Sufficient Conditions for Efficient Indexing Under Different Matchings

Amihood Amir
Department of Computer Science, Bar-Ilan University, Israel
https://u.cs.biu.ac.il/~amir
amir@esc.biu.ac.il

Eitan Kondratovsky
Department of Computer Science, Bar-Ilan University, Israel
kondrae@cs.biu.ac.il

Abstract

The most important task derived from the massive digital data accumulation in the world, is *efficient access* to this data, hence the importance of indexing. In the last decade, many different types of *matching relations* were defined, each requiring an efficient indexing scheme. Cole and Hariharan in a ground breaking paper [Cole and Hariharan, SIAM J. Comput., 33(1):26–42, 2003], formulate *sufficient conditions* for building an efficient indexing for *quasi-suffix collections*, collections that behave as suffixes. It was shown that known matchings, including parameterized, 2-D array and order preserving matchings, fit their indexing settings. In this paper, we formulate more basic sufficient conditions based on the *order relation* derived from the *matching relation* itself, our conditions are more general than the previously known conditions.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases off-the-shelf indexing algorithms, general matching relations, weaker sufficient conditions for indexing

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.6

Funding *Amihood Amir*: Partly supported by ISF grant 1475/18 and BSF grant 2014028.
Eitan Kondratovsky: Partly supported by ISF grant 1475/18.

1 Introduction

Having grown accustomed to the presence of all the web data at our fingertips, it is easy to forget that a mere 25 years ago, an encyclopedia on a CD was the height of information technology. At the root of efficient access of this staggering amount of data is *indexing*. The concept of indexing is that one spends time and effort preprocessing the entire data and constructing auxiliary data structures that will make it possible to efficiently answer future queries of the form: "does input instance I appear in our data?".

The term "appear" is application dependent. It may be an exact matching of a word in a text, it may be searching for a picture of a face in photographs, it may be seeking a gene in the DNA sequence, looking for a tune in a music database, or many other needs in various areas.

The simplest and earliest form of the problem is indexing texts for exact matching. Two strings are said to be matched if and only if they are exactly the same string. This problem has been heavily researched over the last decades and led to the development of two important data structures – the *suffix tree* and the *suffix array*.

The suffix tree is a compacted trie of all the suffixes of T. Its size is $O(n)$ and, starting with Weiner's algorithm [28], various efficient construction algorithms with numerous properties have been proposed (e.g. [23, 26, 11, 4, 18]).

The suffix array [22, 14] is the lexicographically sorted array of suffixes.

The next step was considering more complex matching relations. Two such relations are *parameterized matching* [5, 6, 3] and *order-preserving matching* [17, 19, 16].

Parameterized matching was introduced by Baker [5, 6]. Her main motivation lay in software maintenance, where program fragments are to be considered "identical" even if variable names are different. Baker developed an algorithm to build parameterized suffix trees. Her algorithm essentially takes $O(n(|\Pi| + \log(|\Sigma| + |\Pi|)))$ preprocessing time and $O(m + occ)$ subsequent query time, where Σ and Π are alphabets of fixed symbols and parameter symbols respectively, m is the query's length and occ is the number of pattern occurrences in the text.

Order Preserving matching has been introduced independently by Kubica et al. [19] and by Kim et al. [17]. Crochemore et al. [10] developed a direct suffix tree construction for the order preserving matching, that works in $O(n\sqrt{\log n})$ preprocessing time and $O(m)$ query time if the pattern alphabet is of size $poly(m)$, where m is the pattern length. Otherwise, it can be made by sorting, i.e. in $O(m\sqrt{\log\log m})$ randomized time or in $O(m \log\log m)$ deterministic time. Reporting the pattern occurrences requires adding an extra $O(occ)$ time.

Another matching relation whose indexing problem has been rigorously researched lately is the *histogram matching* (also called *jumbled matching*, *Parikh matching*, or *permuted matching*). In that problem a text T is to be preprocessed. In subsequent queries, a pattern P is given. All locations of T where a permutation of P occurs are sought. Amir et al. [2] showed that this problem is unlikely to have an efficient indexing algorithm. By *efficient indexing*, we mean that preprocessing the text is done in $O(n^{2-\epsilon})$ time for some $\epsilon > 0$, which then allowing for subsequent queries in time $\tilde{O}(m + occ)$, where m is the length of P and occ is the number of locations in T that match P. Recently, the *block mass* [24, 1] problem was considered. It turns out to have an efficient indexing algorithm for constant-size alphabets, but unlikely to have one for unbounded alphabets.

The history of indexing suggests a major difficulty: every matching relation required a *new, ad-hoc* consideration for its indexing version. The desired situation is one where: (1) There are necessary and sufficient conditions on the matching relation for having an efficient indexing algorithm, and (2) when the conditions are sufficient, to have a generic indexing algorithm, so that there will be no need to propose a data structure for every new relation. Finding necessary conditions for efficient indexing is, perhaps, one of the biggest challenges in the area, when the quest is for a generic indexing algorithm.

