Prokrustean Graph: A substring index for rapid k-mer size analysis

Adam Park[1‡] and David Koslicki[1,2,3,‡]

[1] Department of Computer Science and Engineering, The Pennsylvania State University
[2] Department of Biology, The Pennsylvania State University
[3] Huck Institutes of the Life Sciences, The Pennsylvania State University
‡ Corresponding authors: akp6031@psu.edu, dmk333@psu.edu

Abstract. The widespread adoption of k-mers in bioinformatics has led to efficient methods utilizing genomic sequences in a variety of biological tasks. However, understanding the influence of k-mer sizes within these methods remains a persistent challenge, as the outputs of complex bioinformatics pipelines obscure this influence with various noisy factors. The choice of k-mer size is often arbitrary, with justification frequently omitted in the literature and method tutorials. Furthermore, recent methods employing multiple k-mer sizes encounter significant computational challenges.

Nevertheless, most methods are built on well-defined objects related to k-mers, such as de Bruijn graphs, Jaccard similarity, Bray-Curtis dissimilarity, and k-mer spectra. The role of k-mer sizes within these objects is more intuitive and can be described by numerous quantities and metrics. Therefore, exploring these objects across k-mer sizes opens opportunities for robust analyses and new applications. However, the evolution of k-mer objects with respect to k-mer sizes is surprisingly elusive.

We introduce a novel substring index, the Pro$k$rustean graph, that elucidates the transformation of k-mer sets across k-mer sizes. Our framework built upon this index rapidly computes k-mer-based quantities for all k-mer sizes, with computational complexity independent of the size range and dependent only on maximal repeats. For example, counting maximal simple paths in de Bruijn graphs for $k = 1, \ldots, 100$ is achieved in seconds using our index on a gigabase-scale dataset. We present a variety of such experiments relevant to pangenomics and metagenomics.

The Pro$k$rustean graph is space-efficiently constructed from the Burrows-Wheeler Transform. Through this construction, it becomes evident that other modern substring indices inherently face difficulties in exploring k-mer objects across sizes, which motivated our data structure.

Our implementation is available at: https://github.com/KoslickiLab/prokrustean.

1 Introduction

As the volume and number of sequencing reads and reference genomes grow every year, k-mer-based methods continue to gain popularity in computational biology. This simple approach—cutting sequences into substrings of a fixed length—offers significant benefits across various disciplines. Biologists consider k-mers as intuitive markers representing biologically significant patterns, bioinformaticians easily formulate novel methods by regarding k-mers as fundamental units that reflect their original sequences, and engineers leverage their fixed-length nature to optimize computational processes at a very low level. However, understanding the most crucial parameter, k, remains elusive, thereby leaving the accuracy of experimental results obscure to practitioners.

Two central challenges emerge in k-mer-based methods. First, the selection of the k-mer size is often arbitrary, despite its well-recognized influence on outcomes. This issue, though widely acknowledged, remains insufficiently addressed in the literature, with little formal guidance on determine an optimal k size for applications. The reasoning behind these choices is frequently unclear, typically confined to specific, unpublished experimental analyses (e.g., "we found that k=31 was appropriate. . . "). Second, methods attempting to utilize multiple k-mer sizes encounter significant computational burdens. The accuracy benefits of incorporating multiple k-mer sizes are overshadowed by the escalating computational costs with each additional k-mer size. Consequently, researchers often resort to "folklore" k values based on prior empirical results.

The influence of k-mer sizes within each method is a complex function reflecting bioinformatics pipelines that process k-mers. As biological adjustments and engineering strategies complicate the pipelines, their outputs obscure the impact of k-mer sizes with noisy factors, as simplified in the following abstraction:

$$\text{pipeline}\big(\texttt{k-mer-based objects(sequences, k)},\ \text{biological adjustments},\ \text{engineering}\big).$$

Despite the complexity of the pipelines, there always exist mathematically well-defined k-mer-based objects that form the foundation of method formulation. These objects abstract the utilization of k-mers and depend solely on sequences and k-mer sizes. Thus, they offer potential for generalizability and quantification of the influence of k-mers in methods. Indeed, intuitive quantities have been derived from k-mer-based objects; however, there are challenges in actually computing them, as detailed in several examples we now present.

In genome analysis, the number of distinct k-mers is often used to reflect the complexity of genomes. Although the number varies by k-mer sizes, large data sizes restrict experiments to a few k values [9, 15, 39]. A recent study suggests that the number of distinct k-mers across all k-mer sizes provides additional insights into pangenome complexity [10]. Furthermore, the frequencies of k-mers provide richer information, as discussed in [1, 46], but their computation becomes more complicated and sometimes impractical, even with a fixed k-mer size [7].

In comparative analyses, Jaccard Similarity is utilized for genome indexing and searching by being approximated through hashing techniques [26, 34]. Experiments attempting to assess the influence of k-mer sizes on Jaccard Similarity must undergo tedious iterations through various k-mer sizes [8]. Additionally, Bray-Curtis dissimilarity is a k-mer frequency-based measure frequently used to compare metagenomic samples, where experimentalists face resource limitations due to the growing size of sequencing data, even with a fixed k-mer size. Yet, the demand for analyzing multiple k-mer sizes continues to increase [23, 35, 36].

Genome assemblers utilizing k-mer-based de Bruijn graphs are probably the most sensitive to k-mer sizes, and those employing multi-k approaches face significant computational challenges. The choice of k-mer sizes is particularly crucial in *de novo* assemblers, yet they predominantly rely on heuristic methods [19, 27, 38]. The topological features of de Bruijn graphs are succinctly summarized by the compacting process, where vertices represent simple paths called unitigs, but exploring these features across varying k-mer sizes is computationally intensive. Furthermore, the development of multi-k de Bruijn graphs continues to be a challenging and largely theoretical endeavor [22, 40, 43], with no substantial advancements following the heuristic selection of multiple k-mer sizes implemented by metaSPAdes [33].

These examples motivated the pressing need to generalize the exploration of k-mer-based objects across varying k-mer sizes. We introduce a computational framework for rapidly computing quantities derived from k-mer-based objects across a range of k sizes. The objective is clarified in Section 2, with experimental results in Section 3. The underlying substring index and its usage are briefly presented in Section 4 and Section 5.

Due to the length of proofs and algorithms, construction algorithms are left in Supplementary materials. We emphasize that distinctions between the new substring index and other modern substring indexes are only briefly covered in Section 5, while the construction algorithms provide essential insights. The new index can be built from the Burrows-Wheeler Transform (BWT), but the BWT alone is unlikely to achieve the same efficiency as our framework. It appears that modern substring indexes based on longest common prefixes, such as the BWT, suffix tree, FM-index, and r-index, share similar limitations.

2 Setup and main theorem

Let Σ be our alphabet and let $S \in \Sigma^*$ represent a finite-length string, and $\mathcal{U} \subset \Sigma^*$ represent a set of strings. Consider the following preliminary definitions and main result detailing the Pro*k*rustean graph utility:

Definition 1. – *A region in the string S is a triple (S, i, j) such that $1 \leq i \leq j \leq |S|$.*
*** See the example after definition 2 which justifies this non-standard region notation (S, i, j).*
– *The string of a region is the corresponding substring: $str(S, i, j) := S_i S_{i+1} ... S_j$.*
– *The size or length of a region is the length of its string: $|(S, i, j)| := j - i + 1$.*
– *An extension of a region (S, i, j) is a larger region (S, i', j') including (S, i, j):*

$$(S, i, j) \subsetneq (S, i', j') := i' \leq i \leq j \leq j' \text{ and } |(S, i, j)| < |(S, i', j')|.$$

– *The occurrences of S in $\mathcal{U}$ are those regions in strings of $\mathcal{U}$ corresponding to S:*

$$occ_{\mathcal{U}}(S) := \{(S', i, j) \mid S' \in \mathcal{U} \ \& \ str(S', i, j) = S\}.$$

– *S is a maximal repeat in $\mathcal{U}$ if it occurs at least twice and more than any of its extensions: $|occ_{\mathcal{U}}(S)| > 1$, and for all S' a superstring of S, $|occ_{\mathcal{U}}(S)| > |occ_{\mathcal{U}}(S')|$. I.e., a maximal repeat S is one that occurs more than once in $\mathcal{U}$ (i.e. is a repeat), and any extension of which has a lower frequency in $\mathcal{U}$ than S.*
– *Let $\mathcal{R}_{\mathcal{U}}$ represent the set of all maximal repeats in $\mathcal{U}$.*

Theorem 1. ***(Main Result)*** *Given a set of sequences $\mathcal{U}$, there exists a substring representation $\mathcal{G}_{\mathcal{U}}$ of size $O(|\Sigma||\mathcal{R}_{\mathcal{U}}|)$. $\mathcal{G}_{\mathcal{U}}$ can be used to compute k-mer quantities of $\mathcal{U}$ for all $k = 1, \ldots, k_{\max}$ within $O(|\mathcal{G}_{\mathcal{U}}|)$ time and space, where $k_{\max}$ is the length of the longest sequence in $\mathcal{U}$. The k-mer quantities are defined as follows:*

– *(Counts) The number of distinct k-mers, as used in genome cardinality DandD [10].*
– *(Frequencies) k-mer Bray-Curtis dissimilarities, as used in comparing metagenomic samples [36].*
– *(Extensions) The number of maximal unitigs in k-mer de Bruijn graphs, as used in analyses of reference genomes and genome assembly [29, 42].*
– *(Occurrences) The number of edges with annotated length of k in an overlap graph. [45].*

Furthermore, $\mathcal{G}_{\mathcal{U}}$ can be constructed in $O(|bwt|+|\mathcal{G}_{\mathcal{U}}|)$ time and space utilizing bwt, a BWT representation of $\mathcal{U}$.

Proof. Section 4 defines $\mathcal{G}_{\mathcal{U}}$ and derives its size $O(|\Sigma||\mathcal{R}_{\mathcal{U}}|)$. Section 5 introduces the computation of k-mer quantities, and the construction algorithm is introduced in the Supplementary Section S3. □

$\mathcal{G}_{\mathcal{U}}$ is termed the Pro*k*rustean graph of $\mathcal{U}$. The main contribution is that the time complexity for computing k-mer quantities for all $k \in [1, k_{\max}]$ is independent of the range, specifically $O(|\mathcal{G}_{\mathcal{U}}|)$. Notably, $|\mathcal{G}_{\mathcal{U}}|$ is sublinear relative to N, where N is the cumulative sequence length of $\mathcal{U}$. In contrast, any direct computation of k-mer quantities from $\mathcal{U}$ for a fixed k-mer size demands, at best, $O(N)$ time. Extending this computation naively to $k = 1, \ldots, k_{\max}$ increases the complexity to $O(N^2)$ to cover all possible substrings.

3 Results

The construction and four applications of the Pro*k*rustean graph were conducted with various datasets that were randomly selected to represent a broad range of sizes, sequencing technologies, and biological origins. Diverse sequencing datasets were collected, including two metagenome short-read datasets, one metagenome long-read dataset, and two human transcriptome short-read datasets. The full list of datasets is provided in section S1. All experiments were performed on a Mac Studio equipped with an Apple M1 Max of 10 CPU cores and 64 GB memory.

3.1 Pro*k*rustean graph construction with k_{min}

The construction algorithm is introduced in the Supplementary Section S3. The Pro*k*rustean graph grows with the number of maximal repeats, i.e., $O(|\mathcal{G}_{\mathcal{U}}|) = O(|\Sigma| \cdot |\mathcal{R}_{\mathcal{U}}|)$. Also, the size of $\mathcal{G}_{\mathcal{U}}$ can be controlled by dropping maximal repeats of length below k_{min}. Then $\mathcal{G}_{\mathcal{U}}$ requires $O(|\Sigma| \cdot |\mathcal{R}_{\mathcal{U}}(k_{min})|)$ space where $\mathcal{R}_{\mathcal{U}}(k_{min})$ is the set of maximal repeats of length at least k_{min}. Then, subsequent sections cover the results of four applications that compute *k-mer quantities* for $k = k_{min}, \ldots, k_{max}$. We assume k_{max} is always less than $|\mathcal{G}_{\mathcal{U}}|$, even if the largest possible value is chosen, which is typical for sequencing data. Figure 1 shows how the graph size ($O(|\Sigma| \cdot |\mathcal{R}_{\mathcal{U}}(k_{min})|)$) of short sequencing reads decreases as k_{min} increases.

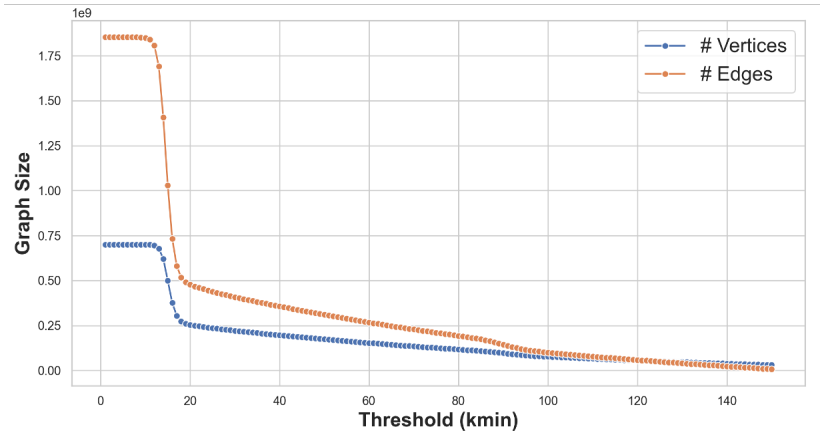


Fig. 1: Size decay of a Pro*k*rustean graph as a function of k_{min}. The sizes of Pro*k*rustean graphs, i.e., number of vertices and edges, versus k_{min} values for a short read dataset ERR3450203 are depicted. The size of the graph rapidly drops around $k_{min} = [11, ..., 19]$, meaning the maximal repeats and locally-maximal repeat regions are dense when k_{min} is under 11. The number of edges falls again around $k_{min} = 100$ because the graph eventually becomes fully disconnected.

Table 1 shows that the size of the Pro*k*rustean graphs is sublinear to that of their inputs. Also, the space occupancy is close to the inputs (read files) with practical k_{min} values. We used $k_{min} = 30$ that is common in genome assemblers and comparative microbiome analyses.

dataset	**reads**	**N**	k_{min}	**time**	**memory**	**output size**	$\lvert\mathcal{V}_\mathcal{U}\rvert$	$\lvert\mathcal{E}_\mathcal{U}\rvert$
SRR20044276 (metagenomic)	0.94M	72M	1	20s	200mb	40mb	8M	21M
SRR20044276 (metagenomic)	0.94M	72M	20	13s	180mb	35mb	2.7M	5.4M
ERR3450203 (metagenomic)	55M	4.5B	20	29min	12.7gb	3.1gb	254M	477M
ERR3450203 (metagenomic)	55M	4.5B	30	28min	11gb	2.7gb	221M	407M
SRR18495451 (metagenomic/long)	0.83M	11.3B	30	90min	43gb	14gb	890M	1.7B
SRR7130905 (human/rna-seq)	45M	6.8B	30	39min	18gb	4gb	280M	580M
SRR21862404 (human/rna-seq)	336M	22.3B	30	99min	37gb	5.9gb	330M	800M

Table 1: Pro*k*rustean graph construction introduced in Section S3. Symbols M, B, mb, and gb denote millions, billions, megabytes, and gigabytes, respectively. N represents the cumulative sequence length of $\mathcal{U}$. $\mathcal{V}_\mathcal{U}$ is the vertex set and $\mathcal{E}_\mathcal{U}$ is the edge set of $\mathcal{G}_\mathcal{U}$. Since $|\mathcal{E}_\mathcal{U}| \ll N$, the time complexity of subsequent applications $O(|\mathcal{G}_\mathcal{U}|) := O(|\mathcal{E}_\mathcal{U}|)$ is sublinear to the input, even if all k-mer sizes are considered.

