given pattern. The now-classical Landau–Vishkin algorithm [55] elegantly solves this problem, achieving a time complexity of $\mathcal{O}(nk)$ through extensive use of LCE queries. A natural extension of k-PME is the problem of *pattern matching with wildcards and k-errors* (k-PMWE), where the pattern and the text may contain wildcards. The algorithm of Landau and Vishkin [55] for k-PMWE can be extended to an $\mathcal{O}(n(k+G))$-time algorithm for k-PMWE in strings with G groups of wildcards (see [4]). Building on their work, Akutsu [4] gave an algorithm for k-PMWE that runs in time $\tilde{\mathcal{O}}(n\sqrt{km})$.[2] In Theorem 5.1, we give an algorithm for k-PMWE with runtime $O(n(k+\sqrt{Gk\log n}))$, which improves on the algorithms of Akutsu [4] and Landau and Vishkin [55] in the regime where $k \ll G \ll m$.

Periodicity arrays capture repetitions in strings and are widely used in pattern matching algorithms, see e.g. [28, 26]. The prefix array of a length-n string T with wildcards stores $\mathsf{LCEW}(1,j)$ for all $1 \le j \le n$. It was first studied in [40], where an $\mathcal{O}(n^2)$-time construction algorithm was given. More recently, Iliopoulos and Radoszewski [41] presented an $O(n\sqrt{n\log n})$-time and $\Theta(n)$-space algorithm. Another fundamental periodicity array is the border array, which stores the maximum length of a proper border of each prefix of the string. When a string contains wildcards, borders can be defined in two different ways [39, 40]. A *quantum border* of a string T is a prefix of T that matches the same-length suffix of T, while a *deterministic border* is a border of a string T' that does not contain wildcards and matches T, see Example 1.4. A closely related notion is that of quantum and deterministic periods and their respective period arrays (see Section 2 for definitions).

▶ **Example 1.4.** The maximal length of a quantum border of $T = \texttt{ab}\lozenge\texttt{bc}$ is 3; note that $\texttt{ab}\lozenge$ matches $\lozenge\texttt{bc}$. The maximal length of a deterministic border of T, however, is 0.

Early work in this area [39, 40] showed that both variants of the border array can be constructed in $\mathcal{O}(n^2)$ time. Iliopoulos and Radoszewski [41] demonstrated that one can compute the border arrays from the prefix array in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space, and the period arrays in $\mathcal{O}(n\log n)$ time and $\mathcal{O}(n)$ space, thus deriving an $\mathcal{O}(n\sqrt{n\log n})$-time, $\mathcal{O}(n)$-space construction algorithm for all four arrays. In Theorem 5.2, we give $O(n\sqrt{G}\log n)$-time, $O(n)$-space algorithms for computing the prefix array, as well as the quantum and deterministic border and period arrays, improving all previously known algorithms when $G = o(n/\log n)$.

2 Preliminaries

A string S of length $n = |S|$ is a finite sequence of n characters over a finite alphabet Σ. The i-th character of S is denoted by $S[i]$, for $1 \le i \le n$, and we use $S[i\,..\,j]$ to denote the *fragment* $S[i]S[i+1]\ldots S[j]$ of S (if $i > j$, then $S[i\,..\,j]$ is the empty string). Moreover, we

[2] Throughout this work, the $\tilde{\mathcal{O}}(\cdot)$ notation suppresses factors that are polylogarithmic in the total length of the input string(s).

use $S[i\,..\,j)$ to denote the fragment $S[i\,..\,j-1]$ of S. A fragment $S[i\,..\,j]$ is a *prefix* of S if $i=1$ and a *suffix* of S if $j=n$.

In this paper, the alphabet Σ contains a special character $\lozenge$ that matches every character in the alphabet. Formally, we define the "match" relation, denoted $\sim$ and defined over $\Sigma \times \Sigma$, as follows: $\forall a,b \in \Sigma : a \sim b \Leftrightarrow a = b \vee a = \lozenge \vee b = \lozenge$. Its negation is denoted $a \nsim b$. We extend this relation to strings of equal length n by $X \sim Y \Leftrightarrow \forall i = 1,\dots,n : X[i] \sim Y[i]$.

Longest common extensions. Let T be a string of length n, and let $i,j \le n$ be indices. The longest common extension at i and j in T, denoted $\mathsf{LCE}_T(i,j)$ is defined as $\mathsf{LCE}_T(i,j) = \max\{\ell \le \min(n-i,n-j)+1 : T[i\,..\,i+\ell) = T[j\,..\,j+\ell)\}$. Similarly, the longest common extension *with wildcards* is defined using the $\sim$ relation instead of equality: $\mathsf{LCEW}_T(i,j) = \max\{\ell \le \min(n-i,n-j)+1 : T[i\,..\,i+\ell) \sim T[j\,..\,j+\ell)\}$.

We focus on data structures for LCEW queries inside a string T, but our results can easily be extended to answer queries between two strings P,Q, denoted $\mathsf{LCEW}_{P,Q}(i,j)$. If we consider $T = P \cdot Q$, then for any $i \le |P|$ and $j \le |Q|$, we have $\mathsf{LCEW}_{P,Q}(i,j) = \min(\mathsf{LCEW}_T(i,j+|P|), |P|-i+1, |Q|-j+1)$. When the string(s) that we query are clear from the context, we drop the T or P,Q subscripts.

Periodicity arrays. The *prefix array* of a string S of length n is an array π of size n such that $\pi[i] = \mathsf{LCEW}(1,i)$.

An integer $b \in [1\,..\,n]$ is a *quantum border* of S if $S[1\,..\,b] \sim S[n-b+1\,..\,n]$. It is a *deterministic border* of S if there exists a string X *without wildcards* such that $X \sim S$ and $X[1\,..\,b] = X[n-b+1\,..\,n]$. Similarly, an integer $p \le n$ is a *quantum period* of S if for every $i \le n-p$, $S[i] \sim S[i+p]$, and it is a *deterministic period* of S if there exists a string X *without wildcards* such that $X \sim S$ and for every $i \le n-p$, $X[i] = X[i+p]$.