Cole and Hariharan [9] presented the first such algorithm. They showed that if the suffixes satisfy some conditions, then an index can be built efficiently. In Sect. 3, we present these conditions in more details.

Both parameterized matching [9] and order preserving matching [10] satisfy these conditions, thus Cole and Hariharan made an important step on the way to generalizing indexing algorithms. They showed a randomized algorithm that can construct the suffix tree in linear time, for alphabets of polynomial size. In addition to being *randomized*, their algorithm is based on McCreight's suffix tree construction algorithm [23] and thus, is *not online*. The online situation was solved by Lee et al. [21]. In this paper, we go a step further. Our contributions are:

1. There is a more natural way of characterizing whether indexing is possible, is by considering the **matching relation**. Thus we present a set of sufficient conditions on the matching relation that, if satisfied, allow efficient indexing. Consequently, when presented with a relation, all that is necessary is to check whether that relation satisfies these conditions. We suggest sufficient conditions that are more natural and less strict than the Cole and Hariharan conditions. We then show a reduction from Cole and Hariharan conditions to our conditions. Moreover, we show a matching relation that satisfies our conditions but not the Cole and Hariharan conditions.

2. We present a very simple generic **online, deterministic** indexing algorithm whose worst case times are $\tilde{O}(n)$ time for preprocessing, and $\tilde{O}(m + occ)$ time for query, where occ is the number of pattern occurrences in the text, and $\tilde{O}$ stands for the complexity up to polynomial logarithmic factor, i.e. $\log^c n$ for some constant c.
3. Our generic scheme allows for an easy algorithm for indexing a new matching which we call *jump forward* for string matching. This will be seen in detail in Sect. 4.3. This matching does not satisfy the Cole and Hariharan conditions.

The novelty of our algorithm is using the matching relation properties rather than using the specific matching to enable us to construct the indexing. These properties allow us to *sort suffixes efficiently.* Our algorithm is slower by a polylog factor than the specialized indexing algorithms for the known *non exact* matching relations, e.g. order preserving matching or parameterized matching. However, we were not driven by the goal of improving the indexing complexity of a given matching relation. Our interest is in presenting a small number of basic sufficient conditions that allows the construction of a **generic** algorithm that can be used as an **off-the-shelf** efficient indexing algorithm (up to polylog factors) for **any** matching relation that satisfies the conditions. An example of the effectiveness of our algorithm was recently demonstrated. Some colleagues were discussing indexing a new matching relation – *Cartesian Tree Matching* [25]. A quick check of the matching relation properties convinced us that an efficient indexing exists. What remained was only shaving some powers off the polylogarithmic factor.

2 Preliminaries

Let Σ be an alphabet. A *string* T over Σ is a finite sequence of letters from Σ. By $T[i]$, for $1 \le i \le |T|$, we denote the i^{th} letter of T. The *empty string* is denoted by ϵ. By $T[i..j]$ we denote the string $T[i] \dots T[j]$ called a *substring* of T (if $i > j$, then the substring is an empty string). A substring is called a *prefix* if $i = 1$ and a *suffix* if $j = |T|$. The prefix of length j is denoted by $T[..j]$. While by $S_i = T[i..]$ we denote the suffix which starts from index i in T. By $lcp(S, S')$ we denote the *longest common prefix (lcp)* of S and S'. In case S and S' are suffixes S_i and S_j, respectively, we denote $lcp(i, j) = lcp(S_i, S_j)$ for short. By T^R we denote the reversal (the mirror image) of T.

A *suffix tree* of a string T, denoted $ST(T)$, is a compacted trie containing all the suffixes of $T\$$, where \$ is a unique character not occurring in Σ. In a compacted trie we define the *depth* of a node to be its number of explicit ancestors, and the *string depth* to be the length of the string it represents. The string of a node x, denoted by $str(x)$, represents the result of reading the letters in path between trie's root and x. In a suffix tree we define the *suffix link* of a node representing the string aS to be a pointer to the node representing S, where S is a string and a is a character. Every explicit node v stores such a link $sl(v)$.

A matching $M \subseteq \Sigma^* \times \Sigma^*$ is a binary relation between strings, where Σ^l is the set of all l-length strings and $\Sigma^* = \bigcup_{l=0}^{\infty} \Sigma^l$. Two strings A and B are said to be matched if and only if $(A, B) \in M$. We consider matchings in which two matched strings are of the same length.

▶ **Example 1** (Exact Matching)**.** Two strings are matched, if they are exactly the same string. Formally, $M_E = \{(A, A) \mid \forall A \in \Sigma^*\}$.

An order relation $R \subseteq \Sigma^* \times \Sigma^*$ is a binary relation between strings. String A is said to be less or equal to string B if and only if $(A, B) \in R$, ARB in short.

