

Gapped Indexing for Consecutive Occurrences

Philip Bille  



Technical University of Denmark, DTU Compute, Lyngby, Denmark

Inge Li Gørtz  

Technical University of Denmark, DTU Compute, Lyngby, Denmark

Max Rishøj Pedersen  

Technical University of Denmark, DTU Compute, Lyngby, Denmark

Teresa Anna Steiner  

Technical University of Denmark, DTU Compute, Lyngby, Denmark

Abstract

The classic string indexing problem is to preprocess a string S into a compact data structure that supports efficient pattern matching queries. Typical queries include *existential queries* (decide if the pattern occurs in S), *reporting queries* (return all positions where the pattern occurs), and *counting queries* (return the number of occurrences of the pattern). In this paper we consider a variant of string indexing, where the goal is to compactly represent the string such that given two patterns P_1 and P_2 and a gap range $[\alpha, \beta]$ we can quickly find the consecutive occurrences of P_1 and P_2 with distance in $[\alpha, \beta]$, i.e., pairs of subsequent occurrences with distance within the range. We present data structures that use $\tilde{O}(n)$ space and query time $\tilde{O}(|P_1| + |P_2| + n^{2/3})$ for existence and counting and $\tilde{O}(|P_1| + |P_2| + n^{2/3} \text{occ}^{1/3})$ for reporting. We complement this with a conditional lower bound based on the set intersection problem showing that any solution using $\tilde{O}(n)$ space must use $\tilde{\Omega}(|P_1| + |P_2| + \sqrt{n})$ query time. To obtain our results we develop new techniques and ideas of independent interest including a new suffix tree decomposition and hardness of a variant of the set intersection problem.

2012 ACM Subject Classification Theory of computation \rightarrow Data structures design and analysis; Theory of computation \rightarrow Pattern matching

Keywords and phrases String indexing, two patterns, consecutive occurrences, conditional lower bound

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.10

Related Version *Full Version:* <https://arxiv.org/abs/2102.02505>

Funding *Philip Bille:* Supported by the Danish Research Council grant DFF-8021-002498.

Inge Li Gørtz: Supported by the Danish Research Council grant DFF-8021-002498.

Max Rishøj Pedersen: Supported by the Danish Research Council grant DFF-8021-002498.

1 Introduction

The classic string indexing problem is to preprocess a string S into a compact data structure that supports efficient pattern matching queries. Typical queries include *existential queries* (decide if the pattern occurs in S), *reporting queries* (return all positions where the pattern occurs), and *counting queries* (return the number of occurrences of the pattern). An important variant of this problem is the *gapped string indexing problem* [6, 8, 10, 14, 27, 28, 31]. Here, the goal is to compactly represent the string such that given two patterns P_1 and P_2 and a *gap range* $[\alpha, \beta]$ we can quickly find occurrences of P_1 and P_2 with distance in $[\alpha, \beta]$. Searching and indexing with gaps is frequently used in computational biology applications [6, 11, 13, 14, 19, 21, 22, 32, 35, 38].



© Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, and Teresa Anna Steiner; licensed under Creative Commons License CC-BY 4.0

32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021).

Editors: Paweł Gawrychowski and Tatiana Starikovskaya; Article No. 10; pp. 10:1–10:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Another variant is *string indexing for consecutive occurrences* [9, 40]. Here, the goal is to compactly represent the string such that given a pattern P and a gap range $[\alpha, \beta]$ we can quickly find *consecutive occurrences* of P with distance in $[\alpha, \beta]$, i.e., pairs of subsequent occurrences with distance within the range.

In this paper, we consider the natural combination of these variants that we call *gapped indexing for consecutive occurrences*. Here, the goal is to compactly represent the string such that given two patterns P_1 and P_2 and a gap range $[\alpha, \beta]$ we can quickly find the consecutive occurrences of P_1 and P_2 with distance in $[\alpha, \beta]$.

We can apply standard techniques to obtain several simple solutions to the problem. To state the bounds, let n be the size of S . If we store the suffix tree for S , we can answer queries by searching for both query strings, merging the results, and removing all non-consecutive occurrences. This leads to a solution using $O(n)$ space and $\tilde{O}(|P_1| + |P_2| + \text{occ}_{P_1} + \text{occ}_{P_2})$ query time, where occ_{P_1} and occ_{P_2} denote the number of occurrences of P_1 and P_2 , respectively¹. However, $\text{occ}_{P_1} + \text{occ}_{P_2}$ may be as large as $\Omega(n)$ and much larger than the size of the output.

Alternatively, we can obtain a fast query time in terms of the output at the cost of increasing the space to $\Omega(n^2)$. To do so, store for each node v in the suffix tree the set of all consecutive occurrences (i, j) where i is the suffix number of a leaf below v in a standard 2D range searching data structure organized by the lexicographic order of j and the distance of the consecutive occurrence. To answer a query, we then perform a 2D range search in the structure corresponding to the locus of P_1 using the lexicographic range in the suffix tree defined by P_2 and the gap range. This leads to a solution for reporting queries using $\tilde{O}(n^2)$ space and $\tilde{O}(|P_1| + |P_2| + \text{occ})$ time, where occ is the size of the output. For existence and counting, we obtain the same bound without the occ term.

In this paper, we introduce new solutions that significantly improve the above time-space trade-offs. Specifically, we present data structures that use $\tilde{O}(n)$ space and query time $\tilde{O}(|P_1| + |P_2| + n^{2/3})$ for existence and counting and $\tilde{O}(|P_1| + |P_2| + n^{2/3}\text{occ}^{1/3})$ for reporting. We complement this with a conditional lower bound based on the set intersection problem showing that any solution using $\tilde{O}(n)$ space must use $\tilde{\Omega}(|P_1| + |P_2| + \sqrt{n})$ query time. To obtain our results we develop new techniques and ideas of independent interest including a new suffix tree decomposition and hardness of a variant of the set intersection problem.

1.1 Setup and Results

Throughout the paper, let S be a string of length n . Given two patterns P_1 and P_2 a *consecutive occurrence* in S is a pair of occurrences (i, j) , $0 \leq i < j < |S|$ where i is an occurrence of P_1 and j an occurrence of P_2 , such that no other occurrences of either P_1 or P_2 occurs in between. The *distance* of a consecutive occurrence (i, j) is $j - i$. Our goal is to preprocess S into a compact data structure that given pattern strings P_1 and P_2 and a gap range $[\alpha, \beta]$ supports the following queries:

- $\text{Exists}(P_1, P_2, \alpha, \beta)$: determine if there is a consecutive occurrence of P_1 and P_2 with distance within the range $[\alpha, \beta]$.
- $\text{Count}(P_1, P_2, \alpha, \beta)$: return the number of consecutive occurrences of P_1 and P_2 with distance within the range $[\alpha, \beta]$.
- $\text{Report}(P_1, P_2, \alpha, \beta)$: report all consecutive occurrences of P_1 and P_2 with distance within the range $[\alpha, \beta]$.

¹ \tilde{O} and $\tilde{\Omega}$ ignores polylogarithmic factors.

We present new data structures with the following bounds:

- **Theorem 1.** *Given a string of length n , we can*
- (i) *construct an $O(n)$ space data structure that supports $\text{Exists}(P_1, P_2, \alpha, \beta)$ and $\text{Count}(P_1, P_2, \alpha, \beta)$ queries in $O(|P_1| + |P_2| + n^{2/3} \log^\epsilon n)$ time for constant $\epsilon > 0$, or*
 - (ii) *construct an $O(n \log n)$ space data structure that supports $\text{Report}(P_1, P_2, \alpha, \beta)$ queries in $O(|P_1| + |P_2| + n^{2/3} \text{occ}^{1/3} \log n \log \log n)$ time, where occ is the size of the output.*

Hence, ignoring polylogarithmic factors, Theorem 1 achieves $\tilde{O}(n)$ space and query time $\tilde{O}(|P_1| + |P_2| + n^{2/3})$ for existence and counting and $\tilde{O}(|P_1| + |P_2| + n^{2/3} \text{occ}^{1/3})$ for reporting. Compared to the above mentioned simple suffix tree approach that finds all occurrences of the query strings and merges them, we match the $\tilde{O}(n)$ space bound, while reducing the dependency on n in the query time from worst-case $\Omega(|P_1| + |P_2| + n)$ to $\tilde{O}(|P_1| + |P_2| + n^{2/3})$ for Exists and Count queries and $\tilde{O}(|P_1| + |P_2| + n^{2/3} \text{occ}^{1/3})$ for Report queries.