▶ **Example 2.1.** Consider string `ab◊b◊bcb`. Its smallest quantum period is 2, while its smallest deterministic period is 4.

For a string S of length n, we define the following arrays of length n:
- the period array π, where $\pi[i] = \mathsf{LCEW}(1,i)$;
- the deterministic and quantum border arrays, B and B_Q, where $B[i]$ and $B_Q[i]$ are the largest deterministic and quantum border of $S[1\,..\,i]$, respectively;
- the deterministic and quantum period arrays, P and P_Q, such that $P[i]$ and $P_Q[i]$ are the smallest deterministic and quantum periods of $S[1\,..\,i]$, respectively.

▶ **Fact 2.2** (Lemmas 12 and 15 [41]). *Given the prefix array of a string S, one can compute the quantum border array and quantum period array in $\mathcal{O}(n)$ time and space, while the deterministic border and period arrays can be computed in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.*

3 Time-Space Trade-off for LCEW

In this section, we prove Theorem 1.2. Recall that $1 \le t \le G$. Following the work of Crochemore et al [27], we define *transition positions* in T, which are the positions at which T transitions from a block of wildcards to a block of non-wildcards characters. We use Tr to denote the set of transition positions in T. Formally, a position $i \in [1\,..\,n]$ is in Tr if one of the following holds:
- $i = n$,
- $i > 1$, $T[i-1] = \lozenge$ and $T[i] \neq \lozenge$.

Note that as T contains G groups of wildcards, there are at most $G+1$ transition positions, i.e., $|\mathsf{Tr}| = \mathcal{O}(G)$. Moreover, by definition, the only transition position i for which $T[i]$ may be a wildcard is n.

Our algorithm precomputes the LCEW information for a subset of evenly distributed transition positions, called *selected positions* and denoted Sel, whose number depends on the parameter t. The set Sel contains one in every t transition position in Tr, along with the last one (which is n). Formally, let $i_1 < i_2 < \ldots < i_r$ denote the transition positions of T, sorted in increasing order, then $\mathsf{Sel} = \{i_{st+1} : s = 0, \ldots, \lfloor (r-1)/t \rfloor\} \cup \{n\}$. Let λ denote the cardinality of Sel, which is $\mathcal{O}(G/t)$.

Additionally, for every $i \in [1\mathinner{.\,.} n]$, we define $\mathsf{next\_tr}[i]$ (resp., $\mathsf{next\_sel}[i]$) as the distance between i and the next transition position (resp., the next selected position) in T. Formally, $\mathsf{next\_tr}[i] = \min\{j - i : j \in \mathsf{Tr} \wedge j \geq i\}$ and $\mathsf{next\_sel}[i] = \min\{j - i : j \in \mathsf{Sel} \wedge j \geq i\}$. These values are well-defined: as n is both a transition and a selected position, the minimum in the above equations is never taken over the empty set. Both arrays can be computed in linear time and stored using $\mathcal{O}(n)$ space. The array next_tr can be used to jump from a wildcard to the end of the group of wildcards containing it, due the following property:

▶ **Observation 3.1.** *For any i such that $T[i] = \diamond$, let $r = \mathsf{next\_tr}[i]$. We have $T[i\mathinner{.\,.} i+r) = \diamond^r$, i.e., the fragment from i until the next transition position (exclusive) contains only wildcards.*

The central component of our data structure is a dynamic programming table, Jump, which allows us to efficiently answer LCEW queries for selected positions. For each selected position i and each (arbitrary) position j, this table stores the distance from i to the last selected position that appears in the common extension on the side of i, i.e. the last selected position i' for which $T[i\mathinner{.\,.} i']$ matches $T[j\mathinner{.\,.} j+i'-i]$. More formally:

$$\forall i \in \mathsf{Sel}, j \in [1\mathinner{.\,.} n] : \textsc{Jump}[i,j] = \max\{i' - i : i' \in \mathsf{Sel} \wedge i' \geq i \wedge T[i\mathinner{.\,.} i'] \sim T[j\mathinner{.\,.} j+i'-i]\}.$$

If there is no such selected position i' (which happens when $T[i] \nsim T[j]$), we let $\textsc{Jump}[i,j] = -\infty$. This table contains $\lambda \cdot n = \mathcal{O}(nG/t)$ entries and allows us to jump from the first to the last selected position in the common extension, thus reducing LCEW queries to finding longest common extensions *to* and *from* a selected position.

Finally, let $T_\#$ be the string obtained by replacing all wildcards in T with a new character "#" that does not appear in T. The string $T_\#$ does not contain wildcards, and for any $i, j \in [1\mathinner{.\,.} n]$, we have $\mathsf{LCEW}_T(i,j) \geq \mathsf{LCE}_{T_\#}(i,j)$.

The data structure. Our data structure consists of

- the Jump table,
- the arrays next_tr and next_sel, and
- a data structure for constant-time LCE queries in $T_\#$, with $\mathcal{O}(n)$ construction time and $\mathcal{O}(n)$ space usage (e.g. a suffix array or a suffix tree [29]).

The Jump table uses space $\mathcal{O}(nG/t)$ and can be computed in time $\mathcal{O}(n(G/t)\log n)$ (see Section 3.1), while the next_tr and next_sel arrays can be computed in $\mathcal{O}(n)$ time and stored using $\mathcal{O}(n)$ space. Therefore, our data structure can be built in $\mathcal{O}(n + n(G/t) \cdot \log n) = \mathcal{O}(n(G/t) \cdot \log n)$ time and requires $\mathcal{O}(n + nG/t) = \mathcal{O}(nG/t)$ space. As shown in Section 3.2, we can use this data structure to answer LCEW queries in T in time $\mathcal{O}(t)$, thus proving Theorem 1.2.

3.1 Computing the Jump Table

In this section, we prove the following lemma.