▶ **Definition 2** (Total Order for M)**.** *Let R be an order relation, and M a matching. $R \subseteq \Sigma^* \times \Sigma^*$ is* total order for M *if the following properties hold.*

- Transitivity: $\forall A, B, C \quad ARB \wedge BRC \Rightarrow ARC$
- Antisymmetry: $\forall A, B \quad ARB \wedge BRA \iff (A, B) \in M$
- Completeness: $\forall A, B \quad ARB \vee BRA$

An order relation R is said to be *consistent with prefixes* (*consistent with suffixes*) if for all A, B such that ARB then $\forall\ l = 1, \ldots, \min\{|A|, |B|\} \quad A[..l]\ R\ B[..l]\ (A[l..]\ R\ B[l..])$.

▶ **Definition 3** (Lexicographic Order). *Given two different strings $A = a_1, a_2, \ldots, a_n$ and $B = b_1, b_2, \ldots, b_m$, the first one is smaller than the second if $a_i < b_i$ for the first i where a_i and b_i differ, otherwise if no such i exists (happens when one of the strings is a prefix of the other) the shorter string is the smaller. The lexicographic order is a total order for M_E and consistent with prefixes.*

▶ **Definition 4** (Quasi-Suffix Collections (QSC) [9]). *An ordered collection of strings $s_1, s_2, ..., s_n$ is called a quasi-suffix collection if and only if the following conditions hold.*

1. $|s_1| = n, |s_2| = n - 1, \cdots, |s_n| = 1$.
2. *No s_i is prefix of another s_j.*
3. *If $lcp(s_i, s_j) = l > 0$ then $lcp(s_{i+1}, s_{j+1}) \geq l - 1$.*

▶ **Definition 5** (Character Oracles [9]). *A data structure which supplies the j^{th} character of the i^{th} string of the quasi-suffix collection, in $O(1)$ time, is called character oracle. Note that the total length of the explicit strings for a quasi-suffix collection of n strings is $O(n^2)$. While character oracle allows holding the collection in smaller space and reading it on demand. Cole and Hariharan reduce the amount of character oracle calls to be linear during the suffix tree construction.*

Our main contributions are the following.

▶ **Theorem 6** (Order Relation based Sufficient Conditions). *Let M be a matching. There is an efficient indexing construction for the matching if there exists an order relation R and the following conditions hold.*

- *R is a* total order for *M and* consistent with prefixes.
- Order derivation: *given $lcp(i, j)$, compute the order between suffix i and j in $\tilde{O}(1)$ time.*
- Lcp derivation: *assuming oracle access to $lcp(i', j')$ for $i', j' > 1$ compute $lcp(1, i)$ where i', j' and i are indexes in the text in $\tilde{O}(1)$ time.*
- Efficient extension by one character: *maintaining lcp derivation and order derivation, to support text extension by one new letter at the beginning. This operation takes amortized $\tilde{O}(1)$ time per extension.*
- Efficient extension by a pattern: *extending the lcp derivation and order derivation, to support lcp derivation of $lcp(P, S_i)$ and order between the pattern and a suffix, takes overall $\tilde{O}(|P|)$ time and space.*

$lcp(i, j)$ is well defined when R is *consistent with prefixes*, because then M is hereditary from right, i.e. dropping same length suffixes of matched strings leads to a matched pair of strings. That is, the prefixes of S_i and S_j from length 1 to length $lcp(i, j)$ are equal. And $lcp(i, j)$ is the length of the longest common prefix.

▶ **Theorem 7.** (The Reduction) *Let M be a matching. Assume a character oracle f such that for every string S, $\{s_i\}_{i=1}^n = \{f(S_i)\}_{i=1}^n$ is a quasi-suffix collection. Then the conditions of Theorem 6 hold for M.*

3 Previous Results

Cole and Hariharan paper [9] guarantees the following. When one wants to index a string T under some matching M, a quasi-suffix collection is required. A compacted trie is then constructed for the supplied collection in $O(n)$ time and space with a failure probability close to inverse exponential. Later, this construction was generalized to be online using the same time and space complexities by Lee et al. [21], by adding letters to the end.

Character oracles are a general abstraction for accessing the quasi-suffix collection's elements. Previously studied character oracles are for parameterized matching [5, 6, 9, 21], 2D matching [13, 9] and order preserving matching [19, 7, 10]. In practice, they are built directly from some transformation f. Each suffix of T might be transformed to a string which is not related to the other suffixes' transformations, these are the $\{s_i\}_{i=1}^{n}$. Based on these transformations the quasi-suffix collection is defined as $s_i = f(S_i)$. It is necessary that the prefixes of any string S be consistent after the transformation, i.e. $f(S[..i]) = f(S)[..i]$, to ensure that the search is well defined, locating the pattern as a prefix of some s_i.

▶ **Example 8** (Parameterized Matching)**.** Let Σ and Π be the alphabets of fixed symbols and parameters symbols respectively. Two strings are said to *parameterized match*, if one string can be transformed into the other by renaming the characters via a *one to one* function on their parameter alphabet.

A predecessor string of a string T has at each location i the distance between i and the location containing the previous appearance of the symbol $T[i]$. The first appearance of each symbol is replaced with a 0. The transformation is defined by the predecessor string and denoted by $f(T)$. The predecessor string is consistent with prefixes because dropping letters from the end of the string does not affect the distances to previous occurrences of the remaining letters.