Regarding subsequent results, note that, to our knowledge, no existing computational techniques perform the exact same tasks across a range of k-mer sizes, making performance comparisons inherently "unfair." Readers are encouraged to focus on the overall time scale to gauge the efficiency of the Pro*k*rustean graph. Additionally, discrepancies in output comparisons may arise due to practice-specific configurations in computational tools, such as the use of only ACGT (i.e., no "N") and canonical k-mers in KMC. Consequently, we tested correctness on GitHub[1] using our brute-force implementations.

3.2 Counting distinct k-mers for $k = k_{min}, \ldots, k_{max}$

The number of distinct k-mers is commonly used to assess the complexity and structure of pangenomes and metagenomes [13, 44]. A recent study by [10] spans multiple k contexts, suggesting that the number of k-mers across all k-mer sizes should be used to estimate genome sizes. The Supplementary Algorithm S1 counts distinct k-mers for all k-mer sizes in time $O(|\mathcal{G}_\mathcal{U}|)$. We employed KMC [28], an optimized k-mer counting library designed for a fixed k size, for comparison, and used it iteratively for all k values.

[1] `https://github.com/KoslickiLab/prokrustean`

Dataset	k	KMC			Prokrustean Graph		
		time	memory	threads	time (build)	memory	threads
SRR20044276	20...150	4.5m	840mb	8	**0.36s** (13s)	80mb	8
ERR3450203	20...150	50m	12gb	8	**30s** (29min)	6gb	8
SRR18495451	30...50000	days	-	8	**88s** (90min)	24gb	8
SRR7130905	30...150	66m	12gb	8	**46s** (39min)	8.5gb	8
SRR21862404	30...150	112m	13gb	8	**74s** (99min)	13.8gb	8

Table 2: Counting distinct k-mers with Pro$k$rustean graphs with Algorithm S1. For comparisons, KMC had to be executed iteratively causing the computational time to increase steadily as the range of k increases. The running time of KMC took about 1-3 minutes for each k-mer size.

3.3 Counting simple paths in de Bruijn graphs for $k_{min}, \ldots, k_{max}$

A maximal unitig of a de Bruijn graph represents a simple path that cannot be extended further. Maximal unitigs are vertices of a compacted de Bruijn graph [20], reflecting a topological characteristic of the graph. More maximal unitigs generally indicate more complex graph structures, which significantly impact genome assembly performance [3, 14]. Therefore, their number across multiple k-mer sizes reflects the influence of k-mer sizes on the complexity of assembly. Algorithm S3 counts maximal unitigs for all k-mer sizes in time $O(|\mathcal{G}_{\mathcal{U}}|)$.

We utilized GGCAT [20] for comparisons, a mature library constructing compacted de Bruijn graphs of a fixed k-mer size. We assumed their construction is equivalent to counting maximal unitigs. Again, computing maximal unitigs with the Pro$k$rustean graph becomes more efficient as additional k-mer sizes are considered. Table 3 displays the comparison, and Figure 2 illustrates one of the outputs.

Dataset	k	GGCAT		Prokrustean Graph	
		time	memory	time	memory
SRR20044276	20..150	13m	310mb	**0.36s**	70mb
ERR3450203	20..150	98m	1.1gb	**21s**	6.1gb
SRR18495451	30..50000	days	-	**127s**	26gb
SRR7130905	30...150	126m	1.4gb	**46s**	8.5gb
SRR21862404	30...150	168m	0.8gb	**76s**	13.8gb

Table 3: Counting maximal unitigs with GGCAT and Algorithm S3 using the Pro$k$rustean Graph. GGCAT was executed iteratively for each k size, with each iteration taking approximately 1-5 minutes. The efficiency gap becomes clearer when there are many possible k-mer sizes, as in long reads. Both methods used 8 threads.

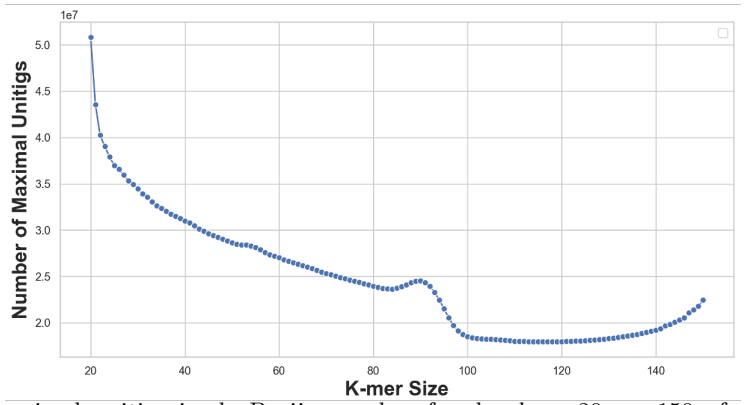


Fig. 2: The number of maximal unitigs in de Bruijn graphs of order $k = 20, \ldots, 150$ of metagenomic short reads (ERR3450203). A complex scenario is revealed: The decrease of the numbers fluctuates in $k = 30, \ldots, 60$, and the peak around $k = 90$ corresponds to the sudden decrease in maximal repeats. Lastly, further disconnections increase contigs and eventually make the graph completely disconnected.

3.4 Computing Bray-Curtis dissimilarities for $k_{min}, \ldots, k_{max}$

The Bray-Curtis dissimilarity is a frequency-based metric that measures the dissimilarity between biological samples. The Bray-Curtis dissimilarity for k-mer sets is defined as follow:

$$BC(\mathcal{U}_A, \mathcal{U}_B, k) = 1 - \frac{\sum_{w \in \{k\text{-mers in } \mathcal{U}_A \cup \mathcal{U}_B\}} 2 \cdot \min(\text{freq}_A(w), \text{freq}_B(w))}{\sum_{w \in \{k\text{-mers in } \mathcal{U}_A \cup \mathcal{U}_B\}} \text{freq}_A(w) + \text{freq}_B(w)}.$$

This quantity is particularly popular in metagenomics analysis [23, 36]. In each citation we found, the dissimilarity scores are presented for a single k-mer size. However, the values are sensitive to the choice of k-mer sizes, so they are often calculated with multiple k-mer sizes for analyses [23, 36, 47], yet most state-of-the-art tools utilize a fixed k-mer size [7]. Algorithm S2 computes Bray-Curtis dissimilarities for all k-mer sizes in $O(|Pairs| \cdot |\mathcal{G}_{\mathcal{U}}|)$, where $\mathcal{G}_{\mathcal{U}}$ contains sequencing reads for samples, and $Pairs$ is the sample pairs compared.

Figure 3 shows that dissimilarities between four example metagenome samples are not consistent in that at some k-mer sizes, one observed pattern of dissimilarity is completely flipped at another k-mer size. I.e. no k size is "correct"; instead, new insights are obtained from multiple k sizes. This task took about 10 minutes with the Pro$k$rustean graph of four samples of 12 gigabase pairs in total, and the graph construction took around 1 hour. Computing the same quantity takes around 5 to 8 minutes for a fixed k-mer size with Simka [7], and no library computes it across k-mer sizes.

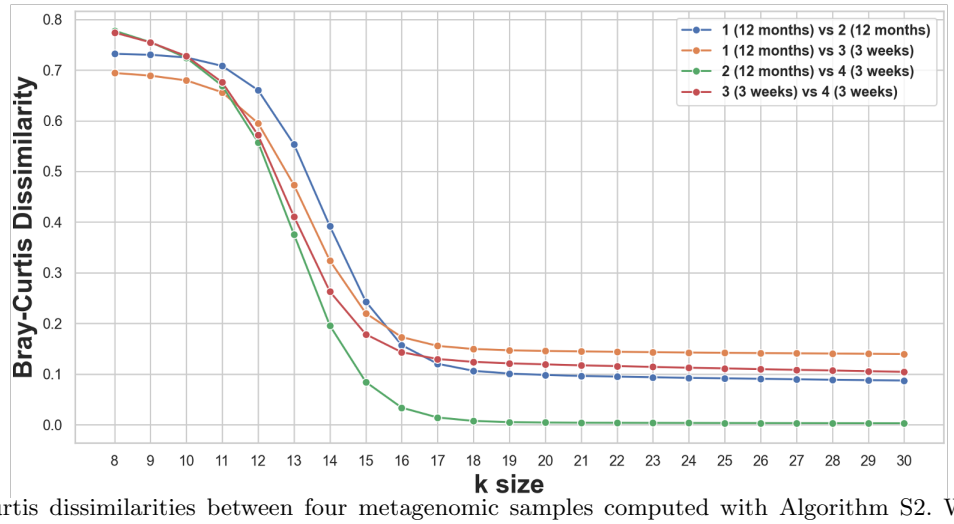


Fig. 3: Bray-Curtis dissimilarities between four metagenomic samples computed with Algorithm S2. We used the sequencing data of four samples derived from infant fecal microbiota, which were studied in [36]. Samples include two from 12-month-old human subjects and two from 3-week-olds. The relative order between values shift as k changes, e.g. compared to all other dissimilarities, the dissimilarity between sample 2 and 4 is highest at $k = 8$ but lowest at $k = 21$. Note that diverse k sizes (10, 12, 21, 31) are actively utilized in practice.

3.5 Counting vertex degrees of overlap graph of threshold k_{min}

Overlap graphs are extensively utilized in genome and metagenome assembly, alongside de Bruijn graphs. Although their definition appears unrelated to k-mers, we can view them as representing common suffix-prefix k-mers of sequences in $\mathcal{U}$. Overlap information is particularly crucial in read classification [2, 16, 31, 41] and contig binning [32, 45] in metagenomics. These applications interpret vertex degrees as (abundant-weighted) read coverage in samples, which exhibit high variances due to species diversity and abundance variability.

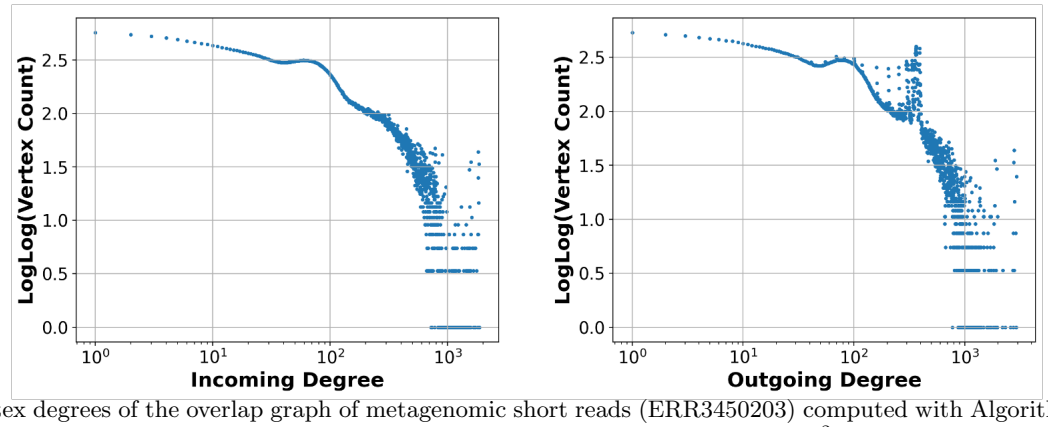


Fig. 4: Vertex degrees of the overlap graph of metagenomic short reads (ERR3450203) computed with Algorithm S4. The number of vertices per each degree is scaled as $\log \log y$. There is a clear peak around 10^2 at both incoming and outgoing degrees, which may imply dense existence of abundant species. The intermittent peaks after 10^2 might be related to repeating regions within and across species.

A common computational challenge is the quadratic growth of overlap graphs relative to the number of reads: $O(|\mathcal{U}|^2)$. In contrast, the Pro*k*rustean graph, which encompasses the overlap graph as a sort of hierarchical form, requires $O(|\mathcal{G}_{\mathcal{U}}|)$ space to access the structure of overlaps, enabling the efficient computation of vertex degrees.

Figure 4 is the result of computing the vertex degrees on a metagenomic short read dataset, as computed by Algorithm S4. With the Pro*k*rustean graph of 27 million short reads (4.5 gigabase pairs in total; ERR3450203), the computation used 8.5 gigabytes memory and 1 minute to count all vertex degrees.

4 Pro*k*rustean graph: A hierarchy of maximal repeats

The key idea of the Pro*k*rustean graph is to recursively capture maximal repeats by their relative frequencies of occurrences in $\mathcal{U}$. Consider Figure 5 depicting repeats in three sequences ACCCT, GACCC, and TCCCG.

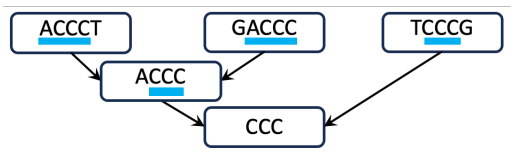


Fig. 5: Recursive repeats. At each sequence on the top level, locally-maximal repeat regions (indicated with blue underlining) are used to denote substrings that are more frequent than its superstring. The arrows then points to a node of the substring, and the process can repeat hierarchically.

Definition 2. (S, i, j) *is* <u>*a locally-maximal repeat region in* $\mathcal{U}$</u> *if:*

$$|occ_{\mathcal{U}}(str(S, i, j))| > |occ_{\mathcal{U}}(S)| \tag{1}$$

and for each extension (S, i', j') *of* (S, i, j)*,*

$$|occ_{\mathcal{U}}(str(S, i', j'))| = |occ_{\mathcal{U}}(S)|. \tag{2}$$

***Note that we often omit "in* $\mathcal{U}$*" when the context is clear.*

A locally-maximal repeat region captures a substring that appears more frequently than the "parent" string, and any extension of the region captures a substring that appears as frequently as the parent string. Consider the previous example $\mathcal{U} = \{$ACCCT, GACCC, TCCCG$\}$. The region $(\text{ACCCT}, 1, 4)$ is a locally-maximal repeat region capturing the substring ACCC in ACCCT. However, $(\text{ACCCT}, 2, 4)$ is not a locally-maximal repeat region because the region capturing CCC in ACCCT can be extended to the left to capture ACCC which occurs more frequently than ACCCT.