▶ **Lemma 3.2.** *Given random access to T, the* JUMP *table can be computed in $\mathcal{O}(n(G/t)\cdot\log n)$ time and $\mathcal{O}(nG/t)$ space.*

Proof. To compute the JUMP table, we leverage the algorithm of Clifford and Clifford for exact pattern matching with wildcards [20]. This algorithm runs in time $\mathcal{O}(n\log m)$, and finds all occurrences of a pattern of length m within a text of length n (both may contain wildcards).

Let $i_1 < i_2 < \ldots < i_\lambda = n$ denote the *selected* positions, sorted in the increasing order. For $r = 1, \ldots, \lambda - 1$, let P_r be the fragment of T from the r-th to the $(r+1)$-th selected position (exclusive), i.e., $P_r = T[i_r \mathinner{.\,.} i_{r+1})$, and let ℓ_r denote the length of P_r, that is $\ell_r = |P_r| = i_{r+1} - i_r$. Then, for every r, we use the aforementioned algorithm of Clifford and Clifford [20] to compute the occurrences of P_r in T: it returns an array A_r such that $A_r[i] = 1$ if and only if $T[i \mathinner{.\,.} i + \ell_r) \sim P_r$.

Using the arrays $(A_r)_r$, the JUMP table can then be computed with a dynamic programming approach, in the spirit of the computations in [27]. The base case is $i = i_\lambda$, for which we have, for all $j \in [1 \mathinner{.\,.} n]$, $\textsc{Jump}[i_\lambda, j] = 0$ if $T[i_\lambda] \sim T[j]$ and $-\infty$ otherwise. We can then fill the table by iterating over all pairs $(r, j) \in [1 \mathinner{.\,.} \lambda - 1] \times [1 \mathinner{.\,.} n]$ in the reverse lexicographical order and using the following recurrence relation:

$$\textsc{Jump}[i_r, j] = \begin{cases} -\infty & \text{if } T[i_r] \nsim T[j] \\ \max(0, \ell_r + \textsc{Jump}[i_{r+1}, j + \ell_r]) & \text{if } T[i_r] \sim T[j], A_r[j] = 1 \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

Computing the arrays $(A_r)_r$ takes $\mathcal{O}(\lambda \cdot n\log n) = \mathcal{O}(n(G/t) \cdot \log n)$ time in total. Computing the JUMP table from the arrays takes constant time per cell, and the table contains $\lambda \cdot n = \mathcal{O}(nG/t)$ cells. Thus, the JUMP table can be computed in time $\mathcal{O}(n(G/t) \cdot \log n)$. ◀

3.2 Answering LCEW Queries

Our algorithm to answer an LCEW query can be decomposed into the following steps:

(a) move forward in T until we reach a selected position or a mismatch,
(b) use the JUMP table to skip to the last selected position in the longest common prefix on the side of the selected position,
(c) move forward until we either reach a mismatch or the end of the text.

Steps (a) and (b) might have to be performed twice, one for each of the "sides" of the query. Steps (a) and (c) can be handled similarly, using LCE queries in $T_\#$ to move forward multiple positions at a time: see Algorithm 1 for the pseudo-code for these steps, and see Algorithm 2 for the pseudo-code of the query procedure.

Analysis of the NextSelectedOrMismatch subroutine (Algorithm 1). Algorithm 1 computes a value ℓ such that $T[i \mathinner{.\,.} i + \ell) \sim T[j \mathinner{.\,.} j + \ell)$, and either $T[i + \ell] \nsim T[j + \ell]$ or at least one of $i + \ell, j + \ell$ is a selected position. In the latter case, $i + \ell$ (resp. $j + \ell$) is the first selected position after i (resp. j). Furthermore, Algorithm 1 runs in time $\mathcal{O}(t)$. These properties are formally stated below. Proofs of Lemmas 3.3–3.5 can be found in the full version of the paper.

▶ **Lemma 3.3.** *Let ℓ be the value returned by Algorithm 1. We have $T[i \mathinner{.\,.} i+\ell) \sim T[j \mathinner{.\,.} j+\ell)$.*

Algorithm 1 Subroutine for LCEW queries.

```
1: function NextSelectedOrMismatch(i, j)
2:     ℓ ← 0
3:     m ← min(next_sel[i], next_sel[j])
4:     while T[i + ℓ] ∼ T[j + ℓ] and i + ℓ ∉ Sel and j + ℓ ∉ Sel do
5:         r ← LCE_{T_#}(i + ℓ, j + ℓ)
6:         ℓ ← min(ℓ + r, m)
7:         d ← 0
8:         if T[i + ℓ] = ◊ then
9:             d ← max(d, next_tr[i + ℓ])
10:        if T[j + ℓ] = ◊ then
11:            d ← max(d, next_tr[j + ℓ])
12:        ℓ ← min(ℓ + d, m)
13:    return ℓ
```

The fact that either $T[i+\ell] \nsim T[j+\ell]$ or at least one of $i+\ell, j+\ell$ is a selected position follows from the exit condition of the `while` loop. If one of them is a selected position, the minimality of its index follows from using $m = \min(\mathsf{next\_sel}[i], \mathsf{next\_sel}[j])$ to bound the value of ℓ throughout the algorithm. This concludes the proof of the correctness of Algorithm 1.

We now turn to proving that Algorithm 1 runs in time $\mathcal{O}(t)$. We use the following properties to bound the number of loop iterations.

▶ **Lemma 3.4.** *In Line 6 of Algorithm 1, either $T[i+\ell] \nsim T[j+\ell]$, or (at least) one of $T[i+\ell], T[j+\ell]$ is a selected position or a wildcard.*

▶ **Lemma 3.5.** *In Line 12 of Algorithm 1, either $T[i+\ell] \nsim T[j+\ell]$, or (at least) one of $T[i+\ell], T[j+\ell]$ is a selected position or a transition position.*

By Lemma 3.5, the number of transition positions between $i+\ell$ or $j+\ell$ and the corresponding next selected position decreases by at least one (or the algorithm exits the loop and returns). The use of m in Lines 6 and 12 ensures that we cannot go over a selected position, and, by construction, there are at most t transition positions between i or j and the next selected position, therefore Algorithm 1 goes through at most $2t$ iterations of the loop. Each iteration consists of one LCE query in $T_\#$ and a constant number of constant-time operations, hence Algorithm 1 takes time $\mathcal{O}(t)$ overall given a data structure for constant-time LCE queries on $T_\#$.