Let $S = ababb$, where $\Pi = \{a, b\}$, then the quasi-suffix collection is $\{00221\$, 0021\$, 001\$, 01\$, 0\$, \$\}$. A special \$ is appended to each transformed suffix to fit Condition 2 of QSC.

The character oracle is based on an observation about the predecessor string. Each time a character is removed from the beginning of S_i it might affect at most one character in the transformation of S_{i+1}. This happens because at most one position distance is replaced by 0. Therefore, by maintaining for each index the distance to the previous occurrence of the same character it is possible to answer $f(S_i)[j]$ in constant time. If $j - pv[j] \geq i$ then $f(S_i)[j] = pv[j]$, otherwise $f(S_i)[j] = 0$, where pv is the distance of each character to its previous occurrence in T.

The power behind the previous results is the optimization of character oracle calls. The amount of oracle calls turns out to be linear even though the quasi-suffix collection size might be quadratic. The revolutionary idea is to create additional nodes during the construction. These nodes are used for every new suffix insertion to locate where to insert the new suffix in efficient time. Their implementation uses amortized constant time per suffix insertion. Condition 3 of QSC guarantees the correctness of suffix links. Let x be a node which is an ancestor of both s_i and s_j and $|str(x)| = lcp(s_i, s_j)$. Removing the first character results in equal strings. By Condition 3 of QSC, $lcp(s_{i+1}, s_{j+1}) \geq |str(x)| - 1 = |str(y)|$ where $y = sl(x)$. However, without this condition it is impossible to define correctly the suffix link because we might have two suffixes whose beginnings are equal, but dropping their first letters would not preserve the equality and we would not be able to search down from the node pointed to by the suffix link.

Two major problems that had been solved by the Cole and Hariharan algorithm are *the branching problem* and *the missing suffix links problem*. In the first, we might have a logarithmic factor due to linear number of children for a certain internal node in the suffix tree. Thus, a perfect hash was introduced in Cole and Hariharan to overcome this logarithmic factor. Next, some internal node suffix links might point to a middle of an edge. This was solved by introducing new nodes called *imaginary nodes* which are added to allow explicit pointing by the suffix links.

4 Our Conditions

We wish to consider the problem of supplying an indexing construction for different matchings from a more general perspective. We would like to know whether, just by looking at the matching relation, it is possible to decide whether it can be indexed efficiently. We decided to go about this by first observing existent indexing constructions and then trying to identify a set of matching properties that are sufficient for constructing an indexing data structure. Afterwords, we introduce two additional conditions which are necessary for construction efficiency. Our conditions are proven to be more general than the ones previously studied. We discuss the connection between sufficient conditions used in previous papers for efficient indexing construction compared to our proposed conditions.

4.1 The Intuition Behind The Choice of Sufficient Conditions

Known indexing data structures, including the Suffix Tree (ST), the Suffix Array (SA) and the Burrows Wheeler Transform (BWT), sort the suffixes lexicographically in their construction procedures, resulting in an index data structure that answers queries using efficient search algorithms. Examples of these algorithms are: a binary search in the SA, search by first letter in the edges of a suffix tree, and search by reducing ranges in BWT using the *FM index* [12]. We observe that all efficient indexes needed both a fast sorting technique and an efficient search algorithm.

For the sake of being able to sort the suffixes, we need to have some order defined on them. This order is required to be a *total order*. *Transitivity* allows sorting the suffixes in a well-defined order. *Antisymmetry* is the property that if a pair of elements are related to each other then they are matched. Hence, *antisymmetry* connects the *order relation* to its matching. Finally, *completeness* allows comparing the pattern with the prefixes of any suffix.

The ability to search efficiently for a pattern P relies on comparison of the pattern with $|P|$-length prefixes of the suffixes. This is possible if the sorting is *consistent with prefixes*, otherwise a pre-sorting of the suffixes will not aid in finding prefixes of a certain length $|P|$. Formally, let a and b be two strings such that $\min\{|a|, |b|\} \geq |P|$. If w.l.o.g. aRb then $a[1..|P|]\ R\ b[1..|P|]$.

It is not enough to have a total order consistent with prefixes for a fast indexing construction. The reason is that the algorithm needs a way to quickly check if xRy is true in order to sort the suffixes. In addition, it is necessary to know during the searching if a pattern P is located at the beginning of some suffix. We use two properties to fill this gap, *order derivation* and *lcp derivation*. *Order derivation* inputs two suffixes i and j and outputs the order between them. *Lcp derivation* receives as input two suffixes i and j and outputs their longest common prefix length. We also want these derivations to be implementable online, supporting extensions by letters added to the beginning of the text. Using the lcp derivation we extend the text from the left by P. Then we can use lcp derivation queries to check if P occurs at any suffix. When finishing the search we undo all the changes that we made to the construction. Note that the extension by P should be achieved in $\tilde{O}(|P|)$ time and space to ensure pattern search efficiency.

However, implementing lcp and order derivation in sublinear time is not a trivial task even for exact matching.