We complement Theorem 1 with a conditional lower bound based on the set intersection problem. Specifically, we use the Strong SetDisjointness Conjecture from [20] to obtain the following result:

- **Theorem 2.** *Assuming the Strong SetDisjointness Conjecture, any data structure on a string S of length n that supports Exists queries in $O(n^\delta + |P_1| + |P_2|)$ time, for $\delta \in [0, 1/2]$, requires $\tilde{\Omega}(n^{2-2\delta-o(1)})$ space. This bound also holds if we limit the queries to only support ranges of the form $[0, \beta]$, and even if the bound β is known at preprocessing time.*

With $\delta = 1/2$, Theorem 2 implies that any near linear space solution must have query time $\tilde{\Omega}(|P_1| + |P_2| + \sqrt{n})$. Thus, Theorem 1 is optimal within a factor roughly $n^{1/6}$. On the other hand, with $\delta = 0$, Theorem 2 implies that any solution with optimal $\tilde{O}(|P_1| + |P_2|)$ query time must use $\tilde{\Omega}(n^{2-o(1)})$ space. Note that this matches the trade-off achieved by the above mentioned simple solution that combines suffix trees with two-dimensional range searching data structures.

Finally, note that Theorem 2 holds even when the gap range is of the form $[0, \beta]$. As a simple extension of our techniques, in the appendix we show how to improve our solution from Theorem 1 to match Theorem 2 in this special case.

1.2 Techniques

To obtain our results we develop new techniques and show new interesting properties of consecutive occurrences. We first consider Exists and Count queries. The key idea is to split gap ranges into large and small distances. For large distances there can only be a limited number of consecutive occurrences and we show how these can be efficiently handled using a segmentation of the string. For small distances, we cluster the suffix tree and store precomputed answers for selected pairs of nodes. Since the number of distinct distances is small we obtain an efficient bound on the space.

We extend our solution for Exists and Count queries to handle Report queries. To do so we develop a new decomposition of suffix trees, called the *induced suffix tree decomposition* that recursively divides the suffix tree in half by index in the string. Hence, the decomposition is a balanced binary tree, where every node stores the suffix tree of a substring of S . We show how to traverse this structure to efficiently recover the consecutive occurrences.

For our conditional lower bound we show a reduction based on the set intersection problem. Along the way we show that set intersection remains hard even if all elements in the instance have the same frequency.

1.3 Related Work

As mentioned, string indexing for gaps and consecutive occurrences are the most closely related lines of work to this paper. Another related area is *document indexing*, where the goal is to preprocess a collection of strings, called *documents*, to report those documents that contain patterns subject to various constraints. For a comprehensive overview of this area see the survey by Navarro [36].

A well studied line of work within document indexing is *document indexing for top- k queries* [12, 23, 24, 25, 26, 33, 34, 37, 39, 42, 43]. The goal is to efficiently report the top- k documents of smallest weight, where the weight is a function of the query. Specifically, the weight can be the distance of a pair of occurrences of the same or two different query patterns [25, 33, 37, 42]. The techniques for top- k indexing (see e.g. Hon et al. [25]) can be adapted to efficiently solve gapped indexing for consecutive occurrences in the special case when the gap range is of the form $[0, \beta]$. However, since these techniques heavily exploit that the goal is to find the top- k *closest occurrences*, they do not generalize to general gap ranges.

There are several results on conditional lower bounds for pattern matching and string indexing [4, 5, 20, 29, 30]. Notably, Ferragina et al. [16] and Cohen and Porat [15] reduce the *two dimensional substring indexing problem* to set intersection (though the goal was to prove an upper, not a lower bound). In the two dimensional substring indexing problem the goal is to preprocess pairs of strings such that given two patterns we can output the pairs that contain a pattern each. Larsen et al. [30] prove a conditional lower bound for the document version of indexing for two patterns, i.e., finding all documents containing both of the two query patterns. Goldstein et al. [20] show that similar lower bounds can be achieved via conjectured hardness of set intersection. Thus, there are several results linking indexing for two patterns and set intersection. Our reduction is still quite different, since we need a translation from intersection to distance.

1.4 Outline

The paper is organized as follows. In Section 2 we define notation and recall some useful results. In Section 3 we show how to answer **Exists** and **Count** queries, proving Theorem 1(i). In Section 4 we show how to answer **Report** queries, proving Theorem 1(ii). In Section 5 we prove the lower bound, proving Theorem 2. In Appendix A we apply our techniques to solve the variant where $\alpha = 0$.

2 Preliminaries

Strings

A *string* S of length n is a sequence $S[0]S[1] \dots S[n-1]$ of characters from an alphabet Σ . A contiguous subsequence $S[i, j] = S[i]S[i+1] \dots S[j]$ is a *substring* of S . The substrings of the form $S[i, n-1]$ are the *suffixes* of S . The *suffix tree* [44] is a compact trie of all suffixes of $S\$$, where $\$$ is a symbol not in the alphabet, and is lexicographically smaller than any letter in the alphabet. Each leaf is labelled with the index i of the suffix $S[i, n-1]$ it corresponds to. Using perfect hashing [18], the suffix tree can be stored in $O(n)$ space and solve the string indexing problem (i.e., find and report all occurrences of a pattern P) in $O(m + \text{occ})$ time, where m is the length of P and occ is the number of times P occurs in S .

For any node v in the suffix tree, we define $\text{str}(v)$ to be the string found by concatenating all labels on the path from the root to v . The *locus* of a string P , denoted $\text{locus}(P)$, is the minimum depth node v such that P is a prefix of $\text{str}(v)$. The *suffix array* stores the suffix

indices of $S\$$ in lexicographic order. We identify each leaf in the suffix tree with the suffix index it represents. The suffix tree has the property that the leaves below any node represent suffixes that appear in consecutive order in the suffix array. For any node v in the suffix tree, $\text{range}(v)$ denotes the range that v spans in the suffix array. The *inverse suffix array* is the inverse permutation of the suffix array, that is, an array where the i th element is the index of suffix i in the suffix array.

Orthogonal range successor

The *orthogonal range successor problem* is to preprocess an array $A[0, \dots, n-1]$ into a data structure that efficiently supports the following queries:

- $\text{RangeSuccessor}(a, b, x)$: return the successor of x in $A[a, \dots, b]$, that is, the minimum $y > x$ such that there is an $i \in [a, b]$ with $A[i] = y$.
- $\text{RangePredecessor}(a, b, x)$: return the predecessor of x in $A[a, \dots, b]$, that is, the maximum $y < x$ such that there is an $i \in [a, b]$ with $A[i] = y$.

3 Existence and Counting

In this section we give a data structure that can answer *Exists* and *Count* queries. The main idea is to split the query interval into “large” and “small” distances. For large distances we exploit that there can only be a small number of consecutive occurrences and we check them with a simple segmentation of S . For small distances we cluster the suffix tree and precompute answers for selected pairs of nodes.

We first show how to use orthogonal range successor queries to find consecutive occurrences. Then we define the clustering scheme used for the suffix tree and give the complete data structure.

3.1 Using Orthogonal Range Successor to Find Consecutive Occurrences

Assume we have found the loci of P_1 and P_2 in the suffix tree. Then we can answer the following queries in a constant number of orthogonal range successor queries on the suffix array:

- $\text{FindConsecutive}_{P_2}(i)$: given an occurrence i of P_1 , return the consecutive occurrence (i, j) of P_1 and P_2 , if it exists, and NO otherwise.
- $\text{FindConsecutive}_{P_1}(j)$: given an occurrence j of P_2 , return the consecutive occurrence (i, j) of P_1 and P_2 , if it exists, and NO otherwise.

Given a query $\text{FindConsecutive}_{P_2}(i)$, we answer as follows. First, we compute the index $j = \text{RangeSuccessor}(\text{range}(\text{locus}(P_2)), i)$ to get the closest occurrence of P_2 after i . Then, we compute $i' = \text{RangePredecessor}(\text{range}(\text{locus}(P_1)), j)$ to get the closest occurrence of P_1 before j . If $i = i'$ then no other occurrence of P_1 exists between i and j and they are consecutive. In that case we return (i, j) . Otherwise, we return NO.

Similarly, we can answer $\text{FindConsecutive}_{P_1}(j)$ by first doing a RangePredecessor and then a RangeSuccessor query. Thus, given the loci of both patterns and a specific occurrence of either P_1 or P_2 , we can in a constant number of RangeSuccessor and RangePredecessor queries find the corresponding consecutive occurrence, if it exists.