LCEW query algorithm. Let ℓ denote the result of Algorithm 2 on some input (i, j). The properties of Algorithm 1 ensure that $T[i \mathinner{.\,.} i+\ell) \sim T[j \mathinner{.\,.} j+\ell)$. As the algorithm returns when it either encounters a mismatch or reaches the end of the string, the matching fragment cannot be extended, which ensures the maximality of ℓ. To prove that Algorithm 2 runs in time $\mathcal{O}(t)$, we show in the full version of the paper that it makes a constant number of loop iterations.

▶ **Lemma 3.6.** *The `while` loop of Algorithm 2 makes at most three iterations.*

Algorithm 2 Algorithm to answer the query LCEW(i, j).

```
function Query(i, j)
    ℓ ← 0
    while i + ℓ ≤ n and j + ℓ ≤ n do
        ℓ ← NextSelectedOrMismatch(i + ℓ, j + ℓ)
        if T[i + ℓ] ≁ T[j + ℓ] then
            return ℓ
        if i + ℓ ∈ Sel then
            ℓ ← ℓ + Jump[i + ℓ, j + ℓ] + 1
        else
            ℓ ← ℓ + Jump[j + ℓ, i + ℓ] + 1
    return ℓ
```

Intuitively, the first call to NextSelectedOrMismatch finds the last selected position in $T[i\,..\,i+\ell)$ or in $T[j\,..\,j+\ell)$, the second call finds the last selected position in the other fragment, and the last call finds either a mismatch or the end of the text. Therefore, Algorithm 2 makes up to three calls to Algorithm 1 plus a constant number of operations, and thus runs in time $\mathcal{O}(t)$.

4 Connection to Boolean Matrix Multiplication

In this section, we describe a fine-grained connection between LCEW and (sparse) Boolean matrix multiplication. In Section 4.1, we use this connection to obtain a lower bound on the preprocessing-query-time product of combinatorial data structures for LCEW. In Section 4.2, we further connect sparse matrices and strings with few groups of wildcards, deriving an efficient multiplication algorithm.

4.1 A Lower Bound for Combinatorial Data Structures

Our lower bound is based on the combinatorial matrix multiplication conjecture which states that for any $\varepsilon > 0$ there is no combinatorial algorithm for multiplying two $n \times n$ Boolean matrices working in time $\mathcal{O}(n^{3-\varepsilon})$. Gawrychowski and Uznański [33, Conjecture 3.1] showed that the following formulation is equivalent to this conjecture:

▶ **Conjecture 4.1** (Combinatorial matrix multiplication conjecture). *For every $\varepsilon > 0$ and $\alpha, \beta, \gamma > 0$, there is no combinatorial algorithm that computes the product of Boolean matrices of dimensions $n^\alpha \times n^\beta$ and $n^\beta \times n^\gamma$ in time $\mathcal{O}(n^{\alpha+\beta+\gamma-\varepsilon})$.*

▶ **Theorem 4.2.** *Under Conjecture 4.1, there is no combinatorial data structure that solves the LCEW problem with preprocessing time $\mathcal{O}(n^a)$ and query time $\mathcal{O}(n^b)$, where a and b are fixed real numbers (independent of n), $a \geq b \geq 0$, and $a + b < 2 - \varepsilon$ for some constant $\varepsilon > 0$.*

Proof. Assume for the sake of contradiction that there exists such a data structure. We can then use it to derive an algorithm that contradicts Conjecture 4.1.

Let α, β be positive constants, and let A and B be rectangular Boolean matrices of respective dimensions $p \times q$ and $q \times p$, where $p = n^\alpha, q = n^\beta$ and $\beta = c \cdot \alpha$ (i.e., $q = p^c$) for a constant c to be fixed later. We encode A into a string S_A of length $pq = p^{c+1}$ in row-major order, that is, $S_A[qi+j+1] = \phi_A(A[i+1, j+1])$ for $i \in [0\,..\,p), j \in [0\,..\,q)$, and B into a string

S_B of length $qp = p^{c+1}$ in column-major order, that is, $S_B[i + qj + 1] = \phi_B(B[i + 1, j + 1])$ for $i \in [0\,..\,q), j \in [0\,..\,p)$, where:

$$\phi_A(x) = \begin{cases} 1 & \text{if } x = 1 \\ \diamond & \text{if } x = 0 \end{cases} \text{ and } \phi_B(y) = \begin{cases} 2 & \text{if } y = 1 \\ \diamond & \text{if } y = 0 \end{cases}$$

It follows from this definition that $\phi_A(x)$ does not match $\phi_B(y)$, i.e., $\phi_A(x) \not\approx \phi_B(y)$, if and only if $x = y = 1$. The following claim is proved in the full version of the paper.

▷ Claim 4.3. We have $(AB)[i, j] = 1 \iff \mathsf{LCEW}_{S_A,S_B}(qi + 1, qj + 1) < q$.

Therefore, we can compute each entry of $C = AB$ using one LCEW query between S_A and S_B. To perform LCEW queries between S_A and S_B, we instantiate our LCEW data structure on the string S_AS_B, which has length $2pq = \mathcal{O}(p^{c+1})$, and therefore has $G = \mathcal{O}(p^{c+1})$ consecutive wildcard groups. Computing C then takes $\mathcal{O}(p^2 \cdot p^{b(c+1)})$ time for the queries, plus $\mathcal{O}(p^{a(c+1)})$ time to build the data structure.