Note, an alternative definition using substring notations, such as *locally-maximal repeats*, instead of regions, is not robust, hence the requirement for us to use region notation. Consider $\mathcal{U} = \{$ACCCTCCG, GCCC$\}$ and S :=ACCCTCCG. Observe that the region $(S, 2, 4)$ capturing CCC is a locally-maximal repeat region because CCC occurs more frequently than S in $\mathcal{U}$, but its immediate left and right extensions capture ACCC and CCCT that occur the same number of times as S. In constrast, a subregion $(S, 2, 3)$ capturing CC is not a locally-maximal repeat region because an extension $(S, 2, 4)$ still captures a string (CCC) which occurs 2 times in $\mathcal{U}$ which is more than S. However, $(S, 6, 7)$, which also captures CC, is a locally-maximal repeat region because $|occ_{\mathcal{U}}(\text{CC})| = 5 > |occ_{\mathcal{U}}(S)| = 1$ and $|occ_{\mathcal{U}}(S)| = |occ_{\mathcal{U}}(\text{TCC})| = |occ_{\mathcal{U}}(\text{CCG})|$. Consequently, defining a *locally-maximal repeat* as $S_{6..7}$ =CC becomes ambiguous when compared with $S_{2..3}$=CC, whereas an explicit expression of a region $(S, 6, 7)$ more accurately reflects the desired property of hierarchy of occurrences.

The Pro*k*rustean graph of $\mathcal{U}$ is simply a graph representation of the recursive structure. Its vertexes represent the substrings and edges represent locally-maximal repeat regions. The theorem below says that all maximal repeats in $\mathcal{U}$ are captured along the recursive description.

Proposition 1 (Complete). *Construct a string set* $\mathbf{R}(\mathcal{U})$ *as follows:*

1. *for each locally-maximal repeat region* (S, i, j) *where* $S \in \mathcal{U}$*, add* $str(S, i, j)$ *to* $\mathbf{R}(\mathcal{U})$ *and*
2. *for each locally-maximal repeat region* (S, i, j) *where* $S \in \mathbf{R}(\mathcal{U})$*, add* $str(S, i, j)$ *to* $\mathbf{R}(\mathcal{U})$*.*

Then it follows that $\mathbf{R}(\mathcal{U}) = \mathcal{R}_{\mathcal{U}}$ *holds.*

Proof. Any string in $\mathbf{R}(\mathcal{U})$ is a maximal repeat of $\mathcal{U}$, so the claim is satisfied if the process captures every maximal repeat of $\mathcal{U}$. Assume a maximal repeat R is not in $\mathbf{R}(\mathcal{U})$. Consider any occurrence of R in some $S \in \mathcal{U}$. Extending the maximal repeat R within S makes the number of its occurrences drop, but since R cannot occur as a locally-maximal repeat region by assumption, it must be included in some locally-maximal repeat region that captures a maximal repeat R_1, hence $R_1 \in \mathbf{R}(\mathcal{U})$, and R is a proper substring of R_1. Now consider any occurrence of R within R_1. This argument continues recursively, capturing maximal repeats $R_1, R_2, ...$, but cannot extend indefinitely as R_{i+1} is always shorter than R_i for every $i \geq 1$. Eventually, R must be occurring as a locally-maximal repeat region of some R_n, which is a contradiction. □

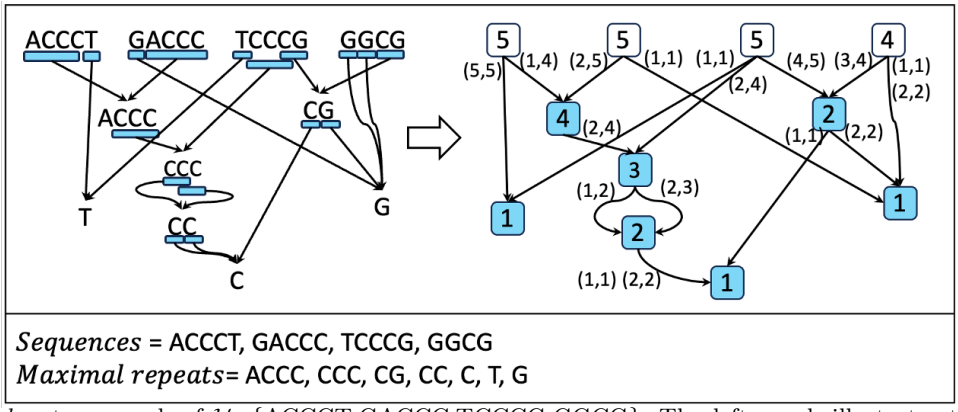


Fig. 6: The Pro*k*rustean graph of $\mathcal{U}$={ACCCT,GACCC,TCCCG,GGCG}. The left graph illustrates the recursion of locally-maximal repeat regions, depicted as blue regions, following the same rule described in Figure 5. The Pro*k*rustean graph on the right represents the same structure that white vertices correspond to sequences ($\mathcal{U}$) and blue vertices represent maximal repeats ($\mathcal{R}_\mathcal{U}$) found through the recursion process shown in the left graph. The strings are not stored, so integer labels store the regions and the size of the substrings.

Therefore, we use maximal repeats as vertices of the Pro*k*rustean graph, along with sequences, so $\mathcal{U} \cup \mathcal{R}_\mathcal{U}$.

Definition 3. *The Prokrustean graph of $\mathcal{U}$ is a directed multigraph $\mathcal{G}_\mathcal{U} := (\mathcal{V}_\mathcal{U}, \mathcal{E}_\mathcal{U})$:*

- *Vertex set $\mathcal{V}_\mathcal{U}$: $v_S \in \mathcal{V}_\mathcal{U}$ if and only if $S \in \mathcal{U} \cup \mathcal{R}_\mathcal{U}$.*
- *Each vertex $v_S \in \mathcal{V}_\mathcal{U}$ is annotated with the string size, $size(v_S) = |S|$.*
- *Edge set $\mathcal{E}_\mathcal{U}$: $e_{S,i,j} \in \mathcal{E}_\mathcal{U}$ if and only if (S, i, j) is a locally-maximal repeat region.*
- *Each edge $e_{S,i,j} \in \mathcal{E}_\mathcal{U}$ directs from v_S to $v_{str(S,i,j)}$ and is annotated with the $interval(e_{S,i,j}) := (i, j)$, representing the locally-maximal repeat region.*

Figure 6 visualizes how locally-maximal repeat regions are encoded in a Pro*k*rustean graph. Next, the cardinality analysis of this representation reveals promising bounds.

Theorem 2 (Compact). $|\mathcal{E}_\mathcal{U}| \leq 2|\Sigma||\mathcal{V}_\mathcal{U}|$.

Proof. Every vertex in $\mathcal{V}_\mathcal{U}$ has at most one incoming edge per letter extension on the right (or similarly on the left), so has at most $2|\Sigma|$ incoming edges. Assume the contrary: a maximal repeat appears as two different locally-maximal repeat regions (S, i, j) and (S', i', j') within $S, S' \in \mathcal{U} \cup \mathcal{R}_\mathcal{U}$, and can extend by the same letter on the right, i.e., $(S, i, j+1)$ and $(S', i', j'+1)$ capture the same string. Extending these regions further together will eventually diverge their strings, because either $S \neq S'$ or the original two regions are differently located even if $S = S'$. Consequently, at least one of them—$(S, i, j+1)$ without loss of generality—results in a decreased occurrence count of its string as it extends. Hence, $|occ_\mathcal{U}(str(S, i, j+1))| > |occ_\mathcal{U}(S)|$. This leads to contradiction because a locally-maximal repeat region (S, i, j) should satisfy $|occ_\mathcal{U}(S, i, j+1)| = |occ_\mathcal{U}(S, 1, |S|)|$. □

Assuming that $|\mathcal{U}| \ll |\mathcal{R}_\mathcal{U}|$, as is overwhelmingly the case in practice—meaning there are significantly more maximal repeats than sequences—we derive that $O(|\mathcal{V}_\mathcal{U}|) = O(|\mathcal{R}_\mathcal{U}|)$. Hence, $O(|\mathcal{G}_\mathcal{U}|) := O(|\mathcal{E}_\mathcal{U}|) = O(|\Sigma||\mathcal{R}_\mathcal{U}|)$ by the theorem. Given that $|\Sigma|$ is typically constant in genomic sequences (eg. $\Sigma = \{A, C, T, G\}$), the graph's size depends on the number of maximal repeats. Letting N be the accumulated sequence length of $\mathcal{U}$, it is a well-known fact that $|\mathcal{R}_\mathcal{U}| < N$ [25], so the graph grows sublinear to the input size. Furthermore, Section 3 restricted $\mathcal{R}_\mathcal{U}$ to maximal repeats of length at least $k_{\min}$. Setting a practical threshold $k_{\min}$ allows for significantly more efficient and configurable space usage.

4.1 Pro*k*rustean graph construction

We leave the construction algorithm in Supplementary *Section S*3. However, we emphasize that the core insight of our study comes from the relationship between the Pro*k*rustean graph and suffix trees. It turns out that a straightforward transformation can be defined with affix trees—the union of the suffix tree of $\mathcal{U}$ and the suffix tree of its reversed strings. A subset of edges of the affix tree of $\mathcal{U}$ corresponds to the reversed edge set of the Pro*k*rustean graph of $\mathcal{U}$. But affix trees are resource-intensive, so our final algorithm utilizes the BWT of $\mathcal{U}$, adding more intermediate steps to compensate for the non-bidirectional substring representation.

5 Framework: Computing *k-mer quantities* for all *k* sizes

We first describe the type of information required to efficiently compute *k-mer quantities* (defined in Theorem 1) to elucidate the uniqueness of the Pro*k*rustean graph. Specifically, why can't other modern substring indexes be used for the computations in previous results? The answer also implies why developing efficient multi-k k-mer-based methods has been challenging.

5.1 A proxy problem: computing substring co-occurrence.

There exists a theoretical void in identifying when a k-mer and a k'-mer serve similar roles within their respective substring sets. Although close k-mer sizes generally yield comparable outputs, dissecting this phenomenon at the level of local substring scopes is unexpectedly challenging. For instance, de Bruijn graphs constructed from sequencing reads with $k = 30$ and $k' = 31$ display distinct yet highly similar topologies [40]. However, anyone formally articulating the topological similarity would find it quite elusive, as vertex mapping and other techniques establishing correspondences between the two graphs fail to consistently explain it.

A common underlying difficulty is that the entire k-mer set undergoes complete transformations as the k-mer size changes. Since the role of a k-mer is assigned within its k-mer set, the role of a k'-mer within k'-mers cannot be "locally" derived. To address this issue, we propose substring co-occurrence as a generalized framework for consistently grouping k-mers of similar roles across varying k-mer sizes.

Definition 4. *A string S <u>co-occurs within</u> a string S' in $\mathcal{U}$ if:*

$$S \text{ is a substring of } S' \text{ and } |occ_{\mathcal{U}}(S)| = |occ_{\mathcal{U}}(S')|.$$

Modern substring indexes built on longest common prefixes of suffix arrays are adept at capturing co-occurrence while extending a substring. In suffix trees, a substring that extends from S to S' along an edge—without passing through a node—indicates preservation of co-occurrence. Similarly, the BWT and its variants represent both S and S' with the same set of suffix array intervals to imply co-occurrence. However, there has been no recognized necessity for capturing co-occurrence for substrings rather than superstrings:

Definition 5. *$co\text{-}substr_{\mathcal{U}}(S)$ is the set of substrings that co-occur within S.*

Modern substring indexes would fail to "smoothly" move over the order (k) unless $co\text{-}substr_{\mathcal{U}}(S)$ is efficiently accessible. For instance, the variable-order de Bruijn graph and its variants extend a compact de Bruijn graph representation built on the Burrows-Wheeler Transform [12]. To address the large substring space, these solutions require non-trivial operation times for both forward moves and order changes [5, 11].

We argue the usefulness of $co\text{-}substr_{\mathcal{U}}(S)$ with a simple yet foundational property.

Theorem 3 (Principle). *For any substring S such that $occ_{\mathcal{U}}(S) > 0$,*

$$S \text{ co-occurs within exactly one sequence or maximal repeat in } \mathcal{U} \cup \mathcal{R}_{\mathcal{U}}.$$

Proof. Assume the theorem does not hold, i.e., S co-occurs within either no string or multiple strings in $\mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$. Consider the former case: S co-occurs within no string in $\mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$, meaning S occurs more than any of its superstrings in $\mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$. Given that S occurs in some string $S' \in \mathcal{U}$ (since $occ_{\mathcal{U}}(S) > 0$), S must be occurring more frequently than S' by assumption. Then, there exists a maximal repeat R such that S is a substring of R and R is a substring of S', because extending S within S' decreases its occurrence at some point. Again, whenever S is a substring of a maximal repeat R, S must be occurring more frequently than R by assumption. Following the similar argument above, there exists another maximal repeat R' such that S is a substring of R' and R' is a proper substring of R. This recursive argument will eventually terminate as R' is strictly shorter than R. Thus, S co-occurs within the last maximal repeat, which contradicts the assumption.

Now consider the latter case where S co-occurs within at least two strings in $\mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$. If S is extended within these two strings, either to the right or left by one step at each time, the extensions eventually diverge into two different strings since the two superstrings differ, thereby reducing the number of occurrences. Hence, S must be occurring more frequently than at least one of those two superstrings, leading to a contradiction. □

This theorem provides two insights. First, the sequences and maximal repeats $\mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$ comprehensively and disjointly cover the entire substring, i.e., every substring in $\mathcal{U}$ is in $\bigsqcup_{S \in \mathcal{U} \cup \mathcal{R}_{\mathcal{U}}}$ *co-stack*$_{\mathcal{U}}(S)$. Second, by computing *co-substr*$_{\mathcal{U}}(S)$ for every $S \in \mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$, we can categorize k-mers of similar roles across various lengths. Therefore, we aim to design a succinct representation of *co-substr*$_{\mathcal{U}}(S)$ that can be accessed by the Pro$k$rustean graph.

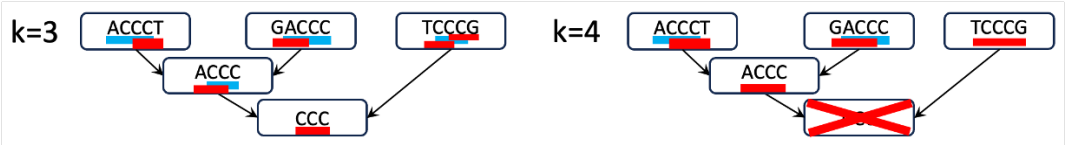


Fig. 7: Complementary regions. Blue underlining depicts locally maximal regions, and red underlining depicts regions that complement k-mers not covered in blue regions, given k values of 3 and 4. Every k-mer appears exactly once within a red region. For example, in the left image, the 3-mer ACC is included in a red region within ACCCC and does not appear again in any other red region. Note that red regions can include multiple co-occurring k-mers. So, in the right image, both 4-mers TCCC and CCCG co-occur within TCCCG.

5.2 An algorithmic idea: co-occurrence stacks.

We apply the Pro$k$rustean graph to compute k-mer quantities for all k sizes within the $O(|\mathcal{G}_{\mathcal{U}}|)$ time and space. Previously, a toy example in Figure 5 depicted blue regions that cover locally-maximal repeat regions. Then, given a k-mer size, a *complementary* region is a maximal region that covers k-mers that are not included in a blue region. See the red regions underlining in Figure 7. An opportunistic property is that k-mers in red regions co-occur within their parent strings. The following notation captures substrings in red regions.