3.2 Data Structure

To build the data structure we will use a cluster decomposition of the suffix tree.

Cluster Decomposition

A cluster decomposition of a tree T is defined as follows: For a connected subgraph $C \subseteq T$, a *boundary node* v is a node $v \in C$ such that either v is the root of T , or v has an edge leaving C – that is, there exists an edge (v, u) in the tree T such that $u \in T \setminus C$. A *cluster* is a connected subgraph C of T with at most two boundary nodes. A cluster with one boundary node is called a *leaf cluster*. A cluster with two boundary nodes is called a *path cluster*. For a path cluster C , the two boundary nodes are connected by a unique path. We call this path the *spine* of C . A *cluster partition* is a partition of T into clusters, i.e. a set CP of clusters such that $\bigcup_{C \in CP} V(C) = V(T)$ and $\bigcup_{C \in CP} E(C) = E(T)$ and no two clusters in CP share any edges. Here, $E(G)$ and $V(G)$ denote the edge and vertex set of a (sub)graph G , respectively. We need the next lemma which follows from well-known tree decompositions [1, 2, 3, 17] (see Bille and Gørtz [7] for a direct proof).

► **Lemma 3.** *Given a tree T with n nodes and a parameter τ , there exists a cluster partition CP such that $|CP| = O(n/\tau)$ and every $C \in CP$ has at most τ nodes. Furthermore, such a partition can be computed in $O(n)$ time.*

Data Structure

We build a clustering of the suffix tree of S as in Lemma 3, with cluster size at most τ , where τ is some parameter satisfying $0 < \tau \leq n$. Then the counting data structure consists of:

- The suffix tree of S , with some additional information for each node. For each node v we store:
 - The range v spans in the suffix array, i.e., $\text{range}(v)$.
 - A bit that indicates if v is on a spine.
 - If v is on a spine, a pointer to the lower boundary node of the spine.
 - If v is a leaf, the local rank of v . That is, the rank of v in the text order of the leaves in the cluster that contains v . Note that this is at most τ .
- The inverse suffix array of S .
- A range successor data structure on the suffix array of S .
- An array $M(u, v)$ of length $\lfloor \frac{n}{\tau} \rfloor + 1$ for every pair of boundary nodes (u, v) . For $1 \leq x \leq \lfloor \frac{n}{\tau} \rfloor$, $M(u, v)[x]$ is the number of consecutive occurrences (i, j) of $\text{str}(u)$ and $\text{str}(v)$ with distance at most x . We set $M(u, v)[0] = 0$.

Denote $M(u, v)[\alpha, \beta] = M(u, v)[\beta] - M(u, v)[\alpha - 1]$, that is, $M(u, v)[\alpha, \beta]$ is the number of consecutive occurrences of $\text{str}(u)$ and $\text{str}(v)$ with a distance in $[\alpha, \beta]$.

Space Analysis

We store a constant amount of words per node in the suffix tree. The suffix tree and inverse suffix array occupy $O(n)$ space. For the orthogonal range successor data structure we use the data structure of Nekrich and Navarro [41] which uses $O(n)$ space and $O(\log^\epsilon n)$ time, for constant $\epsilon > 0$. There are $O(n^2/\tau^2)$ pairs of boundary nodes and for each pair we store an array of length $O(n/\tau)$. Therefore the total space consumption is $O(n + n^3/\tau^3)$.

3.3 Query Algorithm

We now show how to count the consecutive occurrences (i, j) with a distance in the interval, i.e. $\alpha \leq j - i \leq \beta$. We call each such pair a *valid occurrence*.

To answer a query we split the query interval $[\alpha, \beta]$ into two: $[\alpha, \lfloor \frac{n}{\tau} \rfloor]$ and $[\lfloor \frac{n}{\tau} \rfloor + 1, \beta]$, and handle these separately.

3.3.1 Handling Distances $> \frac{n}{\tau}$

We start by finding the loci of P_1 and P_2 in the suffix tree. As shown in Section 3.1, this allows us to find the consecutive occurrence containing a given occurrence of either P_1 or P_2 . We implicitly partition the string S into segments of (at most) $\lfloor n/\tau \rfloor$ characters by calculating τ segment boundaries. Segment i , for $0 \leq i < \tau$, contains characters $S[i \cdot \lfloor \frac{n}{\tau} \rfloor, (i+1) \cdot \lfloor \frac{n}{\tau} \rfloor - 1]$ and segment τ (if it exists) contains the characters $S[\tau \cdot \lfloor \frac{n}{\tau} \rfloor, n-1]$. We find the last occurrence of P_1 in each segment by performing a series of `RangePredecessor` queries, starting from the beginning of the last segment. Each time an occurrence i is found we perform the next query from the segment boundary to the left of i , continuing until the start of the string is reached. For each occurrence i of P_1 found in this way, we use `FindConsecutive P_2` (i) to find the consecutive occurrence (i, j) if it exists. We check each of them, discard any with distance $\leq \frac{n}{\tau}$ and count how many are valid.

3.3.2 Handling Distances $\leq \frac{n}{\tau}$

In this part, we only count valid occurrences with distance $\leq \frac{n}{\tau}$. Consider the loci of P_1 and P_2 in the suffix tree. Let C_i denote the cluster that contains $\text{locus}(P_i)$ for $i = 1, 2$. There are two main cases.

At least one locus is not on a spine

If either locus is in a small subtree hanging off a spine in a cluster or in a leaf cluster, we directly find all consecutive occurrences as follows: If $\text{locus}(P_1)$ is in a small subtree then we use `FindConsecutive P_2` (i) on each leaf i below $\text{locus}(P_1)$ to find all consecutive occurrences, count the valid occurrences and terminate. If only $\text{locus}(P_2)$ is in a small subtree then we use `FindConsecutive P_1` (j) for each leaf j below $\text{locus}(P_2)$, count the valid occurrences and terminate.

Both loci are on the spine

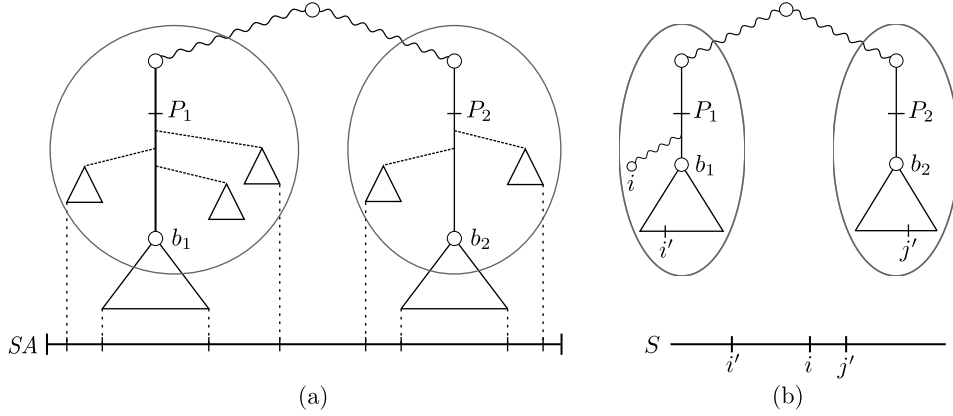
If neither locus is in a small subtree then both are on spines. Let (b_1, b_2) denote the lower boundary nodes of the clusters C_1 and C_2 , respectively. There are two types of consecutive occurrences (i, j) :

- (i) Occurrences where either i or j are inside C_1 resp. C_2 .
- (ii) Occurrences below the boundary nodes, that is, i is below b_1 and j is below b_2 .

See Figure 1(a). We describe how to count the different types of occurrences next.

Type (i) occurrences. To find the valid occurrences (i, j) where either $i \in C_1$ or $j \in C_2$ we do as follows. First we find all the consecutive occurrences (i, j) where i is a leaf in C_1 by computing `FindConsecutive P_2` (i) for all leaves i below $\text{locus}(P_1)$ in C_1 . We count all valid occurrences we find in this way. Then we find all remaining consecutive occurrences (i, j) where j is a leaf in C_2 by computing `FindConsecutive P_1` (j) for all leaves j below $\text{locus}(P_2)$ in C_2 . If `FindConsecutive P_1` (j) returns a valid occurrence (i, j) we use the inverse suffix array to check if the leaf i is below b_1 . This can be done by checking whether i 's position in the suffix array is in $\text{range}(b_1)$. If i is below b_1 we count the occurrence, otherwise we discard it.