Assume first that $a > b$. We set $c = 2/(a - b) - 1 > 0$ to balance the terms of the above two expressions, and obtain an algorithm that computes C in time $\mathcal{O}(p^{2a/(a-b)})$. This contradicts Conjecture 4.1 if this running time is $\mathcal{O}(p^{(2+c-\delta)})$ for some constant $\delta > 0$. In terms of exponents, this happens when $2a/(a - b) < 2 + c - \delta$. Let $\delta = \varepsilon/(a - b) > 0$. We have:

$$\begin{aligned} a + b < 2 - \varepsilon \Leftrightarrow 2a &< a - b + 2 - \varepsilon \\ \Leftrightarrow \frac{2a}{a-b} &< 1 + \frac{2}{a-b} - \frac{\varepsilon}{a-b} \\ \Leftrightarrow \frac{2a}{a-b} &< 2 + \frac{2}{a-b} - 1 - \delta \\ \Leftrightarrow \frac{2a}{a-b} &< 2 + c - \delta \end{aligned} \tag{2}$$

Equation (2) results in a contradiction for all $a > b$ that satisfy $a + b < 2 - \varepsilon$. Assume now that $a = b$. We set $c = 2/\delta$ for $\delta = \varepsilon/2$ and obtain:

$$2 + b(1 + c) < 2 + (1 - \delta)(1 + 2/\delta) = (2 + c)(1 - \delta/2) \leq 2 + c - \delta/2 \tag{3}$$

We hence obtain the desired contradiction for $a = b$ with $a + b < 2 - \varepsilon$ as well, thus concluding the proof. ◀

4.2 Fast Sparse Matrix Multiplication

In this section, we further connect sparse matrices and strings with few groups of wildcards, deriving an algorithm for sparse Boolean matrix multiplication (BMM).

▶ **Observation 4.4.** *Let A and B be sparse matrices with m_{in} ones in total. The string S_AS_B contains $G \leq m_{in} + 1$ groups of consecutive wildcards.*

Furthermore, our reduction to LCEW can also exploit the sparsity of the output matrix C. The algorithm underlying the proof of Theorem 4.2 uses $pr = p^2$ LCEW queries to compute C, one query for each entry. When the output matrix contains at most m_{out} non-zero values, we show in the full version of the paper that we can reduce the number of LCEW queries to $2p + m_{out} - 1$, using the following lemma.

▶ **Lemma 4.5.** *Let t be an integer, and let $i, j < p-t$. We have $\mathsf{LCEW}_{S_A,S_B}(qi+1, qj+1) \geq q \cdot t$ if and only if for every $x < t$, $(AB)[i + x, j + x] = 0$.*

The above lemma readily implies that the answer to an LCEW query at the first indices of a diagonal gives us the length of a longest prefix run of zeroes in this diagonal. A repeated application of this argument implies that computing the entries in the d-th diagonal of C takes $m_d + 1$ LCEW queries, where m_d is the number of non-zero entries in this diagonal. Summing over all $2p-1$ diagonals, this gives a total of $2p - 1 + m_{out}$ queries. As a corollary, we obtain the following algorithm for the multiplication of sparse matrices.

▶ **Theorem 4.6.** *Let A and B be $n \times n$ sparse Boolean matrices such that A and B contain m_{in} non-zero entries and $C = (AB)$ contains m_{out} non-zero entries. There is a deterministic combinatorial algorithm for computing C that runs in time $\mathcal{O}(n\sqrt{m_{in} \cdot (n + m_{out})} \log^2 n)$.*

Proof. We assume that the matrices are given as a list of coordinates of 1 bits, sorted by row index and then by column index: this compact representation has size $\mathcal{O}(m_{in})$.

We can assume without loss of generality that $m_{in} \geq n$, otherwise we can remove the empty rows and columns (i.e., rows and columns with zeroes only) from A and B (an empty row in A induces an empty row in C, while an empty column in B induces an empty column in C), and pad the output with zeroes where necessary. (For sparse matrices, this means offsetting the indices of the non-zeroes.) This procedure takes $\mathcal{O}(m_{in} + m_{out})$ time overall.

Consider the string $S = S_A S_B$ defined in the proof of Theorem 4.2, which has length $2n^2$ and contains $G \leq m_{in} + 1$ groups of wildcards, and let $t = n\sqrt{G/(n + m_{out})}$. If $t \leq G$, we instantiate the data structure of Theorem 1.2 for S with this parameter t, and if $t > G$, we use the data structure of Corollary 1.3. This is always possible as $t \leq n\sqrt{G} \leq G \cdot |S|^{1/2}$. Then, using the argument described just above this theorem, we can compute C using $2n + m_{out} - 1$ LCEW queries.

For $t \leq G$, construction takes time $\mathcal{O}(n^2(G/t) \cdot \log n) = \mathcal{O}(n\sqrt{m_{in} \cdot (n + m_{out})} \log n)$ and answering the queries takes total time $\mathcal{O}((n + m_{out}) \cdot t) = \mathcal{O}(n\sqrt{m_{in} \cdot (n + m_{out})})$. In the other case, constructing the data structure takes $O(n^2)$ time, and answering the queries takes the same time as in the first case.

Accounting for the preprocessing to ensure that $m_{in} \geq n$, the total running time is $\mathcal{O}(n^2 + n\sqrt{m_{in} \cdot (n + m_{out})} \log n + m_{in} + m_{out})$. As $n \leq m_{in} \leq n^2$ and $m_{out} \leq m_{in}^2$, the $n\sqrt{m_{in} \cdot (n + m_{out})}$ term is asymptotically larger than $n^2 + m_{in} + m_{out}$. This yields the desired runtime.

Choosing t requires knowing an estimate of m_{out}: if it is not known, we estimate it using binary search between 1 and n^2. For a given estimate of m_{out}, we run the algorithm, halting and restarting whenever the total query time exceeds the construction time. This search adds an extra $\mathcal{O}(\log n)$ factor in the time complexity. ◀

5 Faster Approximate Pattern Matching and Computation of Periodicity Arrays construction

In this section, we use the data structure of Theorem 1.2 to derive improved algorithms for the k-PMWE problem and the problem of computing periodicity arrays of strings with wildcards.

5.1 Faster Pattern Matching with Errors and Wildcards

We first consider the problem of pattern matching with errors, where both the pattern and the text may contain wildcards. Recall that the edit distance between two strings X, Y, denoted by $\text{ed}(X, Y)$, is the smallest number of insertions, deletions, and substitutions of a character, required to transform X into Y. This problem is formally defined as follows:

k-PMWE
Input: A text T of length n, a pattern P of length m and an integer threshold k.
Output: Every position p for which there exists $i \le p$ such that $\mathrm{ed}(T[i\,..\,p], P) \le k$.