Definition 6. *A string S <u>k-co-occurs within</u> a string S' in $\mathcal{U}$ if:*

every k-mer in S co-occurs within S' in $\mathcal{U}$.

Furthermore, S <u>maximally</u> k-co-occurs within S' in $\mathcal{U}$ if no superstring of S k-co-occurs within S' in $\mathcal{U}$. The red regions in Figure 7 identify maximal k-co-occurring substrings of S', so the Pro$k$rustean graph can be used to efficiently compute co-occurring k-mers of a fixed k-mer size. Now, we extend red regions for all possible k values, as shown in Figure 8. Observe that stack-like structures are formed around locally-maximal repeat regions of the string. The shape motivated the following definition:

Types	***Co-occurrence stacks (red) of*** **S = ACTCCGAAT**	k	Rate	Count	Expressed k-mers
Type 0	(S, T=ACTCCGAAT, k^*=9, d^*=4), δ_l=0, δ_r=0	*9*	-1	1	ACTCCGAAT
		8		2	ACTCCGAA,CTCCGAAT
		7		3	ACTCCGA,CTCCGAA,TCCGAAT
		6		4	ACTCCG,CTCCGA,TCCGAA,CCGAAT
Type 1	(S, T=CTCCGAAT, k^*=5, d^*=2), δ_l=1, δ_r=0	*5*	0	4	CTCCG,TCCGA,CCGAA,CGAAT
		4		4	TCCG,CCGA,CGAA,GAAT
Type 2	(S, T=CCGAA, k^*=3, d^*=3), δ_l=1, δ_r=1	*3*	*1*	*3*	CCG, CGA, GAA
		2		*2*	CG,GA
		1		*1*	G

Fig. 8: Three types of maximal co-occurrence stacks. Maximal co-occurrence stacks of S (sets of red rectangles) are introduced at each row for each type, given two locally-maximal repeat regions in S (blue rectangles). The Rate column describes the change in the count of expressed k-mers as the k-mer size increases by 1. Hence, the rate is $\delta_l + \delta_r - 1$. Also, δ_l and δ_r control the vertical shapes of stacks. For example, the Type 0 stack forms like a rectangle with $\delta_l = \delta_r = 0$, but the Type 1 stack forms like stairs on its left with $\delta_l = 1$.

Definition 7. *Consider two strings S and T where T is a substring of S, along with two numbers: a depth d^* and a k-mer size k^*, each at least 1. A* <u>*co-occurrence stack*</u> *of S in U is defined as a 4-tuple:*

$$(S, T, k^*, d^*)$$

if for each $i = 1, 2, \ldots, d^*$, for $k_i = k^* - (i-1)$, and for <u>*change rates*</u> *$\delta_l := \mathbf{1}_{T \text{ is NOT a prefix of } S}$ and $\delta_r := \mathbf{1}_{T \text{ is NOT a suffix of } S}$, the following holds:*

$$str(T, 1 + \delta_l \cdot (i-1), |T| - \delta_r \cdot (i-1)) \text{ maximally } k_i\text{-co-occurs within } S.$$

A co-occurrence stack <u>expresses</u> a k-mer if the k-mer is a substring of some k-co-occurring substring identified by the stack. A co-occurrence stack is <u>maximal</u> if its expressed k-mers are not a subset of those expressed by any other co-occurrence stack.

Let the set of all maximal co-occurrence stacks of S in $\mathcal{U}$ as $\textit{co-stack}_{\mathcal{U}}(S)$. It is straightforward to check that no two stacks in $\textit{co-stack}_{\mathcal{U}}(S)$ express the same k-mer together. So, we can nicely cover $\textit{co-stack}_{\mathcal{U}}(S)$ with $\textit{co-stack}_{\mathcal{U}}(S)$. We extend the scheme to the substring space of $\mathcal{U}$. Define $\mathcal{S}_{\mathcal{U}} := \bigcup_{S \in \mathcal{U} \cup \mathcal{R}_{\mathcal{U}}} \textit{co-stack}_{\mathcal{U}}(S)$.

Theorem 4 (Framework). *$\mathcal{S}_{\mathcal{U}}$ is a complete substring representation of $\mathcal{U}$ such that every substring (any k-mer of any k value) in $\mathcal{U}$ is expressed by exactly one stack in $\mathcal{S}_{\mathcal{U}}$. $|\mathcal{S}_{\mathcal{U}}|$ is bounded by $O(|\mathcal{G}_{\mathcal{U}}|)$, and can be enumerated in $O(|\mathcal{G}_{\mathcal{U}}|)$ time given $\mathcal{G}_{\mathcal{U}}$.*

Proof. Supplementary Section S2 derives this result. □

5.3 Algorithms

$\mathcal{S}_{\mathcal{U}}$ is used in the four algorithms computing the results in Section 3. For each stack in $\mathcal{S}_{\mathcal{U}}$, constant-time operations are defined, and since $|\mathcal{S}_{\mathcal{U}}|$ is $O(|\mathcal{G}_{\mathcal{U}}|)$, the time complexity of the algorithms is independent of the k-mer size interval $[k_{min}, k_{max}]$.

For example, consider using the stack in the first row of Figure 8 for counting distinct k-mers. The co-occurrence stack $(S, T = ACTCCGAAT, k^* = 9, d^* = 4)$ compactly represents the information of co-occurring substrings. From $k = 6$ to $k = 9$, the counts of expressed k-mers of S consistently decrease by 1, starting from 4. Therefore, for a count vector defined for the interval $[k_{min}, k_{max}]$, an algorithm can implement the contribution of the stack by "adding 4 to the count vector at $k = 6$" and "recording a change rate decrease of 1 at $k = 6$ and restoring it at $k = 9$." Repeating this process for all stacks in $\mathcal{S}_{\mathcal{U}}$ counts all k-mers in $\mathcal{U}$ and takes $O(|\mathcal{S}_{\mathcal{U}})|$ time. Refer to Supplementary Algorithm S1, Algorithm S2, Algorithm S3, and Algorithm S4 for the whole contexts.

6 Conclusion

We have introduced the problem of computing co-occurring substrings $\textit{co-substr}_{\mathcal{U}}(S)$ as a proxy to analyze the influence of k-mer sizes in k-mer-based methods. The Pro$k$rustean graph facilitates access to $\textit{co-substr}_{\mathcal{U}}(S)$ by iterating over $\textit{co-stack}_{\mathcal{U}}(S)$, which is used to implement algorithms that compute k-mer-based quantities across a range of k sizes. Once the Pro$k$rustean graph is constructed using the BWT in a space-efficient manner, the computation of k-mer-based quantities for all k-mer sizes becomes roughly equivalent to scanning the graph. Furthermore, we can control the performance by choosing k_{min}, the smallest maximal repeat size, so that the graph size is adjusted based on the needs of analyzed methods.

The Pro$k$rustean graph and the affix tree have intuitive relationships, as shown in Supplementary Figure S10. The contrast lies in accessing the co-occurrence structure. Suffix trees represent co-occurrence by extending substrings, functioning as a "bottom-up" approach. The Pro$k$rustean graph, however, follows a "top-down" approach, as co-occurring substrings (not superstrings) are identified in constant time. The issue with the bottom-up approach is that not all co-occurrence stacks $\textit{co-stack}_{\mathcal{U}}(S)$ are easily identified, even if bidirectional substring indexes are employed. For example, the Type-2 stack in Figure 8 requires information about two locally-maximal repeat regions within close proximity in a string, which is difficult to identify with substring extending operations. We therefore conjecture that exploring the space of k-mers with modern LCP-based substring indices presents an intrinsic challenge.

The results in Section 3.3 on counting maximal unitigs imply that a Pro$k$rustean graph can smoothly explore de Bruijn graphs of all orders. Thus, the Pro$k$rustean graph can advance the so-called variable-order scheme [11] by representing the union of de Bruijn graphs of all orders. A straightforward approach is to use the maximal co-occurrence stacks $\mathcal{S}_{\mathcal{U}}$ as a vertex set. We have confirmed that an edge set of size $O(|\mathcal{G}_{\mathcal{U}}|)$ can represent all

extensions between the stacks. Since the Pro$k$rustean graph does not grow faster than the number of maximal repeats, this new representation can serve as a compact multi-k assembly graph, more precisely identifying sequencing read errors, SNPs in population genomes, and genome contigs.

Thus, the open problem 5 in [40], which calls for a practical representation of variable-order de Bruijn graphs that generates assembly comparable to that of overlap graphs, can be answered with the Pro$k$rustean graph. Recall that Section 3.5 analyzed the overlap graph of $\mathcal{U}$. The two popular objects used in genome assembly can be accessed together through the Pro$k$rustean graph.

There is abundant literature analyzing the influence of k-mer sizes in various bioinformatics tasks, but little effort has been made to derive rigorous formulations or quantities to explain these phenomena. We expect our framework to contribute as an initial step toward understanding the influence, at least at the base level of k-mer-based objects of sequencing data or reference genomes.

Bibliography

[1] H. N. AlEisa, S. Hamad, and A. Elhadad. K-mer spectrum-based error correction algorithm for next-generation sequencing data. *Computational Intelligence and Neuroscience*, 2022, 2022.

[2] M. Balvert, X. Luo, E. Hauptfeld, A. Schönhuth, and B. E. Dutilh. Ogre: overlap graph-based metagenomic read clustering. *Bioinformatics*, 37(7):905–912, 2021.

[3] A. Bankevich, A. V. Bzikadze, M. Kolmogorov, D. Antipov, and P. A. Pevzner. Multiplex de bruijn graphs enable genome assembly from long, high-fidelity reads. *Nature biotechnology*, 40(7):1075–1081, 2022.

[4] D. Belazzougui. Linear time construction of compressed text indices in compact space. In *Proceedings of the forty-sixth Annual ACM Symposium on Theory of Computing*, pages 148–193, 2014.

[5] D. Belazzougui and F. Cunial. Fully-functional bidirectional burrows-wheeler indexes and infinite-order de bruijn graphs. In *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[6] T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the burrows–wheeler transform. *Journal of Discrete Algorithms*, 18:22–31, 2013.

[7] G. Benoit. Simka: fast kmer-based method for estimating the similarity between numerous metagenomic datasets. In *RCAM*, 2015.

[8] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik. Communication-efficient jaccard similarity for high-performance distributed genome comparisons. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1122–1132. IEEE, 2020.

[9] V. Bonnici and V. Manca. Informational laws of genome structures. *Scientific reports*, 6(1):28840, 2016.

[10] J. K. Bonnie, O. Ahmed, and B. Langmead. Dandd: efficient measurement of sequence growth and similarity. *bioRxiv*, pages 2023–02, 2023.

[11] C. Boucher, A. Bowe, T. Gagie, S. J. Puglisi, and K. Sadakane. Variable-order de bruijn graphs. In *2015 data compression conference*, pages 383–392. IEEE, 2015.

[12] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de bruijn graphs. In *International workshop on algorithms in bioinformatics*, pages 225–235. Springer, 2012.

[13] F. P. Breitwieser, D. N. Baker, and S. L. Salzberg. Krakenuniq: confident and fast metagenomics classification using unique k-mer counts. *Genome biology*, 19:1–10, 2018.

[14] K. Břinda, M. Baym, and G. Kucherov. Simplitigs as an efficient and scalable representation of de bruijn graphs. *Genome biology*, 22:1–24, 2021.

[15] Y. Bussi, R. Kapon, and Z. Reich. Large-scale k-mer-based analysis of the informational properties of genomes, comparative genomics and taxonomy. *PloS one*, 16(10):e0258693, 2021.

[16] M. Cavattoni and M. Comin. Classgraph: improving metagenomic read classification with overlap graphs. *Journal of Computational Biology*, 30(6):633–647, 2023.

[17] D. Cenzato, V. Guerrini, Z. Lipták, and G. Rosone. Computing the optimal BWT of very large string collections. In *In Proc. of the 33rd Data Compression Conference, DCC 2023, 2023*, pages 71–80, 2023.

[18] D. Cenzato and Z. Lipták. A survey of bwt variants for string collections. *Bioinformatics*, page btae333, 2024.

[19] R. Chikhi and P. Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, 30(1):31–37, 2014.

[20] A. Cracco and A. I. Tomescu. Extremely fast construction and querying of compacted and colored de bruijn graphs with ggcat. *Genome Research*, pages gr–277615, 2023.

[21] D. Díaz-Domínguez and G. Navarro. Efficient construction of the bwt for repetitive text using string compression. *Information and Computation*, 294:105088, 2023.

[22] D. D'ıaz-Dom'ınguez, T. Onodera, S. J. Puglisi, and L. Salmela. Genome assembly with variable order de bruijn graphs. *bioRxiv*, pages 2022–09, 2022.

[23] V. B. Dubinkina, D. S. Ischenko, V. I. Ulyantsev, A. V. Tyakht, and D. G. Alexeev. Assessment of k-mer spectrum applicability for metagenomic dissimilarity analysis. *BMC bioinformatics*, 17:1–11, 2016.

[24] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[25] D. Gusfield. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News*, 28(4):41–60, 1997.

[26] L. Irber, P. T. Brooks, T. Reiter, N. T. Pierce-Ward, M. R. Hera, D. Koslicki, and C. T. Brown. Lightweight compositional analysis of metagenomes with fracminhash and minimum metagenome covers. *bioRxiv*, pages 2022–01, 2022.

[27] R. Islam, R. S. Raju, N. Tasnim, I. H. Shihab, M. A. Bhuiyan, Y. Araf, and T. Islam. Choice of assemblers has a critical impact on de novo assembly of sars-cov-2 genome and characterizing variants. *Briefings in bioinformatics*, 22(5):bbab102, 2021.

[28] M. Kokot, M. Długosz, and S. Deorowicz. Kmc 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, 2017.

[29] T. Krannich, W. T. J. White, S. Niehus, G. Holley, B. V. Halldórsson, and B. Kehr. Population-scale detection of non-reference sequence variants using colored de bruijn graphs. *Bioinformatics*, 38(3):604–611, 2022.

[30] H. Li. Fast construction of fm-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014.

[31] X. Liao, M. Li, J. Luo, Y. Zou, F.-X. Wu, Y. Pan, F. Luo, and J. Wang. Improving de novo assembly based on read classification. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 17(1):177–188, 2018.

[32] V. Mallawaarachchi. *Metagenomics Binning Using Assembly Graphs*. PhD thesis, The Australian National University (Australia), 2022.

[33] S. Nurk, D. Meleshko, A. Korobeynikov, and P. A. Pevzner. metaspades: a new versatile metagenomic assembler. *Genome research*, 27(5):824–834, 2017.

[34] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy. Mash: fast genome and metagenome distance estimation using minhash. *Genome biology*, 17:1–14, 2016.

[35] A. E. Pérez-Cobas, L. Gomez-Valero, and C. Buchrieser. Metagenomic approaches in microbial ecology: an update on whole-genome and marker gene sequencing analyses. *Microbial genomics*, 6(8):e000409, 2020.

[36] A. J. Ponsero, M. Miller, and B. L. Hurwitz. Comparison of k-mer-based de novo comparative metagenomic tools and approaches. *Microbiome Research Reports*, 2(4), 2023.

[37] N. Prezza and G. Rosone. Space-efficient computation of the lcp array from the burrows-wheeler transform. *arXiv preprint arXiv:1901.05226*, 2019.