Type (ii) occurrences. Next, we count the consecutive occurrences (i, j) , where both i and j are below b_1 and b_2 , respectively. We will use the precomputed table, but we have to be careful not to overcount. By its construction, $M(b_1, b_2)[\alpha, \min(\lfloor \frac{n}{\tau} \rfloor, \beta)]$ is the number



■ **Figure 1** (a) Any consecutive occurrences (i, j) of P_1 and P_2 is either also a consecutive occurrence of $\text{str}(b_1)$ and $\text{str}(b_2)$, or i or j are within the respective cluster. The suffix array is shown in the bottom with the corresponding ranges marked. (b) Example of a false occurrence. Here (i', j') is a consecutive occurrence of $\text{str}(b_1)$ and $\text{str}(b_2)$, but not a consecutive occurrence of P_1 and P_2 due to i . The string S is shown in bottom with the positions of the occurrences marked.

of consecutive occurrences (i', j') of $\text{str}(b_1)$ and $\text{str}(b_2)$, where $\alpha \leq j' - i' \leq \min(\lfloor \frac{n}{\tau} \rfloor, \beta)$. However, not all of these occurrence (i', j') are necessarily *consecutive* occurrences of P_1 and P_2 , as there could be an occurrence of P_1 in C_1 or P_2 in C_2 which is between i' and j' . We call such a pair (i', j') a *false occurrence*. See Figure 1(b). We proceed as follows.

1. Set $c = M(b_1, b_2)[\alpha, \min(\lfloor \frac{n}{\tau} \rfloor, \beta)]$.
2. Construct the lists L_i containing the leaves in C_i that are below $\text{locus}(P_i)$ sorted by text order for $i = 1, 2$. We can obtain the lists as follows. Let $[a, b]$ be the range of $\text{locus}(P_i)$ and $[a', b'] = \text{range}(b_i)$. Sort the leaves in $[a, a' - 1] \cup [b' + 1, b]$ using their local rank.
3. Until both lists are empty iteratively pick and remove the smallest element e from the start of either list. There are two cases.
 - e is an element of L_1 .
 - Compute $j' = \text{RangeSuccessor}(\text{range}(b_2), e)$ to get the closest occurrence of $\text{str}(b_2)$ after e .
 - Compute $i' = \text{RangePredecessor}(\text{range}(b_1), j')$ to get the closest occurrence of $\text{str}(b_1)$ before j' .
 - e is an element of L_2 .
 - Compute $i' = \text{RangePredecessor}(\text{range}(b_2), e)$ to get the previous occurrence i' of $\text{str}(b_1)$.
 - Compute $j' = \text{RangeSuccessor}(\text{range}(b_1), j')$ to get the following occurrence j' of $\text{str}(b_2)$.

If $\alpha \leq j' - i' \leq \min(\lfloor \frac{n}{\tau} \rfloor, \beta)$ and $i' < e < j'$ decrement c by one. We skip any subsequent occurrences that are also inside (i', j') . As the lists are sorted by text order, all occurrences that are within the same consecutive occurrence (i', j') are handled in sequence.

Finally, we add the counts of the different type of occurrences.

Correctness

Consider a consecutive occurrence (i, j) where $j - i > \frac{n}{\tau}$. Such a pair must span a segment boundary, i.e., i and j cannot be in the same segment. As (i, j) is a *consecutive* occurrence, i is the last occurrence of P_1 in its segment and j is the first occurrence of P_2 in its segment.

With the RangePredecessor queries we find all occurrences of P_1 that are the last in their segment. We thus check and count all valid occurrences of large distance in the initial pass of the segments.

If either locus is in a small subtree we use $\text{FindConsecutive}_{P_2}(\cdot)$ or $\text{FindConsecutive}_{P_1}(\cdot)$ on the leaves below that locus, which by the arguments in Section 3.1 will find all consecutive occurrences.

Otherwise, both loci are on a spine. To count occurrences of type (i), we first compute $\text{FindConsecutive}_{P_2}(i)$ for all leaves i below $\text{locus}(P_1)$ in C_1 and then $\text{FindConsecutive}_{P_1}(j)$ for all leaves j below $\text{locus}(P_2)$ in C_2 . However, any valid occurrence (i, j) where both $i \in C_1$ and $j \in C_2$ is found by both operations. Therefore, whenever we find a valid occurrence (i, j) via $i = \text{FindConsecutive}_{P_1}(j)$ for $j \in C_2$, we only count the occurrence if i is below b_1 . Thus we count all type (i) occurrences exactly once.

To count type (ii) occurrences we start with $c = M(b_1, b_2)[\alpha, \min(\lfloor \frac{n}{\tau} \rfloor, \beta)]$, which is the number of consecutive occurrences (i', j') of $\text{str}(b_1)$ and $\text{str}(b_2)$, where $\alpha \leq j' - i' \leq \min(\lfloor \frac{n}{\tau} \rfloor, \beta)$. Each (i', j') is either also a consecutive occurrence of P_1 and P_2 , or there exists an occurrence of P_1 or P_2 between i' and j' . Let (i', j') be a false occurrence and let w.l.o.g. i be an occurrence of P_1 with $i' < i < j'$. Then i is a leaf in C_1 , since (i', j') is a *consecutive* occurrence of $\text{str}(b_1)$ and $\text{str}(b_2)$. In step 3 we check for each leaf inside the clusters below the loci, if it is between a consecutive occurrence (i', j') of $\text{str}(b_1)$ and $\text{str}(b_2)$ and if $\alpha \leq j' - i' \leq \min(\lfloor \frac{n}{\tau} \rfloor, \beta)$. In that case (i', j') is a false occurrence and we adjust the count c . As (i', j') can have multiple occurrences of P_1 and P_2 inside it, we skip subsequent occurrences inside (i', j') . After adjusting for false occurrences, c is the number of type (ii) occurrences.

Time Analysis

We find the loci in $O(|P_1| + |P_2|)$ time. Then we perform a number of range successor and FindConsecutive queries. The time for a FindConsecutive query is bounded by the time to do a constant number of range successor queries. To count the large distances we check at most τ segment boundaries and thus perform $O(\tau)$ range successor and FindConsecutive queries.

For small distances, if either locus is not on a spine we check the leaves below that locus. There are at most τ such leaves due to the clustering. To count type (i) occurrences we check the leaves below the loci that are inside the clusters. There are at most 2τ such leaves in total. To count type (ii) occurrences we check two lists constructed from the leaves inside the clusters below the loci. There are again at most 2τ such leaves in total. For each of these $O(\tau)$ leaves we use a constant number of range successor and FindConsecutive queries. Thus the time for this part is bounded by the time to perform $O(\tau)$ range successor queries.

Using the data structure of Nekrich and Navarro [41], each range successor query takes $O(\log^\epsilon n)$ time so the total time for these queries is $O(\tau \log^\epsilon n)$. For type (ii) occurrences we sort two lists of size at most τ from a universe of size τ , which we can do in $O(\tau)$ time. Thus, the total query time is $O(|P_1| + |P_2| + \tau \log^\epsilon n)$.

Setting $\tau = \Theta(n^{2/3})$ we get a data structure that uses $O(n + n^3/\tau^3) = O(n)$ space and has query time $O(|P_1| + |P_2| + \tau \log^\epsilon n) = O(|P_1| + |P_2| + n^{2/3} \log^\epsilon n)$, for constant $\epsilon > 0$. We answer an Exists query with a Count query, terminating when the first valid occurrence is found. This concludes the proof of Theorem 1(i).

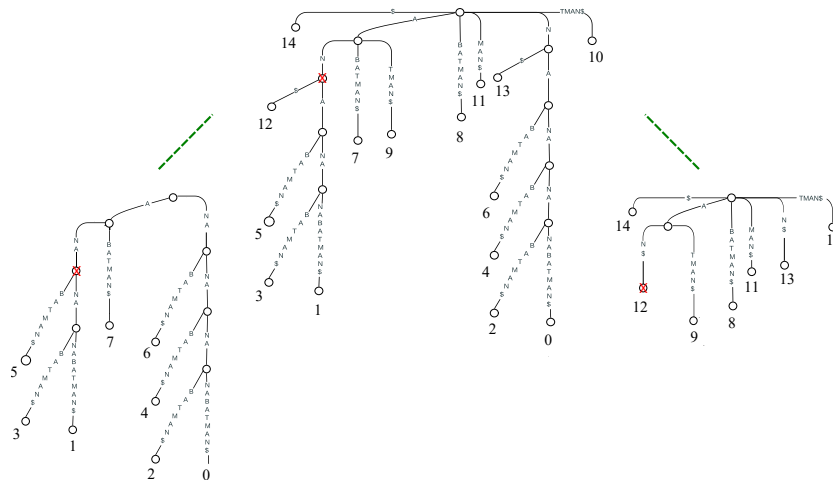


Figure 2 The suffix tree of NANANANABATMAN\$ together with its children trees $T[0, 7]$ and $T[8, 14]$. The red crosses show a node in the parent tree and its successor nodes in the two children trees.