▶ **Example 9.** Let i, j be two suffixes. The order derivation of those suffixes can be determined from comparing $T[i + lcp(i, j) + 1]$ with $T[j + lcp(i, j) + 1]$, while answering $lcp(i, j)$ might require a complex data structure such as Range-Minimum queries [15] on the LCP Array or LCA construction [20] on a suffix tree.

Indeed, computing the *lcp* without any previous knowledge is a hard task. But a modification of the properties can simplify the calculation to be neat and efficient.

▶ **Example 10.** *Lcp derivation* for exact matching can be calculated in constant time based on previous knowledge of $lcp(i + 1, j + 1)$.

$$lcp(i,j) = \begin{cases} lcp(i+1, j+1) + 1, & \text{if } T[i] = T[j] \\ 0, & \text{otherwise.} \end{cases}$$

Moreover, *order derivation* can be answered in constant time when $lcp(i, j)$ is supplied.

The above intuition results in Theorem 6 - our sufficient conditions for efficient indexing. We prove Theorem 6 in Sect. 5, by providing an indexing construction.

▶ **Theorem 11.** *Let M be a matching relation and R a total order that is* consistent with prefixes*. Based on R's properties, the following properties hold for M.*

- Reflexivity*:* $\forall a \quad (a, a) \in M$
- Symmetry*:* $\forall a, b \quad (a, b) \in M \iff (b, a) \in M$
- Transitivity*:* $\forall a, b, c \quad (a, b) \in M \land (b, c) \in M \Rightarrow (a, c) \in M$
- Heredity from right (prefix consistency)*:* $\forall x, y$ *n-length strings* $(x, y) \in M \Rightarrow (x[1..n - 1], y[1..n - 1]) \in M$

Proof. *Reflexivity* is hold, from *completeness*, aRa and from *antisymmetry*, $(a, a) \in M$.

Symmetry is hold, from *antisymmetry*, $aRb \land bRa$, thus from *antisymmetry*, $(b, a) \in M$.

Transitivity is hold, from *antisymmetry*, $aRb \land bRa \land bRc \land cRb$, thus from *transitivity*, $aRc \land cRa$, and from *antisymmetry*, $(a, c) \in M$.

Hereditary from right is hold, from *antisymmetry*, $xRy \land yRx$, from *consistent with prefixes*, $x[..n-1]Ry[..n-1] \land y[..n-1]Rx[..n-1]$, and finally, from *antisymmetry*, $(x[..n-1], y[..n-1]) \in M$. ◀

When presented with a new matching relation, we recommend checking these conditions first because they do not need to use the total order but are depended on the matching itself. If the matching M is reflexive, symmetric, transitive and hereditary from the right, devise a total order for the matching. Finally, provide *order and lcp derivation* properties.

4.2 The Connection To Quasi-Suffix Collections

▶ **Lemma 12.** *Let M be a matching which is an equivalence relation, i.e. reflective, symmetric and transitive. Then the following conditions are equivalent.*

1. *There* exists *a transformation f, consistent with prefixes, such that* $\forall S, T \; f(S) = f(T) \iff (S, T) \in M$*.*
2. *The matching is* hereditary from right*.*

Proof. $1 \Rightarrow 2$**.** It is trivial because the transformation f is consistent with prefixes. Thus, the matching is also consistent with prefixes.

$2 \Rightarrow 1$**.** We will show a construction of f. It is still an open problem if it is possible to calculate a character in f in $\tilde{O}(1)$ time. As a consequence of the reflexive, symmetric, and transitive properties of M, an equivalence relation provides a partition of Σ^* into disjoint equivalence classes. The empty string is transformed into an empty string. We observe the equivalence classes of single characters. Each of those class is transferred to an identical number. This identical number is chosen from $[1, n]$, because there are at most n identical characters. Then we observe substrings of length equals to 2. Each disjoint equivalence classes of those strings is marked with an identical number from $[1, n]$. And the transformation is defined to be the transformation of the string without the last character to which the identical number that corresponds to the full string equivalence class is appended at the end. It is easy to see that this transformation is consistent with prefixes. To support the QSC definition it is required to append \$ at the end to each transformed suffix. ◀

▶ **Lemma 13.** *Let M be a matching which is an equivalence relation. Then the following two conditions are equivalent.*

1. *For every string S, $\forall i, j \;\; lcp(i, j) = l > 0 \;\Rightarrow\; lcp(i + 1, j + 1) \geq l - 1$.*
2. *The matching is* hereditary from left*, i.e. dropping some amount of characters from the beginning of two matched strings results in a matched pair.*

4.3 The Jump Forward Matching

In this section we show a matching that does not satisfy the Cole and Hariharan conditions, yet satisfies our properties.

From Lemma 12, *heredity from right* implies that a character oracle exists, although it does not necessarily follow that such an oracle answers the query in sublinear time. On the other hand, Condition 3 of QSC stipulates that

$$\forall i, j \;\; lcp(i, j) = l > 0 \;\Rightarrow\; lcp(i + 1, j + 1) \geq l - 1 \qquad (1)$$

By Lemma 13, this condition is equivalent to *heredity from left*, i.e. dropping the characters from the beginnings of two matched strings results in matched pair of strings.