Akutsu [4] gave an algorithm for this problem that runs in time $\tilde{\mathcal{O}}(n\sqrt{km})$. Using their framework, we show that the complexity can be reduced to $\mathcal{O}(n(k + \sqrt{kG \log m}))$, where G is the cumulative number of groups of wildcards in P and T (or equivalently, the number of groups of wildcards in $P\$T$). [3]

▶ **Theorem 5.1.** *There is an algorithm for k-PMWE that runs in time $\mathcal{O}(n(k + \sqrt{kG \log m}))$.*

Proof. Akutsu [4, Proposition 1] shows that, if after an α-time preprocessing, LCEW queries between P and T can be answered in time $\beta \ge 1$, then the k-PMWE problem can be solved in time $\mathcal{O}(\alpha + n\beta k)$. First, assume that $G \log m \ge k$. We use the data structure of Theorem 1.2 with $t = \sqrt{(G/k) \cdot \log m} \ge 1$ to answer LCEW queries: we then have $\alpha = \mathcal{O}(n\sqrt{Gk \log m})$ (here we use the standard trick to replace the $\log n$ factor in the construction time with $\log m$, namely, if $n \ge 2m$, we divide T into n/m blocks of length $\le 2m$ overlapping by m characters, and build such a data structure for each block independently) and $\beta = \mathcal{O}(t) = \mathcal{O}(\sqrt{(G/k) \cdot \log m})$. Therefore, the running time of the algorithm is $\mathcal{O}(n\sqrt{Gk \log m})$. Second, if $G \log m < k$, we simply set $t = 1$: the total running time is then $\mathcal{O}(nG \log m + nk) = \mathcal{O}(nk)$. Accounting for both cases, the time complexity of this algorithm is $\mathcal{O}(n(k + \sqrt{Gk \log m}))$. ◀

5.2 Faster Computation of Periodicity Arrays

Our data structure also enables us to obtain efficient algorithms for computing periodicity arrays of a string with wildcards (Theorem 5.2). These algorithms build on and improve upon the results of Iliopoulos and Radoszewski [41].

▶ **Theorem 5.2.** *Let S be a string of length n with G groups of wildcards. The prefix array, the quantum and deterministic border arrays and the quantum and deterministic period arrays of S can be computed in $\mathcal{O}(n\sqrt{G} \log n)$ time and $\mathcal{O}(n)$ space.*

By Fact 2.2, it remains to show that the prefix array of S can be computed in $\mathcal{O}(n\sqrt{G} \log n)$ time and $\mathcal{O}(n)$ space. Recall that the *prefix array* of a string S of length n is an array π of size n such that $\pi[i] = \mathsf{LCEW}(1, i)$. Consequently, π can be computed using n LCEW queries in S. By instantiating our data structure with $t = \sqrt{G}$, we obtain an algorithm running in $\mathcal{O}(n\sqrt{G} \log n)$ time, but its space usage is $\Theta(n\sqrt{G})$. Below, we show how one can slightly modify the data structure of Theorem 1.2 to reduce the space complexity to $\mathcal{O}(n)$, extending the ideas of [41].

▶ **Lemma 5.3.** *Let S be a string of length n with G groups of wildcards. The prefix array of S can be computed in $\mathcal{O}(n\sqrt{G} \log n)$ time and $\mathcal{O}(n)$ space.*

Proof. We add the index 1 to the set of selected positions Sel and preprocess S in $\mathcal{O}(n)$ time and space to support LCE queries on $S_{\#}$ in $\mathcal{O}(1)$ time [29].

Notice that, using the dynamic programming algorithm of Lemma 3.2, for any $r < \lambda$, the row $(\textsc{Jump}[i_r, j], j = 1, \ldots, n)$ of the $\textsc{Jump}$ table can be computed in time $\mathcal{O}(n \log n)$

[3] The additive "k" term in our complexity is necessary because $G \log m$ might be smaller than k. On the other hand, one can assume w.l.o.g. that $m \ge k$. Hence this additional term is hidden in the runtime of Akutsu's algorithm [4].

and $\mathcal{O}(n)$ space from the next row ($\text{JUMP}[i_{r+1}, j], j = 1, \ldots, n$). It suffices to compute the array A_r of occurrences of P_r in time $\mathcal{O}(n \log n)$ using the algorithm of Clifford and Clifford [20], and then apply the recurrence relation of Equation (1).

To answer an $\mathsf{LCEW}(1, j)$ query, we perform the following steps: first, we issue a $\text{JUMP}[1, j]$ query followed by at most t regular LCE queries. If after this process we reach a mismatch or the end of S, we are done. Otherwise, we need to perform another JUMP query from indices (ℓ_j, i_{r_j}), where $i_{r_j} = j + \ell_j - 1$ is a selected position, and then perform at most t more regular LCE queries from the resulting positions. We store the indices (ℓ_j, i_{r_j}) for every value j, grouped by selected position i_{r_j}.

To answer the first batch of $\text{JUMP}[1, j]$ queries, we use the above observation iterated $\lambda - 1$ times to compute the first row of the JUMP table in $\mathcal{O}(n(G/t) \cdot \log n)$ time and $\mathcal{O}(n)$ space. To answer the next batch of queries (i.e., $\text{JUMP}[\ell_j, i_{r_j}]$ queries) in $\mathcal{O}(n(G/t) \cdot \log n)$ time and $\mathcal{O}(n)$ space, we again use the above observation to iterate over the rows of the JUMP table, starting from row i_λ and going up, storing only one row at a time. After computing the row corresponding to a selected position i_r, we answer all JUMP queries with $i_{r_j} = i_r$ and then perform the remaining LCE queries to answer the corresponding $\mathsf{LCEW}(1, j)$ query.