4 Reporting

In this section, we describe our data structure for reporting queries. Note that in Section 3, we explicitly find all valid occurrences *except* for type (ii) occurrences, where we use the precomputed values. In this section, we describe how we can use a recursive scheme to report these.

The main idea, inspired by fast set intersection by Cohen and Porat [15], is to build a recursive binary structure which allows us to recursively divide the problem into subproblems of half the size. Intuitively, the subdivision is a binary tree where every node contains the suffix tree of a substring of S . We use this structure to find type (ii) occurrences by recursing on smaller trees. We define the binary decomposition of the suffix tree next. The details of the full solution follow after that.

4.1 Induced Suffix Tree Decomposition

Let T be a suffix tree of a string S of length n . For an interval $[a, b]$ of *text positions*, we define $T[a, b]$ to be the subtree of T induced by the leaves in $[a, b]$: That is, we consider the subtree consisting of leaves in $[a, b]$ together with their ancestors. We then delete each node that has only one child in the subtree and contract its ingoing and outgoing edge. See Figure 2.

The *induced suffix tree decomposition* of T now consists of a higher level binary tree structure, the *decomposition tree*, where each node corresponds to an induced subtree of the suffix tree. The root corresponds to $T[0, n - 1]$, and whenever we move down in the decomposition tree, the interval splits in half. We also associate a level with each of the induced subtrees, which is their depth in the decomposition tree. In more detail, the decomposition tree is a binary tree such that:

- The root of the decomposition tree corresponds to $T[0, n - 1]$ and has level 0.
- For each $T[a, b]$ of level i in the decomposition, if $b - a > 1$, its two children in the decomposition tree are $T[a, c]$ and $T[c + 1, b]$ where $c = \lfloor \frac{a+b}{2} \rfloor$; we will sometimes refer to these as “children trees” to differentiate from children in the suffix tree.

The decomposition tree is a balanced binary tree and the total size of the induced subtrees in the decomposition is $O(n \log n)$: There are at most 2^i decomposition tree nodes on level i , each of which corresponds to an induced subtree of size $O\left(\frac{n}{2^i}\right)$, and thus the total size of the trees on each of the $O(\log n)$ levels is $O(n)$.

For each node v in $T[a, b]$, we define the *successor node* of v in each of the children trees of $T[a, b]$ in the following way: If v exists in the child tree, the successor node is v . Else, it is the closest descendant which is present. Note that from the way the induced subtrees are constructed, v has at most one successor node in each child tree.

The induced suffix tree decomposition of S consists of:

- Each $T[a, b]$ stored as a compact trie.
- For each $T[a, b]$ we store a sparse suffix array $SA_{[a,b]}$, that is, the suffix array of $S[a, b]$ with the original indices within S .
- For each node v in $T[a, b]$ we store a pointer from v to its successor nodes in each child tree, if it exists, and the interval in $SA_{[a,b]}$ that corresponds to the leaves below v .

Since we store only constant information per node in any $T[a, b]$, the total space usage of this is $O(n \log n)$.

4.2 Data Structure

The reporting data structure consists of:

- The induced suffix tree decomposition for S ,
- An orthogonal range successor data structure on the suffix array, and
- The data structure from Section 3 for each $T[a, b]$ in the induced suffix tree decomposition with parameters n_i and τ_i , where $n_i = \lfloor \frac{n}{2^i} \rfloor$ and $\tau_i = \Theta(n_i^{2/3})$, such that $n_i/\tau_i = \lfloor n_i^{1/3} \rfloor$. The only change is that we do not store an orthogonal range successor data structure for each of the induced subtrees.

Space Analysis

We use the $O(n \log \log n)$ space and $O(\log \log n)$ time orthogonal range successor structure of Zhou [45]. The data structure from Section 3 for each $T[a, b]$ of level i is linear in n_i . Thus, by the arguments of Section 4.1, the total space is $O(n \log n)$.

4.3 Query Algorithm

The main idea behind the algorithm is the following: For large distances, as in Section 3, we implicitly segment S to find all consecutive occurrences of at least a certain distance. For small distances, we are going to use the cluster decomposition and counting arrays to decide whether valid occurrences exist. That is, if one of the loci is in a small subtree, we use $\text{FindConsecutive}_{P_2}(\cdot)$ resp. $\text{FindConsecutive}_{P_1}(\cdot)$ to find all consecutive occurrences. Else, we perform a query as in Section 3 to decide whether any valid occurrences exist, and if yes, we recurse on smaller subtrees.

The idea here is, that in the induced suffix tree decomposition, the trees are divided in half by *text position* - therefore, a *consecutive* occurrence either will be fully contained in the left child tree, fully contained in the right child tree, or have the property that the occurrence of P_1 is the maximum occurrence in the left child tree and the occurrence of P_2 is the minimum occurrence in the right child tree. We will check the border case each time when we recurse.

10:12 Gapped Indexing for Consecutive Occurrences

In detail, we do the following: We find the loci of P_1 and P_2 in the suffix tree. As in the previous section, we check τ_0 segment boundaries with $\tau_0 = \Theta(n^{2/3})$ to find all consecutive occurrences with distance within $[\max(\alpha, \lfloor n^{1/3} \rfloor), \beta]$. Now, we only have to find consecutive occurrences of distance within $[\alpha, \min(\beta, \lfloor n^{1/3} \rfloor)]$ in $T = T[0, n-1]$. In general, let $n_i = \lfloor \frac{n}{2^i} \rfloor$ and $\beta_i = \min(\beta, \lfloor n_i^{1/3} \rfloor)$ and let $T[a, b]$ be an induced subtree of level i .

To find all consecutive occurrences with distance within $[\alpha, \beta_i]$ in $T[a, b]$ of level i , given the loci of P_1 and P_2 in $T[a, b]$, recursively do the following:

- If any of the loci is not on a spine of a cluster, we find all consecutive occurrences using $\text{FindConsecutive}_{P_2}(\cdot)$ resp. $\text{FindConsecutive}_{P_1}(\cdot)$ and check for each of them if they are valid; we report all such, then terminate.
- Else, we use the query algorithm for small distances from Section 3 to decide whether a valid occurrence with distance within $[\alpha, \beta_i]$ exists in $T[a, b]$.

If such a valid occurrence exists, we recurse; that is, set $c = \lfloor \frac{a+b}{2} \rfloor$. We use RangePredecessor to find the last occurrence of P_1 before and including c , and RangeSuccessor to find the first occurrence of P_2 after c . Then we check if they are consecutive (again using RangePredecessor and RangeSuccessor), and if it is a valid occurrence. If yes, we add it to the output. Then, for both $S[a, c]$ and $S[c+1, b]$, we implicitly partition them into segments of size $\lfloor n_{i+1}^{1/3} \rfloor$ and find and output all valid occurrences of distance $> n_{i+1}^{1/3}$. Then we follow pointers to the successor nodes of the current loci to find the loci of P_1 and P_2 in the children trees $T[a, c]$ and $T[c+1, b]$ and recurse on those trees to find all consecutive occurrences of distance within $[\alpha, \beta_{i+1}]$.

Correctness

At any point before we recurse on level i , we check all consecutive occurrences of distance $> n_{i+1}^{1/3}$ by segmenting the current substring of S . By the arguments of the previous section, we will find all such valid occurrences. Thus, on the subtrees of level $i+1$, we need only care about consecutive occurrences with distance in $[\alpha, \beta_{i+1}]$.

By the properties of the induced suffix tree decomposition, a consecutive occurrence of P_1 and P_2 that is present in $T[a, b]$ will either be fully contained in $T[a, c]$, or in $T[c+1, b]$, or the occurrence of P_1 is the last occurrence before and including c and the occurrence of P_2 is the first occurrence after c . We check the border case each time we recurse. Thus, no consecutive occurrences get lost when we recurse. If we stop the recursion, it is either because one of the loci was in a small subtree or that no valid occurrences with distance within $[\alpha, \beta_i]$ exists in $T[a, b]$. In the first case we found all valid occurrences with distance within $[\alpha, \beta_i]$ in $T[a, b]$ by the same arguments as in Section 3. Thus, we find all valid occurrences of P_1 and P_2 .