Thus, we wish to create a matching where *heredity from the left* is not satisfied but our conditions hold. This leads to the following matching definition.

▶ **Definition 14** (Jump Forward Matching)**.** *Let $S \in N^n$, and let $i \in \{1, ..., n\}$. A* forward jump *of index i is index $j = i + S[i]$, if $i + S[i] \leq n$, and is* out *if $i + S[i] > n$.*

The jumps sequence till out *of index i is the sequence $i_1, ..., i_k$, where $i_1 = i$, i_j is the forward jump of i_{j-1}, for $j = 2, ..., k$, and the forward jump of i_k is out. We denote the main jumps sequence to be the jumps sequence till out of index 1 by $f(S)$. Two strings are said to be matched if their main jumps sequences are equal.*

The search of a pattern turns to be a search of the main jumps sequence of the pattern.

A normalization of string S is the replacement of bigger jumps than n by jumps of size n. The normalized string is defined on $\Sigma = \{1, \ldots, n\}$ and is equivalent to the original string.

▶ **Example 15.** Let $S = 2\ 2\ 2\ 2$ and $T = 2\ 3\ 2\ 2$. These two strings are matched because their main jumps sequences are $f(S) = 1\ 3$ and similarly, $f(T) = 1\ 3$.

In this example, $S[2..]$ and $T[2..]$ are not matched because $f(S[2..]) = 1\ 3$ while $f(T[2..]) = 1$. Thus, *heredity from left* does not hold and the matching does not fit the Cole and Hariharan conditions.

▶ **Lemma 16.** *Theorem 6 conditions hold for the jump forward matching.*

Proof. Let $R = \{(X,Y)|f(X) \le f(Y)\}$ be the order relation, where the comparison between $f(X)$ and $f(Y)$ is using the lexicographic order. This order relation is a total order, it is transitive, reflexive and complete. Moreover, it is consistent with prefixes because the transformation is consistent with prefixes, i.e. $\forall S \;\; f(S[..i])$ is a prefix of $f(S)$, where i is an index in the string S.

Note that the *lcp* is defined to be the longest prefix for which the main sequences are equal. A straightforward approach can be used to implement the lcp derivation property.

$$lcp(i,j) = \begin{cases} lcp(i + S[i], j + S[j]) + 1, & \text{if } S[i] = S[j] \\ 1, & \text{otherwise.} \end{cases}$$

The order derivation property is satisfied by returning $f(S[i..])[lcp+1]$ compared to $f(S[j..])[lcp+1]$, where $lcp = lcp(i,j)$. We construct a forest based on the forward jumps, each character is represented by a node in the forest. Its value, the jump, is represented by an edge to its parent node in a tree. In such a way, the maximal jumps sequences are beginning from the leafs, where maximal stands for a jumps sequence that cannot be extended to a longer jumps sequence. Each leaf stores the jumps sequence that begins from its index in the text. For a node, which corresponds to index i, we store a pointer to one the leafs in its subtree. Then $f(S_i)[j]$ is calculated by the leaf's jumps sequence. First, we search to find the position *pos* in which i is located. Afterwards, we return the content at position $pos + j$. The search is implemented by *van-Emde Boas* data structure [27] in $O(\log\log n)$ time and $O(n)$ preprocessing. The indices are stored in the data structure related to the end position of the text, i.e. $n - i$ for an index i. That is, extending the text at the beginning does not affect on existent indices.

Extension of the *order and lcp derivations* by the pattern are done by constructing of the above data structure on the pattern in $O(m)$ time. ◀

4.4 The Reduction

To complete the claim that our conditions are more general than Cole and Hariharan's, we present a reduction from their conditions to our conditions.

▶ **Theorem 7.** (The Reduction) *Let M be a matching. Assume a character oracle f such that for every string S, $\{s_i\}_{i=1}^n = \{f(S_i)\}_{i=1}^n$ is a quasi-suffix collection. Then the conditions of Theorem 6 hold for M.*

Proof.

Order definition. The order relation is the lexicographic order and is a *total order*, it is transitive, reflexive and complete. The quasi-suffix collection is a collection of strings between which the comparison is done using the lexicographic order.

Order consistence with prefixes. Immediate from the order definition.

Lcp derivation. Let S be some string, we create an indexing construction based on Cole and Hariharan [9]. This construction results in a suffix tree of the quasi-suffix collection $\{s_i\}_{i=1}^n$. We then process the suffix tree to support LCA queries [8]. These queries allow the calculation of $lcp(s_i, s_j)$. Thus the lcp query is supported in constant time.

Order derivation. Order derivation is supported based on the character oracle. We compare $s_i[l+1]$ with $s_j[l+1]$ to determine the order, where $l = lcp(i,j)$.