The claimed bounds follow by setting $t = \sqrt{G} \leq G$. ◀

References

1 Amir Abboud, Karl Bringmann, Nick Fischer, and Marvin Künnemann. The time complexity of fully sparse matrix multiplication. In *Proc. of SODA 2024*, pages 4670–4703, 2024. doi:10.1137/1.9781611977912.167.

2 Amir Abboud, Virginia Vassilevska Williams, and Oren Weimann. Consequences of faster alignment of sequences. In *Proc. of ICALP 2014*, pages 39–51, 2014. doi:10.1007/978-3-662-43948-7_4.

3 Peyman Afshani and Jesper Sindahl Nielsen. Data structure lower bounds for document indexing problems. In *Proc. of ICALP 2016*, pages 93:1–93:15, 2016. doi:10.4230/LIPICS.ICALP.2016.93.

4 Tatsuya Akutsu. Approximate string matching with don't care characters. *Inf. Process. Lett.*, 55(5):235–239, 1995. doi:10.1016/0020-0190(95)00111-O.

5 Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2):257–275, 2004.

6 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. A new characterization of maximal repetitions by Lyndon trees. In *Proc. of SODA 2015*, pages 562–571, 2015. doi:10.1137/1.9781611973730.38.

7 Gabriel Bathie, Panagiotis Charalampopoulos, and Tatiana Starikovskaya. Pattern matching with mismatches and wildcards. *CoRR*, 2024. doi:10.48550/arXiv.2402.07732.

8 Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, and Inge Li Gørtz. Finger search in grammar-compressed strings. *Theory of Computing Systems*, 62:1715–1735, 2018. doi:10.1007/S00224-017-9839-9.

9 Philip Bille, Inge Li Gørtz, Patrick Hagge Cording, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind. Fingerprints in compressed strings. *Journal of Computer and System Sciences*, 86:171–180, 2017. doi:10.1016/J.JCSS.2017.01.002.

10 Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In *Proc. of CPM 2015*, pages 65–76, 2015. doi:10.1007/978-3-319-19929-0_6.

11 Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time–space trade-offs for longest common extensions. *Journal of Discrete Algorithms*, 25:42–50, 2014. doi:10.1016/J.JDA.2013.06.003.

12 Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory Comput. Syst.*, 55(1):41–60, 2014. doi:10.1007/S00224-013-9498-4.

13 Or Birenzwige, Shay Golan, and Ely Porat. Locally consistent parsing for text indexing in small space. In *Proc. of SODA 2020*, pages 607–626, 2020. doi:10.1137/1.9781611975994.37.

14 Francine Blanchet-Sadri, Rachel Harred, and Justin Lazarow. Longest common extensions in partial words. In *Proc. of IWOCA 2015*, volume 9538 of *Lecture Notes in Computer Science*, pages 52–64. Springer, 2015. doi:10.1007/978-3-319-29516-9_5.

15 Francine Blanchet-Sadri and Justin Lazarow. Suffix trees for partial words and the longest common compatible prefix problem. In *Proc. of LATA 2013*, pages 165–176, 2013. doi:10.1007/978-3-642-37064-9_16.

16 Francine Blanchet-Sadri and S. Osborne. Computing longest common extensions in partial words. *Discret. Appl. Math.*, 246:119–139, 2018. doi:10.1016/J.DAM.2016.06.007.

17 Béla Bollobás and Shoham Letzter. Longest common extension. *Eur. J. Comb.*, 68:242–248, 2018. doi:10.1016/J.EJC.2017.07.019.

18 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *Proc. of FOCS 2020*, pages 978–989, 2020.

19 Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest palindromic substring in sublinear time. In *Proc. of CPM 2022*, volume 223, pages 20:1–20:9, 2022. doi:10.4230/LIPICS.CPM.2022.20.

20 Peter Clifford and Raphaël Clifford. Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53–54, 2007.

21 Raphaël Clifford, Klim Efremenko, Ely Porat, and Amir Rothschild. From coding theory to efficient pattern matching. In *Proc. of SODA 2009*, pages 778–784, 2009.

22 Raphaël Clifford, Klim Efremenko, Ely Porat, and Amir Rothschild. Pattern matching with don't cares and few errors. *J. Comput. Syst. Sci.*, 76(2):115–124, 2010. doi:10.1016/j.jcss.2009.06.002.

23 Raphaël Clifford, Allan Grønlund, Kasper Green Larsen, and Tatiana Starikovskaya. Upper and lower bounds for dynamic data structures on strings. In *Proc. of STACS 2018*, pages 22:1–22:14, 2018. doi:10.4230/LIPICS.STACS.2018.22.

24 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. of STOC 2004*, pages 91–100, 2004. doi:10.1145/1007352.1007374.

25 Richard Cole and Ramesh Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. of STOC 2002*, pages 592–601, 2002. doi:10.1145/509907.509992.

26 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

27 Maxime Crochemore, Costas S Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Jakub Radoszewski, Wojciech Rytter, Bartosz Szreder, and Tomasz Waleń. A note on the longest common compatible prefix problem for partial words. *Journal of Discrete Algorithms*, 34:49–53, 2015. doi:10.1016/J.JDA.2015.05.003.

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

29 Johannes Fischer and Volker Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proc. of CPM 2006*, pages 36–48, 2006.

30 Nick Fischer. Deterministic sparse pattern matching via the Baur-Strassen theorem. In *Proc. of SODA 2024*, pages 3333–3353, 2024. doi:10.1137/1.9781611977912.119.

31 Zvi Galil and Raffaele Giancarlo. Improved string matching with k mismatches. *ACM SIGACT News*, 17(4):52–54, 1986.

32 Pawel Gawrychowski and Tomasz Kociumaka. Sparse suffix tree construction in optimal time and space. In *Proc. of SODA 2017*, pages 425–439, 2017. doi:10.1137/1.9781611974782.27.

33 Pawel Gawrychowski and Przemyslaw Uznanski. Towards unified approximate pattern matching for Hamming and L_1 distance. In *Proc. of ICALP 2018*, volume 107 of *LIPIcs*, pages 62:1–62:13, 2018. `doi:10.4230/LIPICS.ICALP.2018.62`.