Time Analysis

For finding the loci, we first spend $O(|P_1| + |P_2|)$ time in the initial suffix tree $T[0, n-1]$; after that, we spend constant time each time we recurse to follow pointers. The rest of the time consumption is dominated by the number of queries to the orthogonal range successor data structure, which we will count next.

Consider the recursion part of the algorithm as a traversal of the decomposition tree, and consider the subtree of the decomposition tree we traverse. Each leaf of that subtree is a node where we stop recursing. Since we only recurse if we know there is an occurrence to be found, there are at most $O(\text{occ})$ leaves. Thus, we traverse at most $O(\text{occ} \log n)$ nodes.

Each time we recurse, we spend a constant number of RangeSuccessor and RangePredecessor queries to check the border cases. Additionally, we spend $O(n_i^{2/3})$ such queries on each node of level i that we visit in the decomposition tree: For finding the “large” occurrences, and

additionally either for reporting everything within a small subtree or doing an existence query. For finding large occurrences, there are $O(n_i^{2/3})$ segments to check. The number of orthogonal range successor queries used for existence queries or reporting within a small subtree is bounded by the number of leaves within a cluster, which is also $O(n_i^{2/3})$.

Now, let x be the number of decomposition tree nodes we traverse and let $l_i, i = 1, \dots, x$, be the level of each such node. The goal is to bound $\sum_{i=1}^x \left(\frac{n}{2^{l_i}}\right)^{2/3}$. By the argument above, $x = O(\text{occ} \log n)$. Note that because the decomposition tree is binary we have that $\sum_{i=1}^x \frac{1}{2^{l_i}} \leq \log n$. The number of queries to the orthogonal range successor data structure is thus asymptotically bounded by:

$$\begin{aligned} \sum_{i=1}^x \left(\frac{n}{2^{l_i}}\right)^{2/3} &= n^{2/3} \sum_{i=1}^x \left(\frac{1}{2^{l_i}}\right)^{2/3} \cdot 1 \\ &\leq n^{2/3} \left(\sum_{i=1}^x \left(\frac{1}{2^{l_i}}\right)^{\frac{2}{3} \cdot \frac{3}{2}}\right)^{2/3} \left(\sum_{i=1}^x 1^3\right)^{1/3} \\ &= n^{2/3} \left(\sum_{i=1}^x \frac{1}{2^{l_i}}\right)^{2/3} x^{1/3} \\ &= O(n^{2/3} \text{occ}^{1/3} \log n) \end{aligned}$$

For the inequality, we use Hölder's inequality, which holds for all $(x_1, \dots, x_k) \in \mathbb{R}^k$ and $(y_1, \dots, y_k) \in \mathbb{R}^k$ and p and q both in $(1, \infty)$ such that $1/p + 1/q = 1$:

$$\sum_{i=1}^k |x_i y_i| \leq \left(\sum_{i=1}^k |x_i|^p\right)^{1/p} \left(\sum_{i=1}^k |y_i|^q\right)^{1/q} \tag{1}$$

We apply (1) with $p = 3/2$ and $q = 3$.

Since the data structure of Zhou [45] uses $O(\log \log n)$ time per query, the total running time of the algorithm is $O(|P_1| + |P_2| + n^{2/3} \text{occ}^{1/3} \log n \log \log n)$. This concludes the proof of Theorem 1(ii).

5 Lower Bound

We now prove the conditional lower bound from Theorem 2 based on set intersection. We use the framework and conjectures as stated in Goldstein et al. [20]. Throughout the section, let $\mathcal{I} = S_1, \dots, S_m$ be a collection of m sets of total size N from a universe U . The *SetDisjointness problem* is to preprocess \mathcal{I} into a compact data structure, such that given any pair of sets S_i and S_j , we can quickly determine if $S_i \cap S_j = \emptyset$. We use the following conjecture.

► **Conjecture 4** (Strong SetDisjointness Conjecture). *Any data structure that can answer SetDisjointness queries in t query time must use $\tilde{\Omega}\left(\frac{N^2}{t^2}\right)$ space.*

5.1 SetDisjointness with Fixed Frequency

We define a weaker variant of the SetDisjointness problem: the *f-FrequencySetDisjointness problem* is the SetDisjointness problem where every element occurs in precisely f sets. We now show that any solution to the *f-FrequencySetDisjointness* problem implies a solution to SetDisjointness, matching the complexities up to polylogarithmic factors.

► **Lemma 5.** *Assuming the Strong SetDisjointness Conjecture, every data structure that can answer f -FrequencySetDisjointness queries in time $O(N^\delta)$, for $\delta \in [0, 1/2]$, must use $\tilde{\Omega}(N^{2-2\delta-o(1)})$ space.*

Proof. Assume there is a data structure D solving the f -FrequencySetDisjointness problem in time $O(N^\delta)$ and space $O(N^{2-2\delta-\epsilon})$ for constant ϵ with $0 < \epsilon < 1$. Let $\mathcal{I} = S_1, \dots, S_m$ be a given instance of SetDisjointness, where each S_i is a set of elements from universe U , and assume w.l.o.g. that m is a power of two.

Define the *frequency* of an element, f_e , as the number of sets in \mathcal{I} that contain e . We construct $\log m$ instances $\mathcal{I}_1, \dots, \mathcal{I}_{\log m}$ of the f -FrequencySetDisjointness problem. For each j , $1 \leq j \leq \log m$, the instance \mathcal{I}_j contains the following sets:

- For each $i \in [1, m]$ a set S_i^j containing all $e \in S_i$ that satisfy $2^{j-1} \leq f_e < 2^j$;
- 2^{j-1} “dummy sets”, which contain extra copies of elements to make sure that all elements have the same frequency. That is, we add every element with $2^{j-1} \leq f_e < 2^j$ to the first $2^j - f_e$ dummy sets. These sets will not be queried in the reduction.

Instance \mathcal{I}_j has $O(m)$ sets and every element occurs exactly 2^j times. Further, the total number of elements is at most $2N$. We now build f -FrequencySetDisjointness data structures $D_j = D(\mathcal{I}_j)$ for each of the $\log m$ instances.

To answer a SetDisjointness query for two sets S_{i_1} and S_{i_2} , we query D_j for the sets $S_{i_1}^j$ and $S_{i_2}^j$, for each $1 \leq j \leq \log m$. If there exists a j such that $S_{i_1}^j$ and $S_{i_2}^j$ are not disjoint, we output that S_{i_1} and S_{i_2} are not disjoint. Else, we output that they are disjoint.

If there exists $e \in S_{i_1} \cap S_{i_2}$, let j be such that $2^{j-1} \leq f_e < 2^j$. Then $e \in S_{i_1}^j \cap S_{i_2}^j$, and we will correctly output that the sets are not disjoint. If S_{i_1} and S_{i_2} are disjoint, then, since $S_{i_1}^j$ is a subset of S_{i_1} and $S_{i_2}^j$ is a subset of S_{i_2} , the queried sets are disjoint in every instance. Thus we also answer correctly in this case.

Let N_j denote the total number of elements in \mathcal{I}_j . For each j , we have $N_j \leq 2N$ and thus $N_j^{2-2\delta-\epsilon} \leq (2N)^{2-2\delta-\epsilon}$. Thus, the space complexity is asymptotically bounded by

$$\sum_{j=1}^{\lceil \log m \rceil} N_j^{2-2\delta-\epsilon} = O(N^{2-2\delta-\epsilon} \log m).$$

Similarly, we have $N_j^\delta = O(N^\delta)$ and so the time complexity is asymptotically bounded by

$$\sum_{j=1}^{\lceil \log m \rceil} N_j^\delta = O(N^\delta \log m).$$

This is a contradiction to Conjecture 4. ◀

5.2 Reduction to Gapped Indexing

We can reduce the f -FrequencySetDisjointness problem to Exists queries of the gapped indexing problem: Assume we are given an instance of the f -FrequencySetDisjointness problem with a total of N elements. Each distinct element occurs f times. Assume again w.l.o.g. that the number of sets m is a power of two. Assign to each set S_i in the instance a unique binary string w_i of length $\log m$. Build a string S as follows: Consider an arbitrary ordering e_1, e_2, \dots of the distinct elements present in the f -FrequencySetDisjointness instance. Let $\$$ be an extra letter not in the alphabet. The first $B = f \cdot \log m + f$ letters are a

$$\begin{array}{ll}
S_1 = \{1, 2\} & w_1 = 00 \\
S_2 = \{3, 4\} & w_2 = 01 \\
S_3 = \{1, 3\} & w_3 = 10 \\
S_4 = \{2, 4\} & w_4 = 11 \\
\\
S = \underbrace{00\$10\$ \$ \$ \$ \$ \$ \$}_{1} \underbrace{00\$11\$ \$ \$ \$ \$ \$ \$}_{2} \underbrace{01\$10\$ \$ \$ \$ \$ \$ \$}_{3} \underbrace{01\$11\$ \$ \$ \$ \$ \$ \$}_{4}
\end{array}$$

■ **Figure 3** Instance of f -FrequencySetDisjointness problem reduced to **Exists**. Alphabet $\Sigma = \{0, 1\}$ and fixed frequency $f = 2$, resulting in block size $B = 2 \cdot 2 + 2 = 6$.

concatenation of $w_i\$$ of all sets S_i that e_1 is contained in, sorted by i . This block is followed by B copies of $\$$. Then, we have B symbols consisting of the strings for each set that e_2 is contained in, again followed by B copies of $\$$, and so on. See Figure 3 for an example.