Efficient extension by P. We add P to the suffix tree construction. $lcp(P, S_i)$ is calculated using an LCA query between the pattern and the leaf represented by the suffix S_i. The order is computed by a comparison between $P[l+1]$ and $s_i[l+1]$, where $l = lcp(P, S_i)$. Both *lcp and order derivations* are supplied in constant time. ◀

5 The Construction

In this section, we prove Theorem 6 by providing an efficient indexing construction for a matching M which satisfies the theorem's conditions. Without consistency of the suffixes, suffix links are not well defined during the suffix tree construction and a different structure is required. We use a balanced binary search tree (BST) to store the suffixes in sorted order defined by R. At each phase, a new suffix is inserted from the shortest to the longest. We can also handle text symbols that arrive by prepending to the beginning of the text. The calculation of *lcp derivation* relies on previous knowledge about other lcp values between suffixes which have already been inserted to the BST. We maintain at each node both the lcp of its suffix with the next suffix in the sort, and its lcp with the previous suffix in the sort. The following lemma helps calculate the lcp between every two suffixes in logarithmic time and constant space.

▶ **Lemma 17.** *Let $S_{i_1}, \ldots, S_{i_k}$ be a sorted group of suffixes by some total order R which is consistent with prefixes. Assume w.l.o.g. $i < j$, and $i = i_p$, $j = i_q$, then*

$$lcp(i,j) = \min_{p \le r < q} \{lcp(S_{i_r}, S_{i_{r+1}})\} \tag{2}$$

An identical lemma was proven for exact matching by Manber and Myers [22]. The order there is assumed to be the lexicographic order. Their proof is correct for our case without any modification because it is based on transitivity and consistency with prefixes.

Answering $lcp(i,j)$ for every two suffixes that exist in the BST is possible based on the above lemma.

At each node, we maintain the minimal value of the lcp values in its rooted subtree, by taking into account both lcp with previous and next suffixes. When observing the path between suffixes i and j in the BST, all the suffixes $\{S_{i_r}\}_{r=p}^{q}$ could be iterated to calculate $\min_{p \le r < q}\{lcp(i_r, i_{r+1})\}$. The iteration over the whole subtree can be replaced by constant time accessing of the minimal lcp value stored at each node. This improvement implies a logarithmic time calculation because the BST height is $O(\log n)$.

We are now ready to present how to search for a pattern P. First, we extend the lcp and order derivation, using the efficient extension property, to support *lcp derivation* of $lcp(P, S_i)$ and order between P and S_i. Then, similarly to binary search, we wish to find the range bounds S_{i_p} and S_{i_q}, where $S_{i_p} < S_{i_q}$, inside the sorted group of suffixes $\{S_{i_k}\}_{k=1}^{n}$ in which P is located at the beginning of all elements, and P matches neither a prefix of $S_{i_{p-1}}$ nor a prefix of $S_{i_{q+1}}$.

W.l.o.g. assume that we search for the left corner suffix $S_{i_{p-1}}$. Let S_{i_j} be the root of the BST. If $lcp(P, S_{i_j}) < |P|$ then the next node in the search is the root child followed to the direction of the order between P and S_{i_j}. Otherwise, if $lcp(P, S_{i_j}) = |P|$, P is a prefix of S_{i_j}, and we should check if S_{i_j} is the left corner of the range. Similarly, we check also for $lcp(P, S_{i_{j-1}})$. If this value is smaller than $|P|$ then we finish, otherwise we continue to search to the left. We store a bi-linked list between all suffixes in the sort. It allows to access predecessor and successor suffixes in the sort in constant time.

Note that $lcp(P, S_{i_j})$ is calculated in the following way. Its value is derived from a constant number of lcp values of the form $lcp(P[i'..], S_{i_{j+j'}})$ for some offsets i', j'. If $lcp(P, S_{i_j})$ value is derived from a single lcp value, its calculation is done in $O(m)$ time and constant space by a recursion, where $m = |P|$. Otherwise, because the dependency on i', j' might cause a quadratic amount of lcp calculations, another search algorithm is purposed. We insert each suffix of P to the sorted group of suffixes. In such a way $lcp(P, S_i)$ is calculated in logarithmic time based on Lemma 17.

Time.

Search for matchings with simple lcp derivation. The search time complexity is $O(\log n \cdot (T_{od} + m \cdot T_{ld})) + \tilde{O}(m)$.

Search for matchings with complex lcp derivation. The search time complexity is $O(m \log n \cdot (T_{od} + T_{ld} + \log n) + \log n \cdot (T_{od} + T_{ld} + \log n)) + \tilde{O}(m)$.

Construction time. $O(n \log n \cdot (T_{od} + T_{ld} + \log n))$ using $O(n)$ space.

In all above time formulae, T_{od} and T_{ld} are the time to perform a single order derivation and an lcp derivation query, respectively. Simple lcp derivation stands for lcp derivation calculation which requires at most a single lcp query, otherwise the lcp derivation property is called complex.