[38] A. Prjibelski, D. Antipov, D. Meleshko, A. Lapidus, and A. Korobeynikov. Using spades de novo assembler. *Current protocols in bioinformatics*, 70(1):e102, 2020.

[39] T. R. Ranallo-Benavidez, K. S. Jaron, and M. C. Schatz. Genomescope 2.0 and smudgeplot for reference-free profiling of polyploid genomes. *Nat. comm.*, 11(1):1432, 2020.

[40] R. Rizzi, S. Beretta, M. Patterson, Y. Pirola, M. Previtali, G. Della Vedova, and P. Bonizzoni. Overlap graphs and de bruijn graphs: data structures for de novo genome assembly in the big data era. *Quantitative Biology*, 7:278–292, 2019.

[41] L. M. Rodriguez-r and K. T. Konstantinidis. Estimating coverage in metagenomic data sets and why it matters. *The ISME journal*, 8(11):2349–2351, 2014.

[42] S. Schmidt, S. Khan, J. N. Alanko, G. E. Pibiri, and A. I. Tomescu. Matchtigs: minimum plain text representation of k-mer sets. *Genome Biology*, 24(1):136, 2023.

[43] B. Shariat, N. S. Movahedi, H. Chitsaz, and C. Boucher. Hydavista: towards optimal guided selection of k-mer size for sequence assembly. *BMC genomics*, 15(10):1–8, 2014.

[44] D. Tang, Y. Li, D. Tan, J. Fu, Y. Tang, J. Lin, R. Zhao, H. Du, and Z. Zhao. Kcoss: an ultra-fast k-mer counter for assembled genome analysis. *Bioinformatics*, 38(4):933–940, 2022.

[45] A. Wickramarachchi and Y. Lin. Metagenomics binning of long reads using read-overlap graphs. In *RECOMB International Workshop on Comparative Genomics*, pages 260–278. Springer, 2022.

[46] Z. Yang, H. Li, Y. Jia, Y. Zheng, H. Meng, T. Bao, X. Li, and L. Luo. Intrinsic laws of k-mer spectra of genome sequences and evolution mechanism of genomes. *BMC Evolutionary Biology*, 20:1–15, 2020.

[47] H. Zhai and J. Fukuyama. A convenient correspondence between k-mer-based metagenomic distances and phylogenetically-informed β-diversity measures. *PLOS Computational Biology*, 19(1):e1010821, 2023.

Supplemental Material

S1 Datasets

- Datasets mainly used in applications.
  SRR20044276, ERR3450203, SRR18495451, SRR21862404, SRR7130905

- Human pangenome references (Chromosome 1).
  AP023461.1, CH003448.1, CH003496.1, CM000462.1, CM001609.2, CM003683.2, CM009447.1, CM009872.1, CM010808.1, CM021568.2, CM034951.1, CM035659.1, CM039011.1, CM045155.1, CM073952.1, CM074009.1, CP139523.1, NC_000001.11, NC_060925.1

- Metagenome e.coli references.
  3682 E. coli assemblies in NCBI circa 2020. (https://zenodo.org/records/6577997)

- Metagenome sample for Bray-Curtis, referring to [36].
  ERS11976829, ERS11976830, ERS11976565, ERS11976566

S2 Full framework

We complete the framework in Section 5. Refer to the definitions and figure below which were already introduced in the main text.

Definition S8. *Consider two strings S and T where T is a substring of S, along with two numbers: a depth d^* and a k-mer size k^*, each at least 1. A* <u>*co-occurrence stack*</u> *of S in U is defined as a 4-tuple:*

$$(S, T, k^*, d^*)$$

if for each $i = 1, 2, \ldots, d^*$, for $k_i = k^* - (i-1)$, and for <u>*change rates*</u> *$\delta_l := \mathbf{1}_{T \text{ is NOT a prefix of } S}$ and $\delta_r := \mathbf{1}_{T \text{ is NOT a suffix of } S}$, the following holds:*

$$str(T, 1 + \delta_l \cdot (i-1), |T| - \delta_r \cdot (i-1)) \text{ maximally } k_i\text{-co-occurs within } S.$$

A co-occurrence stack <u>expresses</u> a k-mer if the k-mer is a substring of some k-co-occurring substring identified by the stack. A co-occurrence stack is <u>maximal</u> if its expressed k-mers are not a subset of those expressed by any other co-occurrence stack. It is clear that any k-mer expressed in a co-occurrence stack of S co-occurs within S. So, maximal co-occurrence stacks of S collectively express $co\text{-}substr_{\mathcal{U}}(S)$.

Definition S9. *$co\text{-}stack_{\mathcal{U}}(S)$ is the set of all maximal co-occurrence stacks of S in $\mathcal{U}$.*

A maximal co-occurrence stack of S is <u>type-0</u> if the substring T is S itself, <u>type-1</u> if T is a proper prefix or suffix of S, and <u>type-2</u> otherwise. The numeral in each type's name indicates the number of locally-maximal repeat regions intersecting the regions of substrings identified by the co-occurrence stack. Refer to Figure 8 for intuition on the type names. We further dig into the computational part regarding the types.

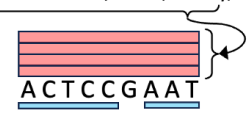
Types	Co-occurrence stacks (red) of S = ACTCCGAAT	k	Rate	Count	Expressed k-mers
Type 0	(S, T=ACTCCGAAT, k^*=9, d^*=4), δ_l=0, δ_r=0 	9	-1	1	ACTCCGAAT
		8		2	ACTCCGAA,CTCCGAAT
		7		3	ACTCCGA,CTCCGAA,TCCGAAT
		6		4	ACTCCG,CTCCGA,TCCGAA,CCGAAT
Type 1	(S, T=CTCCGAAT, k^*=5, d^*=2), δ_l=1, δ_r=0	5	0	4	CTCCG,TCCGA,CCGAA,CGAAT
		4		4	TCCG,CCGA,CGAA,GAAT
Type 2	(S, T=CCGAA, k^*=3, d^*=3), δ_l=1, δ_r=1	3	1	3	CCG, CGA, GAA
		2		2	CG,GA
		1		1	G

Fig. S9: Three types of maximal co-occurrence stacks.

Proposition 2. *A string S maximally k-co-occurs within a string S' in U if and only if its region in S' intersect locally-maximal repeat regions by less than $k-1$, and on both sides, either extends to an end of S' or intersects a locally-maximal repeat region by exactly $k-1$.*

Proof. The forward direction is straightforward: if S k-co-occurs within S', its region cannot intersect any locally-maximal repeat region by k at any case, and if the intersection is under $k-1$ on its edge, by extending S and making the intersection $k-1$, another superstring of S k-co-occurs within S'.

For the reverse direction, assume a region satisfies the conditions. A k-mer appearing in the region cannot occur more than S, as it would mean the region intersects a locally-maximal repeat region by at least k, so S' k-co-occurs within S. S' is also maximal because extending its region increases the size of the intersection to more than $k-1$. □

Proposition 3. *A maximal k-co-occurring substring T and a maximal k'-co-occurring substring T' of S are identified by the same maximal co-occurrence stack if and only if they share identical boundary conditions: on both sides, they either extend to the end of S or intersect the same locally-maximal repeat region by $k-1$ and $k'-1$, respectively.*

Proof. Consider maximal k-co-occurring substring T and k'-co-occurring substring T' of S where $k < k'$. The forward direction is straightforward; a maximal co-occurrence stack (S, S', k^*, d^*), which identifies T and T, identifies S' as a k^*-co-occurring substring of S by definition, and S' either extends to an end of S or intersects an locally-maximal repeat region by $k^* - 1$ on both sides according to Proposition 2. Then, as T and T' are identified as the $(k^* - k + 1)$-th and $(k^* - k' + 1)$-th elements by the stack, respectively, the rates δ_l and δ_r derived from the stack ensure T and T' sharing the equivalent side conditions with S' as described in Proposition 2.

For the reverse direction, consider three types of co-occurrence stacks: 1. $T = T' = S$, i.e., they extend to both ends of S. Then a type-0 stack identifies them. Whenever S k-co-occurs within S, the same holds for $k + 1$ and above until $|S|$. Therefore, $(S, T := S, k^* := |S|, d^*)$ with some maximal d^* identifies T and T' .

2. T and T' are proper prefixes of S. Then a type-1 stack identifies them. The condition says there is a locally-maximal repeat region (S, i, j) intersecting the regions of T and T' in S by $k - 1$ and $k' - 1$, respectively. Without loss of generality, let the intersection be on the left of (S, i, j). Then, a k^*-co-occurring prefix of S with $k^* := |(S, i, j)|$ intersects (S, i, j) by $k^* - 1$; if not, there must be a k^*-mer occurring more than $|occ_{\mathcal{U}}(S)|$ starting between positions 1 and $i - 1$. But k-mers of T start in the positions too, so T does not k-co-occur within S, which leads to contradiction. Thus, $(S, T := str(S, 1, j - 1), k^* := |(S, i, j)|, d^*)$ is a type-1 stack that identifies both T and T'.

3. T and T' are neither prefixes nor suffixes. Then a type-2 stack identifies them. There are two locally-maximal repeat regions (S, i, j) and (S, i', j') that intersect the regions of T and T' by $k - 1$ and $k' - 1$, respectively. Assuming (S, i, j) is smaller, a similar argument as in the previous paragraph shows that a k^*-co-occurring substring intersects both (S, i, j) and (S, i', j') by $k^* - 1$ given $k^* := |(S, i, j)|$, so $(S, T := str(S, j - k^* + 2, i' + k^* - 2), k^* := |(S, i, j)|, d^*)$ identifies T and T'. □

This proposition implies that either zero, one, or two locally-maximal repeat regions in S characterize a co-occurrence stack by identifying substrings of identical side conditions. This corresponds to the classifications—type-0, type-1, and type-2. The proposition also implies that the number of maximal co-occurrence stacks of S does not grow faster than the number of locally-maximal repeat regions in S.

Theorem S5. *co-stack$_{\mathcal{U}}(S)$ completely and disjointly cover co-occurring substrings of S, i.e.,*

$$\text{co-substr}_{\mathcal{U}}(S) = \bigsqcup_{(S,T,k^*,d^*) \in \text{co-stack}_{\mathcal{U}}(S)} \{\text{substrings expressed by } (S, T, k^*, d^*)\}. \tag{3}$$

Also,

$$|\text{co-stack}_{\mathcal{U}}(S)| \leq 1 + 2 \cdot (\text{the number of locally-maximal repeat regions in } S). \tag{4}$$

Hence, given $\mathcal{G}_{\mathcal{U}}$, enumerating co-stack$_{\mathcal{U}}(S)$ takes $O(|\text{outgoing edges of } v_S \text{ in } \mathcal{E}_{\mathcal{U}}|)$ time.

Proof. co-stack$_{\mathcal{U}}(S)$ completely and disjointly covers co-substr$_{\mathcal{U}}(S)$: any k-mer occurring in $\mathcal{U}$ co-occurs within an exactly one string S in $\mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$ by Theorem 3. The k-mer extends to a unique superstring that maximally k-co-occurs within S because appearing in two distinct maximally k-co-occurring substrings imply multi-occurrence of the k-mer in S. The superstring is identified by a unique maximal co-occurrence stack by Proposition 3. So, the k-mer is expressed by exactly one maximal co-occurrence stack.

To demonstrate the inequality, first, consider a type-0 stack on a string S, where all identified substrings are S itself by definition, i.e., extend to ends of S. A single type-0 stack identifies them all by Proposition 3. Hence, just one type-0 stack identifies them all by Proposition 3.

Next, for a type-1 or type-2 maximal co-occurrence stack (S, T, k^*, d^*), we show its longest identified substring always intersects some locally-maximal repeat region of length k^* by exactly $k^* - 1$. Since a proper type-1 stack identifies every k-co-occurring prefix (or suffix) of S whose region intersects the same locally-maximal repeat region by $k - 1$, by Proposition 3, the largest order k^* works too. Then, k^* should be the length of the locally-maximal repeat region; otherwise, we can find a $(k^* + 1)$-co-occurring prefix (or suffix) of S, denoted T', intersecting the locally-maximal repeat region by k^*, so $(S, T', k^* + 1, d^* + 1)$ is a proper co-occurrence stack, resulting in the original stack non-maximal. Similarly, a type-2 stack identifies substrings that intersect two locally-maximal repeat regions on both sides, so checking the length of the smaller region be k^* permits a similar deduction.

Note that each left and right side of a locally-maximal repeat region of length l intersect the region of at most one maximally l-co-occurring substring by exactly $l - 1$. Since such maximal intersection happens with every type-1 or type-2 maximal co-occurrence stack at least once, the number of such stacks is bounded by twice the number of locally-maximal repeat regions, thus is 2(the number of outgoing edges of v_S in $\mathcal{E}_{\mathcal{U}}$) in total.

Lastly, enumerating $\textit{co-stack}_{\mathcal{U}}(S)$ is efficient. A type-0 stack is uniquely defined for a string S where the depth d^* is one more than the length of the largest locally-maximal repeat region. Each side of a locally-maximal repeat region maximally intersects the largest substring identified in at most one stack, either type-1 or type-2. Without loss of generality, a type-1 stack is defined on the left side of a locally-maximal repeat region if it is the largest one among those found on its left, while a type-2 stack pairs with the nearest locally-maximal repeat region of at least the same size on its left. These properties can be verified by a single scan of the locally-maximal repeat regions of a string. □

Lastly, this scheme is extended to the entire substring space.

Proposition 4. *The number of all maximal co-occurrence stacks in $\mathcal{U}$ is $O(|\mathcal{G}_{\mathcal{U}})|$. Precisely,*

$$\sum_{S \in \mathcal{U} \cup \mathcal{R}_{\mathcal{U}}} |\textit{co-stack}_{\mathcal{U}}(S)| \leq |\mathcal{V}_{\mathcal{U}}| + 2|\mathcal{E}_{\mathcal{U}}|.$$

Proof. This is implied by the inequality Equation (4). The term $|\mathcal{V}_{\mathcal{U}}|$ accounts for the constant contribution of 1 from each $S \in \mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$, and $2|\mathcal{E}_{\mathcal{U}}|$ corresponds to the right term because $\mathcal{E}_{\mathcal{U}}$ represents the total set of locally-maximal repeat regions in $\mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$. □

Since k-mers expressed by each stack of S inherit characteristics of S, such as frequency and extensions within $\mathcal{U}$, algorithms can leverage this to compute *k-mer quantities*. We introduce four applications motivated by biological problems that benefit from this approach.

S2.1 Application: Counting distinct k-mers of all k sizes

The core idea leverages the change rate of k-mers within each co-occurrence stack, thereby eliminating the need to consider k-mer counts individually. The number of k-mers expressed by each type-0, 1, or 2 co-occurrence stack changes by -1, 0, or +1 as k increases by 1, respectively, as detailed in the Rate column of Figure S9. This property is utilized in lines 4–7 in Algorithm S1. Recalling that $|\textit{co-stack}_{\mathcal{U}}(S)|$ is bounded by $1 + 2 \times$ (the number of outgoing edges of v_S) in Theorem S5, the nested loop in lines 2-3 runs in $O(|\mathcal{G}_{\mathcal{U}}|)$ time.