For a query for two sets S_i and S_j , where $i < j$, we set $P_1 = w_i$ and $P_2 = w_j$, $\alpha = 0$, and $\beta = B$. If the sets are disjoint, then there are no occurrences which are at most B apart. Otherwise w_i and w_j occur in the same block, and w_j comes after w_i . The length of the string S is $2N \log m + 2N$: In the block for each element, we have $\log m + 1$ letters for each of its occurrences, and it is followed by a $\$$ block of the same length.

This means that if we can solve **Exists** queries in $s(n)$ space and $t(n) + O(|P_1| + |P_2|)$ time, where n is the length of the string, we can solve the f -FrequencySetDisjointness problem in $s(2N \log m + 2N)$ space and $t(2N \log m + 2N) + O(\log m)$ time. Together with Lemma 5, Theorem 2 follows.

6 Conclusion

We have considered the problem of gapped indexing for consecutive occurrences. We have given a linear space data structure that can count the number of such occurrences. For the reporting problem, we have given a near-linear space data structure. The running time for both includes an $O(n^{2/3})$ term, which forms a gap of $O(n^{1/6})$ to the conditional lower bound of $O(\sqrt{n})$. Thus, the most obvious open question is whether we can close this gap, either by improving the data structure or finding a stronger lower bound.

Further, we have used the property that there can only be few consecutive occurrences of large distances. Thus, our solution cannot be easily extended to finding *all* pairs of occurrences with distance within the query interval. An open question is if it is possible to get similar results for that problem. Lastly, document versions of similar problems have concerned themselves with finding all documents that contain P_1 and P_2 or the top- k of smallest distance; conditional lower bounds for these problems are also known. It would be interesting to see if any of these results be extended to finding all documents that contain a (consecutive) occurrence of P_1 and P_2 that has a distance within a query interval.

References

- 1 Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Minimizing diameters of dynamic trees. In *Proc. 24th ICALP*, pages 270–280, 1997.
- 2 Stephen Alstrup, Jacob Holm, and Mikkel Thorup. Maintaining center and median in dynamic trees. In *Proc. 7th SWAT*, pages 46–56, 2000.
- 3 Stephen Alstrup and Theis Rauhe. Improved labeling scheme for ancestor queries. In *Proc. 13th SODA*, pages 947–953, 2002.

10:16 Gapped Indexing for Consecutive Occurrences

- 4 Amihood Amir, Timothy M. Chan, Moshe Lewenstein, and Noa Lewenstein. On hardness of jumbled indexing. In *Proc. 41st ICALP*, pages 114–125, 2014.
- 5 Amihood Amir, Tsvi Kopelowitz, Avivit Levy, Seth Pettie, Ely Porat, and B. Riva Shalom. Mind the gap: Essentially optimal algorithms for online dictionary matching with one gap. In *Proc. 27th ISAAC*, pages 12:1–12:12, 2016.
- 6 Johannes Bader, Simon Gog, and Matthias Petri. Practical variable length gap pattern matching. In *Proc. 15th SEA*, pages 1–16, 2016.
- 7 Philip Bille and Inge Li Gørtz. The tree inclusion problem: In linear space and faster. *ACM Trans. Algorithms*, 7(3):1–47, 2011.
- 8 Philip Bille and Inge Li Gørtz. Substring range reporting. *Algorithmica*, 69(2):384–396, 2014.
- 9 Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, Eva Rotenberg, and Teresa Anna Steiner. String Indexing for Top- k Close Consecutive Occurrences. In *Proc. 40th FSTTCS*, volume 182, pages 14:1–14:17, 2020.
- 10 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.
- 11 Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and David Kofoed Wind. String matching with variable length gaps. *Theoret. Comput. Sci.*, 443, 2012. Announced at SPIRE 2010.
- 12 Sudip Biswas, Arnab Ganguly, Rahul Shah, and Sharma V Thankachan. Ranked document retrieval for multiple patterns. *Theor. Comput. Sci.*, 746:98–111, 2018.
- 13 P Bucher and A Bairoch. A generalized profile syntax for biomolecular sequence motifs and its function in automatic sequence interpretation. In *Proc. 2nd ISMB*, pages 53–61, 1994.
- 14 Manuel Cáceres, Simon J Puglisi, and Bella Zhukova. Fast indexes for gapped pattern matching. In *Proc. 46th SOFSEM*, pages 493–504, 2020.
- 15 Hagai Cohen and Ely Porat. Fast set intersection and two-patterns matching. *Theor. Comput. Sci.*, 411(40-42):3795–3800, 2010.
- 16 Paolo Ferragina, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Two-dimensional substring indexing. *J. Comput. Syst. Sci.*, 66(4):763–774, 2003.
- 17 Greg N Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2):484–538, 1997.
- 18 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $o(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.
- 19 Kimmo Fredriksson and Szymon Grabowski. Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance. *Inf. Retr.*, 11(4):335–357, 2008.
- 20 Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional Lower Bounds for Space/Time Tradeoffs. In *Proc. 15th WADS*, pages 421–436. Springer, 2017.
- 21 Tuukka Haapasalo, Panu Silvasti, Seppo Sippu, and Eljas Soisalon-Soininen. Online dictionary matching with variable-length gaps. In *Proc. 10th SEA*, pages 76–87, 2011.
- 22 K Hofmann, P Bucher, L Falquet, and A Bairoch. The PROSITE database, its status in 1999. *Nucleic Acids Res*, 27(1):215–219, 1999.
- 23 Wing-Kai Hon, Manish Patil, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Indexes for document retrieval with relevance. In *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 351–362, 2013.
- 24 Wing-Kai Hon, Manish Patil, Rahul Shah, and Shih-Bin Wu. Efficient index for retrieving top- k most frequent documents. *J. Discrete Algorithms*, 8(4):402–417, 2010.
- 25 Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Space-efficient frameworks for top- k string retrieval. *J. ACM*, 61(2):1–36, 2014. Announced at 50th FOCS.
- 26 Wing-Kai Hon, Sharma V. Thankachan, Rahul Shah, and Jeffrey Scott Vitter. Faster compressed top- k document retrieval. In *Proc. 23rd DCC*, pages 341–350, 2013.
- 27 Costas S Iliopoulos and M Sohel Rahman. Indexing factors with gaps. *Algorithmica*, 55(1):60–70, 2009.