6 Conclusions and Open Problems

Assuming a matching relation on strings and an order relation of the form smaller or equal ($\leq$) which holds $A \leq B$ and $B \leq A \iff A = B$. We identified a small set of sufficient conditions on the order relation that allows indexing: (1) is total order (2) consistent with prefixes, i.e. the order between two strings is preserved for any pair of equal-length prefixes of them. We presented an efficient indexing algorithm for any matching that fulfills these conditions. Our algorithm relies on four properties: *lcp derivation*, the ability to calculate $lcp(i, j)$ based on other lcp values; *order derivation*, the ability to retrieve the order between suffixes i and j based on $lcp(i, j)$; *efficient extension by one letter*, the ability to extend both *lcp and order derivation* after text extension by one letter at the beginning, in amortized $\tilde{O}(1)$ time; and *efficient extension by the pattern*, the ability to extend both *lcp and order derivation*, to support $lcp(P, S_i)$ and order between the pattern and a suffix, in overall $\tilde{O}(m)$ time and space. An important open problem is to also provide the *necessary* conditions for indexing.

We show that our conditions for indexing are more general than previous known conditions. We prove that the consistency with prefixes is equivalent to a character oracle existence.

We also define a new matching where the previous conditions, in particular *consistency with suffixes*, do not hold but where our conditions hold. When *consistency with suffixes* means that dropping equal length prefixes from two matched strings result in a matched pair of strings. We show an efficient indexing construction which does not rely on suffix links, which cannot be used when there is no heredity from left. An interesting future direction is to define suffix trees that support these kinds of matchings.

References

1 A. Amir, A. Butman, and E. Porat. On the Relationship between Histogram Indexing and Block-Mass Indexing. *Philosophical Transactions of the Royal Society A: Mathematical Physical and Engineering Sciences*, 372(2016), 2014. URL: `http://doi.org/10.1098/rsta.2013.0132`.

2 A. Amir, T.M. Chan, M. Lewenstein, and N. Lewenstein. On the Hardness of Jumbled Indexing. In *Proc. 41st International Colloquium on Automata, Languages and Programming (ICALP)*, pages 114–125, 2014.

3 A. Amir, M. Farach, and S. Muthukrishnan. Alphabet Dependence in Parameterized Matching. *Information Processing Letters*, 49:111–115, 1994.

4 A. Amir and I. Nor. Real-Time Indexing over Fixed Finite Alphabets. In *Proc. 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1086–1095, 2008.

5 B. S. Baker. Parameterized Pattern Matching: Algorithms and Applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.

6 B. S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM J. Comput.*, 26(5):1343–1362, 1997.

7 S. Cho, J. C. Na, K. Park, and J. S. Sim. A fast algorithm for order-preserving pattern matching. *Information Processing Letters*, 115(2):397–402, 2015.

8 R. Cole and R. Hariharan. Dynamic LCA Queries in Trees. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 235–244, 1999.

9 R. Cole and R. Hariharan. Faster Suffix Tree Construction with Missing Suffix Links. *SIAM J. Comput.*, 33(1):26–42, 2003.

10 M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen. Order-preserving Indexing. *Theoretical Computer Science*, 613:122–135, 2016.

11 M. Farach. Optimal Suffix Tree Construction with Large Alphabets. *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.

12 P. Ferragina and G. Manzini. Indexing Compressed Texts. *J. of the ACM*, 52(4):552–581, 2005.

13 R. Giancarlo. A Generalization of the Suffix Tree to Square Matrices, with Applications. *SIAM J. Comput.*, 24(3):520–562, 1995.

14 J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 03)*, pages 943–955, 2003. LNCS 2719.

15 T. Kasai, G. Lee, S. Arikawa, and K. Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In Amihood Amir, editor, *Combinatorial Pattern Matching*, pages 181–192. Springer, 2001.

16 J. Kim, A. Amir, J. C. Na, K. Park, and J. S. Sim. On Representations of Ternary Order Relations in Numeric Strings. *Mathematics in Computer Science*, 11(2):127–136, 2017.

17 J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama. Order Preserving Matching. *Theoretical Computer Science*, 525:68–79, 2014.

18 T. Kopelowitz. On-line Indexing for General Alphabets. In *Proc. 53rd IEEE Symposium on the Foundation of Computer Science (FOCS)*, 2012.

19 M. Kubica, T. Kulczynski, J. Radoszewski, W. Rytter, and T. Walen. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.

20 G.M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

21 T. Lee, J.C. Na, and K. Park. On-Line Construction of Parameterized Suffix Trees. *Information Processing Letters*, 111(5):201–207, 2011.

22 U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 319–327, 1990.

23 E. M. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23:262–272, 1976.

24 J. Ng, A. Amir, and P. A. Pevzner. Blocked Pattern Matching Problem and its Applications in Proteomics. In *Proc. 15th Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 298–319, 2011.

25 S. G. Park, A. Amir, G. M. Landau, and K. Park. Cartesian Tree Matching and Indexing. In *Proc. 30th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2019. To appear.

26 E. Ukkonen. On-line Construction of Suffix Trees. *Algorithmica*, 14:249–260, 1995.

27 P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Mathematical Systems Theory*, 10:99–127, 1977.

28 P. Weiner. Linear Pattern Matching Algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.