Algorithm S1 Count k-mers of a range of k sizes

Input: A Pro$k$rustean graph $(\mathcal{V}_{\mathcal{U}}, \mathcal{E}_{\mathcal{U}})$, k_{max}
Output: The number of distinct k-mers in $\mathcal{U}$ for $k = 1, \ldots, k_{max}$
Make three 0 vectors of length k_{max}: C, ∂C, and $\partial\partial C$.
for $v_S \in \mathcal{V}_{\mathcal{U}}$ **do**
  for $((v_S, i, j), k^*, d^*) \in \textit{co-stack}_{\mathcal{U}}(S)$ **do**
    $\delta = \mathbf{1}_{i \neq 1} + \mathbf{1}_{j \neq size(v_S)} - 1$ // the change rate of the number of k-mers expressed by the stack
    $C[k^* - d^* + 1] \mathrel{+}= (j - i - k^* + 2) - \delta \cdot (d^* - 1)$ // the number of k-mers expressed by the stack for the lowest k-mer size
    $\partial\partial C[k^* - d^* + 2] \mathrel{+}= \delta$ // apply the change rate for the lowest k size
    $\partial\partial C[k^* - 1] \mathrel{-}= \delta$ // remove the change rate for the highest k size
  end
end
for $k = 2 \ldots k_{max}$ **do**
  $\partial C[k] \mathrel{+}= \partial C[k-1] + \partial\partial C[k]$ // total change rate of number of k-mers at each k
  $C[k] \mathrel{+}= C[k-1] + \partial C[k]$ // total number of k-mers at each k
end
return C

S2.2 Application: Computing Bray-Curtis dissimilarities of all k sizes

Recall the definition of the Bray-Curtis dissimilarity for k-mer sets.

$$BC(\mathcal{U}_A, \mathcal{U}_B, k) = 1 - \frac{\sum_{w \in \{k\text{-mers in } \mathcal{U}_A \cup \mathcal{U}_B\}} 2 \cdot \min(\text{freq}_A(w), \text{freq}_B(w))}{\sum_{w \in \{k\text{-mers in } \mathcal{U}_A \cup \mathcal{U}_B\}} \text{freq}_A(w) + \text{freq}_B(w)}.$$

We can compute the Bray-Curtis dissimilarity for all k-mer sizes of samples A and B using $\mathcal{G}_{\mathcal{U}_A \cup \mathcal{U}_B}$, i.e., the Pro$k$rustean graph constructed from the union of their sequence sets $\mathcal{U}_A$ and $\mathcal{U}_B$. For each $v_S \in \mathcal{V}_{\mathcal{U}_A \cup \mathcal{U}_B}$ such that $S \in \mathcal{U}_A \cup \mathcal{U}_B$, whether S is from A or B can be recorded while constructing $\mathcal{G}_{\mathcal{U}_A \cup \mathcal{U}_B}$. Then, $\text{freq}_A(w)$ is $|\text{occ}_{\mathcal{U}_A}(w)|$, accessed through some $v_S \in \mathcal{V}_{\mathcal{U}}$ satisfying w co-occurs within S. The algorithm is contained in Algorithm S2.

Algorithm S2 Compute Bray-Curtis dissimilarities of a range of k sizes

Input: A Pro$k$rustean graph $(\mathcal{V}_{\mathcal{U}}, \mathcal{E}_{\mathcal{U}})$ where $\mathcal{U} = \mathcal{U}_A \bigsqcup \mathcal{U}_B$, k_{max}
Output: Bray-Curtis dissimilarities between samples A and B for $k = 1, \ldots, k_{max}$
Make six 0 vectors of length k_{max}: N, ∂N, $\partial\partial N$, D, ∂D, $\partial\partial D$. ## numerator, denominator
Compute frequencies $freq_A(v_S)$ and $freq_B(v_S)$ for all $v_S \in \mathcal{V}_{\mathcal{U}}$.
for $v_S \in \mathcal{V}_{\mathcal{U}}$ **do**
  for $((v_S, i, j), k^*, d^*) \in \textit{co-stack}_{\mathcal{U}}(S)$ **do**
    Use lines 4-7 in Algorithm S1 to fill $\partial\partial N$: Replace C and $\partial\partial C$ with N and $\partial\partial N$, respectively, and weight the right-hand terms by $2 \cdot min(freq_A(v_S), freq_B(v_S))$.
    Use lines 4-7 in Algorithm S1 to fill $\partial\partial D$: Replace C and $\partial\partial C$ with D and $\partial\partial D$, respectively, and weight the right-hand terms by $freq_A(v_S) + freq_B(v_S)$.
  end
end
for $k = 2 \ldots k_{max}$ **do**
  Use lines 9-10 in Algorithm S1 to fill C, ∂C: Replace C, ∂C, and $\partial\partial C$ with N ∂N, and $\partial\partial N$, respectively.
  Use lines 9-10 in Algorithm S1 to fill D, ∂D: Replace C, ∂C, and $\partial\partial C$ with D ∂D, and $\partial\partial D$, respectively.
end
return $\mathbf{1}$-N/D ## $\mathbf{1}$ is a vector of length k_{max} filled with 1.

Recall that the value assignments in line 5 and 6 utilized co-occurrence stacks for counting k-mers in Algorithm S1. The values $min(freq_A(v_S), freq_B(v_S))$ and $freq_A(v_S) + freq_B(v_S)$ are assigned instead of a constant 1, because they grow linearly to the number of co-occurring k-mers weighted by the frequency-based values. Consequently, vectors N and D contains values of the numerator and denominator part of the Bray-Curtis dissimilarities for all k-mer sizes.

Typically, biological experiments compute these values for pairs of multiple samples. The proposed algorithm can be extended to consider multiple samples, requiring $O(|\text{samples}|^2 \cdot |\mathcal{G}_\mathcal{U}|)$ time for $\mathcal{U}$ the union of all sample read sets.

S2.3 Application: Counting maximal unitigs of all k sizes

Recall that a maximal unitig is a simple path in the de Bruijn graph that cannot be extended further. The idea for computing the number of maximal unitigs is that they are implied by tips, convergences, and divergences, which are indicated by type-0 or 1 co-occurrence stacks. For instance, consider a string S in $\mathcal{U} \cup \mathcal{R}_\mathcal{U}$ with multiple right extensions, such as C and T. If a suffix region in S of length l is an locally-maximal repeat region, it suggests that a k-co-occurring suffix of S where $k > l$ corresponds to a vertex in the respective de Bruijn graph of order k with right extensions C and T. Therefore, identifying the locally-maximal repeat regions on suffixes and prefixes of strings in $\mathcal{U} \cup \mathcal{R}_\mathcal{U}$ is sufficient. This idea is summarized in Algorithm S3.

Algorithm S3 Count maximal unitigs of de Bruijn graphs for a range of k sizes

Input: A Pro$k$rustean graph $(\mathcal{V}_\mathcal{U}, \mathcal{E}_\mathcal{U})$, k_{max}
Output: Numbers of maximal unitigs of de Bruijn graphs of orders $k = 1, \ldots, k_{max}$
Make two 0 vectors of length k_{max}: C, ∂C.
Compute numbers of left/right extensions $left(v_S)$ and $right(v_S)$ for all $v_S \in \mathcal{V}_\mathcal{U}$.
for $v_S \in \mathcal{V}_\mathcal{U}$ **do**
  for $((v_S, i, j), k^*, d^*) \in$ *co-stack*$_\mathcal{U}(S)$ **do**
    if $i = 1$ *and* $(left(v_S) = 0$ *or* $left(v_S) > 1)$ *## tips or convergence* **then**
      $C[k^* - d^* + 1]$+=1
      $\partial C[k^*]$−=1
    end
    if $j = size(v_S)$ *and* $right(v_S) > 1$ *## divergence* **then**
      $C[k^* - d^* + 1]$+=$right(v_S)$
      $\partial C[k^*]$−=$right(v_S)$
    end
  end
end
for $k = 2 \ldots k_{max}$ **do**
  $C[k]$+=$C[k-1]$+$\partial C[k]$
end
return C

This algorithm runs in $O(|\mathcal{G}_\mathcal{U}|)$ time, but can be accelerated in practice. Storing left(v_S) and right(v_S) in the Pro$k$rustean graph is not burdensome due to the small alphabet size ($|\Sigma|$), which allows them to be represented with 1 byte. Also, co-occurrence stacks should not be enumerated at line 4. Instead, it is enough to find two k^* values that are exactly one larger than locally-maximal repeat regions each located at the leftmost and rightmost positions in S, respectively. Then, $C[k^* - d^* + 1]$+=1 and $\partial C[size(v_S)]$−=1 computed for those two k^* values are straightforwardly equivalent to running line 4 to 10. So, assuming left(v_S) and right(v_S) can be accessed in constant time, Algorithm S3 can run in $O(|\mathcal{V}_\mathcal{U}|)$ time. So, the topological characteristics of de Bruijn graphs of all orders can be analyzed within time linear to the number of maximal repeats.

S2.4 Application: Computing vertex degrees of overlap graph

The overlap graph of $\mathcal{U}$ has the vertex set $\mathcal{U}$, and an edge $(v_S, v_{S'})$ if and only if a suffix of S matches a prefix of S'. Define $pref(v_R \in \mathcal{V}_\mathcal{U})$ with a substring R, as the number of R occurring as a prefix in strings in $\mathcal{U}$, which is easily computed through the recursive structure.

$$pref(v_R \in \mathcal{V}_\mathcal{U}) := \sum_{e=(v_{R'}, v_R) \in \mathcal{E}_\mathcal{U}, interval(e)=(1,*)} pref(v_{R'}).$$

Once $pref$ is computed, the outgoing degree of a vertex v_S is the sum of all $pref(v_R)$ such that R is a prefix maximal repeat of S. This computation requires two recursions of depth-first explorations of the Pro*k*rustean graph, resulting in an overall time complexity of $O(|\mathcal{G}_\mathcal{U}|)$. Refer to Algorithm S4 for a brief description.

Algorithm S4 Count vertex outgoing degrees of the overlap graph

Input: A Pro*k*rustean graph $(\mathcal{V}_\mathcal{U}, \mathcal{E}_\mathcal{U})$
Output: Vertex outgoing degrees of the overlap graph of $\mathcal{U}$

```
Make 0 vectors of length |V_U|: pref, incoming, degree, visit.
 Make a stack V.
 Initialize incoming with the number of incoming edges per vertex, i.e., incoming(v_S) = |{(*, v_S) ∈ E_U}|.
 Initialize V and pref: V.push(v_S) and pref(v_S) = 1 for v_S ∈ V_U such that S ∈ U.
 while V is not empty do
    v_S = V.pop().
     Get e = (v_S, v_S') ∈ E_U such that interval(e) = (1, *).
     if e ≠ ∅ then
        pref(v_S')+ = pref(v_S).
         incoming(v_S') − −.
         if incoming(v_S') = 0 then
            V.push(v_S').
        end
    end
end
Re-initialize V: V.push(v_S) for v_S ∈ V_U such that S ∈ U.
 while V is not empty do
    v_S = V.top().
     Get e = (v_S, v_S') ∈ E_U such that interval(e) = (*, size(v_S)).
     if e ≠ ∅ then
        if visit(v_S') = 1 then
            degree(v_S) = pref(v_S') + degree(v_S').
        end
        else
            V.push(v_S').
             Jump to next iteration.
        end
    end
    else
        degree(v_S) = pref(v_S).
    end
    visit(v_S) = 1.
     V.pop().
end
return degree
```

S3 Pro*k*rustean graph construction

In this section, we describe the Pro*k*rustean construction processes, starting with a simpler to describe, but less efficient approach before proceeding to the full implementation.

Recall that Section 5.1 addressed the limitations of LCP-based substring indexes in computing $co\text{-}substr_{\mathcal{U}}(S)$, which motivated the adoption of the Pro*k*rustean graph. LCP-based representations inherently provide a "bottom-up" approach and are at best able to access incoming edges of v_S in the Pro*k*rustean graph. In contrast, the Pro*k*rustean graph supports a "top-down" approach by accessing $co\text{-}substr_{\mathcal{U}}(S)$ through co-occurrence stacks whose number depends on the outgoing edges of v_S.

We start with a more straightforward, but less efficient approach: Section S3.1 employs affix trees—the union of the suffix tree of $\mathcal{U}$ and the suffix tree of its reversed strings—to extract the Pro*k*rustean graph. Although affix trees allow straightforward operations supporting bidirectional extensions above suffix trees, they are resource-intensive in practice. Section S3.2 refines the approach using the BWT of $\mathcal{U}$, incorporating an intermediate model to compensate the bidirectional context lost in moving away from affix trees. Section S3.3 then briefly introduce the implementation, covering recent advancements in BWT-based computations supporting it.

S3.1 Extracting Pro*k*rustean graph from two suffix trees

For ease of explanation, we use two suffix trees to represent the affix tree. Let $\mathcal{T}_{\mathcal{U}}$ denote the generalized suffix tree constructed from $\mathcal{U}$. The reversed string of a string S is denoted by $rev(S)$, and the reversed string set $rev(\mathcal{U})$ is $\{rev(S) \mid S \in \mathcal{U}\}$. A node in a tree is represented as $node(S)$ if the path from the root to the node spells out the string S, with the considered tree being always clear from the context. We assume affix links are constructed, which map between $node(S)$ in $\mathcal{T}_{\mathcal{U}}$ and $node(rev(S))$ in $\mathcal{T}_{rev(\mathcal{U})}$ if both nodes exist in their respective trees.

Additionally, a string preceded/followed by a letter represents an extension of the string with that letter, as in σS or $S\sigma$. Similarly, concatenation of two strings S and S' can be written as SS'. An asterisk (*) on a substring denotes any number (including zero) of letters extending the substring, as in $S*$.

Vertex set Collecting maximal repeats of $\mathcal{U}$ requires information about $|occ_{\mathcal{U}}(\sigma R)|$ and $|occ_{\mathcal{U}}(R\sigma)|$ for each $\sigma \in \Sigma$, for any considered substring R. Most substring indexes built above LCPs provide access to occurrences through the LCP intervals, and the number of occurrences is represented by the length of the corresponding LCP intervals. The reverse representation $\mathcal{T}_{rev(\mathcal{U})}$ is not required yet, implying that the same information is accessible via single-directional indexes like the BWT in the next section.

Edge set A more nuanced approach is required to extract the edge set of a Pro*k*rustean graph. Since an edge in a suffix tree basically implies co-occurrence, it sometimes directly indicates a locally-maximal repeat region, providing a subset of substrings that co-occur within their extensions. That is, an edge from $node(S)$ to $node(S')$ in $\mathcal{T}_{\mathcal{U}}$ implies that S may represent a locally-maximal repeat region in S' capturing a prefix S of S'. However, different scenarios may arise—sometimes the reverse direction should be considered, or both directions, or neither. These variations depend on the combinations of letter extensions around the substring. We present a theorem that delineates these cases, visually explained in Figure S10 for intuition.