34 Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. Optimal dynamic strings. In *Proc. of SODA 2018*, pages 1509–1528, 2018. `doi:10.1137/1.9781611975031.99`.

35 Shay Golan, Tsvi Kopelowitz, and Ely Porat. Streaming pattern matching with d wildcards. *Algorithmica*, 81(5):1988–2015, 2019. `doi:10.1007/S00453-018-0521-7`.

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

37 Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, September 1978. `doi:10.1145/355791.355796`.

38 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984. `doi:10.1137/0213024`.

39 Jan Holub and William F. Smyth. Algorithms on indeterminate strings. In *Proc. of IWOCA 2003*, pages 36–45, 2003.

40 Costas S. Iliopoulos, Manal Mohamed, Laurent Mouchard, Katerina Perdikuri, William F. Smyth, and Athanasios K. Tsakalidis. String regularities with don't cares. In *Proc. of PSC 2002*, pages 65–74, 2002. URL: `http://www.stringology.org/event/2002/p8.html`.

41 Costas S. Iliopoulos and Jakub Radoszewski. Truly Subquadratic-Time Extension Queries and Periodicity Detection in Strings with Uncertainties. In *Proc. of CPM 2016*, volume 54, pages 8:1–8:12, 2016. `doi:10.4230/LIPIcs.CPM.2016.8`.

42 Piotr Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *Proc. of FOCS 1998*, pages 166–173, 1998. `doi:10.1109/SFCS.1998.743440`.

43 Adam Kalai. Efficient pattern-matching with don't cares. In *Proc. of SODA 2022*, pages 655–656, 2002.

44 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proc. of STOC 2019*, pages 756–767, 2019. `doi:10.1145/3313276.3316368`.

45 Dominik Kempa and Tomasz Kociumaka. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In *Proc. of FOCS 2023*, pages 1877–1886, 2023. `doi:10.1109/FOCS57990.2023.00114`.

46 Dominik Kempa and Barna Saha. An upper bound and linear-space queries on the LZ-end parsing. In *Proc. of SODA 2022*, pages 2847–2866, 2022. `doi:10.1137/1.9781611977073.111`.

47 Tomasz Kociumaka. *Efficient Data Structures for Internal Queries in Texts*. Phd thesis, University of Warsaw, October 2018. Available at `https://www.mimuw.edu.pl/~kociumaka/files/phd.pdf`.

48 Roman Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *Proc. of FOCS 1999*, pages 596–604, 1999.

49 Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theoretical Computer Science*, 410(51):5365–5373, 2009.

50 Dmitry Kosolobov. Tight lower bounds for the longest common extension problem. *Information Processing Letters*, 125:26–29, 2017. `doi:10.1016/J.IPL.2017.05.003`.

51 Dmitry Kosolobov and Nikita Sivukhin. Construction of sparse suffix trees and LCE indexes in optimal time and space. *CoRR*, abs/2105.03782, 2021. `arXiv:2105.03782`.

52 Marvin Künnemann. On nondeterministic derandomization of Freivalds' algorithm: Consequences, avenues and algorithmic progress. In *Proc. of ESA 2018*, volume 112 of *LIPIcs*, pages 56:1–56:16, 2018. `doi:10.4230/LIPIcs.ESA.2018.56`.

53 Konstantin Kutzkov. Deterministic algorithms for skewed matrix products. In *Proc. of STACS 2013*, volume 20 of *LIPIcs*, pages 466–477, 2013. `doi:10.4230/LIPIcs.STACS.2013.466`.

54 Gad M Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

55 Gad M. Landau and Uzi Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *Proc. of STOC 1986*, pages 220–230, 1986. doi:10.1145/12130.12152.

56 Florin Manea, Robert Mercas, and Catalin Tiseanu. An algorithmic toolbox for periodic partial words. *Discret. Appl. Math.*, 179:174–192, 2014. doi:10.1016/J.DAM.2014.07.017.

57 Michael S Paterson Michael J Fischer. String-matching and other products. In *SCC*, pages 113–125, 1974.

58 Marius Nicolae and Sanguthevar Rajasekaran. On string matching with mismatches. *Algorithms*, 8(2):248–270, 2015. doi:10.3390/a8020248.

59 Marius Nicolae and Sanguthevar Rajasekaran. On pattern matching with k mismatches and few don't cares. *Inf. Process. Lett.*, 118:78–82, 2017. doi:10.1016/j.ipl.2016.10.003.

60 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully Dynamic Data Structure for LCE Queries in Compressed Space. In *Proc. of MFCS 2016*, volume 58 of *LIPIcs*, pages 72:1–72:14, 2016. doi:10.4230/LIPIcs.MFCS.2016.72.

61 Nicola Prezza. Optimal substring equality queries with applications to sparse text indexing. *ACM Trans. Algorithms*, 17(1), December 2021. doi:10.1145/3426870.

62 Mihai Pătrașcu. Unifying the landscape of cell-probe lower bounds. *SIAM J. Comput.*, 40(3):827–847, 2011. doi:10.1137/09075336X.

63 Yuka Tanimura, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, Simon J. Puglisi, and Masayuki Takeda. Deterministic Sub-Linear Space LCE Data Structures With Efficient Construction. In *Proc. of CPM 2016*, volume 54 of *LIPIcs*, pages 1:1–1:10, 2016. doi:10.4230/LIPIcs.CPM.2016.1.

64 Yuka Tanimura, Takaaki Nishimoto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Small-space LCE data structure with constant-time queries. In *Proc. of MFCS 2017*, 2017. doi:10.4230/LIPICS.MFCS.2017.10.

65 I Tomohiro. Longest common extensions with recompression. In *Proc. of CPM 2017*, volume 78, page 18, 2017. doi:10.4230/LIPICS.CPM.2017.18.

66 Dirk Van Gucht, Ryan Williams, David P. Woodruff, and Qin Zhang. The communication complexity of distributed set-joins with applications to matrix multiplication. In *Proc. of PODS 2015*, pages 199–212, 2015. doi:10.1145/2745754.2745779.