- 28 Orgad Keller, Tsvi Kopelowitz, and Moshe Lewenstein. Range non-overlapping indexing and successive list indexing. In *Proc. 11th WADS*, pages 625–636, 2007.
- 29 Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *Proc. 27th SODA*, pages 1272–1287, 2016.
- 30 Kasper Green Larsen, J Ian Munro, Jesper Sindahl Nielsen, and Sharma V Thankachan. On hardness of several string indexing problems. *Theoret. Comput. Sci.*, 582:74–82, 2015.
- 31 Moshe Lewenstein. Indexing with gaps. In *Proc. 18th SPIRE*, pages 135–143, 2011.
- 32 Gerhard Mehdau and Gene Myers. A system for pattern matching applications on biosequences. *Bioinformatics*, 9(3):299–314, 1993.
- 33 J. Ian Munro, Gonzalo Navarro, Jesper Sindahl Nielsen, Rahul Shah, and Sharma V. Thankachan. Top-k term-proximity in succinct space. *Algorithmica*, 78(2):379–393, 2017. Announced at 25th ISAAC.
- 34 J. Ian Munro, Gonzalo Navarro, Rahul Shah, and Sharma V. Thankachan. Ranked document selection. *Theor. Comput. Sci.*, 812:149–159, 2020.
- 35 Eugene W. Myers. Approximate matching of network expressions with spacers. *J. Comput. Bio.*, 3(1):33–51, 1992.
- 36 Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):1–47, 2014.
- 37 Gonzalo Navarro and Yakov Nekrich. Time-optimal top-k document retrieval. *SIAM J. Comput.*, 46(1):80–113, 2017. Announced at 23rd SODA.
- 38 Gonzalo Navarro and Mathieu Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *J. Comput. Bio.*, 10(6):903–923, 2003.
- 39 Gonzalo Navarro and Sharma V. Thankachan. New space/time tradeoffs for top-k document retrieval on sequences. *Theor. Comput. Sci.*, 542:83–97, 2014. Announced at 20th SPIRE.
- 40 Gonzalo Navarro and Sharma V. Thankachan. Reporting consecutive substring occurrences under bounded gap constraints. *Theor. Comput. Sci.*, 638:108–111, 2016. Announced at 26th CPM.
- 41 Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In *Proc 13th SWAT*, pages 271–282, 2012.
- 42 Rahul Shah, Cheng Sheng, Sharma V. Thankachan, and Jeffrey Scott Vitter. Top-k document retrieval in external memory. In *Proc. 21st ESA*, pages 803–814, 2013.
- 43 Dekel Tsur. Top-k document retrieval in optimal space. *Inf. Process. Lett.*, 113(12):440–443, 2013.
- 44 Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th FOCS*, pages 1–11, 1973.
- 45 Gelin Zhou. Two-dimensional range successor in optimal time and almost linear space. *Inf. Process. Lett.*, 116(2):171–174, 2016.

A Gapped Indexing for $[0, \beta]$ Gaps

In this section, we consider the special case where the queries are one sided intervals of the form $[0, \beta]$. We give a data structure supporting the following tradeoffs:

- **Theorem 6.** *Given a string of length n , we can*
- (i) *construct an $O(n)$ space data structure that supports $\text{Exists}(P_1, P_2, 0, \beta)$ queries in $O(|P_1| + |P_2| + \sqrt{n} \log^\epsilon n)$ time for constant $\epsilon > 0$, or*
 - (ii) *construct an $O(n \log n)$ space data structure that supports $\text{Count}(P_1, P_2, 0, \beta)$ and $\text{Report}(P_1, P_2, 0, \beta)$ queries in $O(|P_1| + |P_2| + (\sqrt{n} \cdot \text{occ}) \log \log n)$ time, where occ is the size of the output.*

Note that since the results match (up to log factors) the best known results for set intersection, this is about as good as we can hope for. We mention here that for this specific problem, a similar tradeoff follows from the strategies used by Hon et al. [25]. The results from

10:18 Gapped Indexing for Consecutive Occurrences

that paper include (among others) a data structure for documents such that given a query of two patterns P_1 and P_2 and a number k , one can output the k documents with the closest occurrences of P_1 and P_2 . Thus, the problem is slightly different, however, with some adjustments, the results from Theorem 6 follow (up to a log factor). We show a simple, direct solution.

The data structure is a simpler version of the data structure considered in the previous sections. The main idea is that for each pair of boundary nodes u and v , we do not have to store an array of distances, but only one number that carries all the information: the smallest distance of a consecutive occurrence of $\text{str}(u)$ and $\text{str}(v)$. Thus, for existence, we can cluster with $\tau = \sqrt{n}$ to achieve linear space, and we do not need to check large distances separately. For the reporting solution, we store the decomposition from Section 4.1, and use the matrix M to decide where to recurse. In the following we will describe the details.

Existence data structure

For solving Exists queries in this setting, we cluster the suffix tree with parameter $\tau = \sqrt{n}$. Again, we store the linear space orthogonal range successor data structure by Nekrich and Navarro [41] on the suffix array. For each pair of boundary nodes (u, v) , we store at $M(u, v)$ the minimum distance of a consecutive occurrence of $\text{str}(u)$ and $\text{str}(v)$. The total space is linear. To query, we proceed similarly as in Section 3 for the “small distances”: We find the loci of P_1 and P_2 . If any of the loci is not on the spine, we check all consecutive occurrences using $\text{FindConsecutive}_{P_2}(\cdot)$ resp. $\text{FindConsecutive}_{P_1}(\cdot)$. If both loci are on the spine, denote b_1, b_2 the lower boundary nodes of the respective clusters. Find $M(b_1, b_2)$. If $M(b_1, b_2) \leq \beta$, we can immediately return YES: If a valid occurrence (i', j') of $\text{str}(b_1)$ and $\text{str}(b_2)$ exists, then either (i', j') is a consecutive occurrence of P_1 and P_2 , or there exists a consecutive occurrence of smaller distance. Otherwise, that is if $M(b_1, b_2) > \beta$, all valid occurrences (i, j) have the property that either i is in the cluster of $\text{locus}(P_1)$ or j is in the cluster of $\text{locus}(P_2)$, and we check all such pairs using $\text{FindConsecutive}_{P_2}(\cdot)$ resp. $\text{FindConsecutive}_{P_1}(\cdot)$. The running time is $O(|P_1| + |P_2| + \sqrt{n} \log^\epsilon n)$.

Reporting data structure

For the reporting data structure, we store the decomposition of the suffix tree as described in Section 4.1 and the $O(n \log n)$ space orthogonal range successor data structure by Zhou [45] on the suffix array. For each induced subtree of level i in the decomposition, we store the existence data structure we just described.

Reporting algorithm

The algorithm follows a similar, but simpler, recursive structure as in Section 4. We begin by finding the loci of P_1 and P_2 . If either of the loci is not on a spine, we find all consecutive occurrences using $\text{FindConsecutive}_{P_2}(\cdot)$ resp. $\text{FindConsecutive}_{P_1}(\cdot)$, check if they are valid, report these, and terminate. If both loci are on a spine, we check $M(b_1, b_2)$ for the lower boundary nodes b_1 and b_2 . If $M(b_1, b_2) > \beta$, all valid occurrences (i, j) have the property that either i is in the cluster of $\text{locus}(P_1)$ or j is in the cluster of $\text{locus}(P_2)$. We check all such pairs using $\text{FindConsecutive}_{P_2}(\cdot)$ resp. $\text{FindConsecutive}_{P_1}(\cdot)$, report the valid occurrences, and terminate. If $M(b_1, b_2) \leq \beta$, we recurse on the children trees. That is, we check the border case and follow pointers to the loci in the children trees.

Analysis

The space is $O(n \log n)$, just as in Section 4.

For time analysis, we spend $O(\sqrt{\frac{n}{2^{l_i}}})$ orthogonal range successor queries on the nodes in the decomposition tree of level l_i where we stop the recursion. For all other nodes we visit in the tree traversal, we only spend a constant number of queries. In total, we visit $O(\text{occ} \log(n/\text{occ}) + \text{occ})$ decomposition tree nodes (by following the analysis in [15]), and we spend $O(\sqrt{\frac{n}{2^{l_i}}})$ orthogonal range successor queries on $O(\text{occ})$ many such nodes.

We use the same notation as in Section 4. By $x = O(\text{occ})$ we now denote the number of nodes where we stop the algorithm and output. Since each such node can be seen as a leaf in a binary tree, $\sum_{i=1}^x \frac{1}{2^{l_i}} \leq 1$. We use the Cauchy-Schwarz inequality (which is a special case of Hölders with $p = q = 2$). We get as an asymptotic bound for the number of orthogonal range successor queries:

$$\begin{aligned} \sum_{i=1}^x \sqrt{\frac{n}{2^{l_i}}} &= \sqrt{n} \sum_{i=1}^x \sqrt{\frac{1}{2^{l_i}}} \cdot 1 \\ &\leq \sqrt{n} \sqrt{\sum_{i=1}^x \frac{1}{2^{l_i}}} \sqrt{\sum_{i=1}^x 1} \\ &\leq \sqrt{nx} = O(\sqrt{n \cdot \text{occ}}). \end{aligned}$$

Note that since $\text{occ} \log(n/\text{occ}) = O(\text{occ} \sqrt{n/\text{occ}}) = O(\sqrt{n \cdot \text{occ}})$, this brings the total number of orthogonal range successor queries to $O(\text{occ} + \sqrt{n \cdot \text{occ}})$. Using the data structure by Zhou [45], the time bound from Theorem 6 follows.