We define terms distinguishing occurrence patterns of extensions. A string R is <u>left-maximal</u> if $|occ_{\mathcal{U}}(\sigma R)| < |occ_{\mathcal{U}}(R)|$ for every $\sigma \in \Sigma$, and <u>left-non-maximal with σ</u> if $|occ_{\mathcal{U}}(\sigma R)| = |occ_{\mathcal{U}}(R)|$, and <u>right-maximality</u> and <u>right-non-maximality</u> are symmetrically defined.

Theorem S6. *Consider a maximal repeat $R \in \mathcal{R}_{\mathcal{U}}$ and letters $\sigma_l, \sigma_r \in \Sigma$. The following three cases define a bijective correspondence between selected edges in the suffix trees ($\mathcal{T}_{\mathcal{U}}$ and $\mathcal{T}_{rev(\mathcal{U})}$) and the edges ($\mathcal{E}_{\mathcal{U}}$) of the Prokrustean graph of $\mathcal{U}$.*

Case S6.1 *(Prefix Condition) The following three statements are equivalent.*
- *(a) $R\sigma_r$ is left-maximal.*
- *(b) There exists a string T of form $R\sigma_r*$ in $\mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$ such that an edge from $node(R)$ to $node(T)$ is in $\mathcal{T}_{\mathcal{U}}$.*
- *(c) There exists an edge $e_{T,i,j}$ in $\mathcal{E}_{\mathcal{U}}$ satisfying $str(T, i, j+1) = R\sigma_r$ and $i = 1$.*

Case S6.2 *(Suffix Condition) The following three statements are equivalent.*
- *(a) $\sigma_l R$ is right-maximal.*
- *(b) There exists a string T of form $*\sigma_l R$ in $\mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$ such that an edge from $node(rev(R))$ to $node(rev(T))$ is in $\mathcal{T}_{rev(\mathcal{U})}$.*

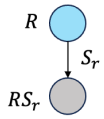
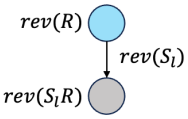
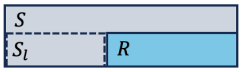
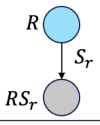
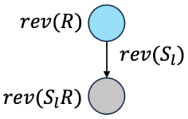
Scenarios	Suffix tree	Reversed suffix tree	Locally-maximal repeat regions
1. An edge in the suffix tree implies a locally-maximal repeat region at a prefix.			$(S, 1, \|R\|)$ where $S := RS_r$
2. An edge in the reversed suffix tree implies a locally-maximal repeat region at a suffix.			$(S, \|S_l\| + 1, \|S\|)$ where $S := S_lR$ 
3. One edge in the suffix tree and one in the reversed suffix tree imply a locally-maximal repeat region not extending to both ends.			$(S, \|S_l\|+1, \|S\| - \|S_r\|)$ where $S := S_lRS_r$

Fig. S10: Deriving locally-maximal repeat regions from edges of suffix trees. From a node of a maximal repeat R or $rev(R)$, the extension S is identified by adding strings on edge annotations. Then, the three scenarios depict how each edge(s) locates each locally-maximal repeat region that captures R within S.

(c) *There exists an edge* $e_{T,i,j} \in \mathcal{E}_{\mathcal{U}}$ *satisfying* $str(T, i-1, j) = \sigma_l R$ *and* $j = |T|$.

Case S6.3 *(Interior Condition) The following three statements are equivalent.*

(a) $R\sigma_r$ *is left-non-maximal with* σ_l *and* $\sigma_l R$ *is right-non-maximal with* σ_r.

(b) *There exist strings* S_l *and* S_r, *and* $T := S_l R S_r$ *of form* $*\sigma_l R \sigma_r *$ *in* $\mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$ *such that an edge from* $node(R)$ *to* $node(RS_r)$ *is in* $\mathcal{T}_{\mathcal{U}}$ *and an edge from* $node(rev(R))$ *to* $node(rev(S_l R))$ *is in* $\mathcal{T}_{rev(\mathcal{U})}$.

(c) *There exists an edge* $e_{T,i,j}$ *in* $\mathcal{E}_{\mathcal{U}}$ *satisfying* $str(T, i-1, j+1) = \sigma_l R \sigma_r$.

Proof. Case S6.1 $(a) \implies (b)$: We proceed arguing in the contrapositive. Assume $T := RS_r \notin \mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$ which is non-maximal on either left or right. Observe that the definition of the construction of the suffix tree $\mathcal{T}_{\mathcal{U}}$ implies $|occ_{\mathcal{U}}(R\sigma_r)| = |occ_{\mathcal{U}}(RS_r)|$ and RS_r being right-maximal. Since RS_r is right-maximal, RS_r must be left-non-maximal with some $\sigma \in \Sigma$, so $|occ_{\mathcal{U}}(\sigma RS_r)| = |occ_{\mathcal{U}}(RS_r)|$. Since $|occ_{\mathcal{U}}(R\sigma_r)| = |occ_{\mathcal{U}}(RS_r)|$, $|occ_{\mathcal{U}}(\sigma RS_r)| = |occ_{\mathcal{U}}(RS_r)| = |occ_{\mathcal{U}}(R\sigma_r)|$ meaning $R\sigma_r$ is left-non-maximal with some σ.

Case S6.1 $(b) \implies (c)$: Since $R, T \in \mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$ holds and $T[|R|+1] = \sigma_r$, showing $(T, 1, |R|)$ is a locally-maximal repeat region is enough to show $e_{T,1,|R|}$ satisfies the statement. Let $T := RS_r$. First, the region's string R occurs more than RS_r because R is a maximal repeat. Next, as $(RS_r, 1, |R|)$ is a prefix region, every extension includes the smallest extension $(RS_r, 1, |R|+1)$, so the string of any of its extensions occurs at least $|occ_{\mathcal{U}}(R\sigma_r)|$, but $\mathcal{T}_{\mathcal{U}}$ implies $|occ_{\mathcal{U}}(R\sigma_r)| = |occ_{\mathcal{U}}(RS_r)|$, so the extensions co-occur within RS_r. Hence, $(T, 1, |R|)$ is a locally-maximal repeat region and $str(T, 1, |R|+1) = R\sigma_r$.

Case S6.1 $(c) \implies (a)$: Since $T, 1, |R|$ is a locally-maximal repeat region, its extension captures $R\sigma_r$, so $|occ_{\mathcal{U}}(R\sigma_r)| = |occ_{\mathcal{U}}(T)|$ holds. Also, $T \in \mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$ implies T is left- and right-maximal, so $|occ_{\mathcal{U}}(\sigma T)| < |occ_{\mathcal{U}}(T)|$ holds for any $\sigma \in \Sigma$. Therefore, $|occ_{\mathcal{U}}(\sigma R\sigma_r)| < |occ_{\mathcal{U}}(R\sigma_r)|$ for any $\sigma \in \Sigma$, hence $R\sigma_r$ is left-maximal.

**Case S6.2 is symmetrically argued in line with Case S6.1.

Case S6.3 $(a) \implies (b)$: The non-maximalities in (a) imply $|occ_{\mathcal{U}}(\sigma_l R)| = |occ_{\mathcal{U}}(R\sigma_r)| = |occ_{\mathcal{U}}(\sigma_l R\sigma_r)|$. Also, by co-occurrences implied by two suffix trees, $|occ_{\mathcal{U}}(S_l R)| = |occ_{\mathcal{U}}(\sigma_l R)|$ and $|occ_{\mathcal{U}}(RS_r)| = |occ_{\mathcal{U}}(R\sigma_r)|$ hold. Hence, $|occ_{\mathcal{U}}(S_l R)| = |occ_{\mathcal{U}}(S_l RS_R)| = |occ_{\mathcal{U}}(RS_R)|$ is satisfied, meaning $S_l RS_R$ is left- and right-maximal that $T := S_l RS_R \in \mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$.

Case S6.3 $(b) \implies (c)$: The maximal repeat R occurs more than $S_l RS_r$ and the string of any extension occurs at least $|occ_{\mathcal{U}}(\sigma_l R)|$ or $|occ_{\mathcal{U}}(R\sigma_r)|$ times, while $|occ_{\mathcal{U}}(\sigma_l S)| = |occ_{\mathcal{U}}(S_l SS_R)| = |occ_{\mathcal{U}}(SS_R)|$ is implied by the two suffix trees. Hence, $(T, i, j) := (S_l RS_r, |S_l|+1, |S_l|+|R|)$ is a locally-maximal repeat region such that $str(T, i-1, j+1) = \sigma_l R\sigma_r$.

Case S6.3 $(c) \implies (a)$: This statement is trivial because the locally-maximal repeat region implies $|occ_{\mathcal{U}}(\sigma_l R)| = |occ_{\mathcal{U}}(S_l RS_R)| = |occ_{\mathcal{U}}(R\sigma_r)|$, hence $|occ_{\mathcal{U}}(\sigma_l R)| = |occ_{\mathcal{U}}(R\sigma_r)| = |occ_{\mathcal{U}}(\sigma_l R\sigma_r)|$ holds, which implies non-maximalities on both sides of R. □

The computation of extracting the edge set of the Pro*k*rustean graph is realized by collecting incoming edges for each vertex: Explore $\mathcal{T}_{\mathcal{U}}$ to identify each maximal repeat $R \in \mathcal{R}_{\mathcal{U}}$, and verify the maximality conditions in items labeled (a) in the theorem. Upon satisfying (a) of any case, use the corresponding (b) to identify a superstring T from the suffix trees $\mathcal{T}_{\mathcal{U}}$ and $\mathcal{T}_{rev(\mathcal{U})}$. The corresponding (c) then confirms that the region of

R within T is indeed a locally-maximal repeat region in $\mathcal{U}$. The precise region (T, i, j) is inferred from the decomposition of T outlined in (b). For instance, if T is $S_l R$ as in Case S6.2, then (T, i, j) is the suffix region $(T, |S_l| + 1, |T|)$, so an edge from v_T to v_R is labeled $(|S_l| + 1, |T|)$.

This approach covers every edge in the Pro*k*rustean graph. Observe that items labeled (c) distinguish locally-maximal repeat regions by the letters on immediate left or right. Each region is unique with respect to the letter extension, as previously established in the proof of Theorem 2. Hence, for each maximal repeat R, the conditions met in (a) have bijectively mapped incoming edges to v_R, thus the entire edge set $\mathcal{E}_{\mathcal{U}}$ is identified through Theorem S6.

We have leveraged a bidirectional substring index, but if only one direction is supported, i.e., $\mathcal{T}_{rev(\mathcal{U})}$ is unavailable, finding T as outlined in (b) becomes challenging. This limitation with single-directional substring indexes motivates the development of an enhanced algorithm.

S3.2 Extracting Pro*k*rustean graph from Burrows-Wheeler transform

This section addresses two issues of the previous computation when implemented in practice. Firstly, suffix tree representations generally consume substantial memory, which hinders scalability when handling large genomic datasets such as pangenomes. Secondly, bidirectional indexes are more space-consuming, expensive to build, and less commonly implemented than single-directional indexes. Instead, BWTs enable traversal of the same suffix tree structure using sublinear space relative to the input sequences, and are supported by a range of actively improved construction algorithms.

Since the bidirectional extensions in items of (b) in Theorem S6 are not directly applicable to the BWT or other single-directional substring indexes, we utilize a subset of occurrences of maximal repeats as an intermediate representation of locally-maximal repeat regions. This approach necessitates a consistent way of utilizing suffix orders within the generalized suffix arrays of $\mathcal{U}$.

Before describing the construction of the Pro*k*rustean from a BWT, we need the following definitions.

Definition S10. *Let the rank of a region (S, i, j) be the rank of the suffix of S starting at i in the generalized suffix array of $\mathcal{U}$, which is implied by a substring index being used. Then, the <u>first occurrence</u> of a string R, $occ_{\mathcal{U}}(R)_{\mathbf{1}}$, is the lowest ranked occurrence among $occ_{\mathcal{U}}(R)$.*

Note that suffix ordering is consistent only within a sequence but varies across sequences depending on implementations. For example, among BWTs that employ different strategies, some consider the global lexicographical order, yielding $occ_{\mathcal{U}}(AA)_{\mathbf{1}} = (\text{CCAA}, 3, 4)$ for $\mathcal{U} = \{\text{GGAA, CCAA}\}$, while others impose a strict order on input sequences, such as $\mathcal{U} = \{\text{GGAA}\#_1, \text{CCAA}\#_2\}$, resulting in $occ_{\mathcal{U}}(AA)_{\mathbf{1}} = (\text{GGAA}, 3, 4)$. This variability is extensively discussed in [18]. In any scenario, the identical Pro*k*rustean graph will be generated as long as the first occurrences are consistently considered.

The following definition of projected occurrences considers specific occurrences of maximal repeats, as an intermediate representation of the edges of the Pro*k*rustean graph.

Definition S11. *$ProjOcc_{\mathcal{U}}$ is a subset of occurrences of strings in $\mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$, where elements are specified as follows:*

1. *For every sequence $S \in \mathcal{U}$, the entire region of the sequence, i.e., $(S, 1, |S|) \in ProjOcc_{\mathcal{U}}$.*
2. *For every maximal repeat $R \in \mathcal{R}_{\mathcal{U}}$ and letters $\sigma_l, \sigma_r \in \Sigma$,*
   - *(1) If $R\sigma_r$ is left-maximal, then $(S, i, j-1) \in ProjOcc_{\mathcal{U}}$ given $(S, i, j) := occ_{\mathcal{U}}(R\sigma_r)_{\mathbf{1}}$.*
   - *(2) If $\sigma_l R$ is right-maximal, then $(S, i+1, j) \in ProjOcc_{\mathcal{U}}$ given $(S, i, j) := occ_{\mathcal{U}}(\sigma_l R)_{\mathbf{1}}$.*
   - *(3) If $R\sigma_r$ is left-non-maximal with σ_l and $\sigma_l R$ is right-non-maximal with σ_r, then $(S, i+1, j-1) \in ProjOcc_{\mathcal{U}}$ given $(S, i, j) := occ_{\mathcal{U}}(\sigma_l R\sigma_r)_{\mathbf{1}}$.*

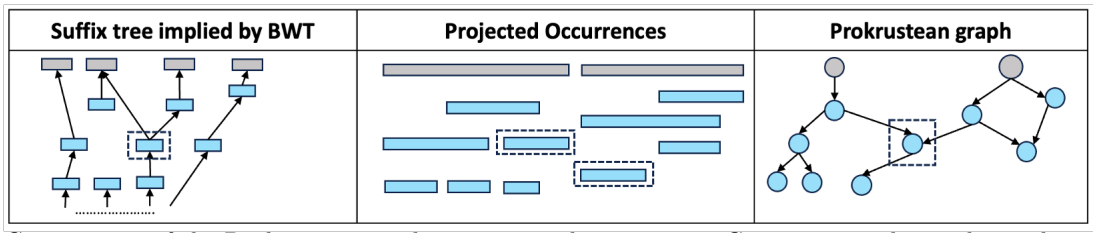


Fig. S11: Construction of the Pro*k*rustean graph via projected occurrences. Gray vertices, depicted as either rectangles or circles, represent sequences in $\mathcal{U}$, while blue vertices denote maximal repeats of $\mathcal{U}$. The suffix tree is shown flipped to better illustrate the correspondence, providing intuition for the "bottom-up" approach. Dotted rectangles in each diagram refer to a maximal repeat R. First occurrences of substrings like $\sigma_l R$ and $R\sigma_r$ are identified through the construction of $ProjOcc_{\mathcal{U}}$. Then, the nested structure of projected occurrences derive incoming edges of v_R in the Pro*k*rustean graph.

Refer to Figure S11 for intuition that $ProjOcc_{\mathcal{U}}$ is indirectly deriving the locally-maximal repeat regions in $\mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$. Although the same maximality conditions are used in both Theorem S6 and the definition of $ProjOcc_{\mathcal{U}}$, the occurrences in $ProjOcc_{\mathcal{U}}$ are organized in a nested manner, allowing locally-maximal repeat regions to be inferred through their inclusion relationships. So, $ProjOcc_{\mathcal{U}}$ is a projected image of the Pro*k*rustean graph of $\mathcal{U}$. The decoding rule for $ProjOcc_{\mathcal{U}}$, necessary for reconstructing the Pro*k*rustean graph, is articulated in the following proposition and theorem.

The proposition below is useful to prove the main theorem.

Proposition 5. *For every maximal repeat $R \in \mathcal{R}_{\mathcal{U}}$, $occ_{\mathcal{U}}(R)_{\mathbf{1}}$ is in $ProjOcc_{\mathcal{U}}$.*

Proof. If a maximal repeat is a whole sequence in $\mathcal{U}$, it is trivially in $ProjOcc_{\mathcal{U}}$. So, assume every occurrence of R in $\mathcal{U}$ is extendable by some letters.

Let $(S, i, j) := occ_{\mathcal{U}}(R)_{\mathbf{1}}$, and define $\sigma_l := S[i-1]$ and/or $\sigma_r := S[j+1]$. One of σ_l or σ_r might not be defined, so consider $(S, i-1, j) = occ_{\mathcal{U}}(\sigma_l R)_{\mathbf{1}}$ without loss of generality.

If $\sigma_l R$ is right-maximal, then $occ_{\mathcal{U}}(R)_{\mathbf{1}}$ trivially belongs to $ProjOcc_{\mathcal{U}}$ by 2-(2) of Definition S11. Otherwise, $\sigma_l R$ is right-non-maximal with some σ_r, so $(S, i, j+1) = occ_{\mathcal{U}}(R\sigma_r)_{\mathbf{1}}$ must hold.

Consequently, whether $R\sigma_r$ is left-maximal or left-non-maximal with σ_l, $occ_{\mathcal{U}}(R)_{\mathbf{1}} \in ProjOcc_{\mathcal{U}}$ holds because conditions 2-(1) or 2-(3) of Definition S11 apply, respectively. The symmetric argument assuming $(S, i, j+1) = occ_{\mathcal{U}}(R\sigma_r)_{\mathbf{1}}$ confirms the same results. □

The following theorem elaborates the decoding rule. Let the relative occurrence of (S, i, j) within (S, i', j') be $(str(S, i', j'), i - i' + 1, j - i' + 1)$ defined only if $(S, i, j) \subsetneq (S, i', j')$.

Theorem S7. *$e_{T,p,q} \in \mathcal{E}_{\mathcal{U}}$, i.e., (T, p, q) is a locally-maximal repeat region in some $T \in \mathcal{U} \cup \mathcal{R}_{\mathcal{U}}$ if and only if there exist $(S, i, j), (S, i', j') \in ProjOcc_{\mathcal{U}}$ satisfying:*

- *$(S, i, j) \subsetneq (S, i', j')$, and*
- *the relative occurrence of (S, i, j) within (S, i', j') is (T, p, q), and*
- *no region $(S, x, y) \in ProjOcc_{\mathcal{U}}$ satisfies $(S, i, j) \subsetneq (S, x, y) \subsetneq (S, i', j')$.*

Proof. The forward direction is composed of three parts. We show there exists some $(S, i, j) \in ProjOcc_{\mathcal{U}}$ corresponding to (T, p, q) that is specified by Theorem S6, and then show there exists some $(S, i', j') \in ProjOcc_U$ such that the relative occurrence of (S, i, j) in the region is (T, p, q). Lastly, the third statement is shown.

First, there exists an occurrence $(S, i, j) \in ProjOcc_{\mathcal{U}}$ that captures the same string and shares some left or right extension with (T, p, q). (T, p, q) must be satisfying exactly one of three items labeled (c) in Case S6.1, Case S6.2, and Case S6.3 conditioned by extensions of (T, p, q). Then the corresponding item of (a) describes a maximality condition around R that is equally described in 2-(1), 2-(2), and 2-(3) in Definition S11 to derive elements in $ProjOcc_{\mathcal{U}}$. So, there is always some (S, i, j) in $ProjOcc_{\mathcal{U}}$ chosen for (T, p, q) by the agreement in letter extensions, i.e., either $S[i-1] = T[p-1]$ or $S[j+1] = T[q+1]$ or both holds.

Next, there exists some (S, i', j') such that the relative occurrence of (S, i, j) within (S, i', j') is (T, p, q). The construction of (S, i, j) showed either $S[i-1] = T[p-1]$ or $S[j+1] = T[q+1]$ holds, so assume $S[i-1] = T[p-1]$ without loss of generality. Since (T, p, q) is a locally-maximal repeat region, $str(T, p-1, q)$ co-occurs within T and hence $str(S, i-1, j)$ co-occurs within T. Therefore, (S, i, j) can be extended to some (S, i', j') so that T is captured and i is at $i' + p - 1$ reflecting the position of (T, p, q) within T. Similarly, $j = j' + q - 1$ holds, and hence (T, p, q) is recovered by $(str(S, i', j'), i - i' + 1, j - j' + 1)$.

We need $(S, i', j') \in ProjOcc_{\mathcal{U}}$ to finish showing the second statement. If T is in $\mathcal{U}$, then trivially $(S, i', j') = (S, 1, |S|)$ holds and $(S, 1, |S|) \in ProjOcc_{\mathcal{U}}$ is applied by item 1. in Definition S11. Otherwise, T is in $\mathcal{R}_{\mathcal{U}}$, then (S, i', j') is $occ_{\mathcal{U}}(T)_{\mathbf{1}}$ because the first occurrence of some subtring co-occurring within T was utilized for identifying (S, i, j). Proposition 5 confirms that $occ_{\mathcal{U}}(T)_{\mathbf{1}} \in ProjOcc_{\mathcal{U}}$.

Lastly, assume some (S, x, y) in $ProjOcc_{\mathcal{U}}$ satisfies $(S, i, j) \subsetneq (S, x, y) \subsetneq (S, i', j')$. Then a maximal repeat $R'(:= str(S, x, y))$ exists such that R' is a superstring of $R(:= str(T, p, q))$ and T is a superstring of R'. Then R' could be captured by extending (T, p, q) and R' occurs more than T, so (T, p, q) is not a locally-maximal repeat region, leading to a contradiction.

For the backward direction, assume (T, p, q) is not a locally-maximal repeat region for contradiction. Then some locally-maximal repeat region (T, p', q') must be found by extending (T, p, q) because $R := str(T, p, q)$ occurs more than T. Then by following the the same steps in the first paragraph utilizing Theorem S6 and Definition S11, (T, p', q') is used to identify an occurrence (S'', i'', j'') in $ProjOcc_{\mathcal{U}}$ that shares some letter extension. So, letting $R' := (T, p', q')$, a substring of form either $\sigma_l R'$ or $R'\sigma_r$ is used to find a first occurrence and the substring co-occurs within T, so (S'', x'', y'') appears within $occ_{\mathcal{U}}(T)_{\mathbf{1}}$ so that its relative occurrence within $occ_{\mathcal{U}}(T)_{\mathbf{1}}$ is (T, p', q'). Therefore, $(S'', i'', j'') \subsetneq occ_{\mathcal{U}}(T)_{\mathbf{1}}$. Also, $(S, i, j) \subsetneq (S'', i'', j'')$ because (T, p', q') is an extension of (T, p, q) that (S'', i'', j'') is an extension of (S, i, j) too. Lastly, $occ_{\mathcal{U}}(T)_{\mathbf{1}} = (S, i', j')$ indeed holds because of $(S, i, j) \subsetneq (S, i', j')$. That is, if $occ_{\mathcal{U}}(T)_{\mathbf{1}}$ is not (S, i', j') but some other occurrence of T in S'', then the corresponding occurrence of R in $ProjOcc_{\mathcal{U}}$ must be found in S'' instead of (S, i, j). Thus, the presence of (S'', i'', j'') leads to contradiction. □

Therefore, two occurrences in their "closest" containment relationship in $ProjOcc_{\mathcal{U}}$ imply their strings form a locally-maximal repeat region. Since every occurrence in $ProjOcc_{\mathcal{U}}$ originates from either a maximal repeat or a sequence, each represents a vertex in the Pro*k*rustean graph. Therefore, collecting these relative occurrences constructs the edge set of the Pro*k*rustean graph. Refer to Section S3 for the algorithm.

S3.3 Implementation

We briefly introduce the techniques implemented in our code base: https://github.com/KoslickiLab/prokrustean.

Firstly, the BWT is constructed from a set of sequences using any modern algorithm supporting multiple sequences [17, 21, 30]. The resulting BWT is then converted into a succinct string representation, such as a wavelet tree, to facilitate access to the "nodes" of the implied suffix tree. For this purpose, we used the implementation from the SDSL project [24]. The traversal is built on foundational works by Belazzougui et al. [4] and Beller et al. [6], as detailed by Nicola Prezza et al. [37]. Their node representation, based on LCP intervals, enables constant-time access to $occ_{\mathcal{U}}(R\sigma)$ and $occ_{\mathcal{U}}(\sigma R)$ for each substring R and letter $\sigma \in \Sigma$. This capability is crucial for verifying the maximality conditions outlined in Definition S11. Also, the first occurrences $occ_{\mathcal{U}}(R\sigma)_{\mathbf{1}}$ and $occ_{\mathcal{U}}(\sigma R)_{\mathbf{1}}$ can be identified as the start of the LCP interval of $R\sigma$ and σR within the representation, respectively.

Therefore, $ProjOcc_{\mathcal{U}}$ is collected by exploring the nodes of the suffix tree implicitly supported by the BWT of $\mathcal{U}$. The exploration requires $O(\log|\Sigma| \cdot |\mathcal{T}_{\mathcal{U}}|)$ time in total, because succinct string operations take $O(\log|\Sigma|)$ time to move over and extract node representations. Identifying a maximal repeat takes $O(|\Sigma|)$ time and then collecting its first occurrences used in $ProjOcc_{\mathcal{U}}$ takes $O(|\Sigma|^2)$, because each combination of occurrence, e.g., $|occ_{\mathcal{U}}(\sigma_l R \sigma_r)|)$ is accessible in constant time. In summary, constructing $ProjOcc_{\mathcal{U}}$ takes $O(\log|\Sigma| \cdot |\Sigma|^2 \cdot |\mathcal{T}_{\mathcal{U}}|)$ time and $O(|BWT| + |ProjOcc_{\mathcal{U}}|)$ space where $|BWT|$ is the size of the succinct string representing the BWT of $\mathcal{U}$.

Lastly, reconstructing the Pro*k*rustean graph using Theorem S7 takes $O(|ProjOcc_{\mathcal{U}}|)$ time, which is introduced in Section S3. Note that the computation need not consider every combination of occurrences in $ProjOcc_{\mathcal{U}}$; it leverages the property that an occurrence $(S, i, j) \in ProjOcc_{\mathcal{U}}$ can imply locally-maximal repeat regions with up to two extensions in S found in $ProjOcc_{\mathcal{U}}$. Therefore, $ProjOcc_{\mathcal{U}}$ is grouped by each sequence $S \in \mathcal{U}$, and the edge set $\mathcal{E}_{\mathcal{U}}$ can be built from each group. Hence, the total space usage is $O(|BWT| + |ProjOcc_{\mathcal{U}}| + |\mathcal{G}_{\mathcal{U}}|)$, where $O(|ProjOcc_{\mathcal{U}}|) = O(|\mathcal{G}_{\mathcal{U}}|)$. The space usage is primarily output-dependent that around $2|\mathcal{G}_{\mathcal{U}}|$ in practice, and the most time-consuming part is the construction of $ProjOcc_{\mathcal{U}}$ via traversing the implicit suffix tree.

S3.4 Algorithm: from projected occurrences to Pro*k*rustean graph

The algorithm below reduces the task of reconstructing the Pro*k*rustean graph from $ProjOcc_{\mathcal{U}}$ to a problem designed for a specific string $S \in \mathcal{U}$. For the subset of regions $\{(S', i, j) \in ProjOcc_{\mathcal{U}} \mid S' = S\}$, determine their inclusion relationships to identify the closest related pairs. This is easily parallelized as it processes the intervals

for each S separately. And the running time is linear to the size of the subset of regions per each S, hence $|ProjOcc_{\mathcal{U}}|$ in total.

An important assumption in the algorithm is that for any region $(S, i, j) \in ProjOcc_{\mathcal{U}}$, within the subset $\{(S', i, j) \in ProjOcc_{\mathcal{U}} \mid S' = S\}$, there are at most two regions (S, i', j') such that $(S, i, j) \subsetneq (S, i', j')$ and no (S, x, y) exists where $(S, i, j) \subsetneq (S, x, y) \subsetneq (S, i', j')$. It is straightforward to check the property.

Algorithm S5 Find recursive inclusions of intervals

Input: A set of intervals I pre-ordered by first and then second element.
Output: Interval pairs such that each (I, I') satisfies $I \subsetneq I'$ and no other interval I'' satisfies $I \subsetneq I'' \subsetneq I'$.

```
Set Ints := Ints[pos] = intervals in I starting from pos. (accessed by Ints[pos][idx]).
 Set IdxMap := IdxMap[pos] = 0 for each starting position pos in Ints.
 Set PostMap := PostMap[pos] = nextPos per each adjacent positions (pos,nextPos) in Ints, and nextPos
 is NULL if pos is rightmost.
 Set curPos = The rightmost position of Ints.
 Set curIdx = 0
 Set Pairs = ∅
 while curIdx ≠ NULL do
    if curIdx > 0 then
       //Adjacent intervals sharing start positions completely satisfy the conditions.
        Pairs =Pairs∪{(Ints[curPos][curIdx − 1], Ints[curPos][curIdx])}
    end
    Set postPos = PostMap[curPos]
     while postPos ≠ NULL do
       postIdx= IdxMap[postPos] if Ints[postPos][postIdx] ⊄ Ints[curPos][curIdx] then
          break while
       end
       while Ints[postPos][postIdx + 1] ⊂ Ints[curPos][curIdx] do
          postIdx += 1
       end
       Pairs =Pairs∪{(Ints[postPos][postIdx], Ints[curPos][curIdx])}
        if postIdx is NOT last in Ints[postPos] then
          IdxMap[postPos] = postIdx + 1
       end
       else
          IdxMap[postPos] =NULL
           PostMap[curPos] = PostMap[postPos]
       end
       postPos = PostMap[curPos]
    end
    if curIdx is NOT last in Ints[curPos] then
       curIdx += 1
    end
    else if curPos is not first in Ints then
       curPos= previous position of curPos in Ints
        curIdx= IdxMap[curPos]
    end
    else
       curIdx= NULL
    end
 end
Return Pairs
```